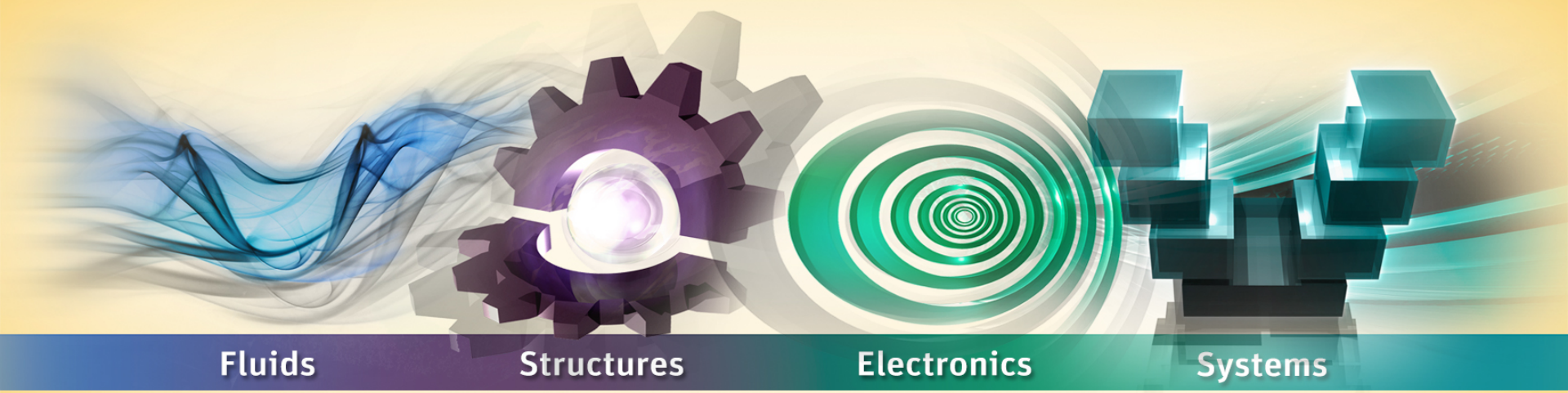
ANSYS

Realize Your Product Promise®

SCADE TRAINING



SCADE Suite Compiler Verification Kit

ESTEREL Technologies

AGENDA

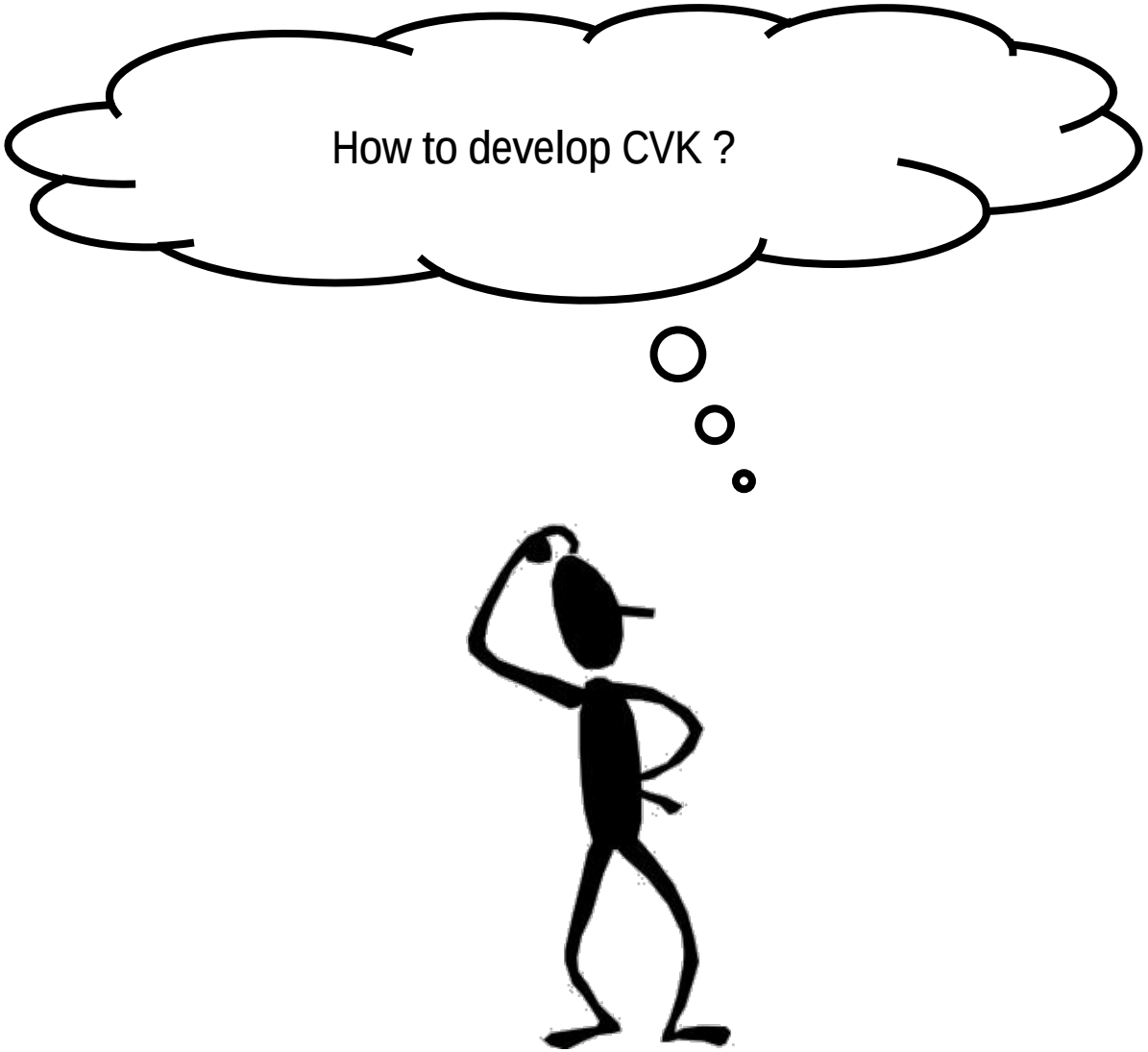
Introduction to the strategy for developing CVK

Verification process of SCADE Suite development environment

CVK product overview

Adapt CVK to cross platform and targe

Exercise: CVK execution on a host platform



Strategy for Developing CVK

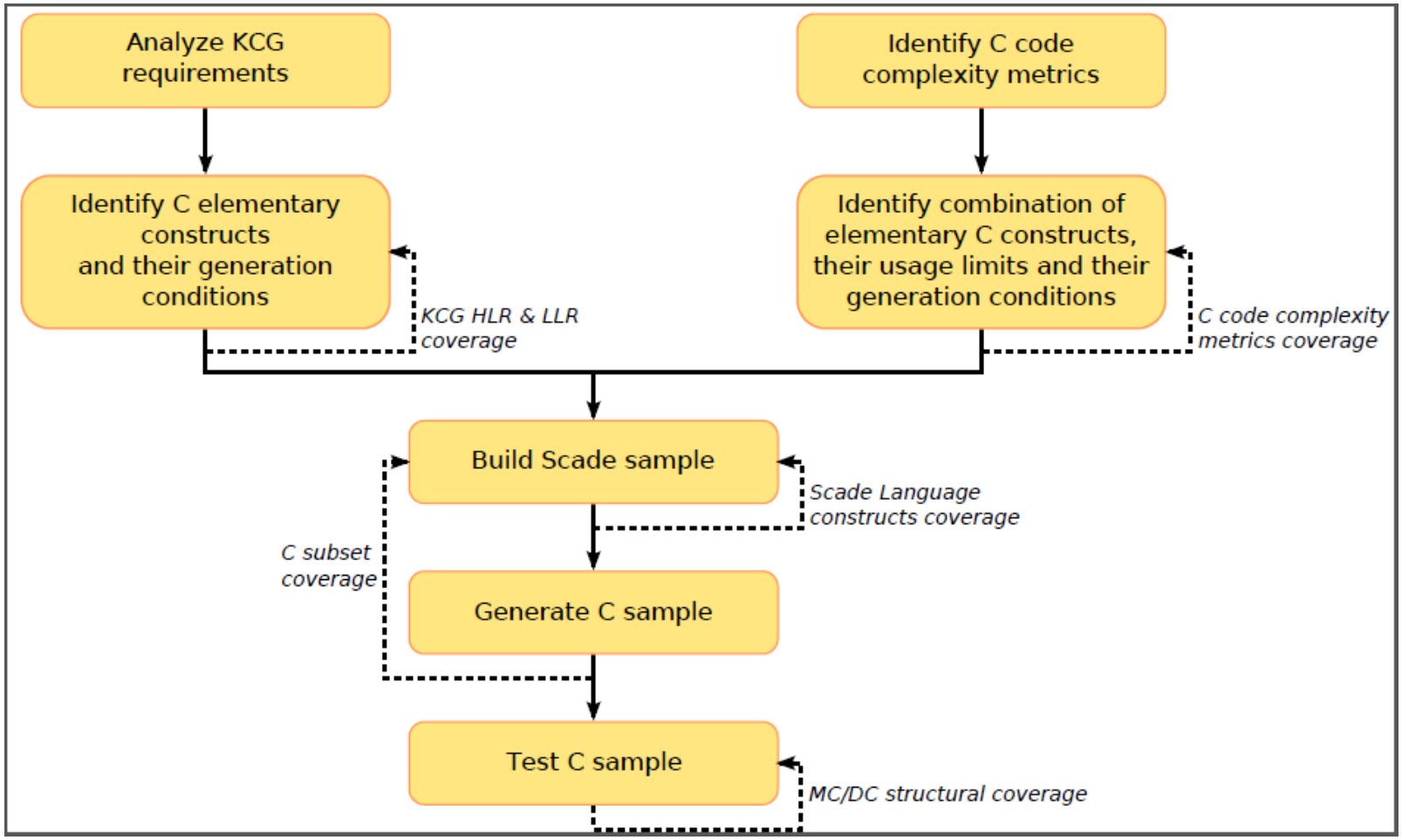
The Esterel Technologies R&D has performed following steps:

- Identify the C elementary constructs generated with KCG:
  - By analyzing the KCG software requirements - High Level and Low-Level Requirements (HLR and LLR)
- Define relevant complexity metrics for KCG-generated code:
  - By analyzing compilers limits defined in C standards (ANSI, ISO) and compilers documentation
- Identify the combination of elementary C constructs generated by KCG that make sense in the compiler verification
  - Combinations directly based on complexity metrics previously identified
  - Objective to cover a finite set of combinations with a high level of complexity

Strategy for Developing CVK

- Build the C sample:
  - A suite of Scade samples, covering all Scade constructs, built as material for C code generation
  - Each elementary C construct and its combination, generated from Scade samples root nodes with appropriate KCG options
  - Coverage of this C subset by the C sample, required and verified
- Develop a test harness, exercising the C sample with input vectors and verifying that output vectors conform to the expected ones
- Test execution on a host platform to verify:
  - Conformance of outputs to the expected ones
  - Modified Condition/Decision Coverage (MC/DC) coverage at a C code level
- Test execution for each selected target platform to verify:
  - The adaptation to a CVK specific cross-environment capabilities (portability)
  - The correctness of effective output vectors on this platform

Strategy for Developing CVK



CVK Definition

A tests suite whose purpose is to verify that a compiler correctly compiles code generated by SCADE Suite KCG

Developed for the verification of a development environment containing KCG

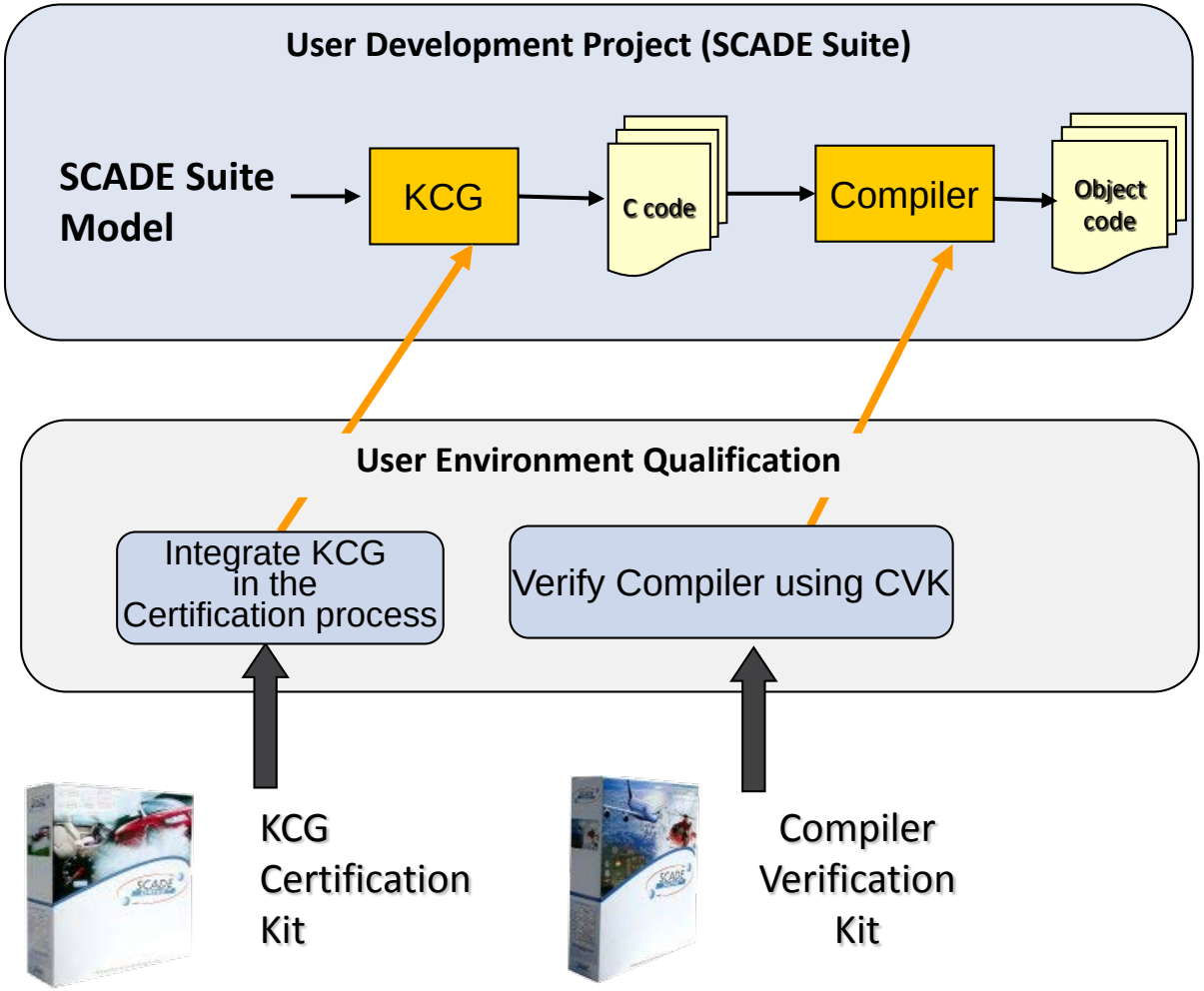
Supports early verification of the correctness and consistency between the development tools chain and the target platform

Addresses the verification of target compatibility with the architecture and LLR contained in SCADE models

Does not validate the C compiler or processor

Not an executable software and not a tool

KCG and CVK Role

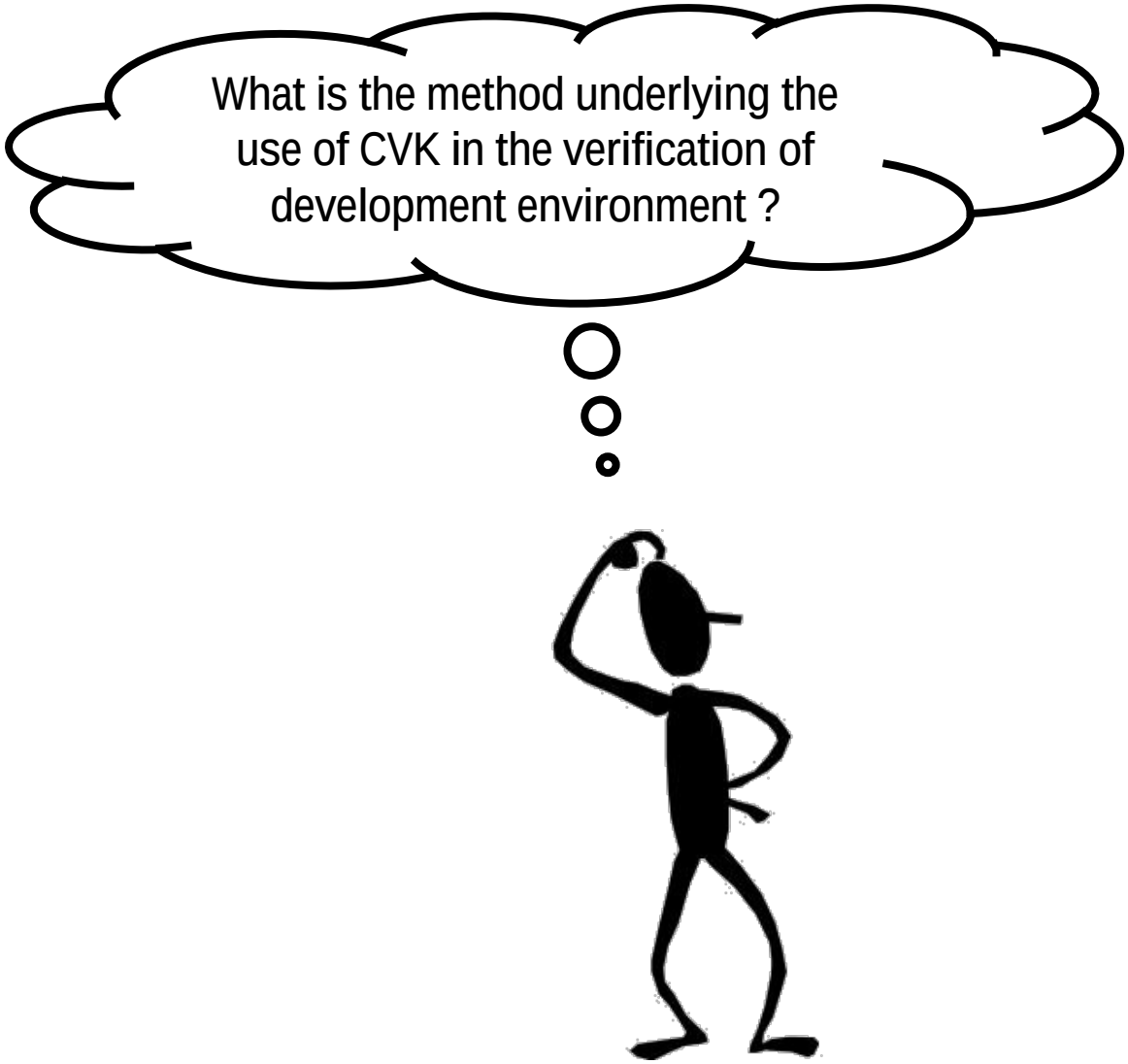


- **KCG qualification ensures that Source Code conforms to LLR developed with SCADE Suite**
- **CVK checks that the C compiler correctly compiles the C code generated by KCG**

KCG and CVK Role

Verification activities to be performed by the applicant, on the basis of the CVK C-sample

- Generating C Source Code from CVK SCADE model and comparing with reference C-sample
- Compiling/linking/loading the C Source Code on target
- Running the test cases on target and verifying that expected results are obtained



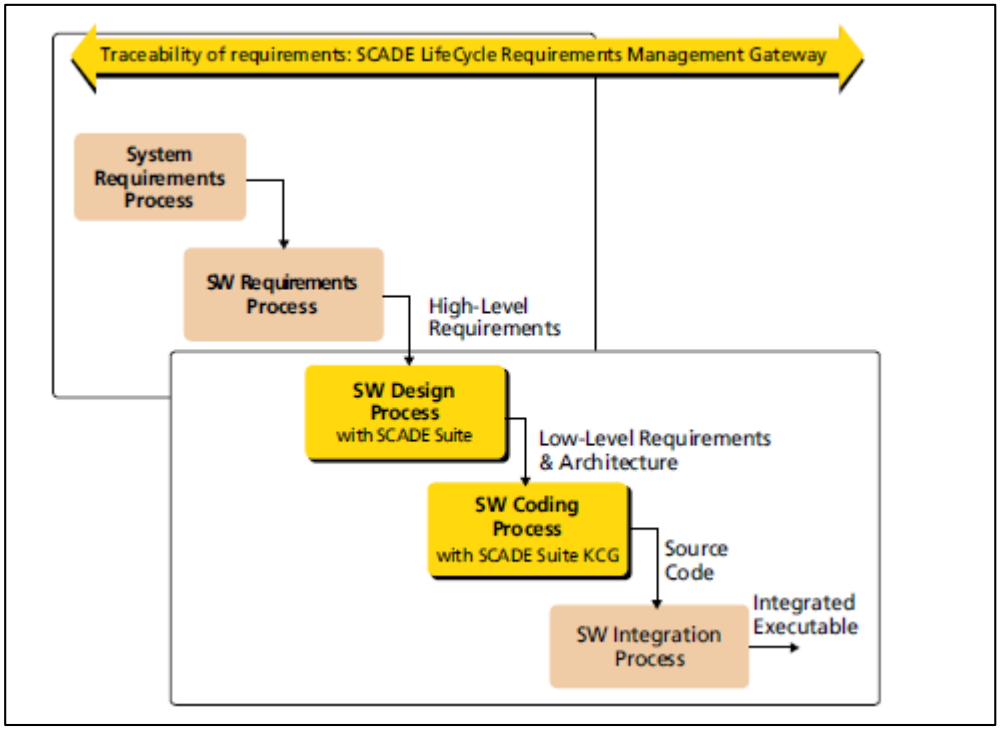
CVK IN SCADE SUITE DEVELOPMENT VERIFICATION

SCADE SUITE DEVELOPMENT FLOW

COMBINED TESTING PROCESS

VERIFICATION USING CVK

SCADE Suite Development Flow



- **Developing with SCADE Suite editor, the Software Requirements down to a level where they can be coded (LLR)**
- **Coding performed automatically with KCG, the qualifiable code generator**

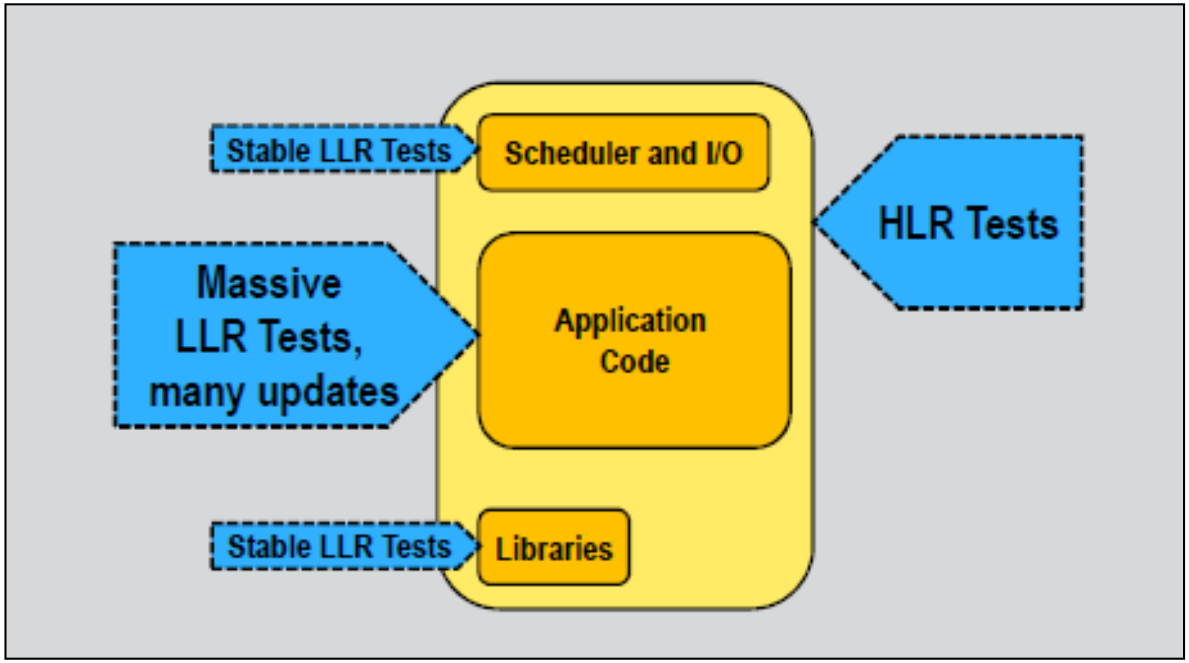
CVK IN SCADE SUITE DEVELOPMENT VERIFICATION

SCADE SUITE DEVELOPMENT FLOW

COMBINED TESTING PROCESS

VERIFICATION USING CVK

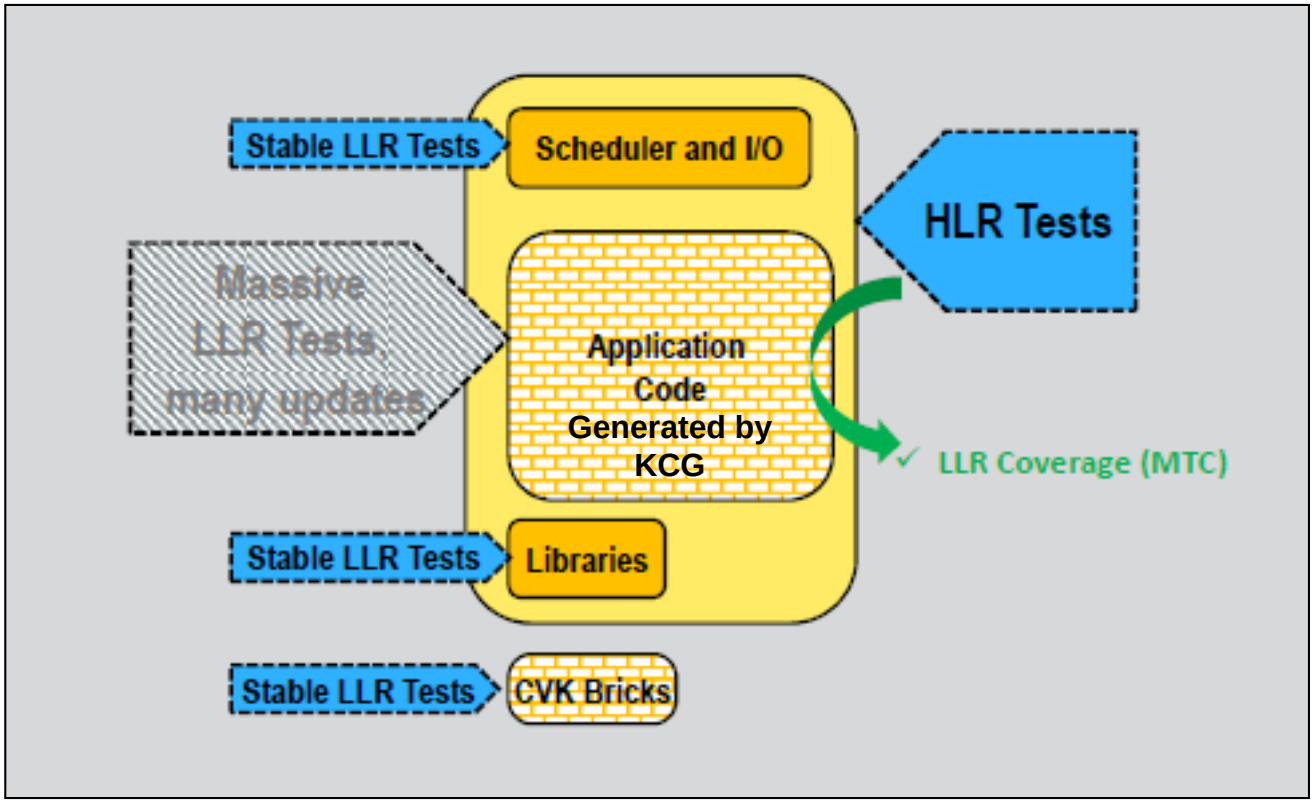
Combined Testing Process (CTP)



Traditional Testing Process (without SCADE)

In a traditional development process, testing mixes the search for design, coding and compilation errors

Combined Testing Process (CTP)



CTP

Optimization by CTP of the testing effort, using a divide-and-conquer approach for the Software part (SW) developed with SCADE tools

CTP

Benefits from KCG qualification and from the characteristics of the generated code:

- Conformance of the SCADE LLR to the HLR
  - Verified at the model level, by review and a combination of HLR-based testing on host and on target platforms
    - Using SCADE LifeCycle Testing Environment tool (TE)
    - Using Model Test Coverage verifying the coverage of SCADE LLR by the HLR-based test cases
- Conformance of the Source Code to the SCADE LLR
  - Ensured by KCG qualification
- Verification of “source code to object code transformation” by the compiler
  - Verified by analysis and low-level testing, with MC/DC structural code coverage:
    - For the complete C hand-written code
    - On a representative sample of the generated code, named “C sample”, which is contained in CVK

CTP

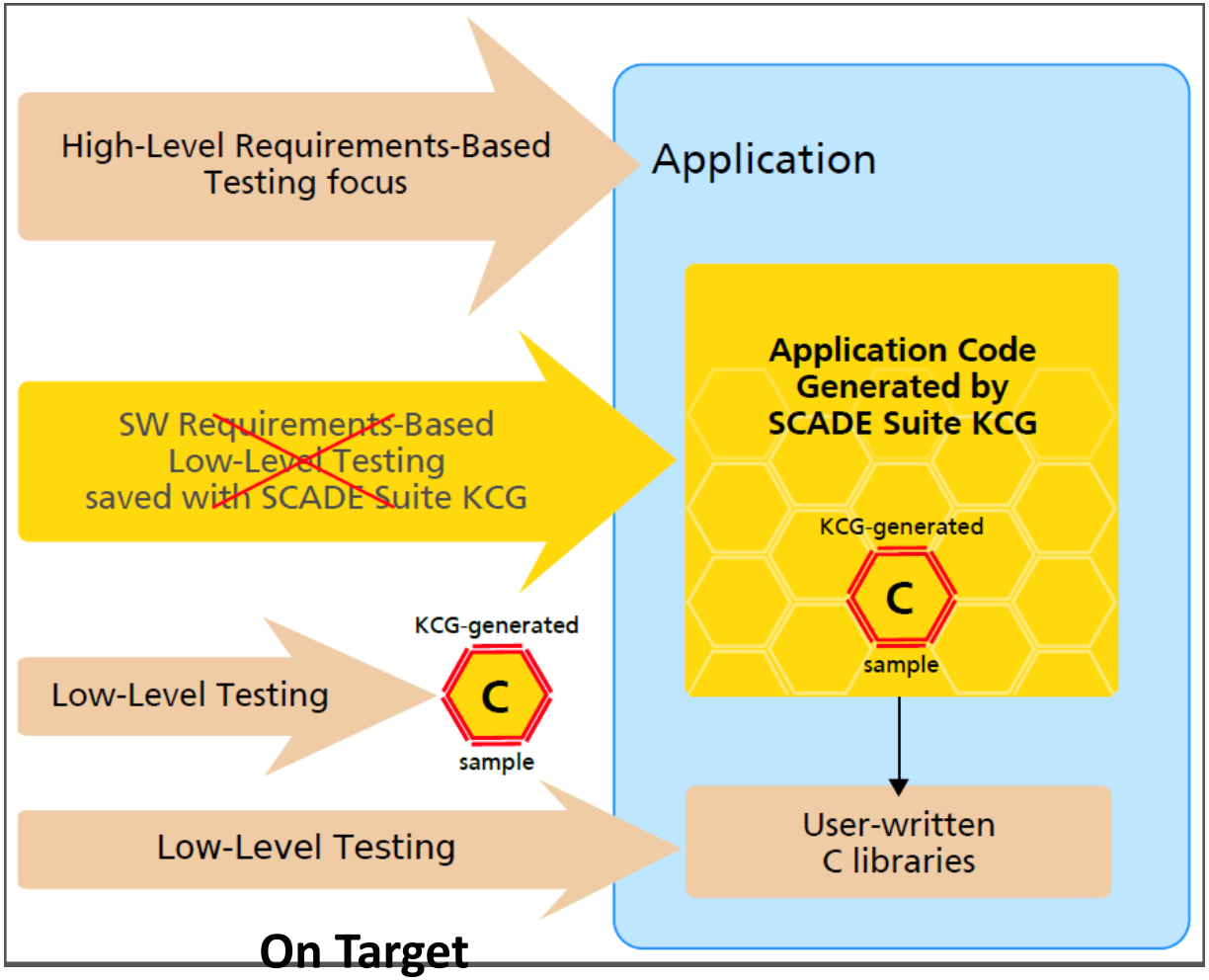
Combined testing on the host platform

- Executes all test cases that can be done on the host
- Analyzes the test results
- Code and model coverage shortfalls analyzed and resolved
- Also prepares the target testing

CTP

- Includes:
  - LLR-based tests of the C hand-written code: tested traditionally, with structural code coverage analysis and resolution
  - HLR-based simulation tests, with SCADE LifeCycle QTE tool in order to verify the SCADE LLR
  - SCADE LLR reviews
- Its Objective:
  - conformance to the HLR
  - HLR coverage
  - LLR coverage (for SCADE LLR, LLR coverage is model coverage), in order to verify the complete coverage of LLRs by the HLR-based test cases

CTP



CTP

Combined testing on the target platform

- Covers following steps:
  - Preparing the coding environment
    - Installation of the Source to Object Code Compiler/Linker with a stable version
      - Appropriate compiler options are selected (for example, no or little optimization)
      - The same environment is used for C manual code and KCG-generated code in the project

- Verifying by review and Low Level testing a representative sample of the KCG generated code
  - **CVK** contains such a representative sample, named "**C sample**"
    - The C sample is tested with full structural code coverage analysis (MC/DC), using the test suite contained in CVK
  - If there is C imported code, integration testing between SCADE Suite-generated code and imported code is performed

CTP

- Verifying the source to object code traceability
  - *Guidelines for Approving Source Code to Object Code Traceability, CAST-12 Position Paper December 2002*

    "If the compiler generates Object Code that is not directly traceable to Source Code statements, then, additional verification should be performed on the Object Code to establish the correctness of such generated code sequences.

    A compiler-generated array-bound check in the Object Code is an example of Object Code that is not directly traceable to the Source Code"

- Performing systems requirements-based software and hardware / software integration tests
  - Cover All systems requirements allocated to software are covered by those tests
  - Coverage of the HLR and LLR has to be achieved by the host and target testing

CVK IN SCADE SUITE DEVELOPMENT VERIFICATION

SCADE SUITE DEVELOPMENT FLOW

COMBINED TESTING PROCESS

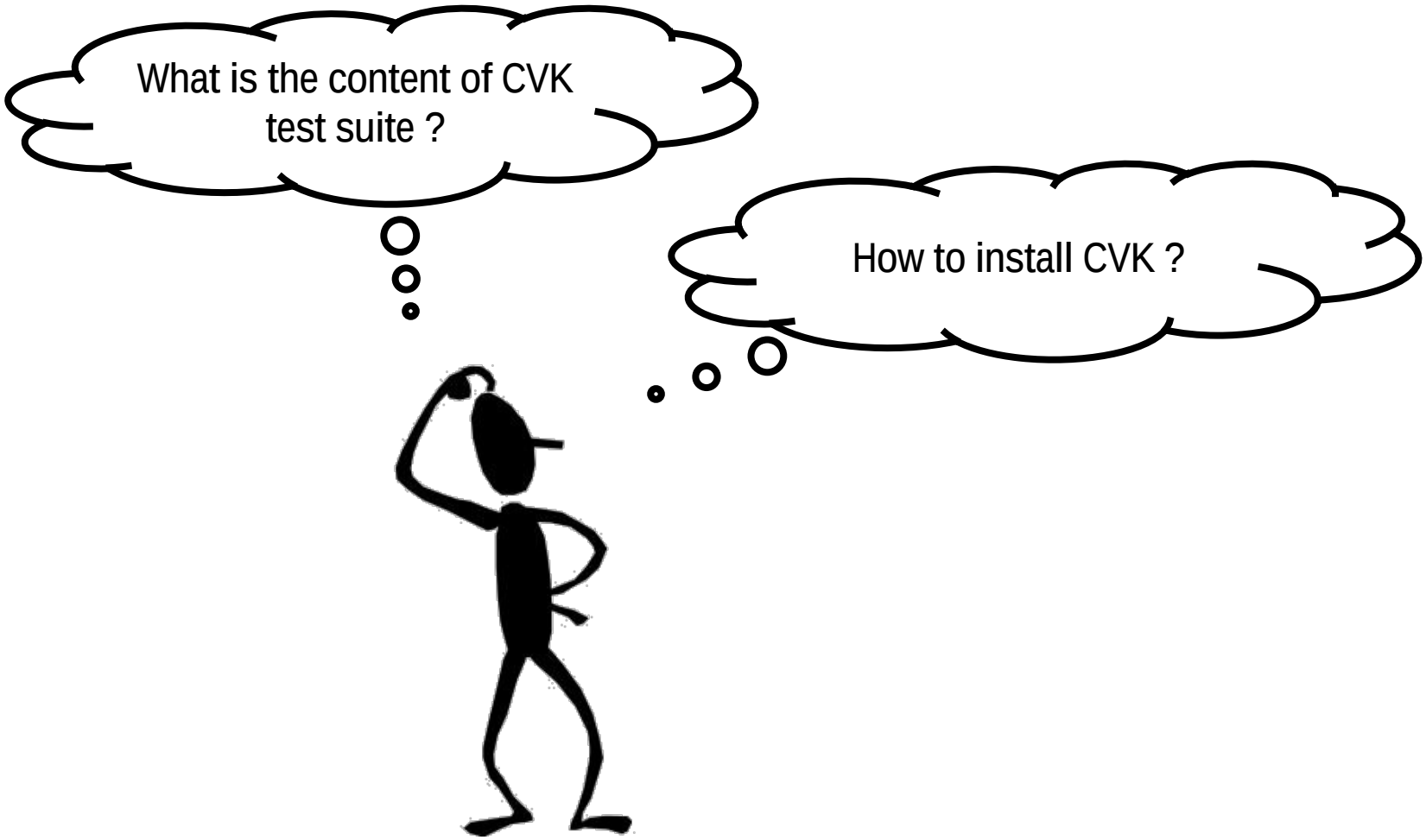
VERIFICATION USING CVK

Verification using CVK

Compiler verification with CVK completes, for KCG-generated code only, the verification of

- Object Code, as summarized in DO-178B Table A-6 , "*Testing Of Outputs of Integration Process*", by verifying that the Object Code conforms to the source code

- Structural coverage, as summarized in DO-178B Table A-7, "*Verification Of Verification Process Results*" by verifying that all building blocks of the generated code have been structurally covered

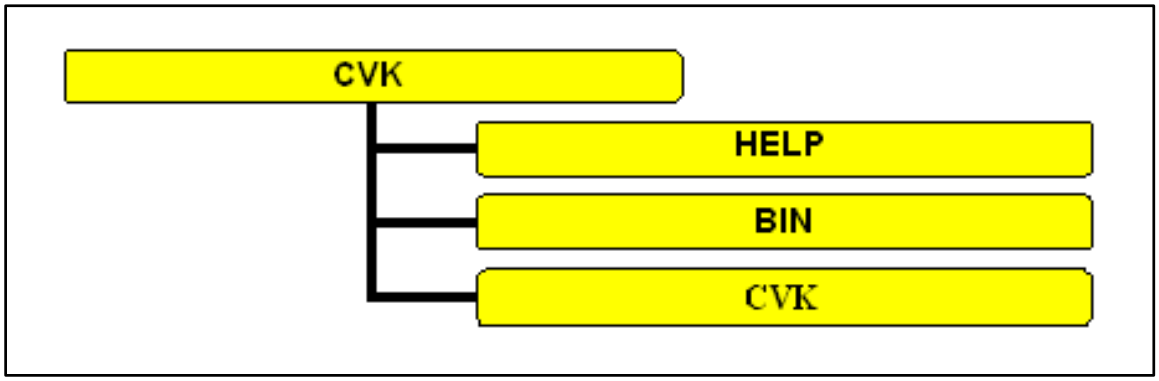


CVK PRODUCT OVERVIEW

CVK PACKAGE CONTENT

CVK INSTALLATION

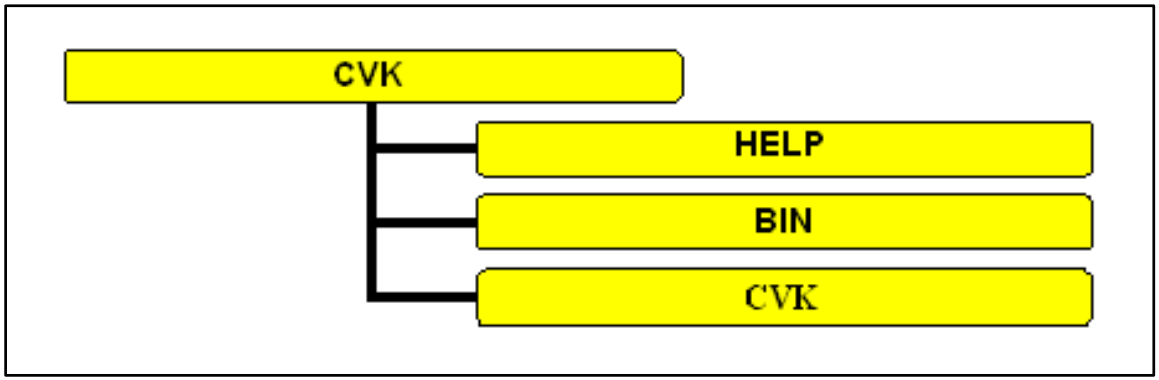
CVK Package Content



Help directory contains documentation

- The CVK User Manual that provides
  - the installation and use instructions
  - The customization guidelines
- The CVK Reference Manual that contains
  - The Scade samples description
  - The test cases description including a specification of the C sample
- The Release Notes and associated documents describing fixed and remaining issues

CVK Package Content



Bin directory contains

- Utilities to check the CVK installation integrity

CVK directory contains

- The Test DataBase (TDB) in following folders
  - *CONFIGURATION*
  - *DATA*
  - *RESULTS*
  - *SCRIPTS*

CVK Package Content: TDB

- *CONFIGURATION* folder contains Tcl configuration script files to customize the CVK environment according to the user environment
  - **`CVK_Environment.tcl`**: defines the **TDB procedures environment** such as paths and compilation options
  - **`CVK_TestCasesList.tcl`**: test cases list run during one full TDB execution
  - **`CVK_TestCasesList_real_free.tcl`:** test cases list run during one full TDB execution except tests cases that uses real as input, output or sensor
  - **`CVK_BuildCustom.tcl`**: customizable procedures that generate the commands for compiling, linking and running the test programs
  - **`CVK_UnitTest_TestCasesList.tcl`**: test cases list run for unit testing
  - **`CVK_UnitTest_TestCasesList_real_free.tcl`**: test cases run for unit testing except tests cases that uses real as input, output or sensors

CVK Package Content: TDB

- *DATA* folder contains
  - **C sample** to verify the C compiler
  - **Scade samples**
    - Enables CVK users to (re)generate the C sample with SCADE Suite KCG in their host and cross platform environment
    - Allows comparing the generated code with the expected results provided in the CVK test suite and verifying SCADE Suite KCG installation
  - **Test cases** exercising the C sample with full MC/DC coverage
  - **TestCCode folder that contains C source files that are common to all test cases**
    - `cvk_def_<W16|W32|UW32|SP|DP>.h`: user macros for integers (WORD16, WORD32, UNSIGNED WORD32, MAX_INT, SAFE_MAX_INT,…) and floating point numbers (SIMPLE_PRECISION, DOUBLE_PRECISION, EPSILON, MAX_FLT, SAFE_MAX_FLT,…) implementation
    - `cvk_def_common.h`: user implementation for bool and char types (kcg_bool and kcg_char)
    - `macro_user.h`: user macros for initialization/end of test programs
    - `check.[c|h]`: user function to analyze the computed result with regard to the expected result
    - C template files to define user Assume and Guarantee macro functions

CVK Package Content: TDB

- C Sample contains
  - All elementary C constructs that may be generated by KCG from a SCADE Suite model
  - Additional combinations of C constructs to check the behavior of the C compiler on “typical” combinations generated by SCADE Suite KCG
  - Complex and deeply nested C code to stress the compiler
- A C sample element is defined by the combination of a SCADE model, package, root node and a set of KCG code generation options, that are used to generate this C sample element with KCG
- The test cases list is built with the objective that every element of the C subset is contained at least once in the generated C sample
  - This objective is verified by generated code analysis

CVK Package Content: TDB

- Scade Sample
  - composed of three main SCADE Suite models
    - CVK_Basic: covers elementary Scade constructs with a low level of complexity

    - CVK_Combination: contains “typical” combination of Scade elementary constructs with a medium level of complexity

    - CVK_Limit: contains complex and/or deeply nested expressions, data structures and calls, that stress the compiler
      - Upon request, Esterel Technologies will provide variants of them with different parameter values

CVK Package Content: TDB

- Test cases contain test vectors that ensure full MC/DC coverage of the C sample
  - contain for numeric operators, both normal-range values, singular points and min/max values
- Test Harness
  - Small main programs
  - Each one exercises the code of one test case (generated from one root node located in a package of a SCADE model with one KCG options set)
  - Cyclically
    - gets the input vector from memory, defined in **<RootNodeName>**_io.c
    - calls the cyclic function generated by SCADE Suite KCG
    - retrieves the output vector and compares it to the expected one, defined in **<RootNodeName>_io.c**
  - If all output vectors match the expected one, the test harness **exits with return code 97**
    - This value is chosen because it is very unlikely that a program returns it accidentally

CVK Package Content: TDB

Specification regarding the options set (configuration)

- <ConfigurationName> = configuration name
- Following KCG options are common to all configurations and not listed in the table
  -user_config cvk_param.h, to take into account CVK parameters set by the user into KCG generated code
  -target C, to generate C code
  -target_dir <path>, to take into account CVK parameters set by the user into KCG generated code

Configuration	KCG Options	Purpose with respect to C Compiler
Standard	-O 1	Typical profile for modern compiler, with a good level of traceability and a reasonable level of complexity
StdWrap	-O 1 –wrap_c_obs	The same as “Standard” configuration except that arithmetic, relational and logical operators are wrapped to C protected macro definitions
StdNoBitwise	-O 1 –significance_length 127	The same as "Standard" configuration except that no bitwise C operator is used in generated code
StdSignLength	-O 1 –significance_length 127	The same as “Standard” configuration with a value set to 127 for significant initial characters of SCADE Suite KCG generated identifiers
Optim	expall -O 2	High degree of optimization. May generate complex expressions and decrease traceability. May increase nesting of expressions and control structures in the C code
Global	-global_root_context -O 1	Idem “Standard” except the inputs and context of the root node are global variables
Debug	-debug –O 0	Decomposed C code and maximum of visibility
DebugMacroOnAssert	-debug –O 0 -macro_on_assert	The same as "Debug" with use of macro on assertion, macros that shall be defined by user
O0NoDebug	–O 0	Decomposed C code (high number of locals)
O0NoDebugStatic	–O 0 -static	The same as “O0NoDebug” configuration with the C “static” keyword for each local variable declaration

CVK Package Content: TDB

Data of each test harness are located in:

- *DATA/<ModelName>/<PackageName>/<RootNodeName>/<TestCase Name>/TestHarness*

  where

<TestCaseName> = test case identification by appending the code generation configuration name to the root node name:

<TestCaseName> = <RootNodeName>_<ConfigurationName> such as “Booleans_Standard”

- Example*: DATA/CVK_Basic/Logical/Booleans/Booleans_Standard*

CVK Package Content: TDB

Implementation size and precision are defined as follows:

- For integer implementation, there are three configurations covered by test cases
  - Int (W16 = WORD16)
  - Long int (W32 = WORD32)
  - Unsigned long int (UW32 = UNSIGNEDWORD32)
- For real implementation, there are two configurations covered by test cases
  - Float (SP = SIMPLE_PRECISION,)
  - Double (DP = DOUBLE_PRECISION)
- kcg_bool, kcg_char, kcg_real and kcg_int default definition (from bool, char, real and int scade predefined types) are given in file named kcg_types.h via a “typedef” declaration
  - The user could redefine last ones in definition files:
  - cvk_def_common.h and cvk_def_<W16|W32|UW32|SP|DP>.h
- Comparison functions used by the test harness to compare actual output vectors with expected values must be checked with unit tests included in the CVK package

CVK Package Content: TDB

- *RESULTS* folder is generated during first CVK execution, contains
  - **Test reports** stored in ***LogFiles*** sub-folder and named:

    `<script name>_<configuration>_<date>_<time>.txt`

    where

    `<script name>`: name of Tcl script used (without prefix CVK_)

    `<configuration>:` configuration name passed as argument

    `<date>` ::= M-DD-YY (D: day, M: month in letters, Y: Year)

    `<time>` ::= HH-MM-SS (hours, minutes, seconds)
  - `Build.log` and `Run.log` stored in *current_<configuration>* sub-folder, containing the test harness build and execution traces

    <configuration> is either SP_W16|SP_W32|SP_UW32|DP_W16|DP_W32|DP_UW32 depending on what is defined in ***CONFIGURATION/CVK_Environment.tcl*** by the user
  - Data generated for each test case named <TestCaseName> are stored in *Current/<TestCaseName>* sub-folder

    Example: *Current/ArithmeticInt_Standard*

CVK Package Content: TDB

- *SCRIPTS* folder contains Tcl scripts files that run automated CVK test procedures as follows:
  - Installation verification Tcl script files:
    - `CVK_Check.tcl` to verify the CVK installation integrity (not KCG one)
    - `CVK_KCG_Verification.tcl` to check that code generated by SCADE Suite KCG in the user environment from the Scade sample is the same as the reference C sample delivered with CVK
  - Common Tcl script files:
    - `CVK_Tools.tcl`, Tcl CVK library, implements several utility Tcl procedures
    - `CVK_Init.tcl` called by all other Tcl scripts in order to initialize their launch:
      - Load `CVK_Tools.tcl`
      - Check the command line
      - Set the following CVK environment variables:
        - SCADE KCG installation path (defined in ***CONFIGURATION/CVK_Environment.tcl***)
        - KCG settings
        - CVK version
        - CVK banner
        - result file name (default one is `CVKResult_${date}.txt`)

CVK Package Content: TDB

- Metrics Tcl script file:
  - `kcgmetrics2.tcl`, Tcl CVK library, to compute the two following metrics:
    - Number of characters in a logical source line and
    - Number of nesting levels for #included files.
  - To execute it, run the following command in the output directory of KCG:

```
tclsh kcgmetrics2.tcl -scade <SCADE_installation> [-I <include_dir>] [-h]
```

Where:

- <SCADE_installation> refers to the SCADE installation directory
- [-I <include_dir>] parameter is optional and allows specifying additional include directories
- [-h] parameter is optional and displays help message

- Building test harness Tcl script files:
  - `CVK_ExecutableGeneration.tcl` to generate test programs (test harness)
    - Load `CVK_Init.tcl` when the test cases list is empty
    - Load `CVK_BuildCustom.tcl` (from *CONFIGURATION* folder)
    - Run `CVK_ScriptFilesGeneration.tcl` to generate
      - <nodename>_build.* file (.bat for Windows) in charge of the test harness compiling/linking
      - <nodename>_run.* (.bat for Windows) in charge of running the test harness
    - Run `CVK_ExecutableBuild.tcl` to execute the <nodename>_build.* file

CVK Package Content: TDB

- Executing test harness Tcl script files:
  - `CVK_ExecutableRun.tcl` to run the test harness previously generated by the `CVK_ExecutableGeneration.tcl` script
    - Load `CVK_Init.tcl` when the test cases list is empty
- Building and executing test harness Tcl script files:
  - `CVK_ExecutableGenRun.tcl` to build and run the test harnesses
    - Execute `CVK_ExecutableGeneration.tcl`
    - Execute `CVK_ExecutableRun.tcl`

CVK PRODUCT OVERVIEW

CVK PACKAGE CONTENT

CVK INSTALLATION

CVK Installation

Prerequisites:

- SCADE Suite KCG is already installed
- Need a Tcl distribution version 8.4 or later (CVK automated test procedures are made of Tcl scripts)
  - Get tcl from *http://www.tcl.tk/software/*, free of charge

    Or reference the path of tcl binaries from SCADE Suite installation in the PATH environment variable: *<SCADE_DIR>\contrib\tcl\bin*

Install it on a root path as *"C:\CVK64"*

- This avoids problems because of the Windows path length limitation when executing *F_127_StdSignLength* test case

CVK Installation

Host Platform:

- Select for the user a platform with the following criteria:
  - `SCRIPTS/CVK_KCG_Verification.tcl` must be executed in the platform where KCG is run in production, listed in KCG Tool Accomplishment Summary document (TAS, in the certification Kit)
  - For other CVK scripts execution, it must be the cross compilation platform which is used in production
  - Update if any in `CONFIGURATION/CVK_BuildCustom.tcl`
    - compiler / linker tool names
    - object files target directory
    - object files extension
    - compiler / linker options

CVK Installation

Target Platform:

- Platform on which the real application is embedded
- CVK must be adapted to the user cross platform and target:
  - Cross compiler/linker
  - Operating system
  - Hardware

CVK Installation

Check the installation integrity

- Use "CVK*SCRIPTS**CVK_Check.tcl*"
  - Verifies that the CVK files are present and unchanged with respect to the reference product
  - Based on verification of checksums of each file, except documentation files
  - Verifies CVK, not SCADE Suite KCG product installation
- Execute on a batch window in the *SCRIPTS* folder:

```
tclsh CVK_Check.tcl
```

- This verification is passed if the following message is displayed

  Success: 0 different signature(s) over 2288 analyzed file(s)

Tip: By convention "tclsh" means the name of the user tcl shell. So, if the tcl shell name is not "tclsh", replace "tclsh" by the correct one such as "tclsh84"

Exercise 1

Exercise

The aim of this exercise is

- To run CVK on a host platform to verify the CVK flow
  - Step 1: check the CVK installation integrity
  - Step 2: update CVK user environment variables to the host platform
  - Step 3: update compiler / linker settings to the host platform
  - Step 4: verify the CVK and KCG installation data
  - Step 5: verify CVK comparison functions
  - Step 6: - build and run test harnesses on host
    - check the passed status of the test cases execution

Exercise 1

Exercise

Step 1: check the installation integrity

- Launch a Dos batch window
- Update PATH environment variable according to:
  - SCADE installation bin directory path (that contains KCG 6.4)
  - Tcl installation bin directory path

```
Path =
<SCADE_DIR>\bin;<SCADE_DIR>\contrib\tcl\bin;%path%
```

- Run the integrity check

CVK Installation

Update CVK user environment variables

- Edit the "CVK\CONFIGURATION\CVK_Environment.tcl" file
  - Set the scadeKCGPath variable to the KCG installation folder path

    Example:

```
set scadeKCGPath {C:\Program Files\Esterel Technologies\KCG6.4}
```

  - Set the following variables to expected values:
    - *realImplementation* variable with SP or DP (SIMPLE or DOUBLE PRECISION)
    - *intImplementation* variable with W16, W32 or UW32 (WORD16, WORD32 or UNSIGNEDWORD32)

    Example:

```
set intImplementation {W16}
set realImplementation {SP}
```

CVK Installation

Check the CVK and KCG installation data

- Use "*CVK\SCRIPTS\CVK_KCG_Verification.tcl*"
  - Verifies that code generated by KCG in the user environment from the SCADE sample is the same as the reference C sample delivered with CVK
  - The verification is applied either to:
    - All CVK test cases (default)
    - The list of test cases given in the command line if there are
- Execute on a batch window in the *SCRIPTS* folder:

```
tclsh CVK_KCG_Verification.tcl
```

  - The following message is displayed

    "Tests Results are stored in <pathname_of_the_results_file>"
  - The status is "OK" for all test cases in this results log file (*RESULTS* folder)

Exercise 1

Exercise

Step 2: update CVK user environment variables to the host platform

- Edit ..\CONFIGURATION\CVK_Environment.tcl
- Update following variables:
  - scadeKCGPath to the user local KCG 6.4 installation path
  - realImplementation to SIMPLE_PRECISION (SP)
  - intImplementation to WORD16 (W16)

```
set scadeKCGPath {<SCADE_DIR>}
set realImplementation {SP}
set intImplementation {W16}
```

Exercise 1

Exercise

Step 3: update compiler/linker settings to the host platform

- Edit ..\CONFIGURATION\CVK_BuildCustom.tcl
- Update GenBuild procedure as follows:

```
# Compiler name
    puts $batFileRef "@set vCompiler=gcc.exe"
# Linker name
   puts $batFileRef "@set vLinker=gcc.exe"
# Object files directory
    puts $batFileRef "@set vObjDir=win32"
 # Object files extension
    puts $batFileRef "@set vObjExt=.obj"
 # Compiler options
     puts $batFileRef "@set vCFlags=-O2 -pedantic"
 # Linker options
     puts $batFileRef "@set vLDFlags=-o \"${rootNodeName}${vExeExt}\""“
```

Exercise 1

- Update GenRun procedure as follows:
  - puts $batFileRef "\"${exePath}\""
- Update PATH environment variable according to GNU gcc binaries installation directory

Exercise

Exercise 1

Exercise

Step 4: verify the CVK and KCG installation data

- Run CVK_KCG_Verification.tcl
- Check that no error message is raised
- Tests Results are stored in <CVK_DIR>\CVK\RESULTS\LogFiles\KCG_Verification_<date>-<time>.txt
- When "_CVK_KCG_Verification end" is displayed, check in the results log file that the status is "OK" for all test cases

ANSYS

ESTEREL Technologies

Exercise

SOLUTION

Solution: Step 4

Exercise

In the result log file:

```
##########################################################################
##########################

>> CVK Test Case KCG verification summary <<

====================================

 * Total number              : 117

----------------------------------

 *  Successful KCG verification        : 117

 *  Failed KCG verification            : 0

====================================
```

How to run CVK tests suite on a target ?

USING CVK

PREREQUISITES BEFORE USING CVK

ADAPTING CVK TO CROSS PLATFORM AND TARGET

GENERATE AND RUN EXECUTABLE TEST CASES

Prerequisites Before Using CVK

Verify CVK comparison functions to ensure that the user cross-compiler compiles correctly these functions:

- Use CVK test suite that contains, with <type>= {bool, int, float}:
  - Nominal unit tests:
    - compare_<type>_nominal tests that check that reference and expected outputs are the same ones
    - For floating number tests comparison, three cases are covered:
      - The reference output values are exactly the same as expected
      - The reference output values are greater than expected ones but the difference between reference and expected outputs is below the tolerance threshold
      - The reference output values are lower than expected but the difference between reference and expected outputs is below the tolerance threshold

Prerequisites Before Using CVK

- Robustness unit tests
  - compare_<type>_incorrect tests that check that reference and expected outputs are not the same ones
  - For floating number tests comparison, two cases are covered:
    - The reference output values are greater than expected ones and the difference between reference and expected outputs exceeds the tolerance threshold
    - The reference output values are lower than expected and the difference between reference outputs and expected outputs exceeds the tolerance threshold

Prerequisites Before Using CVK

- Test procedures to run previous unit tests are the same ones as for the other CVK tests
  - Execute on a batch window in the *SCRIPTS* folder:
    - `tclsh CVK_ExecutableGeneration.tcl -f ../CONFIGURATION/CVK_UnitTest_TestCasesList.tcl`

      This execution is passed if:
      - The following message is displayed

        "*Tests Results are stored in <pathname_of_the_results_file>*"
      - The status is “OK” for all test cases in this results log file

    - `tclsh CVK_ExecutableRun.tcl –f ../CONFIGURATION/CVK_UnitTest_TestCasesList.tcl`

      This execution is passed if:
      - The following message is displayed

        "*Tests Results are stored in <pathname_of_the_results_file>*"
      - The status is “OK” for all nominal test cases and “NOK” for all robustness ones

Prerequisites Before Using CVK

Verify that the complexity of code generated by KCG from its SCADE application is in the limits scope covered by CVK

- Check metrics values as defined in ANSI-ISO/C after code generation by analyzing the generated C source code
- Check that each maximum value associated to a metric is less than or equal to the limit value tested in CVK
  - Metrics are split in several categories:
    - Data Structures
    - Control Structures
    - Program Size

Prerequisites Before Using CVK

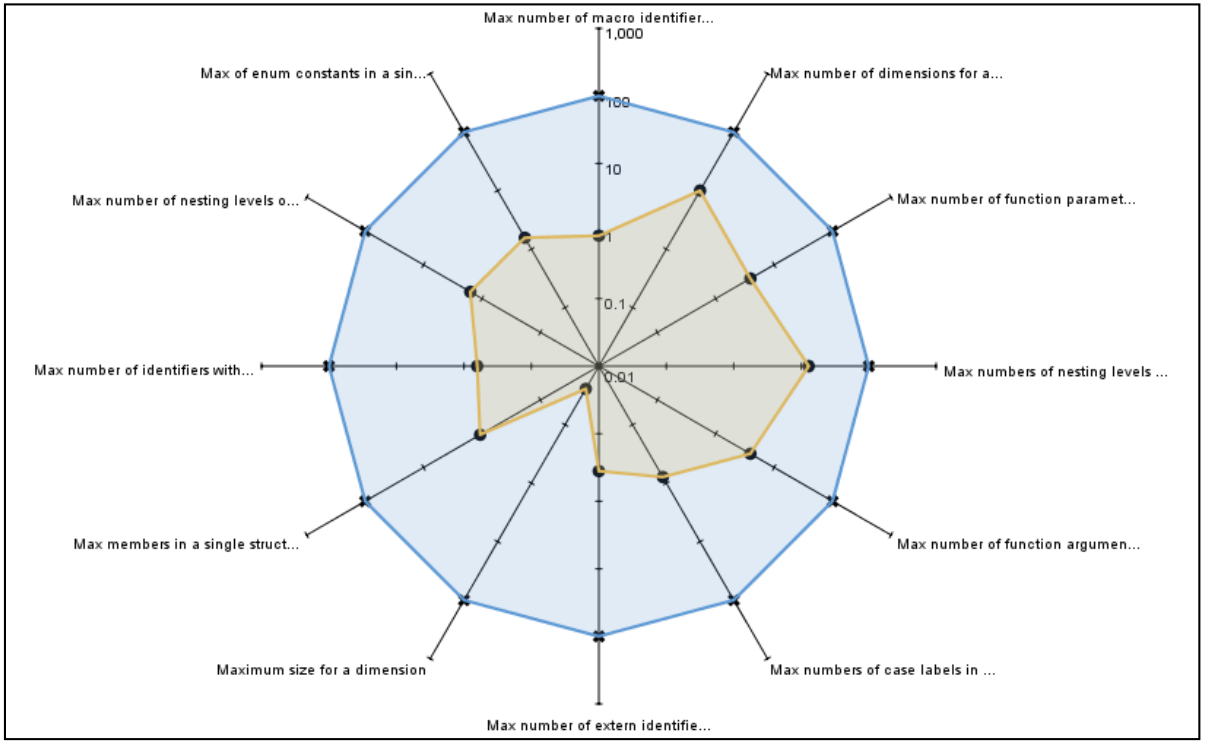
- KCG provides most values for these metrics in a file named `kcg_metrics.txt`
  - This file is systematically generated when KCG is launched
  - Each value corresponds to the maximum value computed on the generated C code
- Some others metrics must be computed by scripts (kcgmetrics2.tcl and user ones) or manually

Note: kcg_metrics.txt is available for the CVK_Limit model

Prerequisites Before Using CVK

The SCADE LifeCycle Dashboard

- A mean to allow Safety and Quality Managers to check that the code generated by KCG meets the safe usage domain of cross compiler
- Merges metrics generated by KCG on your application and data coming from CVK



USING CVK

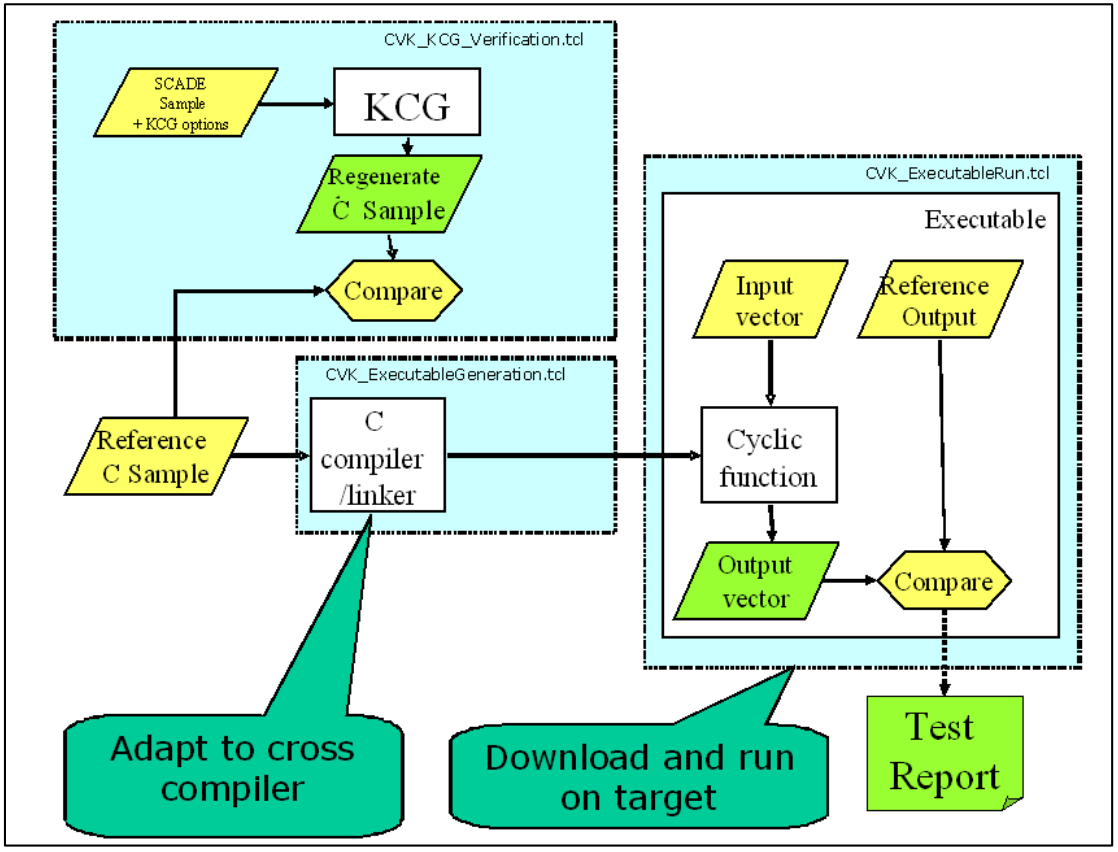
PREREQUISITES BEFORE USING CVK

ADAPTING CVK TO CROSS PLATFORM AND TARGET

GENERATE AND RUN EXECUTABLE TEST CASES

Adapting CVK to Cross and Target Platform

CVK Flow Overview



- **Once all tests have been passed successfully on host platform, these tests can be adapted and run on cross and target environments**

Adapting CVK to Cross and Target Platform

Adapting Test Harness To The cross Compiler And Target

- Update if need be, PREAMBLE and POSTAMBLE, user macros defined in `TestCode/macro_user.h` file
  - These macros are called in `CVK_MAIN` function of `<RootNodeName>_test.c`
    - Redefine PREAMBLE to perform target-specific initialization functions
    - Redefine POSTAMBLE to perform target-specific termination functions such as communicate the pass/fail status in another way than with the return code

Adapting CVK to Cross and Target Platform

Example

CVK_Basic\Logical\Booleans\Booleans_Standard

Default declarations:

```
#define PREAMBLE()
#define POSTAMBLE(nCR) exit(nCR)
```

```
int CVK_MAIN(void)
{
 int result = 0;
 int nCR = 1;
 int nNbCycl;
 PREAMBLE();
 INIT_Booleans();
 /* Main loop */
 for (nNbCycl=0; nNbCycl<NB_CYCL; nNbCycl++) {
  VAR_Booleans_BoolInput1 = Inputs[nNbCycl].BoolInput1;
  VAR_Booleans_BoolInput2 = Inputs[nNbCycl].BoolInput2;
  VAR_Booleans_BoolInput3 = Inputs[nNbCycl].BoolInput3;
  PERFORM_Booleans();   )
  if(CheckResults(nNbCycl)) {
              result = result + 1;
            }
 }/* end Main loop */

 if(result == NB_CYCL) {
    nCR = 97;
  }
 POSTAMBLE(nCR);
 }
```

Adapting CVK to Cross and Target Platform

Adapting Types Implementation

- Redefine if need be, type definitions for
  - Boolean and character in macros located in `TestCode/cvk_def_common.h`

```
#define kcg_char kcg_char
typedef char kcg_char;
```

Default declarations:

```
#define kcg_bool kcg_bool
typedef long kcg_bool;
```

  - integer and float defined in macros located in

    `TestCode/cvk_def_<|W16|W32|UW32|SP|DP>.h`

    Default declarations:

```
#define WORD16
#define kcg_int kcg_int
typedef short int kcg_int;
```

```
#define WORD32
#define kcg_int kcg_int
typedef int kcg_int;
```

```
#define UWORD32
#define kcg_int kcg_int
typedef unsigned long kcg_int;
```

```
#define SIMPLE_PRECISION
#define kcg_real kcg_real
typedef float kcg_real;
```

```
#define DOUBLE_PRECISION
#define kcg_real kcg_real
typedef double kcg_real
```

Adapting CVK to Cross and Target Platform

Adapting Compiling and Linking Procedure

- Update if need be, `CONFIGURATION/CVK_BuildCustom.tcl`
  - `GenBuild` procedure to create
    - in one way a batch file (such as .bat or .sh file) to execute compilation routines
    - or another way a Makefile and a batch file that execute make command
    - In the PARAMETERS section, update compiler / linker tool names, object files target directory, object files extension and compiler / linker options

```
# Compiler name
  puts $batFileRef "@set vCompiler=gcc.exe"
# Linker name
  puts $batFileRef "@set vLinker=gcc.exe"
# Object files directory
  puts $batFileRef "@set vObjDir=win32"
# Object files extension
  puts $batFileRef "@set vObjExt=.obj"
# Compiler options
  puts $batFileRef "@set vCFlags=-O2 -pedantic"
# Linker options
  puts $batFileRef "@set vLDFlags=-o \"${rootNodeName}${vExeExt}\"“
```

Default values are set for gcc tool and Windows OS to produce a .bat file

  - `GenRun` procedure, to generate the executable run as a batch file:

```
puts $batFileRef "\"${exePath}\""
```

Adapting CVK to Cross and Target Platform

- Find in the CONFIGURATION folder several customizations of CVK_BuildCustom.tcl named CVK_BuildCustom_<type>.tcl where
  - <type> = gcc / gcc_unix / VC6 / VC2012, respectively for the use of Windows gcc , Unix gcc, Visual C++ 6.0 (VC) and VC 2012.

Downloading and Running Test Harnesses

- Specific to the target environment (user customization)
- Two cases:
  - The execution environment does not support automation
    - Must download and run manually each test program
    - Collect manually the status (for instance with a debugger)
  - The execution environment supports automation
    - Insert directives in `CONFIGURATION/CVK_BuildCustom.tcl` that control downloading the program and retrieve results according to the current target platform

USING CVK

PREREQUISITES BEFORE USING CVK

ADAPTING CVK TO CROSS PLATFORM AND TARGET

GENERATE AND RUN EXECUTABLE TEST CASES

Generate and Run Executable Test Cases

Execute on a batch window in the *SCRIPTS* folder:

- For generation of all executable test cases on the cross environment:
  `tclsh CVK_ExecutableGeneration.tcl`
- For running all executable test cases on the target platform:
  `tclsh CVK_ExecutableRun.tcl`
- For generation and running all executable test cases:
  `tclsh CVK_ExecutableGenRun.tcl`
- Each previous execution is passed if:
  - The following message is displayed
    "*Tests Results are stored in <pathname_of_the_results_file>*"
  - The status is "OK" for all test cases in this results log file (located in RESULTS folder)
- **Test Results**
  - Passed/failed results are summarized in the *RESULTS/LogFiles* folder
  - Details of build/run can be found in the *.log files of *CURRENT/<TestCaseName>* folder
- If a test is failed then users must verify the settings of the compiler that must be C standards (ANSI, ISO) compliant

Generate and Run Executable Test Cases

Optional arguments:

- Use "-v", to trace details to the output batch window (verbose mode)
- Use <TestCasesList>, to specify that the procedure has to be applied only to a list of test cases (white separated test case names)
  - Example: to generate `ArithmeticInt_Global` and `ArithmeticInt_Standard` test harnesses

```
tclsh CVK_ExecutableGeneration.tcl
      ArithmeticInt_Global ArithmeticInt_Standard
```

- Use «`-f <FileName>.tcl`», to run the procedure on a test cases subset that are specified in the `<FileName>` Tcl file
  - Example: to generate the test harness for the arithmetic test cases only

```
tclsh CVK_ExecutableGeneration.tcl –f
../CONFIGURATION/CVK_ArithmeticTestCasesList.tcl
```

Where the content of `CVK_ArithmeticTestCasesList.tcl` is created as follows in the *CONFIGURATION* folder:

```
CVK_Basic Arithmetic ArithmeticInt Standard {}
CVK_Basic Arithmetic ArithmeticInt Debug {}
CVK_Basic Arithmetic ArithmeticInt Global {}
CVK_Basic Arithmetic ArithmeticReal Standard {}
```

Exercise 1

Exercise

Step 5: verify CVK comparison functions

- Run executable generation for unit test cases
  - Check that the following message is displayed

    Tests Results are stored in
    <CVK_DIR>\CVK\RESULTS\LogFiles\ExecutableGeneration_<date>_<time>.txt
  - Check in this results log file that the status is “OK” for all test cases

- Launch executable run for unit test cases
  - Check that the following message is displayed

    Tests Results are stored in
    <CVK_DIR>\CVK\RESULTS\LogFiles\ExecutableRun_SP_W16_<date>_<time>.txt
  - Check in this results log file that the status is “OK” for all test cases

Exercise 1

Exercise

Step 6: build and run test cases on host

- Run executable generation for test cases
  - Check that the following message is displayed:

    Tests Results are stored in
    <CVK_DIR>\CVK\RESULTS\LogFiles\ExecutableGeneration_<date>_<time>.txt
  - Check in this results log file that the status is “OK” for all test cases

- Launch executable run for test cases
  - Check that the following message is displayed :

    Tests Results are stored in
    <CVK_DIR>\CVK\RESULTS\LogFiles\ExecutableRun_SP_W16_<date>_<time>.txt
  - Check in this results log file that the status is “OK” for all test cases

ANSYS

ESTEREL Technologies

Exercise

SOLUTION

Solution: Step 5

Exercise

Executable generation, in the result log file:

```
#########################################################################
#
>> CVK Test Case generation summary <<
=====================================
 * Total number                : 6
-------------------------------------
 *  Successful generation     : 6
 *  Failed generation         : 0
=====================================
```

Executable run, in the result log file:

```
#########################################################################
#
>> CVK Test Case execution summary <<
=====================================
 * Total number                : 6
-------------------------------------
 *  Successful execution      : 3
 *  Failed execution          : 3
=====================================
#########################################################################
#
```

Solution: Step 6

Executable generation, in the result log file:

```
#########################################################################
#
>> CVK Test Case generation summary <<
====================================
 * Total number               : 117
------------------------------------
 *  Successful generation     : 117
 *  Failed generation         : 0
====================================
```

Executable run, in the result log file:

```
#########################################################################
#
>> CVK Test Case execution summary <<
====================================
 * Total number               : 117
------------------------------------
 *  Successful execution      : 117
 *  Failed execution          : 0
====================================
#########################################################################
#
```

Exercise

ANNEX

Verification using CVK

DO-178B Table A-6, "*Testing Of Outputs of Integration Process*"

	Objective	Verification Method	Applicability by SW Level
1	Executable Object Code complies with high level requirements	Test cases covering normal HLR conditions both on host and target (**SCADE LifeCycle TE**)	A, B, C, D
2	Executable Object Code is robust with high level requirements	Test cases covering abnormal HLR conditions (robustness cases) both on host and target (**TE**)	A, B, C, D
3	Executable Object Code complies with low level requirements	SCADE Suite KCG Qualification **CVK sample testing on the target**	A*, B*, C
4	Executable Object Code is robust with low level requirements	Test arithmetic exception handler and/or usage of robust arithmetic blocks (robustness analysis and testing): **not part of CVK**	A*, B, C
5	Executable Object Code is compatible with target computer	HW / SW integration testing: Qualitative compatibility is ensured by nature of generated code (portable code) Quantitative compatibility is evaluated by resource	A, B, C, D

* The objective should be satisfied with independence

Verification using CVK

DO-178B Table A-7, "*Verification Of Verification Process Results*"

	Objective	Verification Method	Applicability by SW Level
1	Test procedures are correct	Test procedure review, concerns both host and target	A*, B, C
2	Test results are correct and discrepancies explained	Test results review, concerns both host and target	A*, B, C
3	Test coverage of high-level requirements is achieved	Coverage of HLR by test cases	A*, B, C, D
4	Test coverage of low-level requirements is achieved	Model Test Coverage (**TE-MTC, QMTC**)	A*, B, C
5	Test coverage of software structure (modified condition/decision) is achieved	KCG qualification MC/DC on **CVK "C sample"** MC/DC coverage on SCADE model	A*
6	Test coverage of software structure (decision coverage) is achieved	DC on **CVK "C sample"**	A*, B*
7	Test coverage of software structure (statement coverage) is achieved	Statement coverage on **CVK "C sample"**	A*, B*, C

* The objective should be satisfied with independence

Verification using CVK

DO-178C:

- **Objective MB.A-4.3: Low level requirements are compatible with target computer**
- CVK allows analysis of compatibility of the cross compiler and target with respect to:
  - Complexity of expressions
  - Complexity of control structures
  - Rounding to zero

- **Objective MB.A-4.10: Architecture is compatible with target computer**
- CVK allows analysis of compatibility of the cross compiler and target with respect to:
  - Complexity of data structures nesting
  - Number of arguments in a function call

CVK Package Content: CVK_Basic

SCADE Package Name	Test Cases	Objective	Implementation
Arithmetic	ArithmeticInt	Cover basic integer operators (+, -, *, div, mod, unary minus). Also Cover use of public and private operators	W16 / W32 / UW32
Arithmetic	ArithmeticReal	Cover basic real operators (+, -, *, /, unary minus) and real to int conversion	SP / DP
Arithmetic	ArithmeticIntCast	Conversion from int to real	W16 / W32 / UW32; SP / DP
Arrays	ArrayAccess	Static and dynamic array projection	W16 / W32 / UW32
Arrays	ArrayAccessWithSize	Array access with size parameter	W16 / W32 / UW32
Arrays	ArrayConstructors	Array constructor operators	W16 / W32 / UW32
Arrays	Concatenation	Array concatenation	W16 / W32 / UW32
Arrays	Iterators	Cover iterators operators (fold|map ,fold|map<i,w,wi>, mapfold)	W16 / W32 / UW32
Arrays	Reverse	Array reverse operator	W16 / W32 / UW32
Arrays	Slice	Cover array slice operator	W16 / W32 / UW32
Arrays	SumIntWithSizes	Use function with two size parameters	W16 / W32 / UW32
Arrays	Transpose	Cover array transpose operator	W16 / W32 / UW32
ClockedBlocks	Booleans	If block construct	W16 / W32 / UW32
ClockedBlocks	Enumerates	When block construct	N/A
Comparison	ComparisonChar	Cover comparison operator on char type (<>, =)	N/A
Comparison	ComparisonEnum	Cover comparison operator on enum type (<>, =)	N/A
Comparison	ComparisonInt	Cover comparison operator on int type (<, <=, >, >=, <>, =)	W16 / W32 / UW32
Comparison	ComparisonReal	Cover comparison operator on real type (<, <=, >, >=,<>, =)	SP / DP
ConditionalRestart	Activate	Conditional node activation (with default value, with initial default value, on clock, on not clock, on match value clock)	W16 / W32 / UW32
ConditionalRestart	Restart	Conditional node restart construct	W16 / W32 / UW32
Contract	Assume	Assume contract	W16 / W32 / UW32
Contract	AssumeGuarantee	Both assume and guarantee contract	W16 / W32 / UW32
Contract	Guarantee	Guarantee contract	W16 / W32 / UW32

CVK Package Content: CVK_Basic

SCADE Package Name	Test Cases	Objective	Implementation
GroupType	ConditionalSumProdInt	Cover group type	W16 / W32 / UW32
ImportExport	InstanceImportedOperator	Imported node and function	W16 / W32 / UW32
ImportExport	ComparisonImportedType	Imported type	W16 / W32 / UW32
Logical	Booleans	Cover boolean operators (and, or, xor, not, =, <>, #)	N/A
Polymorphism	NumPolymorphismInstInt	Use polymorphic numeric operator with int type	W16 / W32 / UW32
Polymorphism	NumPolymorphismInstIntReal	Use function with two parametric types	W16 / W32 / UW32; SP / DP
Polymorphism	NumPolymorphismInstReal	Use polymorphic numeric operator with real type	SP / DP
Polymorphism	PolymorphismInstance	Use polymorphic operator with bool type	N/A
Selection	CaseOf	Case…of operator when condition is int input, or bool input, or enum input or char input	W16 / W32 / UW32
Selection	IfThenElse	If…then…else operator with flows of bool, char and int values	W16 / W32 / UW32
Selection	IfThenElseReal	If…then…else operator with flows of real and int values	W16 / W32 / UW32; SP / DP
StateMachine	Automaton	Automaton with normal, fork, history, strong and weak transitions	N/A
StateMachine	Parallelism	Parallel composition of state machine	N/A
StateMachine	Synchro	Automaton with synchronization transition	N/A
Structures	Projection	Use of structure projection	W16 / W32 / UW32
Structures	StructConstructor	Use structure constructor	W16 / W32 / UW32
TextualOperators	InstanceFunction	Textual operator	W16 / W32 / UW32
TextualOperators	SpecializeSumInt	Specialize imported polymorphic function with int type	W16 / W32 / UW32
TextualOperators	SpecializeSumReal	Specialize imported polymorphic function with real type	SP / DP
TextualStateMachine	EvenTimes	Textual State Machine	W16 / W32 / UW32
TextualStateMachine	EvenTimesNotify	Variant of Textual State Machine	W16 / W32 / UW32

CVK Package Content: CVK_Basic

SCADE Package Name	Test Cases	Objective	Implementation
Time	Fby	Fby operator for single flow, implicit group flows and array	W16 / W32 / UW32
Time	Init	Init operator for single flow, implicit group flows and structure	W16 / W32 / UW32
Time	Merge	Merge with boolean condition and enumerate condition	W16 / W32 / UW32
Time	Pre	Pre operator for single flow, implicit group flows and structure	W16 / W32 / UW32
Time	Times	Times operator	W16 / W32 / UW32

CVK Package Content: CVK_Combination

SCADE Package Name	Test Cases	Objective	Implementation
Arithmetic	ArithmeticAdvanced	Combine real and integer I/O with integer and real operators	W16 / W32 / UW32; SP / DP
Arithmetic	ArithmeticIntAdvanced	Combine integer operators (+, -, *, div, mod, unary minus)	W16 / W32 / UW32
Arithmetic	ArithmeticRealAdvanced	Combine real operators (+,-, *, /, unary minus)	W16 / W32 / UW32 SP / DP
Arithmetic	RobustIntDiv	Implementation of robust div where result is numerator if divide by zero	W16 / W32 / UW32
Arithmetic	RobustIntProd	Implementation of robust product so result is in bound	W16 / W32 / UW32
Logical	MediumBooleans	Combine boolean operators (and, or, xor, not, #)	N/A
StateMachine	ABC	ABC automaton implementation	N/A
StateMachine	ABRO	ABRO automaton implementation	N/A

CVK Package Content: CVK_Limit

SCADE Package Name	Test Cases	Objective	Implementation
Arithmetic	combine_int_A_10	Combine int operators with level of parenthesizes below 10	W16 / W32 / UW32
Arithmetic	combine_int_N_10	Combine int operators with level of parenthesizes above 10	W16 / W32 / UW32
Arithmetic	combine_float_A_10	Combine float operators with level of parenthesizes below 10	SP / DP
Arithmetic	combine_float_N_10	Combine float operators with level of parenthesizes above 10	SP / DP
Arithmetic	sum_int	Test 63 as limit number of nesting levels of parenthesized expressions within a full expression of int number	W16 / W32 / UW32
Arithmetic	sum_float	Test 63 as limit number of nesting levels of parenthesized expressions within a full expression of float number	SP / DP
Control	large_match_block_255	Test 255 as limit of number of case labels for a switch statement	W16 / W32 / UW32
Control	nested_if_block_31	Test 31 as limit of number of nesting levels of compound statements (blocks), iteration control structures and selection control structures	W16 / W32 / UW32
Data	array_depth10	Test number of dimensions for an array	W16 / W32 / UW32
Data	array_width4095	Test 4095 as limit of maximum size for a dimension	W16 / W32 / UW32
Data	cte_ext_adder_4095	Test 4095 as limit of number of external identifiers in one translation unit	W16 / W32 / UW32
Data	enum_255	Test 255 as limit of number of enumeration constants in a single enumeration	N/A
Data	c_macro_adder_4095	Test 4095 as limit of number of macro identifiers simultaneously defined in one preprocessing translation unit	W16 / W32 / UW32
Data	f_io_63_root	Test 63 as limit of number of parameters in one function definition and number of arguments in one function call	W16 / W32 / UW32
Data	f_io_255_root	Test 255 as limit of number of parameters in one function definition and number of arguments in one function call	W16 / W32 / UW32
Data	F127	Test 127 as number of significant initial characters in an internal identifier or a macro name and number of significant initial characters in an external identifier	W16 / W32 / UW32
Data	locals_511	Test 511 as limit of number of identifiers with block scope declared in one block	W16 / W32 / UW32
Data	locals_2047	Test 2047 as limit of number of identifiers with block scope declared in one block	W16 / W32 / UW32
Data	nf_leaf_62	Test structure or union with 62 levels of definitions in a single struct-declaration-list	W16 / W32 / UW32

CVK Package Content: CVK_Limit

SCADE Package Name	Test Cases	Objective	Implementation
Data	nf_leaf_45	Test structure or union with 45 levels of definitions in a single struct-declaration-list	W16 / W32 / UW32
Data	nf_leaf_30	Test structure or union with 30 levels of definitions in a single struct-declaration-list	W16 / W32 / UW32
Data	nf_leaf_25	Test structure or union with 25 levels of definitions in a single struct-declaration-list	W16 / W32 / UW32
Data	nf_leaf_14	Test structure or union with 14 levels of definitions in a single struct declaration-list	W16 / W32 / UW32
Data	nf_width	Test 1025 as limit of number of members in a single structure or union	W16 / W32 / UW32
Data	nf_width_255	Test 256 as limit of number of members in a single structure or union	W16 / W32 / UW32
Logical	deep_atoms	Test deeply combination of boolean operators	N/A
Data2	c_macro_adder_2000	Test 2000 as limit of number of macro identifiers simultaneously defined in one preprocessing translation unit	W16 / W32 / UW32
Data3	cte_ext_adder_512	Test 512 as limit of number of external identifiers in one translation unit	W16 / W32 / UW32
Data3	array_combine_3_5	Test number of dimensions for an array	W16 / W32 / UW32
Data3	locals_4800	Test 4800 as limit of number of identifiers with block scope declared in one block	W16 / W32 / UW32
Data3	nf_width_1600	Test 1602 as limit of number of members in a single structure or union	W16 / W32 / UW32

Prerequisites Before Using CVK

Metric name	To be verified	Generated by KCG	Reference in KCG metrics
Numbers of levels of nested structure or union definitions in a single structure-declaration-list	Yes	No	
Number of members in a single structure or union	Yes	Yes	Max members in a single structure
Number of dimensions for an array	Yes	Yes	Max number of dimensions
Maximum size for a dimension	Yes	Yes	Max size for a dimension
Constants in a single enumeration	Yes	Yes	Max of enum constants
Number of Case labels for a switch statement	Yes	Yes	Max number of case labels in a switch
Number of nesting levels of compound statements (blocks), iteration control structures and selection control structures	Yes	Yes	Max number of nesting levels in compound statement
Number of nesting levels of parenthesized declarators within a full declarator	No	No	
Number of pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration	Yes	Yes	Max number between 1 and Max
Number of dimensions Number of nesting levels of parenthesized expressions within a full expression	Yes	Yes	Max level of nested parenthesis
Number of significant initial characters in an internal identifier or a macro name	Yes	No	
Number of significant initial characters in an external identifier	Yes	No	
Number of external identifiers in one translation unit	Yes	Yes	Max number of external identifiers
Number of identifiers with block scope declared in one block	Yes	Yes	Max number of identifiers declared in a block
Number of macro identifiers simultaneously defined in one preprocessing translation unit	Yes	Yes	Max number of macro identifiers
Number of parameters in one function definition	Yes	Yes	Max number of function parameters in definition
Number of arguments in one function call	Yes	Yes	Max number of function arguments in a call

Prerequisites Before Using CVK

Metric name	To be verified	Generated by KCG	Reference in KCG metrics
Number of parameters in one macro definition	No	No	
Number of arguments in one macro call	No	No	
Others Number of nesting levels of conditional inclusion	No	No	
Number of nesting levels for #included files	Yes	No	
Number of characters in a logical source line	No	No	From [KCG-TOR], requirement KCG-273, "A line in the generated code shall not exceed 2048 characters, unless composed of a single identifier longer than 2048 characters". This metric depends of "Number of significant initial characters in an internal or external identifier or a macro name".
Number of characters in a string literal	No	No	
Number of bytes in an object (in a host environment only)	Yes	No	

Prerequisites Before Using CVK

Criteria	Maximum values (CVK)	Intermediate levels (CVK)
Number of levels of nested structure or union definitions in a single structure declaration-list	63	15, 26, 31, 46
Number of members in a single structure or union	1602	256, 1025
Number of dimensions for an array	10	3
Maximum size for a dimension	4095	5
Number of enumeration constants in a single enumeration	255	
Numbers of case labels in a switch	255	
Numbers of nesting levels in compound statement(blocks), iteration control structures and selection control structures	31	
Number of nesting levels of parenthesized declarators within a full declarator	1	
Number of pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	10	3
Number of nesting levels of parenthesized expressions in a full expression	63	
Number of significant initial characters in an internal identifier or a macro name	127	
Number of significant initial characters in an external identifier	127	
Number of external identifiers in one translation unit	4281	512, 4095
Number of identifiers with block scope declared in one block	4800	511, 2047
Number of macro identifiers simultaneously defined in one preprocessing translation unit	4157	2013, 4095
Number of parameters in one function definition	255	63
Number of arguments in one function call	255	63
Number of parameters in one macro definition	2	
Number of arguments in one macro call	3	
Number of nesting levels of conditional inclusion	2	
Number of nesting levels for #included files	63	15, 26, 31, 46
Number of characters in a logical source line	831	
Number of characters in a string literal	N/R	
Number of bytes in an object (in a hosted environment only)	N/R	