TML User Documentation

Institut SC

Exported on 09/03/2021

Table of Contents

1 Create a TML Project

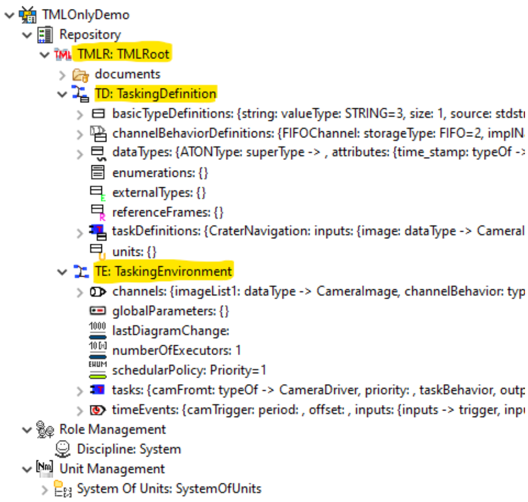
- If not existing create a virsat project.
- Open the Repository and activate the concepts "tml.structural", "tml.behavioral" and "tml.configuration".

2 Modelling Structure

It is possible to use TML in a project with only software aspects (TML-Only) or integrated into a system model.

2.1 TML-Only Structure

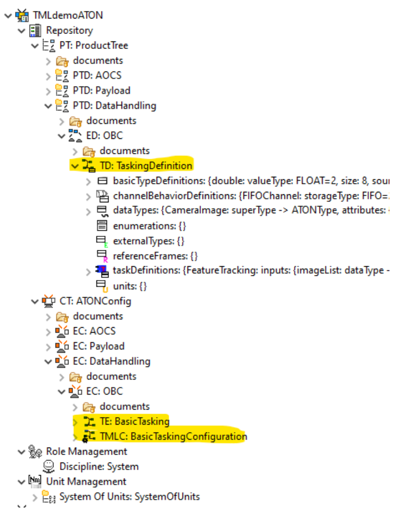
If TML is intended to be used without other Virtual Satellite concepts, a TMLRoot element can be created underneeth the repository.



This TMLRoot element then contains the TaskingDefiniton element with all its type definitions. The TaskingEnvrionemnt specifies the actuall instances of these types / tasks and their data connection in the software.

2.2 Product Structure based System Model

If TML is integrated into a system model, it should comply to the patters of the Virtual Satellite Product Structure; more information can be found here[1].

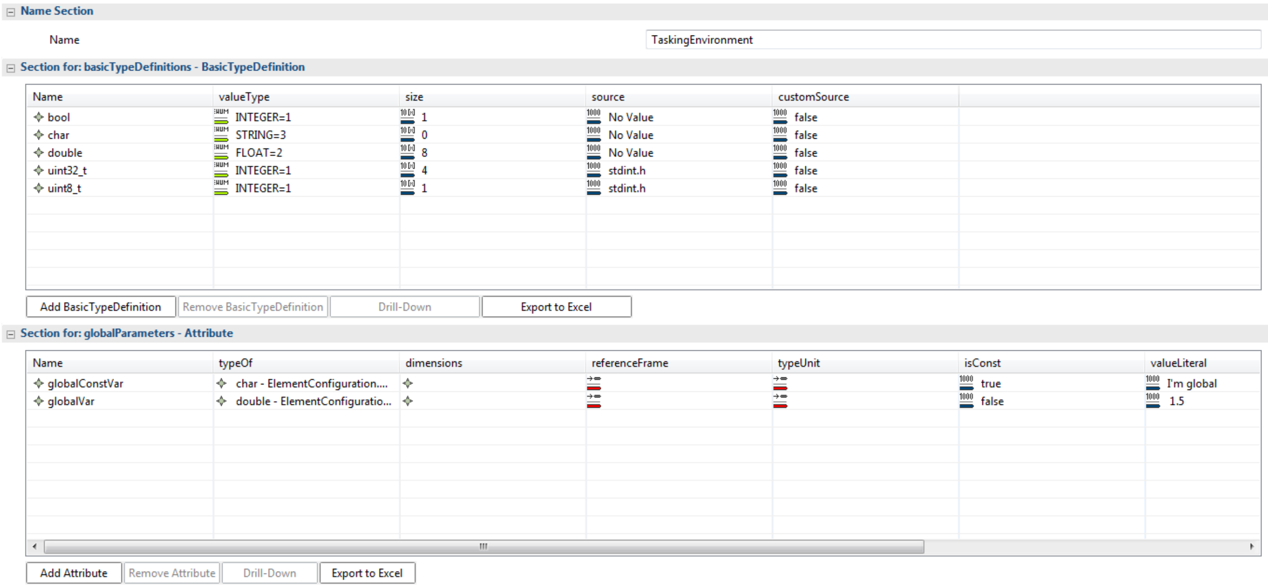


1 https://github.com/virtualsatellite/VirtualSatellite4-Core/releases/download/Release_4.12.1/VirSat_Core_User_Manual.pdf

3 Editors

3.1 Virsat Editor

Create a tasking defintion element. Double-click the new model element and specify the tasking's name, its basic types and global parameters (if any) by using the virsat edior:



In C++11, no sources for primitive c-types is required. Ideally, basic-types shall be defined in following manner:

- boolean - bool
- char arrays and string - char
- unsigned char - uchar
- double - double
- float - float
- integer - int
- unsigned int - uint
- 8-bit integer - int8_t
- 8-bit signed integer - uint8_t
- 16-bit integer - int16_t
- 16-bit signed integer - uint16_t
- 64-bit integer - int64_t
- 64-bit signed integer - uint64_t
- long - long
- unsigned long - ulong

3.2 Datatype Editor

Create datatypes by using the textual editor. Right click the tasking definition model element and choose "Add DataType" or selecting an existing one and using "Edit DataType".

Create a datatype the keyword "DataType" and specify a name. Add attributes within the brackets. General syntax of attribute definition is <attribute name> : <attribute type>; Details can be specified by adding brackets to attributes.

Attribute types can be other data types, enumerations or basic types (specified with the virsat editor)

See an example data type:

```
NavigationState
DataType NavigationState extends AbstractType {
    intProp :  uint8_t;
    position : double[3, name=x];
    velocity : double[3, name=x];
    matrix : double[2, name=x][2, name=x];
}
```

**Note: Name of the dimension of high-dimensional attributes is irrelevant in-term of the generated source code.

3.3 Enumeration Editor

Create enumerations by using a textual editor. Right click the tasking definition model element and choose "Add Enumeration" or selecting an existing one and using "Edit Enumeration".

Enumerations are created with the "Enum" keyword. Add literals into brackets. Integer values of the literals are optional.

See an example enumeration:

```
ComponentState
Enum ComponentState {
    UNDEFINED = 0, OK, ERROR
}
```

**Note: Enum literals can have an integer keys. If no key is defined, then default key starting from 0.. is used. If only first key is defined the rest of the keys are created incrementally starting from the first key value.

3.4 Task Definition Editor

Create task definitions by using a textual editor. Right click the tasking defintition model element and choose "Add TaskDefinition" or selecting an existing one and using "Edit TaskDefinition".

TaskDefinitions are created with the "Task" keyword. Specifiy inputs, outputs and parameters by using the common TML attribute syntax <attribute name> : <attirbute type>.

See an exmaple task defintion

```
NavigationFilter
Task NavigationFilter {
    inputs {
        estimatedCrater10Pos: Position;
        estimatedCrater45Pos: Position;
    }
    outputs {
        currentPosition: NavigationState;
    }
    parameters {
        componentState : ComponentState;
        startVelocity : double[3, name=x];
    }
}
```

**Note: It is also possible to define tasks without inputs, outputs and parameters.

3.5 Channel Behavior Editor

Create channel behavior definitions by using a textual editor. Right click the tasking definition model element and choose "Add ChannelBehaviorDefinition" or selecting an existing one and using "Edit ChannelBehaviorDefinition".

Create channel definitions by using the "Channel" keyword. Specify a storage type; options are DOUBLE_BUFFER, FIFO, LIFO, EVENT_ONLY, CUSTOM. Optionally you can add parameters.

See an example of a channel definition:

FifoChannel

```
Channel FifoChannel : FIFO {
    size : INTEGER;
    chParam : FLOAT;
}
```

Additionally, define a custom channel definition. Following example shows the custom channel definition which implements fifo from the tasking framework.

TaskingFifoChannel

```
Channel TaskingFifoChannel : CUSTOM implementation: "fifo.h" {
    size : INTEGER;
}
```

*

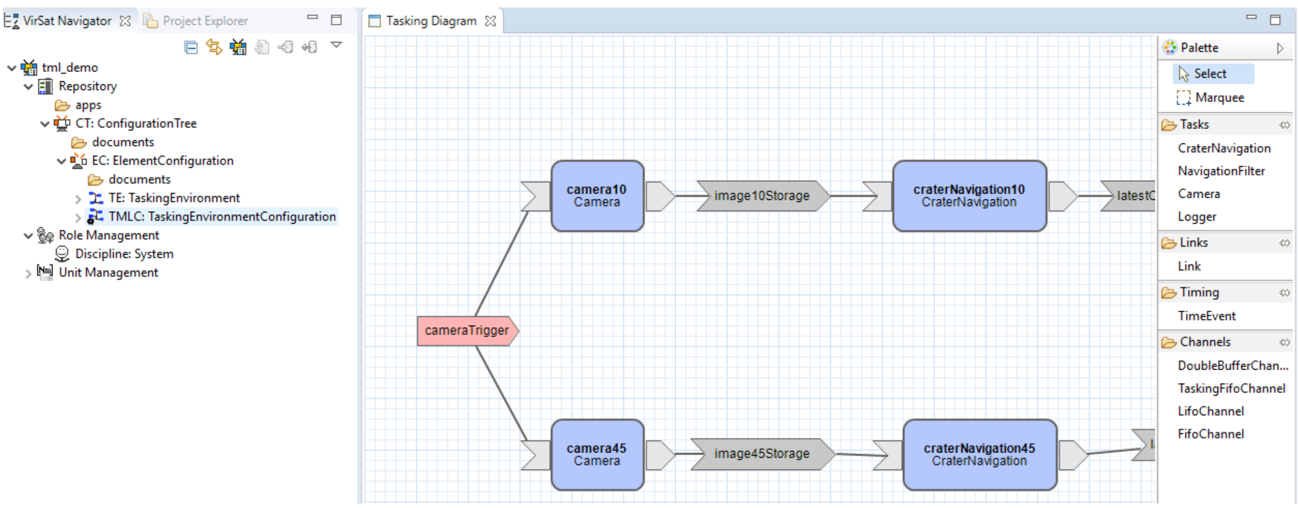
**Note: Channels which have size, must have a integer type parameter defined with name "size", the name is case-insensitive.

3.6 Task Diagram Editor

Open the task diagram editor by right-clicking the environment and choos "Open Task Diagram Editor". The diagram can be used to create instances of tasks and channels. Connectors can be used to connect tasks and channels. Take care that datatypes of inputs / outputs match the type of the channel.

General properties of diagram elemtents can be seen with the property view below the diagram. To specify values of channel parameters specified in the Channel Behavior Edior, switch to the tab "Behavior Parameters".

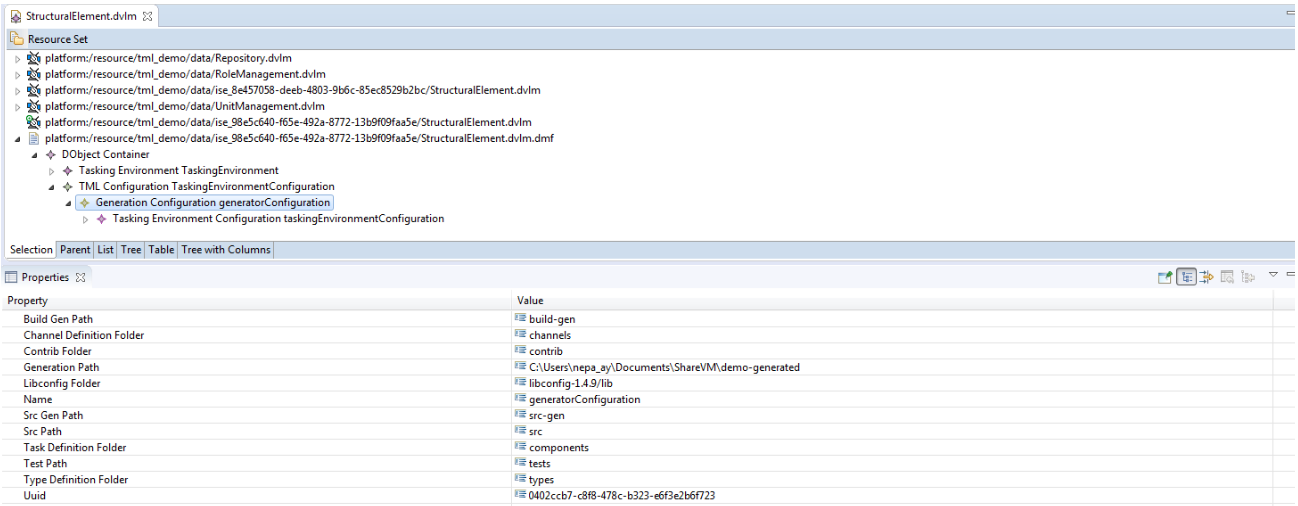
See a screenshot of a example diagram:



3.7 Tree Editor

The tree editor can be used to editor model properties like the configuration attributes. Right-click the environment and choose "Open Tree Edior" to open the edior

An screenshot of an example can be seen here:

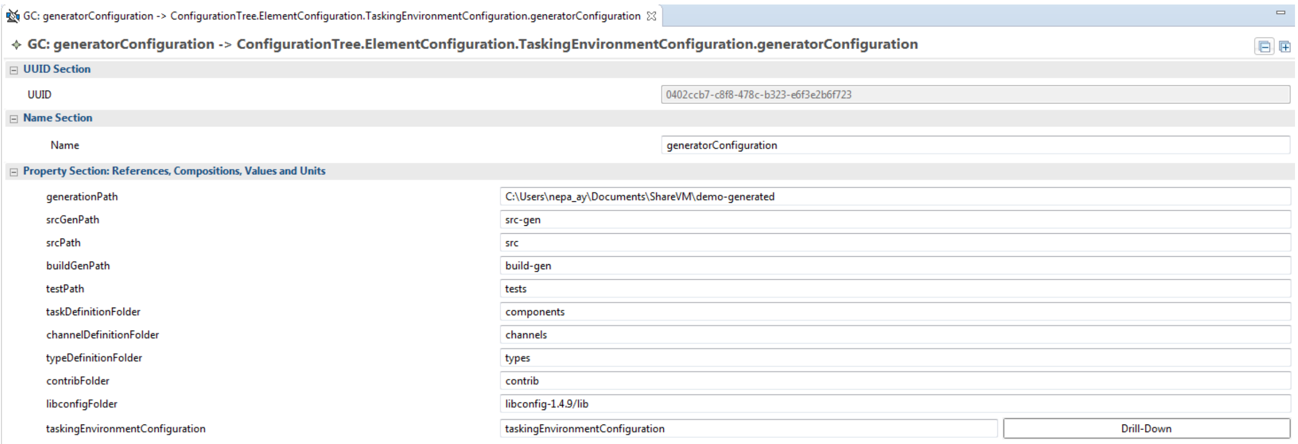


4 Code Generation

4.1 Create generation configuration

Configuration for the code generation can be defined in Tasking Environment Configuration. Right-click the environment and choose "Create Generation Configuration" to open the configuration editor.

An screenshot of an example can be seen here:



When the Tasking Environment Configuration is created for the first time, properties are have their default values.

Under Property Section following properties are given:

- **generationPath :** Root directory for the code generation.
- **srcGenPath :**Folder in Root directory where abstract classes are generated. Content of this folder should not be edited.
- **srcPath :**Folder in Root directory where concrete classes are generated.Content of this folder should be edited.
- **buildGenPath :** Folder in Root directory wherescons build files are generated.
- **testPath :** Folder in srcGenPath directory where unit-test files are generated.
- **taskDefinitionFolder :**Folder in srcPath and srcGenPath, where task and component classes are generated.
- **channelDefinitionFolder :** Folder in srcPath and srcGenPath, where channel classes are generated.
- **typeDefinitionFolder :**Folder in srcPath and srcGenPath, where data-types and enum files are generated.
- **contribFolder :**Folder in Root directory where all external libraries should be located. e.g. Tasking and LibConfig (if used) must be in this folder.
- **libConfigFolder :** (Optional) - Tasks in the current environment can be configured using a LibConfig file. To use the LibConfig library, specify the name of the folder where the LibConfig source files are located, if left empty, the library is not used and also the configuration loader class is not generated. This folder must be placed inside /contribFolder. Source code of the library can be found here[2] under /lib folder. Library files must be placed in *contribFolder*, like:
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / grammar.c
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / scanner.c
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / scanctx.c
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / strbuf.c
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / libconfig.c
  - *generationPath* / *contribFolder* / libconfig-1.4.9 / lib / libconfigcpp.c++

[2] https://github.com/hyperrealm/libconfig

- **taskingEnvironmentConfiguration :** Reference to the domain element (Tasking Environment)

Optionally, these parameters can also be edited from the Tree Editor.

4.2 Generate source code

After creating the Tasking Environment Configuration, right click the configuration model and choose "Generate Source Code" to generate the source code. With default configuration, the source code is generated into *generationPath* (/cpp) folder of the current project. To see the files, open the project explorer and refresh the project.

4.2.1 Generation Gap

Source code for tasks/components, datatypes and channels are generated using the generation-gap pattern.

Every information that comes from the model are given in the abstract implementation of that specific module. These abstract implementations are generated in *srcGenPath* sub-folder in the *generationPath* directory. Previously generated files in *srcGenPath* are replaced everytime the code generation is triggered, therefore no files in this directory should be altered manually.

In the *srcPath* sub-folder inside *generationPath*, the concrete implementation of each modules are generated but only if they are non-existing. These concrete implementation extend their respective abstraction from *srcGenPath* sub-folder. User shall edit these files as required.

Source Code

Source files for all components/tasks, channels and data-types are generated in *srcPath* and *srcGenPath* following the generation-gap pattern. All abstract classes have “A” prefix in their class name.

Source and header files for following modules are generated:

4.2.1.1 Tasking Environment

Source file as well as header for the tasking-environment class is created directly in *srcGenPath.*

In the abstract class of the tasking-environment, all components/tasks, channels as well as task-inputs are declared; also the parameter instances for all components and tasks are created here. Task-inputs as well as task-events are associated with their respective channel instances, and are added in the respective task instances. Additionally for task-events, “*setTiming*” method is called, to initialize the timer.

Furthermore, when the configuration file is used (i.e valid path of the LibConfig library is available in the Generation Configuration), the name of the config-file shall be passed as constructor argument so that the configuration is loaded.

4.2.1.2 Configuration Loader

The configuration loader class provides methods to load the LibConfig configuration file and get parameters of each component. Component’s local parameter as well as tasking-environment global parameters can be loaded. Since the configuration loader class is completely generated and there is no need of manual implementation, it directly provides the concrete implementation in the *srcGenPath*.

4.2.1.3 Channels

In the *channel* folder in *srcGenPath* and *srcPath* channels are generated as templates. There is no separate source file. If the channel definition has a size parameter, channel template is generated with size as template <typename T, unsigned int SIZE>.

Channel Interface (*AChannel.h*) is defined as template and extends the Tasking::TaskChannel. It provides pure virtual methods to allocate, push, and pop data. All channel classes must extend AChannel class and implement these methods.

For every channel definitions, channel files are generated following generation gap pattern. Abstract channel class implements AChannel interface and declares channel parameters as protected member variables. Concrete channel class extending the abstract channel class is generated in *srcPath/channels* folder. Method stub for all methods in AChannel interface are generated for manual implementation.

When the channel-behavior-definition is defined as FIFO, the abstract channel class implementing FIFO is generated in the same folder, which also has the method implementation for push and pop. This class stores data in a Queue class, which is also generated in the same folder. This channel class is used for unit-testing and shall be used as an example for implementing channels.

For other storage types: LIFO and DoubleBuffer no channel implementation is currently available, so user should implement them in the concrete class. The same is valid for the Custom channels.

4.2.1.4 Components/Tasks

Currently it is not possible to define Task and Component separately via language. Task definitions without any input are considered to be normal components so they do not extend Tasking::Task class. Pointers to the parameter object and all output channels are given as constructor argument. Inputs, outputs and parameters defined in the task-definition are implemented as sub-classes of the Component/Task class. Inputs and Outputs sub-classes hold pointers to object datatypes.

In the abstract component/task classes, initialize() method from Tasking::Task is overridden which initializes the parameters and gets pointer to all input channels. Furthermore, following methods:

- virtual void init(Parameters *parameters)
- virtual void execute(void)
- virtual void terminate(void)
- virtual void output(const Inputs &inputs, Outputs &outputs)
- virtual void update(const Inputs &inputs)

are defined as pure virtual and user is expected to implement them in the corresponding concrete class.

4.2.1.5 Datatypes

Datatype classes are also generated following generation gap pattern. Datatypes classes have additional "Datatypes" namespace i.e <*TaskingEnvironmentName*>::DataTypes::<*DatatypeClassName*>. All parameters are declared as public member variables in the abstract implementation. In the abstract datatype class, following methods:

- void serializeLogHeader(std::ofstream& stream)
- void serializeLog(std::ofstream& stream)
- void deserializeLog(std::string line)

implementations are generated, which could be used for serialization and deserialization purposes. Very large arrays are not serializable, so they are not used in these methods. The maximum serializable array size is currently fixed to 100; in the future this shall be configurable.

When a datatype extends an external type, the external type definition in VirSat editor must contains all constructor arguments in the same order as in actual external datatype class. These parameters are printed as constructor argument of the generated datatype classes and are passed to the super class's constructor (the external datatype class). No member variable are created for external datatype classes.

4.2.1.6 Enumeration Types

Enum datatypes are defined in <*TaskingEnvironmentName*>::DataTypes::NS<*EnumName*> namespace. When integer key to all enum entries are provided in the definition then they are used, otherwise they are generated incrementally starting from the integer key of the first entry. If no integer key in defined for the first entry, its key is assumed to be 0.

For enumerations, inline methods for input and output stream operators (>> and <<) are generated. Furthermore, methods to convert the enum object to the corresponding integer key and to the string literal is provided.

4.2.1.7 Configuration Files

When the LibConfig library is used to configure the tasking-environment, two configuration files are generated: one for testing purposes in *srcGenPath/tests* folder and another one in *srcPath*. The one in *srcPath* has same name as the tasking-environment with ".cfg" extension, like <*TaskingEnvironmentName*>.cfg. Initially, all values are in their initial states, user must edit this file and set values as required.

4.2.1.8 Test Files

Currently following test-cases are available, namely:

- **Tasking Environment Test** – creates an instance of the tasking environment and initializes its parameters
- **Configuration Loader Test** – Tests the loading mechanism of the configuration loader for all tasks/ component classes from the LibConfig file for test <TestConfigFile>.cfg. This test file is generated only when LibConfig library is used in the project.
- **Channel Test** – For each channel-behavior-definitions, a channel test-case for all available datatypes are generated and tested using the TestTask. The TestTask is a Tasking::Task implementation which just sets its member variable "m_executed" to true when being executed, and sets it back to false when the initialize() methods is called. TestTask class is also generated in the *srcGenPath*/*tests* folder. Test file is also generated for the custom channels, which will obviously fail if the test cases are executed before the user has implemented its methods.

4.2.1.9 Build Scripts

Cmake and Scons build scripts are generated which build the tasking environment as a static library. Build-scripts in all folders except the one in *srcGenPath* and its sub-folders can be edited manually. Build scripts are generated in following folders:

- *rootPath* – CMakeList.txt, SConstruct and SConscript for the project. SConscript in this folder builds the tasking environment library. When using CMake, library is built in *srcPath.*

- *contribPath* – CMakeList.txt and SConscript to build Tasking framework and LibConfig libraries (if used). When further libraries are intended to be used they must be placed in *contribPath* folder and these build scripts shall be edited accordingly.
- *srcPath* – Only CMakeList.txt is generated in this folder which builds a static library of all source code in this directory.
- *srcGenPath* – Only CMakeList.txt is generated in this folder which builds a static library of all source code in this directory, except the one in tests folder, and finally link it to the project library.
- *srcGenPath/tests* – CMakeList.txt and SConscript is generated in this folder which builds an executable for the unit test.The <TestConfigFile>.cfg file is copied to the location of the test executable automatically while building.