An Integrated Model-Driven Approach for Mechatronic Systems Testing

Roberto S. Silva Filho, Christof J. Budnik
Siemens Corporation, Corporate Research and Technology
Software Development Technologies
755 College Road East
Princeton, NJ, USA 08540
{Roberto.Silva-Filho, Christof.Budnik}@siemens.com

***Abstract*— Mechatronic systems integrate mechanical, electrical and software subsystems. They are increasingly important in mission-critical infrastructures in different domains including automotive, healthcare, energy and transportation sectors. As such, the adequate testing and validation of such infrastructures are of prime importance. Mechatronic systems testing has been supported by different isolated industrial and research approaches including automated test infrastructures, model-based testing, and test execution engines. While these approaches have individual benefits, they are usually applied in isolation, suffering from different weaknesses. This paper reports on an integrated model-driven approach for mechatronic systems testing, where existing approaches are combined and enhanced. The result is unprecedented levels of automation and testability. We discuss the design and implementation of our approach, illustrating its use with a hybrid calculator example.**

Keywords: Mechatronic systems testing, Model-based testing, Automated Testing Infrastructures.

I. INTRODUCTION

Mechatronics is an engineering approach that manages the synergistic integration of mechanics, electronics, control theory, and computer science within product design and manufacturing, in order to develop, improve and/or optimize its functionality [1]. Mechatronics is central to corporations as SIEMENS and its many business units where an increasing number of healthcare, industrial and energy products require the seamless integration of overly complex and heterogeneous subsystems.

Traditional engineering approaches, adopted in the construction of these systems in isolation, are usually inadequate or do not scale to the complexity involved in the development of these systems of systems.

In particular, software has become an important part of today's mechatronic systems. Taking the automotive industry as an example, some experts predict that the relative cost of software and electronics in a new cars will be around 50 percent for conventional vehicles and 80 percent for hybrids within 10 years [2]. This shift from mechanics to electronic and software defines different challenges in the verification, validation and certification of such systems.

As a response to this need, Siemens Corporate Technology is to research and develop novel techniques and tools for the development of these systems of systems. In this context, model-based system engineering is a key element. Models provide an integrated way to tame the complexity of large systems by enabling the precise and multi-faceted specification of the different concerns involved in their analysis and design. Moreover, when combined with automated tools, models provide the basis for streamlining system development tasks including: analysis, design, implementation, and verification & validation.

Verification and validation are important activities in the design and development of mechatronic systems. Depending on the size of the system and the selected testing technique, the number of test cases can be extremely large. Model based testing (or MBT) is a methodology that supports the automation of quality assurance tasks [3]. It automates the quality assurance process of complex systems by the automatic generation of test cases from system specification models. This approach not only saves developers time, by relieving them from the burden on manually generating tests, but also improves test coverage, supports different optimization strategies, and achieves unprecedented control over different test parameters.

Currently, the testing of mechatronic systems has been supported by a loosely-integrated combination of commercial and industry-specific including: hardware abstraction frameworks [4], [5], [6], [7] test execution engines (e.g. NI TestStand, MathWorks Simulink, and NUnit), and a new set of model-based testing tools [3], [8], [9]. While each individual approach comes with its own benefits, their integration is still not leveraged in practice.

This paper discusses the design and implementation of an integrated model-driven approach for mechatronic systems testing. It discusses the main elements of the approach, including the model-based language, its transformations, and shows how these have been integrated into existing test execution engines and infrastructures in the automation of mechatronic systems within Siemens.

II. BACKGROUND

A. Mechatronic Systems Testing Requirements

The heterogeneous and complex nature of mechatronic systems pose different challenges for testing [10]. In particular, automated mechatronic testing infrastructures must be able to support the following requirements.

Heterogeneity. Mechatronic systems have not only software but also hardware (including electro-mechanic) components. In order to support automated testing, these components must be accessible through software interfaces that support their automatic set-up, execution and monitoring.

External Dependencies. It is usually the case that the components and sub-systems of mechatronic systems are produced by different vendors, many of which may not be available during the early development states of a product. For example, in an automobile, the power train control units, the anti-lock braking control units, and the cruise control computer can be produced by different vendors. During the development of a new car, the main control board can still be tested against these third-party components using simulated controllers, thus supporting parallel development and integration testing of sub-systems.

Internal Configurability. Mechatronic systems are many times designed as product families, sharing commonalities, while supporting specific features in each family member. For example, fire panels support a core set of features, while can be customized to operate using different network protocols, power sources and user interfaces, which often require the use of different hardware components.

Deployment Configurability. Mechatronic systems must also support different deployment scenarios. A good example is in the domain of fire safety panels that must control alarms, sensors and actuators ranging from dozens of devices in a building floor, to thousands of panels and sensors, in a campus. These panels must be tested against different deployment scales before the actual deployment and commissioning occurs.

Maintainability. The test infrastructure should allow the evolution of the system under test with minimal impact on existing tests. In other words, changes in non-relevant implementation details must not affect the existing test scripts. Moreover, new tests should be able to be added at a minimum impact on existing tests and features.

Non-Functional Requirements (NFR). Mechatronic systems are often employed in support of mission-critical activities. As such, they are subject to different norms and regulations. For example, fire safety systems must generate alarms within 10 seconds of a fire being detected. They also must have high survivability, i.e. they must continue working in case of power outages and disconnection with other parts of the system. Hence, a test infrastructure must support the validation of a system under different conditions and scenarios.

Script & Execution Automation. Finally, as the scale and complexity of systems increase, and with it the systems testing costs, approaches that automate the execution, generation and reporting of test activities become increasingly important.

B. Existing Approaches and Related Work

The literature of software architecture of test harnesses for testing embedded and mechatronic systems appears somewhat sparse. Tsai et al. [11] present a process to develop object-oriented test frameworks for testing embedded systems, and focus on ease of test script development. Their process produces a test harness for a specific system, rather than a customizable framework. Fin et al. Lieberman et al. [12] show a test framework with abstraction layers similar to those of our approach and a separate configuration file; however, the top layer abstracts to test instruments, rather than the components of the SUT (System under Test). Chegi [13] describes an architecture that allows testers to transparently interchange instrumentation without rewriting scripts; the abstraction is at the level of instruments, not the SUT. None of these approaches include facilities as the easily creating and updating configurations by an integrated solution rather supporting the testing of mechatronic systems by a loosely-integrated combination of commercial and industry-specific solutions as illustrated in Figure 1. These include: hardware abstraction frameworks [4], test execution engines, and model-based testing tools [3], [14]. While these approaches individually address specific issues on different levels of abstraction, they remain loosely integrated, hindering their combined adoption in industrial settings.

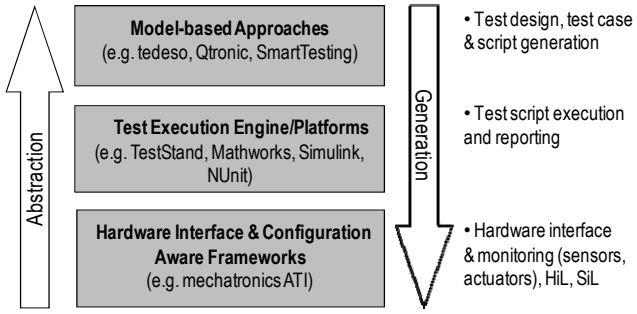


Figure 1. Existing test approaches

For example, test frameworks such as the mechatronics ATI (Automated Test Infrastructure) [4], shown in Figure 2, are used within SIEMENS for testing fire alarm systems. They support testing of systems under different deployment configurations and scenarios. The ATI focus on providing a software interface to hardware devices, allowing test scripts to be produced for different hardware configurations.

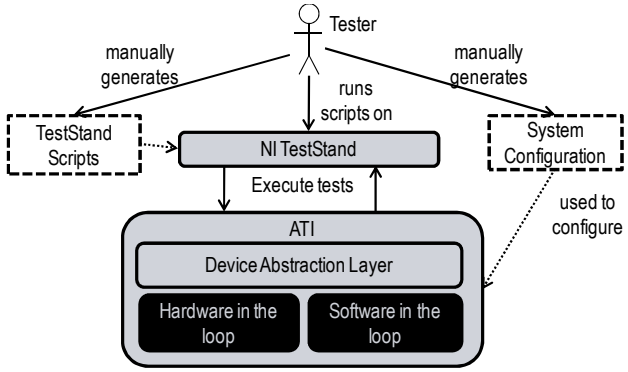


Figure 2. Automated Test Infrastructure (ATI) main elements and process

In particular, the ATI supports strategies as "Hardware in the Loop" (HiL) and "Simulator (or Software) in the Loop" (SiL), allowing the testing of (incomplete) embedded and mechatronic systems under different configurations. While frameworks such as these address the heterogeneity, external dependencies, internal configurability and maintainability of test infrastructures, their extension, configuration and use is largely manual, time consuming and error-prone.

On another level of abstraction, off-the-shelf test execution engines such as TestStand and Simulink, automate the execution of test cases, and the generation of test reports. They support the manual specification of complex test scripts, written against a high-level SUT software APIs. As such, they are good fit for ATI frameworks, providing test execution automation. In spite of their benefits, test execution engines still require the manual development of test cases, a time consuming, and error-prone activity, that requires lots of skill. They also produce test scripts that are tightly coupled to the underlying infrastructure, requiring constant test maintenance as the underlying SUT changes and evolve.

Recently, model-based testing approaches (or MBT) such as tedeso (former TDE/UML [14]) have been used in support of test generation. Tests are produced based on abstract test cases from high-level system specifications written in standardized specification languages such as UML. This transformation is performed based on existing algorithms [15] combined with different test coverage strategies [16]. The results are unprecedented guarantees and automation [17]. As such, MBT approaches support the amenability of test cases by the automatic generation of scripts. They also support the testing of non-functional requirements by supporting the concept of scenarios enriched with measurable performance attributes.

For example, tedeso is an extensible model-based testing tool that supports different testing stages: from system specification, rule-based model verification, test generation, and code and report generation. A distinctive feature of tedeso is its support for extensibility and configurability of its features by means of plug-ins (represented as white balloons in Figure 3).

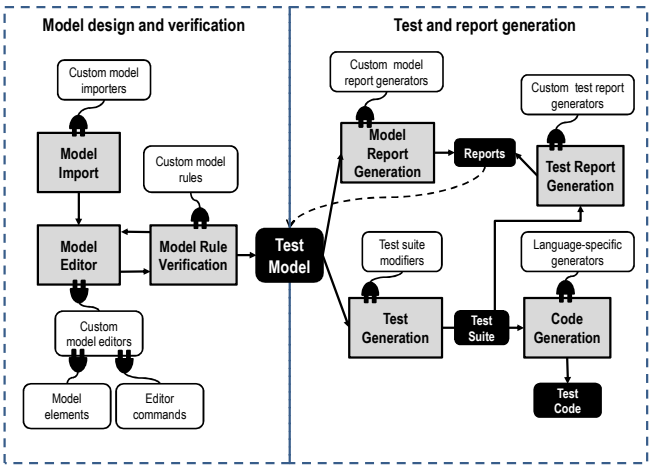


Figure 3. tedeso general approach elements

Through the use of these mechanisms, tedeso can be integrated with existing tools and approaches towards customer needs. While MBT approaches focus on test generation strategies and coverage algorithms, they do not address the problem of heterogeneity, external dependencies and the generation of configurations. They are generally not responsible for executing the generated test cases, nor do they handle the interface between hardware and software components. Instead, they rely on existing test execution and frameworks.

By integrating these individual approaches, and addressing some of their weaknesses, we propose an integrated model-based approach for mechatronic systems testing. This combined approach addresses all the requirements previously discussed as illustrated in Table I.

Table I. Existing approaches and their support for mechatronic systems requirements

	ATI	**Execution Engines**	**MBT**	**Integrated MBT**
Heterogeneity	x			x
External dependencies	x			x
Internal Conf.	x			x
Deploym. Conf.				x
Maintainability	x		x	x
NFR			x	x
Script Autom.			x	x
Exec. Autom.		x		x

III. INTEGRATED MODEL BASED TESTING

In this section, we present the main elements of our integrated model-based testing approach (or iMBT). We discuss how they have been combined and extended in support of mechatronic systems testing, and exemplify their use with a calculator example. This example was designed to be didactic, and is used to preserve the confidentiality of SIEMENS fire alarm systems, where the approach has been applied.

As illustrated in Figure 2, before applying iMBT, the testing of existing fire safety systems was supported by an Automated Test Infrastructure (or ATI), which provided a software framework for representing hardware devices that supports the execution of test scripts. The ATI provides a set of interfaces and implementations that handle the communication between test program and the system under test (SUT). For providing a software layer isolating the SUT from the test program, the ATI supports the use of simulated devices, and the testing of the system under different configurations. The ATI also provides a set of convenient services such as logging, threading and internationalization, and promotes best practices such as the separation between interface and implementation, and the use of configuration, execution and verification methods, that facilitate the development of test scripts.

In this testing environment, however, software testers are responsible for manually writing test scripts, implementing

ATI components, and providing configuration files and simulated devices to represent the SUT. While the execution of test cases is automated by execution engines such as TestStand or Simulink, the development of test scripts and configuration files is manual.

A. *Integrated Model-Based Testing Extensions*

As illustrated in Figure 4, iMBT integrates the capabilities of three main approaches: model-based testing tools (tedeso), execution engines (NI TestStand http://www.ni.com/teststand) and existing hardware interface & configuration aware frameworks (ATI).

In doing so, it automates the otherwise manual process of test script development, and system configuration & deployment. This automation facilitates the reuse of existing specification models, that can be imported into tedeso, and augmented with behavior data used for test, configuration and deployment generation. The outcome of our approach is higher levels of productivity, repeatability and automation if compared to the manual production of test scripts and configurations due to an expressive modeling that enables the automated generation of test script and system configurations.

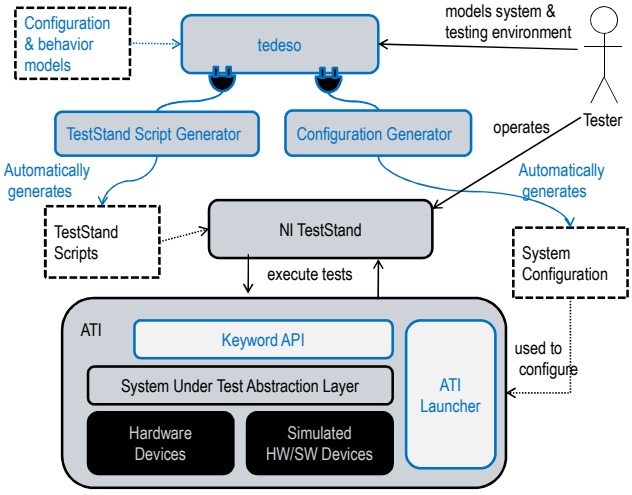


Figure 4. Integrated MBT approach (iMBT) automated process

This integrated approach required the adaptation and extension of existing tools and frameworks. In particular, we illustrate these extensions as highlighted blocks in the diagram of Figure 5. The MBT layer (implemented in tedeso) was extended with behavior, configuration and deployment idioms, as well as a test generator and a configuration & deployment generator tailored to these languages.

The execution layer (provided by NI TestStand) was reused without adaptation. Scripts generated by the MBT layer are used to test the system under different deployment and internal configurations provided by the underlying ATI. The ATI layer (based on the FireFinder ATI [4]), was extended with two new components: the Deployment-aware ATI Launcher, and the keyword processor.

The **keyword processor layer** provides high-level abstractions used by the test scripts, providing "set", "sense" and "action" methods used to put the system in an initial state, and to verify the results after the execution of certain steps. The ATI launcher is responsible for managing the deployment of ATI components in a distributed setting.

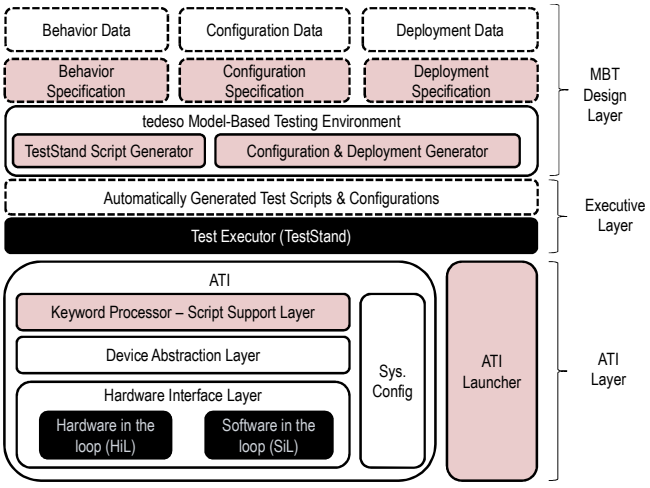


Figure 5. iMBT approach: main architectural elements

B. *iMBT Main Elements*

System Behavior Specification. The system behavior specification defines the expected behavior of the SUT, based on use cases and activity diagrams. It specifies sequences of steps, decision nodes, transitions and alternative paths in scenarios that the system must support. In our approach we use standard UML 2.0 use cases and activity diagrams to express the system behavior.

Behavior Data Specification. The system behavior data specifies the data set used during the tests. In particular, tedeso uses the category partition method [15], which prescribes a set of categories (data classes) and choices (data instances) of these categories. When associated with behavioral models by means of notes, categories and choices will determine the generated tests covering different sets of choices and/or paths in the models. The result is a set of test procedures that are used to derive executable test code.

System Configuration Specification. A configuration specification represents the sub-components of the SUT and their relations. For example, a typical configuration includes the many components, buses, and user interface elements of a fire panel. In our approach, the system configuration is specified using UML 2.0 component model.

System Configuration Data. The configuration data is used to annotate the system configuration specification with information necessary to set-up each component/sub-system. For example, IP address, polling intervals, log level options, whether the component is enabled, etc. In a way similar to the behavior model, the configuration data is associated to each component in the configuration model by means of notes.

System Deployment Specification. A deployment specification represents system nodes and subsystem

instances throughout a distributed system. In the case of a fire safety system, for example, it defines the sensors and actuators connected to fire panels; or in the case of a client/server system, the different types and instances of servers and clients. In our approach, we use UML 2.0 deployment diagrams to specify deployments.

Deployment Data. Deployment data represent information used to initialize the components in a deployment configuration, and to initialize instances of these components. For example, information about the host name, IP address, number of instances, and initialization parameters used by the configuration manager. As with the configuration specification, the system data specification is implemented by a slightly modified UML 2.0 class diagrams as supported by tedeso.

Test Code Generator. The behavior specification, annotated with its respective data specification, is used to produce a set of test scripts. These scripts are sent to a test executor, in our case NI TestStand, which interprets them and generate method calls to the SUT via ATI.

Configuration & Deployment Generator. This tedeso component translates deployment and configuration specifications into ATI configuration files. A configuration includes information about how the ATI sub-components are inter-connected (configuration), as well as the number of instances of each major system component (deployment).

ATI Launcher. This component creates multiple ATI instances, based on the generated deployment configuration file. For example, this is important in cases where multiple instances of the SUT and its clients (test executors) must be loaded, thus creating simulated deployment scenarios.

ATI Layer Customizations. It is important to mention that the ATI layer must also be configured and extended. This configuration includes a customized keyword processor, tailored to the SUT, and sub-components representing hardware and software elements of the SUT. Currently, there is no automation support for generating SUT-specific ATI components, and this step must be performed manually. Once implemented, components can be combined and reused in different configurations.

IV. CASE STUDY: CONFIGURABLE CALCULATOR

In this section, we illustrate the use of the iMBT approach in the testing of a configurable calculator. In doing so, we show how the elements of the approach are integrated in the testing of a system.

Consider a calculator with two different modes: a standard mode (with the four basic operations: +, -, / and *), and a scientific mode, with additional (!, log, ^) operators. Calculators supporting the two models have similar electronic components, with the exception of two different processors, designed for each mode. Using this approach, the company can provide different calculator models: one that has both: scientific and standard functionality, as well as a purely scientific calculator, and a purely standard calculator.

Our goal in this case study is to automate the test generation and execution activities, achieving higher degrees of test coverage, while relieving test engineers from the manual development of test cases. Using our iMBT approach, the following steps, are required: ATI extension, System modeling, Code generation and Test Execution. These steps are further explained in the following sections.

If compared to the existing manual approach, the system modeling and test execution steps replace the manual development of test cases.

A. *Stage1: ATI Extension*

The first step in the instrumentation of the calculator is the implementation of software interfaces to relevant hardware devices, implement proxies for these components, and to provide a keyword API for the whole system.

The provided hardware device proxies will interface with the calculator components via existing software monitors, connected to the real hardware components as illustrated in Figure 6.

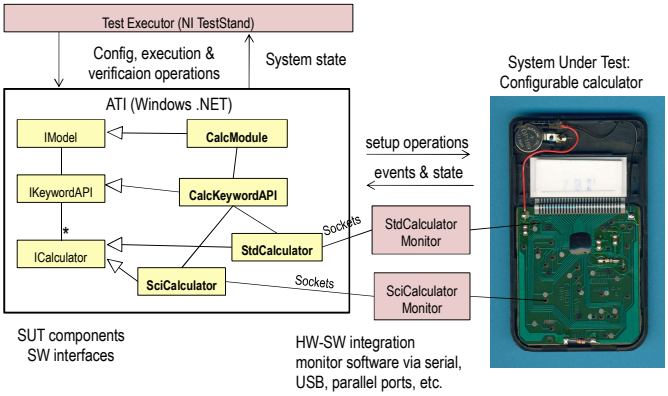


Figure 6. Extension of the ATI for the Calculator Example

Every component within the ATI must implement a stable interface. This rule allows the component implementation to vary, while their interface contracts are preserved. Using this approach, simulated devices can be used instead of the hardware components.

Besides the individual components that define the SUT, the developer should also provide a keyword processor API. The function of a keyword processor is to provide a façade to the SUT, defining a convenient set of domain-specific commands that are especially designed for test purposes. The keyword processor interacts with the system components represented in the ATI, and copes with the variability of the system.

Figure 7 shows the method used in the keyword API for the calculator example. These methods must belong to three categories:

- System set-up methods, for example: the *SwitchMode()* command, that selects a particular mode of operation;
- System state verification methods, for example, the *CheckValue()* command which checks for the state of the system;
- System execution methods, for example, the *InputNumber()* and *InputOperation()* commands that perform the data input and operations of the calculator.

The Keyword API methods must all return a Boolean status to indicate their success/failure during test execution.

This strategy allows the verification of the system through micro-steps. In particular, the verify methods are generally used to check the response from asynchronous method calls, in particular those that may take too much time to execute, thus supporting higher system parallelism.

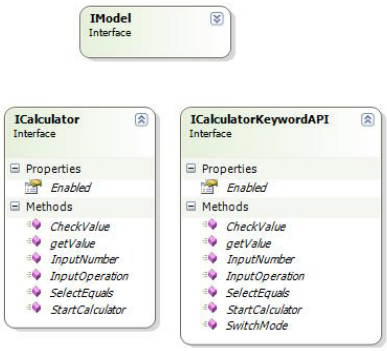


Figure 7. Calculator Example: ATI Classes & Interfaces

B. *Stage 2: System Specification*

Once the ATI is extended with the appropriate SUT sub-components, and a keyword API, the next stage is the specification of the system using UML diagrams in tedeso.

A system specification, as illustrated in Figure 8, is a collection of diagrams that include behavior models (use cases and activity diagrams), configuration models (describing system components and their inter-connection), and deployment models (with bindings between system components and hosts in a distributed system). These models follow a particular test idiom; i.e., they are annotated with special guards and notes (TDE/UML notes) that represent input/output data (passed as method parameters), initialization properties (passed to constructors), and data variables (used to multiply the number of instances of elements and data paths).

One advantage of the use of models is the possibility of reutilizing system specifications defined in the system design state. These design models can also be easily imported in tedeso, and then converted into test models. Test models are then used by tedeso for test script and configuration generation.

C. *Stage 3: Test Script and Configuration Generation*

With the specification of the system, defined in the different diagrams of Figure 8, tedeso can now generate test scripts according to existing path and data coverage algorithms, and test configurations representing internal and deployment system arrangements.

The test generation in tedeso, follows an intermediate step. Instead of generating code straight from models, abstract test cases are first produced. They represent test cases, procedures and steps. In a second step, these models are converted into executable code. During this code generation step, users can specify the target deployment or configuration models. The result of test generation is a list of NI TestStand test sequences (.seq files) representing each one of the test cases of the system. These scripts are generated to specific ATI configurations, supporting the testing of the SUT with the right set of components.

D. *Stage 4: Test Script Execution*

Finally, the user can then run the generated tests against the ATI and SUT by opening and executing the generated scripts in TestStand. TestStand also generates a standard report at the end of the execution.

V. SPECIFICATION LANGUAGE

In this section, we describe key elements of the specification language used in the system, and the code generation transformations.

The first step in support of model-based generation of test cases is the definition of idioms to support the behavior, data input/output, and to represent the different test configurations the system must support. These idioms are built upon the existing support for UML diagrams in tedeso, which currently supports a sub-set of UML, including: use case, activity, component, deployment and class diagrams.

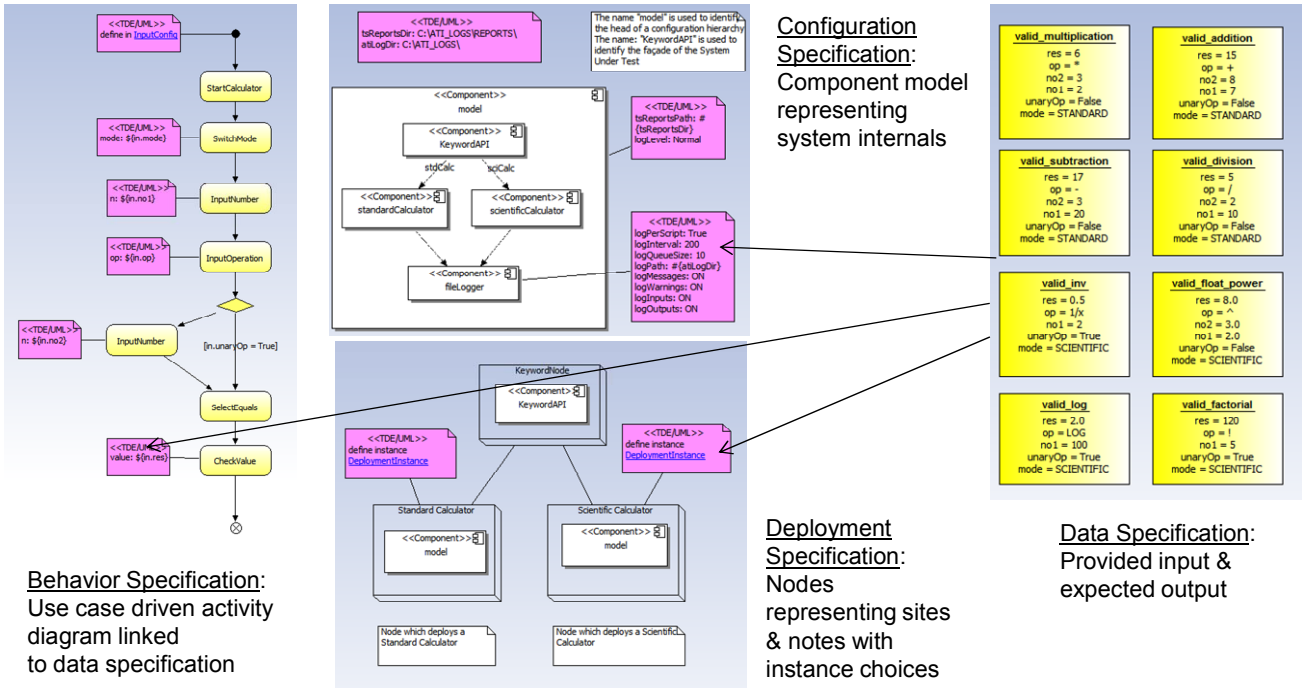


Figure 8. System specification test models in tedeso

A. Behavior Specification Language

The behavior specification language is based on the concept of use cases and activity diagrams. Use cases represent typical and exceptional scenarios that the SUT must support. Activity diagrams are used to specify individual use cases. They are represented as a sequence of steps, transitions and decision nodes (see Figure 9).

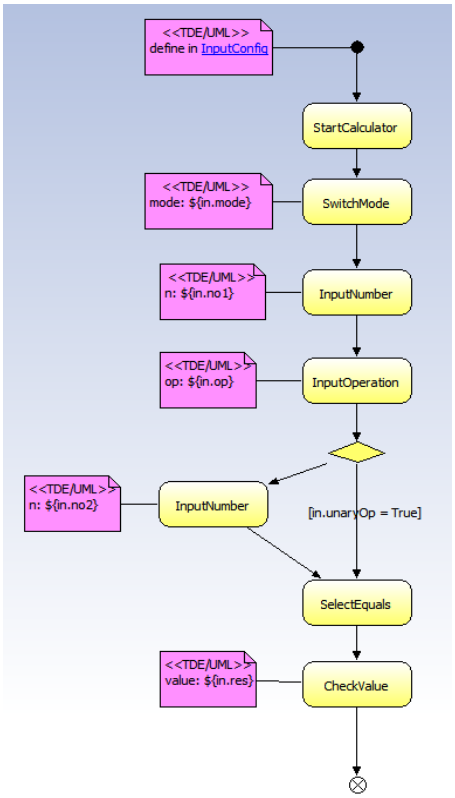


Figure 9. Configurable calculator behavior model annotated with data input and expected output choices

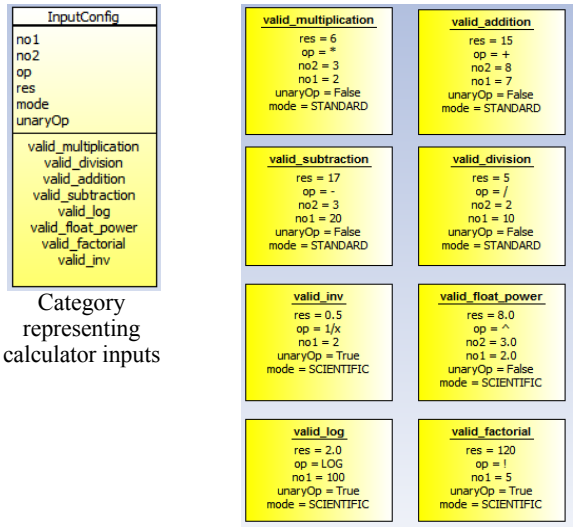
In the model of Figure 9, we describe a general scenario where a set of system operations are represented as activity steps. In particular, it describes a typical use case of a calculator. The use case consists on starting the calculator, selecting a mode (SCIENTIFIC or STANDARD), and inputting numbers and operators. The user can type either a unary (log, !, 1/x) or binary (+, -, *, /, ^) operation. Note that the four basic operators (+, -, *, /) are found in both, the scientific and standard calculators. Hence, the right calculator mode must be first selected, in an early step.

Tedeso activity diagrams were extended to support the following conventions:

- Every activity step represents one method call in the keyword API. In fact, their names must be exact matches to that API.
- The parameters of each activity step are expressed as properties ("<attribute_name>: <value>" pairs) in the notes attached to each step. These properties can have constant or variables associated to them. Variables, represented in ${} represent choices (category instances) defined in the data model, that are automatically bound to attributes during test generation. In the example, the variable "*in*" is bound to the choices within *InputConfig* category through the expression: *"define in InputConfig"* in the beginning of the activity diagram.
- As with the activity step names, the names of the step parameters must match the names of the parameter methods in the keyword API.
- Variables of type *InputConfig* can also be used in transition nodes to select one path over another in the activity diagram.

B. Data Specification Language

The data specification language is used to define data records (categories), and their instances (choices). These data records are linked to steps and decision nodes as shown in Figure 9, parameterzing the activity diagram.



Category representing calculator inputs

Choices: instances of *InputConfig* category that represent valid input/output sets.

Figure 10. Choices and categories of the data model

Figure 10 shows the *InputCategory* and its individual choices, used in the calculator activity diagram of Figure 9. Categories and choices are basic elements of tedeso category partition method [15]. Categories are represented as classes. These classes may have attributes. Instances of these classes are called choices. Choices are category instances which attributes are bound to constants as shown in the second column of Figure 10. They are shown in the body of each category. For example, *valid_multiplication*, *valid_division*, and *valid_addition* are named instances of *InputConfig*.

C. Translation between Activity Diagrams and Keyword API Calls

By using activity name conventions, and the attached data elements to the behavior model, tedeso applies the category partition method to automatically iterate over all category instances attached to a model, generating a test case to each choice in the model. A set of test cases is called a test suite.

The intermediate test suite is used by code generators to produce test executor-specific code. For example, Figure 11 illustrates the transformations applied to notes and data by the code generator.

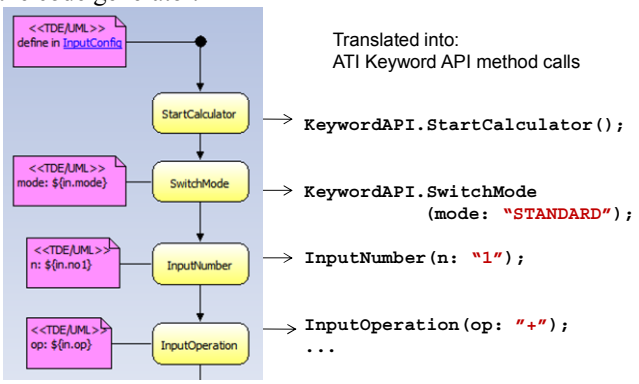


Figure 11. Code generation: from specification models into code

D. *System Configuration Specification Language*

The internal system configuration, representing its internal components and inter-connections is expressed using the UML component model. Figure 12 shows the component model for a simple standard calculator. Every component in the model is mapped to an interface and associated class in the ATI. Properties within TDE/UML notes are passed to class constructor parameters when the ATI is initialized. As with the behavior model, these properties can be parameterized in terms of choices within categories. Dependencies between components as well as containment are translated to component references passed to constructors.

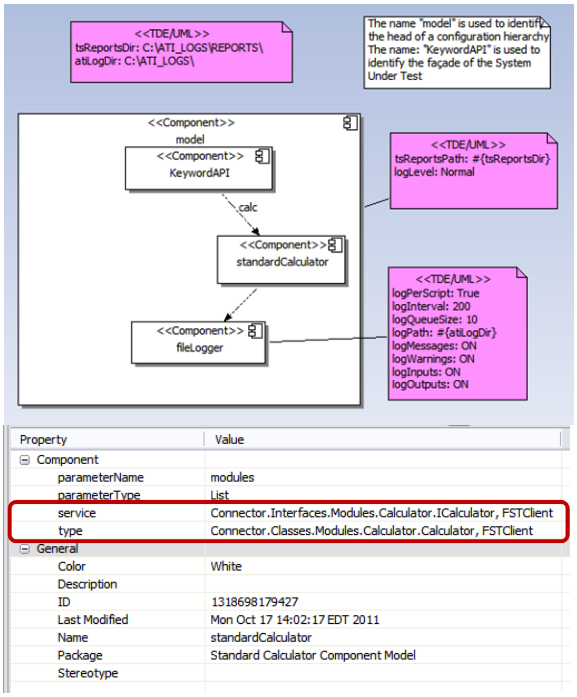


Figure 12. Components represent classes in the ATI. Interfaces and implementations are defined for each component

Figure 13 lists the configuration language used to initialize the ATI with specific components as results of the translation from the component model (see Figure 12).

In the example of Figure 13, the *fileLogger* component is implemented by the class *Utilities.Logger.FileLogger* from the *Utilities.dll* of the ATI. This class implements the interface *Utilities.Logger.Ilogger* from the same dll. An instance of *fileLogger* is then passed as a parameter to the standard calculator constructor. The dependence between *KeywordAPI* and *standardCalculator* is also converted into a constructor reference to the *KeywordAPI* class, using a parameter named "calc".

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <properties>
      <atiLogDir>C:\ATI_LOGS\</atiLogDir>
      <tsReportsDir>C:\ATI_LOGS\REPORTS\</tsReportsDir>
   </properties>
   <components>
      <component id="standardCalculator"
         service="Connector.Interfaces.Modules
                  .Calculator.ICalculator, FSTClient"
         type="Connector.Classes.Modules
                  .Calculator.Calculator, FSTClient">
         <parameters>
            <logger>${fileLogger}</logger>
         </parameters>
      </component>
      <component id="KeywordAPI"
         service="Connector.Interfaces.Modules
                  .Calculator.ICalculatorKeywordAPI,
FSTClient"
         type="Connector.Classes.Modules.Calculator
                  .CalculatorKeywordAPI, FSTClient">
         <parameters>
            <calc>${standardCalculator}</calc>
         </parameters>
      </component>
      <component id="fileLogger"
         service="Utilities.Logger.ILogger, Utilities"
         type="Utilities.Logger.FileLogger, Utilities">
         <parameters>
            <logMessages> ON </logMessages>
            <logPath> #{atiLogDir} </logPath>
            <logInputs> ON </logInputs>
            <logPerScript> True </logPerScript>
            <logWarnings> ON </logWarnings>
            <logOutputs> ON </logOutputs>
            <logQueueSize> 10 </logQueueSize>
            <logInterval> 200 </logInterval>
         </parameters>
      </component>
      ...
```

Figure 13. Configuration language used by the ATI derived from component diagrams

Using this strategy, multiple configurations can be produced such as a calculator with standard and scientific modes, as well as simpler calculators with either scientific or standard modes only.

Note that each configuration must include a Keyword API as a top-level component. This API provides a stable set of methods that are used to drive the SUT under different test cases. This approach also allows the underlying change of the system configuration with minimum impact in the test specification.

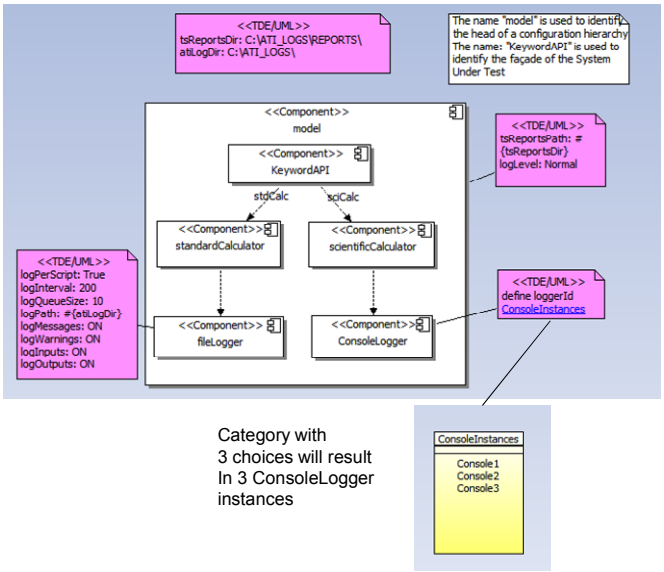


Figure 14. Multiple component instances can be generated by attaching variables to components

As with the behavior model, variables can be declared and attached to individual components shown in Figure 14. This dialect is used by tedeso to create multiple instances of a component within a configuration. For example, by attaching a variable called *loggerId* to the *ConsoleLogger* component, three instances, named "Console1", "Console2" and "Console3", will be created and passed to the *scientificCalculator* constructor as references.

E. Deployment Specification Language

The deployment specification language is similar to the configuration language. It, however, introduces the concept of nodes. Nodes represent hosts in a distributed system, where components (and their sub-components) will be deployed.

Figure 15 shows a deployment configuration where a *KeywordAPI*, controlling the whole system, is deployed in one host, while multiple instances of standard and scientific calculators are deployed in multiple hosts.

The number of instances is a function of the choices attached to each node in the diagram. According to the choices within *DeploymentInstance* category, there will be 6 hosts running standard calculator instances and 6 hosts running scientific calculator instances. Since a note with "*define component CalculatorInstance*" is attached to the scientific calculator within a node, 3 instances of *scientificCalculator* will be created in each node with this type of component, totalizing 18 instances throughout the system.

Note that parameters such as IP address or hostname can be passed to individual nodes as properties. These properties will be used to locate the host in the network.

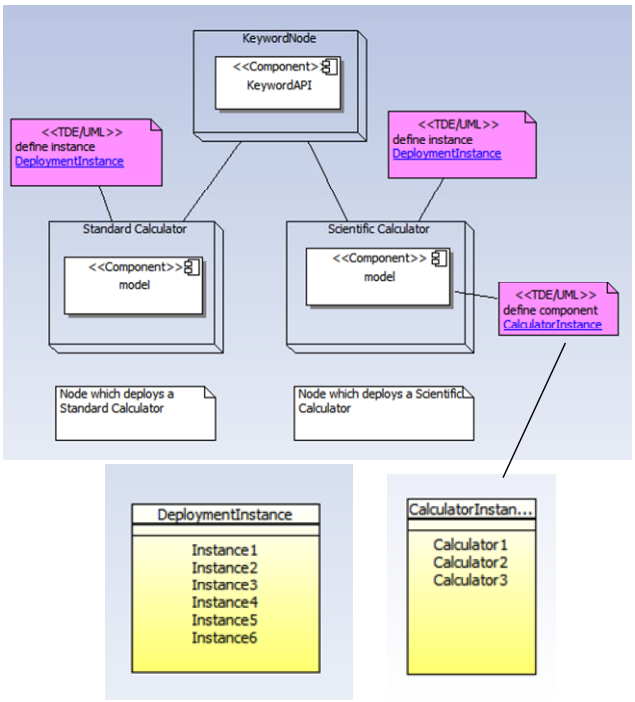


Figure 15. Deployment language example

VI. DISCUSSION

The automated approach described in this paper has many benefits to mechatronic systems testing. The use of a model-based tool as tedeso, supports different data and path converge algorithms that can significantly increase the test coverage of a system. MBT also provides automation, relieving the system tester from the burden of manually generating tests and configurations. The separation between models, abstract test cases, and executable code allows the use of different test execution engines. The use of a keyword API provides a common façade to the SUT, facilitating the code generation process, and isolating the MBT approach from system implementation. The separation between component interfaces and implementations in the ATI also support strategies as Simulator-in-the-Loop, coping with external dependencies and heterogeneity. Finally, the combination of behavior and configuration models as part of the test generation process support multiple deployments & configurations.

A prototype of the approach was implemented, as discussed in the paper. In particular, the following models and generators are currently supported. The specification languages for the ATI-specific behavior, data, component & deployment allow the modeling of a mechatronic system in sufficient detail for testing, i.e., the generation of test code as executable script and test environment configurations. The TestStand code generator leverages the use of test models described by UML activity diagrams to derive test cases that are aligned with the test environment under consideration for test execution. The ATI Configuration & deployment code generator enables the generation of various test environment configurations and deployments including simulations that

are developed in the electronics or mechanical engineering domains, without imposing demands that those simulations be complete at low levels of abstraction before software test can start. Finally, the integration between the configuration model and the behavior model leverages the execution of test cases under the generated configuration and deployment which are interfaced with the SUT. By generating the configurations automatically from models, following model driven development principles, we not only reduced the number of failures due to incorrect configuration but also make the system configuration process less tedious and error-prone.

In this paper, we introduced an integrated model-based test approach by adapting existing approaches in order to support the generation of test environment configurations and deployments for mechatronic systems testing.

VII. FUTURE WORK

A challenging aspect of the approach is deployment configuration generation. Currently, the deployment is achieved by generating individual configuration files, one for each node in the system. Thus, an implication for future research work is the implementation of the ATI Launcher, a framework that will automate the process of activation and set-up of deployment configurations. Such infrastructure must be able to start and stop its main components in tandem, thus supporting the testing of non-functional requirements.

Another challenge faced in the use of existing MBT tools was the lack of a way to specify configurations that crosscut different models. A language that allows the definitions of configurations as first-class entities would facilitate the synchronization between models, for example, selecting the appropriate deployment model for a given behavior model, a step that requires user input at the current implementation at code generation time.

Future work includes the analysis of project data and experiences on benefits, costs and drawbacks of the adopted MBT approach in industry.

REFERENCES

[1] T. R. Hsu, "Mechatronics. An overview," *IEEE Transactions on Components, Packaging, and Manufacturing Technology, Part C,* vol. 20, pp. 4-7, 1997.

[2] R. N. Carette, "This Car Runs on Code," in *IEEE Spectrum*: IEEE, 2009.

[3] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice, *in 21st international conference on Software engineering.* Los Angeles, California, 1999.

[4] S. P. Masticola and R. Subramanyan, "Experience With Developing a High-Productivity Test Framework for Scalable Embedded and Mechatronic Systems," in *SME/IEEE 2009,* San Diego, CA, 2009, pp. 175-184.

[5] P. Muir and J. Horner, "Mechatronic objects for real-time control software development," in *1998 SPIE International Symposium on Intelligent Systems and Advanced Manufacturing: Mechatronics Conference*, 1998.

[6] M. Jenko, N. Medjeral, and P. Butala, "Component-based software as a framework for concurrent design of programs and platforms – an industrial kitchen appliance embedded system," *Microprocessors and Microsystems,* vol. 25, pp. 287-296, 2001.

[7] K. Kondo and M. Yoshida, "Use of hybrid models for testing and debugging control software for electromechanical systems," *IEEE/ASME Transactions on Mechatronics,* vol. 10, pp. 275 -284, 2005.

[8] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI Testing Using a Model-driven Approach," in *Intl. Workshop on Automation of Software Test* Shanghai, China: ACM, 2006.

[9] W. Schafer, "Model driven development with mechatronic UML," in *IEEE Symposium on Visual Languages and HumanCentric Computing* Corvallis, Oregon, 2008, pp. 9-10.

[10] W. Schafer and H. Wehrheim, "The Challenges of Building Advanced Mechatronic Systems," in *Future of Software Engineering*, Minneapolis, MN, 2007, pp. 72 -84.

[11] W.-T. Tsai, Y. Na, R. A. Paul, F. Lu, and A. Saimi, "Adaptive scenario-based object-oriented test frameworks for testing embedded systems," in COMPSAC, 2002, pp. 321–326.

[12] I. Lieberman, S. Insalaco, and A. Van Oosbree, "A software approach for a common munitions test architecture," in Proceedings of AUTOTESTCON 2000, Anaheim, CA, 2000, pp. 108– 119.

[13] D. Cheji, "A software architecture for building interchangeable test systems," in Proceedings of AUTOTESTCON 2001, Valley Forge, PA, 16-22 2001.

[14] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, "A UML-based approach to system testing," Innovations in Systems and Software Engineering, vol. 1, pp. 12-24, 2005.

[15] T. J. Ostrand and M. J. Balcer, "The Category-partition Method for Specifying and Generating Fuctional Tests," Commun. ACM, vol. 31, pp. 676-686, 1988.

[16] R. S. Silva Filho, C. J. Budnik, W. M. Hasling, M. McKenna, and R. Subramanyan, "Supporting Concern-Based Regression Testing and Prioritization in a Model-Driven Environment," in 2nd Intl Workshop on Software Test Automation (STA 2010), Seoul, Korea, 2010.

[17] A. D. Neto, R. Subramanyan, M. Vieira, G. H. Travassos, and F. Shull, "Improving Evidence about Software Technologies: A Look at Model-Based Testing," IEEE Software, vol. 25, pp. 10-13, May-June 2008.