


SmartTS: A Component-based and Model-Driven Approach to Software Testing in Robotic Software Ecosystem






Vineet Nagrath¹, Christian Schlegel²
Service Robotics Research Center
Technische Hochschule Ulm
89075 Ulm, Germany

Abstract—Validating the behaviour of commercial off-the-shelf components and of interactions between them is a complex, and often a manual task. Treated like any other software product, a software component for a robot system is often tested only by the component developer. Test sets and results are often not available to the system builder, who may need to verify functional and non-functional claims made by the component. Availability of test records is key in establishing compliance and thus selection of the most suitable components for system composition. To provide empirically verifiable test records consistent with a component's claims would greatly improve the overall safety and dependability of robotic software systems in open-ended environments. Additionally, a test and validation suite for a system built from the model package of that system empirically codifies its behavioural claims. In this paper, we present the “SmartTS methodology”: A component-based and model-driven approach to generate model-bound test-suites for software components and systems. SmartTS methodology and tooling are not restricted to the robotics domain. The core contribution of SmartTS is support for test and validation suites derived from the model packages of components and systems. The test-suites in SmartTS are tightly bound to an application domain's data and service models as defined in the RobMoSys (EU H2020 project) compliant SmartMDS toolchain. SmartTS does not break component encapsulation for system builders while providing them complete access to the way that component is tested and simulated.





Keywords—Model-Driven Engineering (MDE); Component-based Software Engineering (CBSE); Model-Driven Testing (MDT); Component-based Software Testing (CBST); Service Robotics; Software Quality; Automated Software Testing

I. INTRODUCTION

A software product may be difficult to understand and modify, it might be prone to misuse and difficult to use, it might not integrate well with another piece of software and it might work on only a particular machine and only under some very hard assumptions. Unless one looks under the hood to measure software on these parameters, the *quality* of the software can not be judged purely on the basis that it works and was delivered on time [1]–[3]. Fig. 1 shows key characteristics that can be used to evaluate the overall quality of a software product ¹. A good software product can be qualified as testable if it performs well on the following quality parameters as suggested by Boehm et al. [1].

- B1 **Accountability**: *Code*² allows for mechanisms to measure its usage, e.g. code instrumentation .
- B2 **Accessibility**: Code allows selective usage of its parts and provides side entrances for test access .
- B3 **Communicativeness**: Code allows specification of inputs and provides corresponding outputs in a usable form .
- B4 **Self-descriptiveness**: Code provides enough information, e.g., a test model, for its *use*³ and *verification*⁴ .
- B5 **Structured**: Code is organized into definite interdependent parts, i.e. the software system is composed of part-wise/component-based testable units .

In summary, a testable code has an established verification criterion and supports performance evaluation. Testing in the software industry is broadly categorised into *unit*, *integration* and *system testing* [4]–[7]. A fourth level, namely, *acceptance testing* may be added on some occasions (see Fig. 2).








- L1 **Unit testing** as the name suggests tests individual *code components* .
- L2 When different code modules are joined, the test of data flowing through the *interface* is **integration testing** .
- L3 A system prepared by several code units is tested against *functional and non-functional system-level requirements* in **system testing** .
- L4 An additional *enforcement* check performed against a contract, after the final delivery of the software product is **acceptance testing** .

High speed, repeatable, low effort and inexpensive testing improves the overall quality of the software product and its development workflow. A well-structured test and validation mechanism can be made cheaper when automated. Removing the human element from the testing and enforcement equation

²Code in software engineering is short for *source code* unless stated otherwise.

³Component blocks, ports (required and provided services), connectors.

⁴Objectives, assumptions, constraints, revisions and usage history.

¹  Throughout this paper, Indicators like      &  are internal connections between notions presented in the paper.

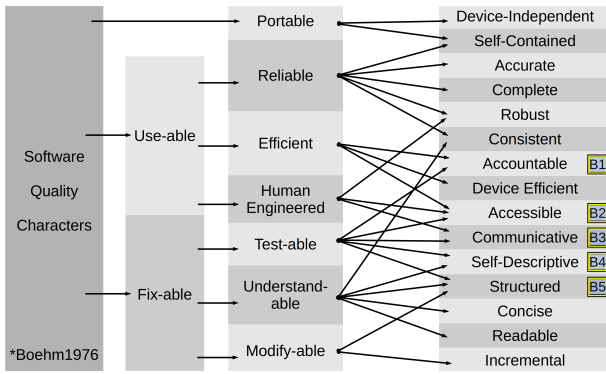


Fig. 1. Software Quality Characteristics Tree [1].

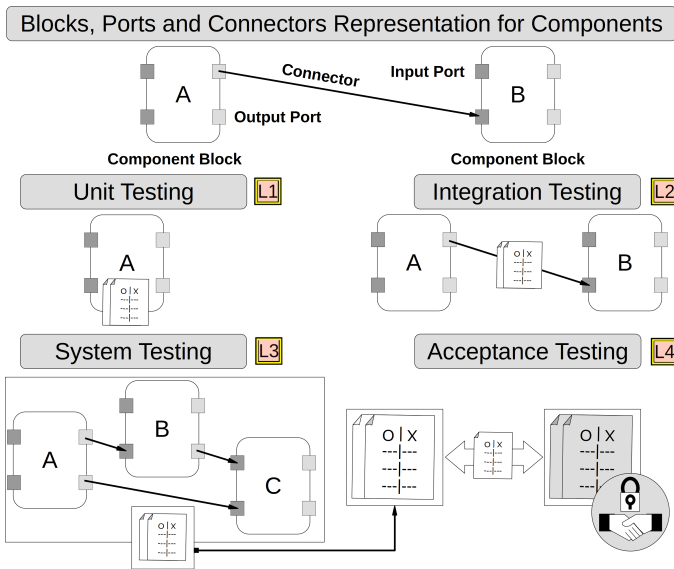


Fig. 2. Software Test Levels. Blocks-ports-connectors Representation for Software Code Modules.

is of great value for high complexity, incrementally growing software systems. Machine delivered tests allow for frequent regression checks on legacy systems in a long-term life cycle of a software product. The importance of automation in software testing was realized very early in the development of software development as an industrial process [8], [9]. A trade-off exists between the degree of automation in testing and the regularity by which those tests will be executed, which tilts heavily in favour of automation as a system grows more elaborate and mission-critical [10]. Thus, it will be safe to assume that any modern software development methodology will be incomplete without a comprehensive plan to automate testing in all test levels (L1-L4)@ and across all test quality parameters (B1-B5)@.

Testing robotic software differs from software testing in general for several reasons.

- R1 **Robots interact with the physical world** with sensors and actuators which are inherently prone to noise and faulty execution [11] R1.
- R2 **Environments**, where robots work, are often open-

ended with very few assumptions [12]. There is a physical danger involved while working with robotic systems, especially during their testing R2.

- R3 **Safety concerns** are high especially when robots work near humans R3.
- R4 **Robot cost** is high and their availability for extensive testing is low R4.
- R5 Robots, sensors mounted on those robots and the software driving the two are often built by different vendors with a wide scope of utilization in mind. Robust and verifiable hardware and software composition thus become ever more important for robotic systems [13] R5.
- R6 Very few off-the-shelf software components fit real-world applications, a large portion of code has to be custom made for a particular robot R6.
- R7 Robots are built using several hardware components like gears, wheels and consumables which wear down, get damaged or replaced over time. The challenge is to reuse, with confidence, the existing software against worn-out or some new, slightly different hardware [11], [14] R7.
- R8 Difficult to specify what constitutes a correct behaviour for a robotic system [15] i.e. it is not always clear what needs to be tested. The challenge comes in particular from the open-ended world for which full coverage testing is not possible R8.
- R9 People from various domains work on a robot, not all of them are trained software engineers R9.
- R10 Lack of communities and uniform standards for the robotic industry. Standards for robotic hardware and software should be made abstract and encapsulated to hide the intellectual property of a business, while not compromising its usability and configurability for the end-user R10.

A recent extensive study on the challenges of testing robotic systems concluded with three important themes describing major challenges in testing robots. Following are the themes and suggested solutions reasoned in the study [16].

- A1 **Real-world complexities:** A robot's interactions with the real world is a key difficulty in testing robotic software A1.
 - A1S1 Rapid development of reliable simulators with better Application Programming Interface (APIs) and User Interfaces (UI), leading to better automated simulation testing A1S1.
 - A1S2 Research on tools and techniques for automated testing of robotic software A1S2.
- A2 **Communities and standards:** Community-driven standards promote product quality and incentivise member businesses A2.
 - A2S1 Developing a robotic software ecosystem with special emphasis on software quality standards A2S1.

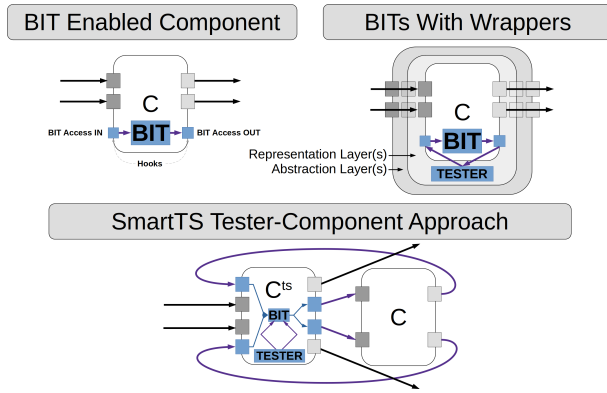


Fig. 3. SmartTS Approach in the Context of BITs and Test Wrappers.

A2S2 Guidelines and tools to promote healthy practices for the growth of the ecosystem [A2S2].

A3 Component integration: Robotic software demands an integrated (hardware + software) approach to system testing [A3].

A3S1 Better availability of hardware components for testing within the community [A3S1].

A3S2 Promote tools like hardware-in-the-loop simulation borrowed from other industries [A3S2].


A3S3 Develop better test oracles⁵ to ensure that the testing apparatus knows with confidence the results of running a test [A3S3].

In the past decade, there have been some application-specific proposals for systems for testing software for robots [17]–[27]. These approaches present tools and algorithms to address the above [A1–A3]@ concerns but they are either too specific for a particular robotic application [18]–[20] or are generic extensions of general software testing techniques and standards [21], [22] with little to no planning specifically for robotic software testing [R1–R10]@. In either case, none of these is community-driven or providing tools or standards for businesses to follow. A notable exception to this is the Robot Testing Framework (RTF) [23] which is the right step towards test automation by a plug-in based approach to testing that is independent of platform, middleware and programming language used for writing the test plug-ins. There are works focusing on improving simulation testing [24]–[26] and model-driven performance testing [27]. An idea borrowed from the computer hardware industry [28], the *built-in test enabled components* [29]–[31], is to have a functionally separate maintenance mode to provide access to **Built-In Tests (BITs)**. Using **test-wrappers** is another common approach that works along with the BIT approach to envelop the software component in a single or multi-layer software wrapper that is transparent to both the component being tested and its peers in the environment. These wrappers, when used with BITs, enable *testing without breaking component encapsulation*. The *RESOLVE* [32] approach is one such approach, it proposes a two-layer wrapper to achieve *automated black-box testing of*

software components.

The *SmartTS* approach to component testing is to create a tester component whose model is derived from the model of the component being tested. This tester component implements the BITs for automated testing of the component and its code is transparent (white-boxed) to the ecosystem, thus the *component encapsulation* is maintained (black-boxed) while no additional operational overhead is attached to the component for implementation and execution of BITs in maintenance/test mode (See Fig. 3).

According to our experience of working with robots in the service robot industry, *community-driven models* with a special focus on *Component-Based Software Development (CBSD)* works best for the development of the robotic software component. Although there exist several *Model-Based Engineering (MDE)* approaches for software development in general [31], [33]–[37], approaches with a special focus on the robotic industry are essential for the growth of the robotics industry (Multi-annual roadmap [38], the European SPARC Robotics [39] initiative). One such effort towards creating an ecosystem for model-driven and component-based development of robotic software is the EU H2020 *RobMoSys: Composable Models and Software for Robotics* [40], [41] project. *Meta-models* that promote *separation of concerns* [42], [43] along different roles such as robotic experts and application domain experts are highly desirable for the industry. MDE supports the separation of concerns and of roles since it provides operational modules dedicated to use by specific stakeholders. The *RobMoSys* approach has a special emphasis on a clear separation of concerns and roles and promotes community building for efficient collaboration between stakeholders. Other model-based efforts towards a *robotic software ecosystem* [44]–[47] are also taking separation of concerns and roles as an essential part of their working philosophy, which will be essential for their success [48].

In CBSD for *software-intensive service robotic systems*, validating the behaviour of a supplied component and its interactions with other components is a complex, and often a manual task. In *EU Robotics Strategic Research, Innovation and Deployment Agenda 2020 on AI, Data and Robotics Partnership* [49], *trustworthiness* was identified as one of the *core characteristics* that *robotics and AI systems* need to display. **Trustworthiness**  is a property of the system derived from the trustworthiness of its constituents and their interactions. Treated like any other software product, a software component for a robot system is tested by the component developer (and/or component tester) at the vendor's (component supplier) end. Test-sets and records of test-results are often not available to the system builder, who may need them to verify functional and non-functional claims made by the vendor about the component. Availability of test records is key in establishing compliance and thus selection of the most suitable component for system composition. To provide empirically verifiable test records consistent with a component's claims would greatly improve the overall safety and dependability of robotic software systems in open-ended environments. It is of added benefit that when a system is composed of several components, a part of the system's test and validation suite is automatically generated from the test-suites of the constituent components. This further helps

⁵Test oracle (or simply Oracle) is software engineering terminology for any mechanism by which a test script determines whether a test case has passed or not.

empirically codify a system's functional and non-functional behavioural claims. To the best of our knowledge, there is an absence of a **wholistic model-driven approach towards CBSD for robotic systems, that integrates support for test and validation suites⁶ within the model-package⁷ of a robotic software component or system.** Inclusion of test sets and expected results within the *model-package* in *model-driven component-based software development* enables the following modelling and transformational⁸ functionalities.

- E1 Meta-modelling framework to define and codify **component-test-model** [E1](#).
- E2 **Automated generation of user-editable component-level test-suites** from *component* and *domain model packages*. Automated **model to data transformations (M2D)** of user-ascribable *test-data sets* from *domain* and *user test documents*.
- E3 **Automated testing** at vendor's side [E3](#).
- E4 Empirically verified **adequate component selection** during system composition [E4](#).
- E5 Meta-modelling framework to define and codify **system-test-model** [E5](#).
- E6 **Automated generation of system-level test-suites** from *component-level test suites* and *system-test-model packages* [E6](#).
- E7 **Does not break component encapsulation** [E7](#).
- E8 **Does not break system encapsulation** to *enforcement and verification agents* [E8](#).

In this paper, we present the “**SmartTS methodology**”: **A component-based and model-driven approach to generate model-bound test-suites for software components and systems.** The test-suites in *SmartTS* are tightly bound to an *application domain's data and service models* as defined in the *RobMoSys* [40] (EU H2020 project) compliant *SmartMDS* [50], [51] *Toolchain*. *SmartTS* provides *automated generation, execution and transformation of test-suite models and test-suite results* across a *service domain, component and system models*, enabling *automated testing and verification of components and systems*. *Component test-suite results* are used for *selecting an appropriate component for composition*. *System test-suite results* are used for *documenting or sensing system behaviour during composition, acceptance testing, enforcement or for run-time diagnosis*. *SmartTS* does not break *component encapsulation* for *system builders* while providing them complete access to the way that a *component* is *tested and simulated* (Supporting *composition and separation of roles*).

The rest of the paper is organized as follows. Section titled **SmartTS Overview** introduces the intended *goals and contributions* of the *SmartTS toolchain*. It presents the *principles and methodologies* that have inspired *SmartTS*. Sections

⁶A *test and validation suite* is a set of testable statements about a software component or a multi-component system, which when true indicates the validity of a particular behavioural claim of the entity.

⁷A set of models that collectively define an entity.

⁸A model transformation in MDE is an automated mechanism of transforming one entity into another, where the entity could mean a model or text (including raw data or code written in a programming language).

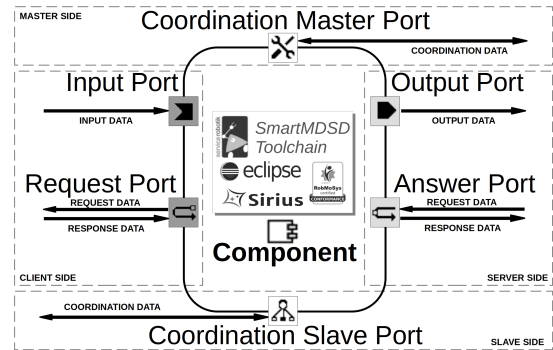


Fig. 4. Anatomy of a SmartMDS Component.

SmartTS and the SmartMDS Toolchain and SmartTS in the Context of RobMoSys present how *SmartTS* integrates with the existing *SmartMDS toolchain* and the *RobMoSys ecosystem*. This paper puts the focus on **new meta-models, domain-specific languages (DSLs), roles, views, model-to-model (MMT), model-to-text (M2T) and model-to-data (M2D) transformations** that *SmartTS* introduces. Section **SmartTS in Action** presents a few use-cases for *SmartTS* to showcase its impact. An illustrative sketch of *SmartTS* tooling based on *Eclipse* features and plug-ins is provided without going into how it is implemented. Finally, Section **Conclusions and Future Works** draws some conclusions on the presented work and outline future research direction and works.

II. SMARTTS OVERVIEW

The *core philosophy and contribution* of the *SmartTS methodology* is to embed models describing the tests for a component within the model package of the *component*. A *tester component* is then generated from the *model package* that without breaking *component encapsulation* implements *built-in tests* and drives *automated testing* for the *component*. The *tester component* generates *empirically verifiable test records* which are distributed with the *component* to support its *claims*. **Since the trustworthiness of a system is derived from its components, the system-level test-suite is partly derived from tester components and models of components that constitute the system.** *Model-driven and component-based software development* form the base on which the *SmartTS methodology* is placed. In this section, we will walk through the *methodology* and present the *mechanism* by which *SmartTS* proposes a *component-based and model-driven approach to software testing* in a *robotic software ecosystem*.

SmartTS is a member of the *RobMoSys/SmartMDS ecosystem*. In this paper, we are presenting the *SmartTS methodology* in the *context* of its *core ecosystem (RobMoSys/SmartMDS)*. **The principles and mechanisms described here though can be transported as-is to any component-based and model-driven software ecosystem.** Fig. 4 shows the *anatomy* of a typical *SmartMDS component*. It is typical in *CBSD* to represent *components and systems* using the *blocks ports connectors notation* (see Fig. 2). In this paper, we use a *custom blocks ports connectors notation* (Fig. 5) to present the *SmartTS methodology*. Note that *SmartMDS components* can have any number of *input, output, request or answer ports* and exactly one *coordination & configuration*

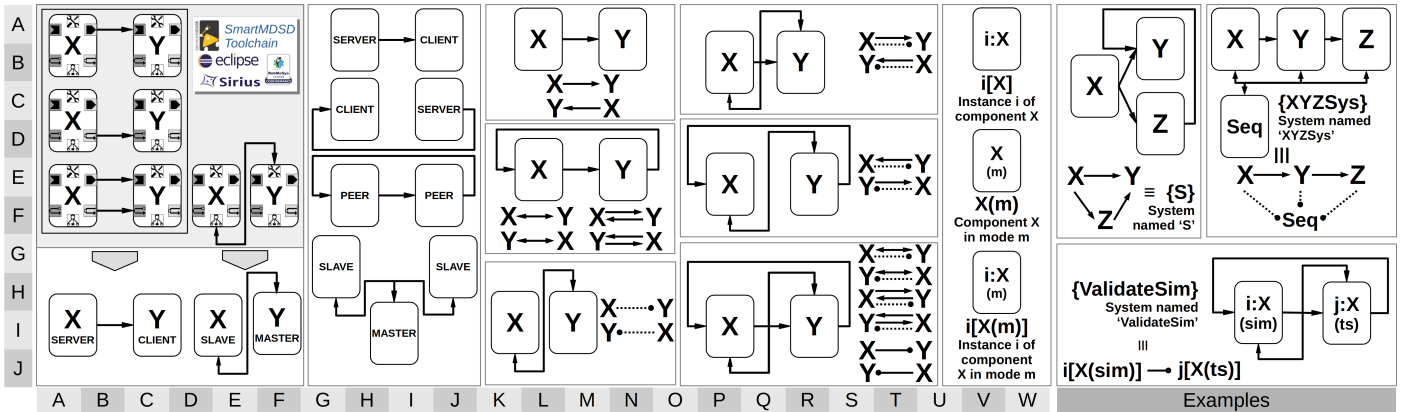


Fig. 5. Custom Blocks Ports Connectors Notation used in SmartMDS::SmartTS Methodology.

i	C	Component 'C'	
ii	C^{ts}	SmartTS Tester Component for Component 'C'	
iii	C^{ts(test)}	Component 'C ^{ts} ' in Test mode	
iv	C^{ts(train)}	Component 'C ^{ts} ' in Train mode	
v	C^{ts(sim)}	Component 'C ^{ts} ' in Simulate mode	
vi	C^{tr}	Test Results for Component 'C'	
vii	{C^{test}}	A System for Testing Component 'C'	
viii	{C^{train}}	A System for Training Component 'C ^{ts(sim)} '	
ix		$C \xrightarrow{\text{TRANSFORMS TO}} C^{ts}$	E7
x		$\{C^{test}\} \equiv \{C \rightarrow C^{ts(test)}\}$	L1
xi		$\{C^{test}\} \xrightarrow{\text{GENERATES}} C^{tr} \xrightarrow{\text{VALIDATES}} C$	E1 E2 B1 L4
xii		$\{C^{train}\} \equiv \{C \rightarrow C^{ts(train)}\} \xrightarrow{\text{TRAINS}} C^{ts(sim)}$	B4
xiii	{S}	A System 'S' containing Component 'C'	
xiv	{SsC}	Simulating Component 'C' in System 'S'	
xv	{SpC}	Simulating Peers for Component 'C'	
xvi	{X^{tr}}	Test Results for System 'X'; X: SsC, SpC, ..., S ^{sim}	
xvii	{S^{sim}}	Simulating System 'S'	
xviii		$\{S\} \equiv \{A, B, C\} \equiv \{A \rightarrow B\}$	E8
xix		$\{SsC\} \equiv \{A, B, C^{ts(sim)}\} \xrightarrow{\text{GENERATES}} \{SsC\}^{tr}$	B2 B5
xx		$\{SpC\} \equiv \{A^{ts(sim)}, B^{ts(sim)}, C\} \xrightarrow{\text{GENERATES}} \{SpC\}^{tr}$	L2
xxi		$\{S^{sim}\} \equiv \{A^{ts(sim)}, B^{ts(sim)}, C^{ts(sim)}\} \xrightarrow{\text{GENERATES}} \{S^{sim}\}^{tr}$	L3
xxii		$\{SsA\}^{tr}, \{SpA\}^{tr}, \{SsB\}^{tr}, \{SpB\}^{tr} \xrightarrow{\text{TRANSFORMS TO}} \{S\}^{tr}$	E3 E4
xxiii		$A^{tr}, B^{tr}, C^{tr}, \{S\}^{tr} \xrightarrow{\text{VALIDATES}} S$	E5 B1 L4

Fig. 6. SmartTS Transformations and Validation Mechanisms.

slave port unlike what one may infer from the simplified representation in Fig. 4. If a component has a master port of a coordination & configuration interface integrated then it can coordinate and control other components. Furthermore, the four-side arrangement of ports in Fig. 4 is only for representation and the SmartMDS toolchain GUI doesn't tightly bound

these ports to particular sides of a component. SmartMDS Services [52] is an item being transported (communication object [53]) in a particular manner (communication pattern [54]). Depending on the communication pattern, the SmartMDS component could possess an input, output, request or answer port (Fig. 4). The 'send' communication pattern is one-way while a 'query' communication pattern is for two-way communication of communication objects. A publish/subscribe mechanism is available for one-to-many communication using 'push' (distribution) and 'event' (asynchronous notification) communication patterns. Coordination ports are for a two-way exchange of 'coordination' patterns ('parameters', 'states', 'dynamic wiring' and 'monitoring data'). These 'coordination' patterns are internally built on top of 'data' patterns ('send', 'query', 'push' and 'event'). A system built using the SmartMDS toolchain has a default coordination master (e.g. a sequencer) with all constituent components as its coordination clients, in a configuration similar to the system XYZSys (examples)⁹.

For the benefit of the reader, it is enough to retain that a SmartMDS component typically acts as a service consumer as well as a service provider at the same time. It is coordinated by a global coordination master component (sequencer) which in normal usage is hidden from the user. A SmartTS tester component to a component would thus become a consumer to every service provided by the component and provider to all services requested by the component. It will also act as a coordination master to the component and the system built using the component and its tester component would have a configuration similar to the one shown in (GO-JU)⁹ between component X (HP⁹) and its tester component Y (HR⁹). In shorthand notation, this system would be written as IT⁹. The reader is advised to go through the notation given in Fig. 5. A component can have more than one instance in a system (CV⁹) and can have differently named operating modes (FV⁹). Systems are represented with their names in curly brackets (examples⁹). Mapping of SmartMDS component notation (Fig. 4) to SmartTS custom notation is given in (AA-JF)⁹.

Fig. 6 shows key SmartTS transformations and validation mechanisms. A component C (Fig. 6.i) is transformed to its

⁹See alphanumeric coordinates in Fig. 5

tester component C^{ts} (Fig. 6.ii, Fig. 6.ix). In *SmartMDS*D, the component code is generated from its component model package. The same model package is transformed into the model package for the component C^{ts} . This tester component provides an empty code template which is later filled to implement BITs for component C. Once BITs are implemented and linked to associated test and validation data (discussed later in the Section **SmartTS and the SmartMDS**D Toolchain), the component C^{ts} is deployed to test component C (Fig. 6.x). The component C^{ts} has three principal operating modes namely test, train and simulate (Fig. 6.iii-v). In test mode, the component C^{ts} is deployed with component C to form the test system $\{C^{test}\}$ for component C (Fig. 6.vii, x). The test results C^{tr} from the system $\{C^{test}\}$ are used to validate the claims made by the component (Fig. 6.vi, xi). The component C^{ts} is deployed in train mode to form the training system $\{C^{train}\}$ for component C (Fig. 6.viii, xii). This training system $\{C^{train}\}$ trains the component C^{ts} to work in the simulated mode. Note that the C^{ts} is simulated against BITs which may not match the BITs implemented for its test mode. The difference between these two sets of BITs is only in terms of the motivation behind their existence.

SmartTS Tester Component

SmartTS tester component for a component is a consumer to every service provided by the component, provider to all services requested by the component and it acts as a coordination master to the component.

SmartTS Test System

SmartTS test system is a system with a component and its tester component deployed to execute the BITs implemented by the tester component and generate corresponding test results.

SmartTS Trainer System

SmartTS trainer system is a system with a component and its trainer component deployed to execute the BITs implemented by the trainer component and generate a fully trained simulator component.

SmartTS Simulator Component

SmartTS simulator component is a tester component operating in the simulate mode. The simulator component can reproduce the service and coordination behaviour of the component for a specific set of BITs.

In Principal, once trained, the component C^{ts} in simulate mode can reproduce the service and coordination behaviour of component C for a specific set of BITs. This simulated mode component C^{ts} is then used in various simulated variants ($\{SsC\}$, $\{SpC\}$ and $\{S^{sim}\}$): Fig. 6.xiv, xv, xvii, xix-xxi) of a given system $\{S\}$ (Fig. 6.xiii, xviii). Results (Fig. 6.xvi, xix-xxii) from these simulated variants of the system are transformed to a single set of simulation test results $\{S\}^{tr}$ for the system $\{S\}$ (Fig. 6.xxii). **Trustworthiness** (Conformance to claims and agreed upon BITs) is a property

of the System ($\{S\}$) derived from the trustworthiness of its constituents (A^{tr} , B^{tr} and C^{tr}) and their interactions ($\{SsA\}^{tr}$, $\{SpA\}^{tr}$, $\{SsB\}^{tr}$, $\{SpB\}^{tr}$, $\{SsC\}^{tr}$, $\{SpC\}^{tr}$, $\{S^{sim}\}^{tr}$). The simulation test results $\{S\}^{tr}$ for the system $\{S\}$ along with test results of its constituents (A^{tr} , B^{tr} and C^{tr}) are used to validate the claims made by the system (Fig. 6.xxiii).

III. SMARTTS AND THE SMARTMDS

D TOOLCHAIN

*SmartMDS*D toolchain [50], [51] is a *RobMoSys* [40] compliant model-driven tooling for component-based robotic software development based on the *SMARTSOFT* methodology [55]. *SmartTS*: Test-suite extensions for *SmartMDS*D toolchain, presented for the first time through this paper is an addition to the existing *SmartMDS*D toolchain and provides constructs for modelling built-in contract testing in systems built using the *SmartMDS*D toolchain. *SmartTS* provides models to associate a test and validation suite with any of the existing *SmartMDS*D models. It also allows for the creation and usage of data elements associated with the test and validation suites. **Eclipse features and plug-ins for SmartTS are available for download** [56]. Context and video tutorials on the use of *SmartTS* will soon be available online at *SRRC* wiki web page [57]. Fig. 7 shows the key elements of the *SmartTS* methodology.

*SmartMDS*D and *SmartTS* elements span across two tiers (1-20, 20-50)¹⁰ and involve four main actor-groups. Domain experts (2C,15N)¹⁰, component developers (24D, 48L)¹⁰, system builders (23D, 48R)¹⁰ and behaviour developers (22C¹⁰). The two tiers in the *SmartTS* methodology correlate with tier-2 and tier-3 of *RobMoSys* (Section **SmartTS in the Context of RobMoSys**). Component developers, system builders and behaviour developers are tier-3 ecosystem users (22E, 25N, 25R)¹⁰. A tier-3 ecosystem user exchanges content (20P, 5K, 20T, 20V)¹⁰ and writes models (28C, 36C, 48C, 33T)¹⁰ that conforms to (20B, 31J)¹⁰ domain-specific models (7C, 6G)¹⁰ defined by the domain experts in tier-2. The domain experts write the *SmartMDS*D domain model package (7C¹⁰) which contains several domain models (9A-19D)¹⁰ about services (19B¹⁰), data (12B¹⁰), dependency (15B-18B)¹⁰, modes (13B¹⁰), parameters (9B¹⁰), tasks (10B, 11B)¹⁰ and documentation (14B¹⁰). These models are later implemented or imported by tier-3 users (22E¹⁰) in *SmartMDS*D component (28C¹⁰), system (36C¹⁰) and behaviour (48C¹⁰) models. *SmartMDS*D component model package (28C¹⁰), written by component developers (24D¹⁰) consists of component models (30A-35D)¹⁰ that describe a component's parameters (30B¹⁰), code structure (35B¹⁰), ports (35B¹⁰), dependency objects (34B, 35B)¹⁰, skills (31B¹⁰) and documentation (33B¹⁰). System builders (23D¹⁰) write the *SmartMDS*D system model package (36C¹⁰) containing models (38A-46D)¹⁰ that describe structure (39B, 46B)¹⁰, operations (38B, 40B, 42B)¹⁰, dependency (44B, 45B)¹⁰ and documentation (43B¹⁰) for the system. Behaviour developers (22C¹⁰) write the *SmartMDS*D behaviour model (48C¹⁰) that specifies how domain tasks (11B¹⁰) are realized (50C¹⁰) in a particular behaviour. The *SmartTS* models (7F-18J, 34F-48J)¹⁰ are derived (17F, 48F)¹⁰ from the existing *SmartMDS*D domain, component, system and behaviour models [58].

¹⁰See alphanumeric coordinates in Fig. 7

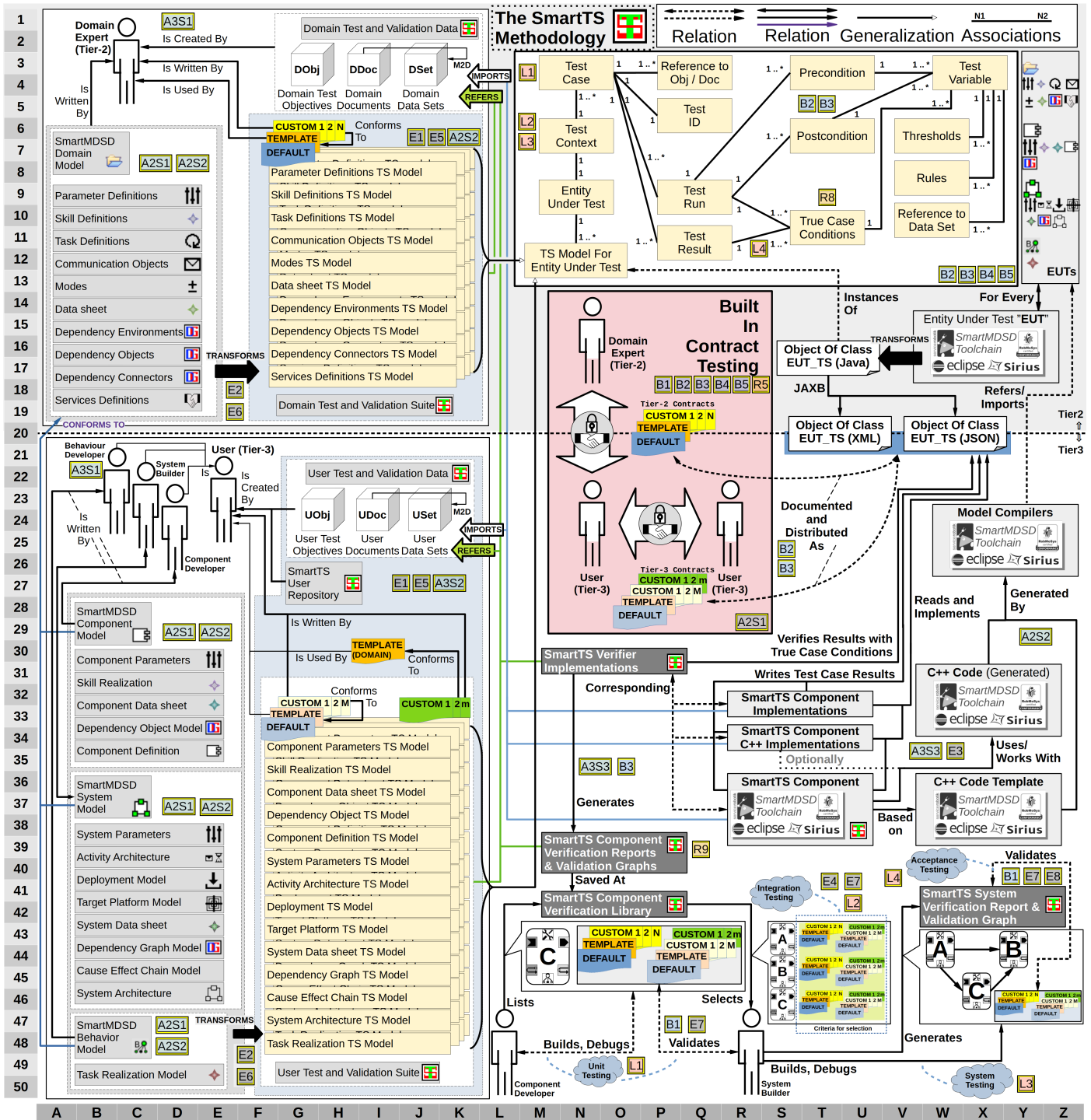


Fig. 7. The SmartTS Methodology.

Tier-2 SmartTS models (7F-18J)¹⁰ are packaged together to form the domain test and validation suite (19G¹⁰). A default (6G¹⁰) domain test and validation suite package is transformed (17F¹⁰) from the existing SmartMDS domain model package. For every model in the SmartMDS domain model package, a corresponding SmartTS model is generated. The transformation also produces a template (6G¹⁰) to write any number of custom (6G¹⁰) domain test and validation

suite packages. These custom packages are written by domain experts using the template (6G¹⁰→4F¹⁰→2C)¹⁰. Similarly, for tier-3 SmartTS models (34F-48J)¹⁰ in user test and validation suite (49G¹⁰), the default, template and custom (33G¹⁰) user test and validation suite packages are transformed (48F¹⁰) and written (33G¹⁰→26F¹⁰→22E)¹⁰. Additionally, a tier-3 user (22E¹⁰) can write a custom user package (33T¹⁰) that conforms to any template from a domain test and validation suite template

(30I↔6G)¹⁰.

Once written and locked (20N↔6G, 28P↔33G, 28P↔33T)¹⁰, the domain/user test and validation suite packages (6G, 33G, 33T)¹⁰ are provided to tier-3 users (25N, 25R)¹⁰ and establishes the foundation for built-in contract testing (14M-29R)¹⁰ in the ecosystem. Domain test and validation suite packages (20P↔6G)¹⁰ govern contract testing between tier-2 domain experts (15N¹⁰) and tier-3 ecosystem users (25N¹⁰) as conforming parties. Similarly, user test and validation suite packages (28P↔(33G, 33T))¹⁰ that govern contract testing amongst (25N↔25P↔25R)¹⁰ any number of different tier-3 ecosystem users (25N, 25R)¹⁰.

These tier-3 contracts (28P¹⁰) can be written for claims made by a component developer, requirements made by a system builder, mutually agreed upon behaviour or any other contractual requirement that any of the models from a SmartMDS model package should adhere to. The tier-3 contracts (28P¹⁰) in built-in contract testing could also be shared between different component developers as standard tests that all components of a particular kind should pass or between system builders as standard tests for quality assurance or enforcement-related requirements. Standard tests for domain requirements, quality assurance and enforcement can be distributed as tier-2 contracts (20P¹⁰) between domain experts (15N¹⁰) and ecosystem users (25N¹⁰).

A SmartTS model (7F-18J, 34F-48J)¹⁰ is required to define sets of data to be used in tests. This data is modelled as domain/user test and validation data (1G, 22G)¹⁰ in SmartMDS domain repository (1A-19K)¹⁰ or SmartTS user repository (21F-50K)¹⁰. SmartTS test and validation data package contains documentation for uniquely identifiable test objectives (3G, 24G)¹⁰ and documents (3I, 24I)¹⁰. Domain/user data sets (3J, 24J)¹⁰ are transformed from respective documents (3I→3K→3J, 24I→24K→24J)¹⁰. These model to data transformations (M2D: 3K, 24K)¹⁰ compile the data range mentioned in the documents to create the data for uniquely identifiable data sets mentioned in the documents. SmartTS models (7F-18J, 34F-48J)¹⁰ refers (12L→5K, 41L→(25K, 5K))¹⁰ to these data sets, which are later imported ((5K, 25K)→(33S,34S,38S))¹⁰ during test execution.

Every SmartTS model (7F-18J, 34F-34J)¹⁰ for an entity under test ((3Y-13Z)↔(9A-19D, 30A-35D, 38A-46D, 50C))¹⁰ is a generalization (12L→12N, 41L→12N)¹⁰ for a common SmartTS model class (12N¹⁰). The SmartTS model class for an entity under test is structured (3L-13Y)¹⁰ to incorporate context (9N, 6N, 3P)¹⁰, runs (3N, 5P, 9P, 11P, 10T)¹⁰ and conditions (3T, 6T, 3X, 6V, 8V, 10V)¹⁰ for tests. For every entity under test (15Y¹⁰) from the SmartMDS model package (17X¹⁰) is transformed to Java instances of SmartTS model class (17V, 17T)¹⁰ and documented and distributed (24T¹⁰) as XML and JSON documents (17T→18T→(20T, 20V))¹⁰ to work as contracts (20P, 28P)¹⁰ for built-in contract testing (14M-29R)¹⁰.

The SmartMDS compilers (24X¹⁰) generate background C++ code (33X¹⁰) and a template (38X¹⁰) based on which the component developer can create the SmartTS component (38S¹⁰). The SmartTS component works with the background C++ code (38S→35X→33X)¹⁰ and implements the contracts (38S→27V→20W→(20T, 20V))¹⁰. It is also possible to im-

plement components created without the use of SmartMDS toolchain (33S, 34S)¹⁰, and yet adhere to the SmartTS contracts ((33S, 34S)→27V→20W→(20T, 20V))¹⁰. Any component implementing SmartTS contracts imports domain/user data sets when they are referred to in the SmartTS model ((33S, 34S, 38S)→(5K, 25K))¹⁰. A SmartTS verifier implementation (30N¹⁰) corresponding to (33P¹⁰) the SmartTS component implementation verifies the results generated by SmartTS components against the contracts (30N→30V→20V→(20T, 20V))¹⁰ and generates SmartTS component verification reports (model, text) and validation graphs (graphics) (39N, 33P→37N→39N→42N)¹⁰.

These verification reports and validation graphs are listed in the SmartTS component verification library (39N→42N, 48L→46L→44M→42N)¹⁰ after they are used by component developers for debugging (unit-testing: 50N)¹⁰ the component (48L→48N→46P)¹⁰. System builders (48R)¹⁰ selects (48R→46R→44S→42N)¹⁰ and validates (48R→48Q→46P)¹⁰ components, and builds a system (44W, 48R→50X→44W)¹⁰ after performing integration (41S¹⁰) and system (50X¹⁰) testing. System builder also generates system verification report (model, text) and validation graphs (graphics) for the newly composed system (48R→48V→42W)¹⁰. System contracts (44V¹⁰) are a collection of contracts that exist between tier-2 and tier-3 parties associated with the system. System contracts are validated against system verification report and validation graphs during acceptance testing (42W→40Y→44Y)¹⁰.

The SmartTS platform-independent model (A-K)¹⁰, based on the SmartTS meta-model (1L-20Z)¹⁰ is thus condensed into shareable documents (20T, 20V)¹⁰ to be received and later implemented (20L-40Z)¹⁰ and tested (41L-50Z)¹⁰ for specific platforms. Fig. 8 shows key users, models, transformations in the SmartTS workflow as explained in Sections II and III.

IV. SMARTTS IN THE CONTEXT OF ROBMO SYS

RobMoSys: Composable models and software for robotics [40], [41] is an EU H2020 funded project (2017-2020, Grant number 732410) to create better modelling standards and tooling for robotic systems. RobMoSys has a three-tier ecosystem for model-driven, component-based software development for robotic systems (RobMoSys: Wiki [41]). Fig. 9 shows the three tiers of the RobMoSys ecosystem and the roles that participate in these tiers. Members of a lower-tier conform to models defined by members of a higher-tier in the ecosystem. SmartMDS toolchain [50], [51] is a RobMoSys conformant toolchain that enables ecosystem users to share components and compose systems that are according to the principles dictated by RobMoSys. SmartTS is an addition to SmartMDS tooling and provides a model-based methodology for software testing in the RobMoSys ecosystem.

V. SMARTTS IN ACTION

Fig. 10 shows some of the key features of SmartTS acting along with the SmartMDS toolchain. Tier-2 domain experts and Tier-3 users transform SmartMDS models to contracts (Fig. 10.a) and documents (Fig. 10.b). SmartTS documents are transformed into data sets (Fig. 10.c) which are referred to in

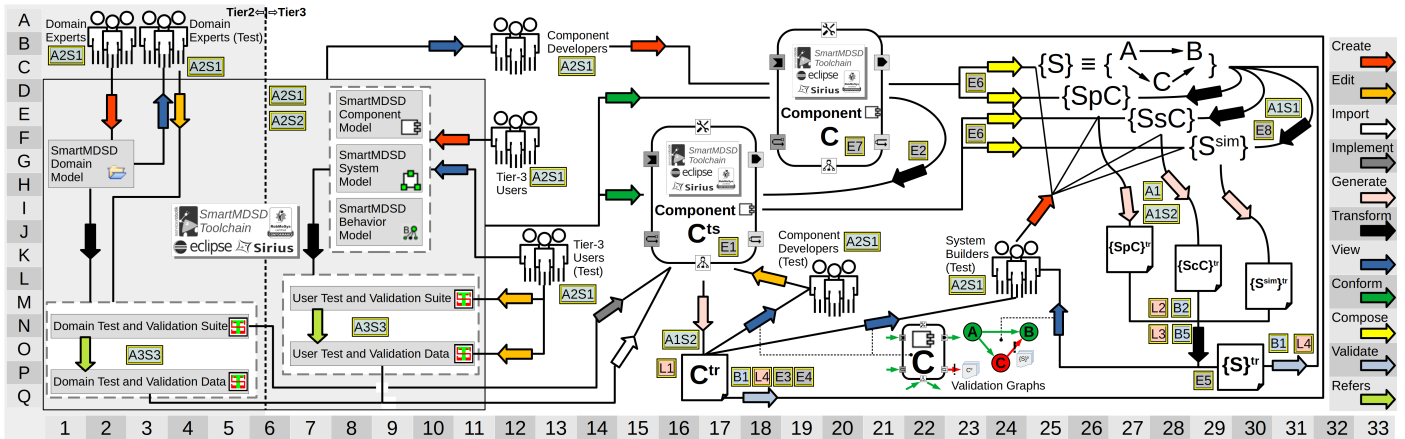


Fig. 8. Key users, Models, Transformations in the SmartMDS::SmartTS Workflow.

SmartTS contracts. Component models are transformed to their SmartTS tester component (Fig. 10.(d,e,f)) which is used to test (Fig. 10.g) or simulate (Fig. 10.h) the component. **SmartTS tooling as it stands today is functionally complete for the workflow described in Fig. 8.** Automation and visualization of some workflow elements (e.g. validation graphs) is planned to further improve user experience. **Eclipse features and plug-ins for SmartTS are available for download [56].** Context and video tutorials on the use of SmartTS will soon be available online at the SRRC wiki web page [57].

VI. CONCLUSIONS AND FUTURE WORKS

Validating the behaviour of commercial off-the-shelf components and system interactions is enhanced by the availability of empirically verifiable test records consistent with a component's claims. The **trustworthiness of a system is derived from the trustworthiness of its constituents.** Test and validation suite for a system can be built using model-driven test and validation suites of its components. In this paper, we presented the **“SmartTS methodology: A component-based and model-driven approach to generate model-bound test-suites for software components and systems”.** The test-suites in SmartTS are tightly bound to an application domain's data and service models as defined in RobMoSys (EU H2020 project) compliant SmartMDS toolchain. SmartTS does not break component encapsulation for system builders while providing

them complete access to the way that component is tested and simulated. At present, the SmartTS functionality is partially consolidated in the SmartMDS toolchain (E1-E8). Plans to automate remaining SmartTS transformations are marked for incorporation in future releases of the SmartMDS toolchain as SmartDBE (Smart digital business ecosystem) features and plug-ins [56].

ACKNOWLEDGMENT

This paper is supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) in the programme “Development of Digital Technologies (PAiCE)” under grant agreement No. 01MA17003D, project *SeRoNet - Eine Plattform zur arbeitsteiligen Entwicklung von Serviceroboter-Lösungen* [59] and by the European Union's Horizon 2020 research and innovation programme under grant agreement No 732410, project *RobMoSys: Composable Models and Software for Robotics* [40]

REFERENCES

- [1] B. W. Boehm, J. R. Brown, and M. Lipow, “Quantitative evaluation of software quality,” in *Proceedings of the 2nd International Conference on Software Engineering*, 1976, pp. 592–605.
- [2] B. Kitchenham and S. L. Pfleeger, “Software quality: the elusive target [special issues section],” *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.
- [3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.
- [4] K. E. Wiegers, *Creating a software engineering culture*. Pearson Education, 1996.
- [5] G. J. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing : Module (Unit) Testing*. John Wiley and Sons, Ltd, 2012, ch. 5, pp. 85–111. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119202486.ch5>
- [6] —, *The Art of Software Testing: Higher-Order Testing*. John Wiley and Sons, Ltd, 2012, ch. 6, pp. 113–142. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119202486.ch6>
- [7] —, *The Art of Software Testing: Usability (User) Testing*. John Wiley and Sons, Ltd, 2012, ch. 7, pp. 143–155. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119202486.ch7>
- [8] R. W. Gray, “Automation versus Manual Testing,” in *SAE Technical Paper*. SAE International, 02 1966. [Online]. Available: <https://doi.org/10.4271/660694>
- [9] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, “On the Automated Generation of Program Test Data,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 293–300, 1976.

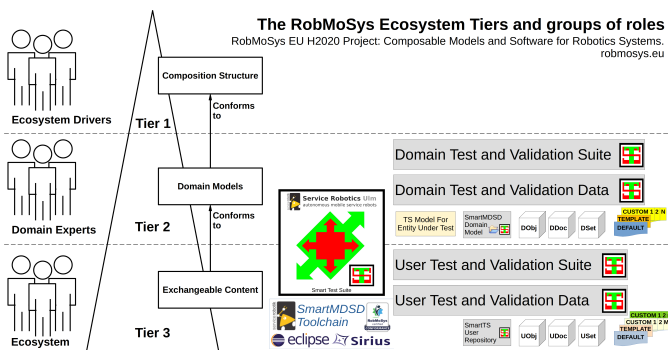
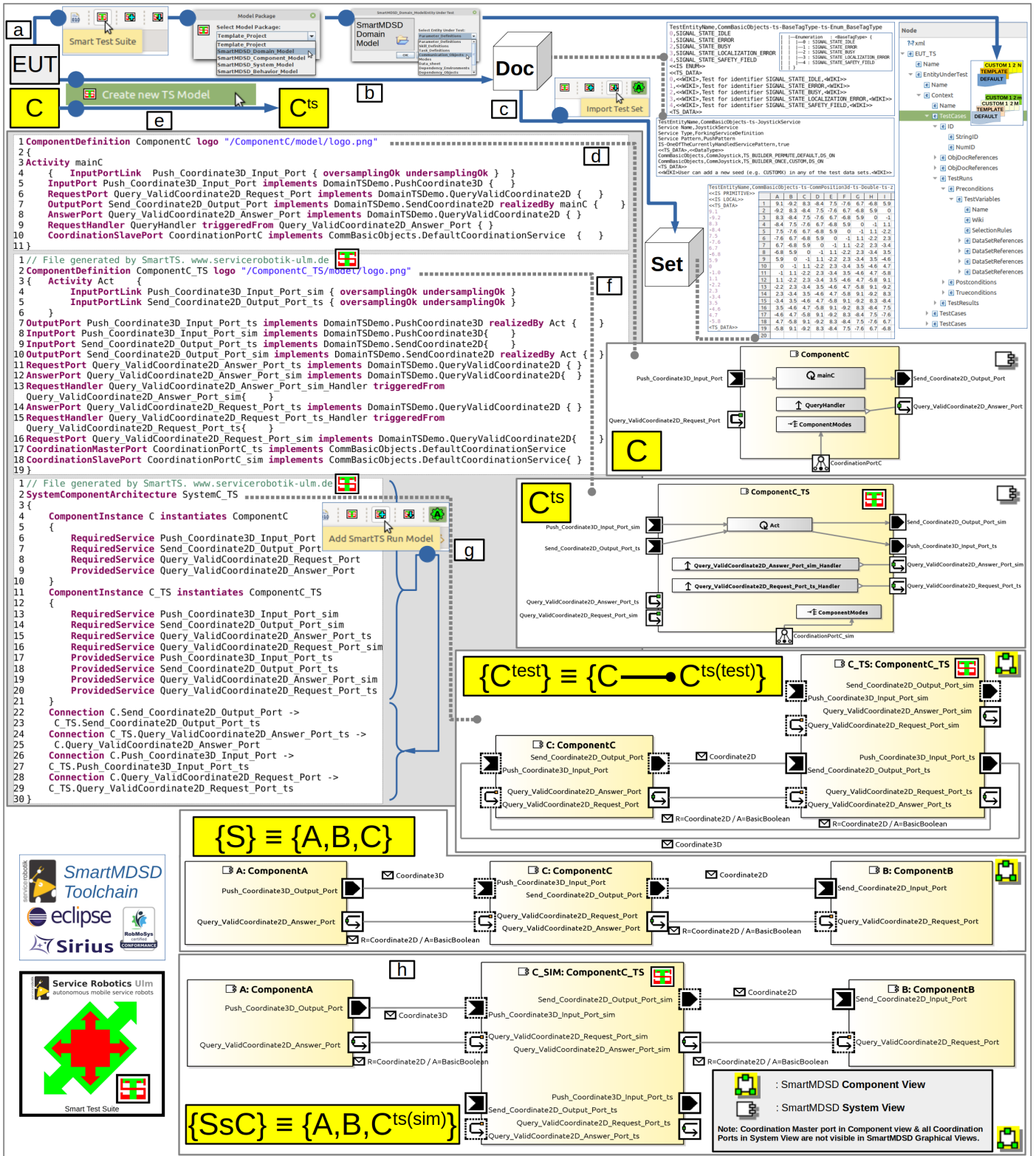


Fig. 9. SmartTS in RobMoSys Ecosystem.



a. Entity under test (EUT) is transformed to domain/user contracts in form of a SmartTS model. EUT is any SmartMDS model from domain, component, system or behaviour model package.
 b. Entity under test (EUT) is transformed to domain/user documents in SmartTS domain/user repository (model to text transformation).
 c. Domain/user documents in SmartTS domain/user repository is transformed to corresponding domain/user data sets in domain/user repository (model to data transformation).
 d. SmartMDS component-definition model for component named C.
 e. SmartMDS component-definition model for component named C is transformed to its SmartTS tester component named C_TS (model to model transformation).
 f. SmartMDS component-definition model for SmartTS tester component named C_TS for component C.
 g. SmartMDS System-component-architecture model for SystemC_TS (test) with one instance each of components C and C_TS. Connections are automatically made between components.
 h. SmartMDS System-component-architecture model for SystemSSC with component C_SIM simulating component C.

Fig. 10. SmartTS in Action. Creation of a SmartTS Contract for a Domain Communication Object and its Integration with the SmartTS Test Component.

- [10] J. Chacón Montero, A. Jimenez Ramirez, and J. Gonzalez Enriquez, "Towards a Method for Automated Testing in Robotic Process Automation Projects," in *2019 IEEE ACM 14th International Workshop on Automation of Software Test (AST)*, 2019, pp. 42–47.
- [11] H. Li, *Communications for control in cyber physical systems: theory, design and applications in smart grids*. Morgan Kaufmann, 2016.
- [12] L. Esterle and R. Grosu, "Cyber-physical systems: challenge of the 21st century," *E & I Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 299–303, 2016.
- [13] M. A. S. Brito, S. R. S. Souza, and P. S. L. Souza, "Integration testing for robotic systems," *Software Quality Journal*, Nov 2020. [Online]. Available: <https://doi.org/10.1007/s11219-020-09535-w>
- [14] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," in *2018 IEEE ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 276–285.
- [15] D. Marijan, A. Gotlieb, and M. K. Ahuja, "Challenges of testing machine learning based systems," in *2019 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2019, pp. 101–102.
- [16] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A Study on Challenges of Testing Robotic Systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 96–107.
- [17] R. Pietrantuono and S. Russo, "Robotics software engineering and certification: Issues and challenges," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 308–312.
- [18] T. Bretl and S. Lall, "Testing static equilibrium for legged robots," *IEEE Transactions on Robotics*, vol. 24, no. 4, pp. 794–807, 2008.
- [19] S. Sheng and N. Becker, "Challenges in standardizing ram testing for small unmanned robotic systems," in *2013 Proceedings Annual Reliability and Maintainability Symposium (RAMS)*. IEEE, 2013, pp. 1–6.
- [20] M. Farrell, R. C. Cardoso, L. A. Dennis, C. Dixon, M. Fisher, G. Kourtis, A. Lisitsa, M. Luckcuck, and M. Webster, "Modular verification of autonomous space robotics," *arXiv preprint arXiv:1908.10738*, 2019.
- [21] M. Mossige, A. Gotlieb, and H. Meling, "Testing robot controllers using constraint programming and continuous integration," *Information and Software Technology*, vol. 57, pp. 169–185, 2015.
- [22] Y. K. Chung and S.-M. Hwang, "Software testing for intelligent robots," in *2007 International Conference on Control, Automation and Systems*. IEEE, 2007, pp. 2344–2349.
- [23] A. Paikan, S. Traversaro, F. Nori, and L. Natale, "A generic testing framework for test driven development of robotic systems," in *International Workshop on Modelling and Simulation for Autonomous Systems*. Springer, 2015, pp. 216–225.
- [24] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 331–342.
- [25] V. Mayoral-Vilches, N. García-Maestro, M. Towers, and E. Gil-Uriarte, "Devsecops in robotics," *arXiv preprint arXiv:2003.10402*, 2020.
- [26] A. Afzal, D. S. Katz, C. L. Goues, and C. S. Timperley, "A study on the challenges of using robotics simulators for testing," *arXiv preprint arXiv:2004.07368*, 2020.
- [27] J. Wienke, D. Wigand, N. Koster, and S. Wrede, "Model-Based Performance Testing for Robotics Software Components," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*, 2018, pp. 25–32.
- [28] B. Konemann, G. Zwiehoff, and J. Mucha, "Built-in test for complex digital integrated circuits," *IEEE Journal of Solid-State Circuits*, vol. 15, no. 3, pp. 315–319, 1980.
- [29] R. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.
- [30] Y. Wang, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance," in *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*, 1999, pp. 186–189.
- [31] H.-G. Gross, I. Schieferdecker, and G. Din, "Model-based built-in tests," *Electr. Notes Theor. Comput. Sci.*, vol. 111, pp. 161–182, 01 2005.
- [32] M. Sitariman and B. Weide, "Component-based software using RESOLVE," *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 21–22, 10 1994.
- [33] H.-G. Gross, *Component-Based Software Testing with UML*. Springer-Verlag Berlin Heidelberg, 2005.
- [34] P. Feiler, D. Gluch, and J. Hudak, "The Architecture Analysis and Design Language (AADL): An Introduction," p. 145, 02 2006.
- [35] MARTE, "A UML Profile for MARTE: modeling and analysis of real-time embedded systems." <http://www.omg.org/spec/MARTE/1.1/>, 2011, accessed: 2021-03-15. [Online]. Available: <http://www.omg.org/spec/MARTE/1.1/>
- [36] OMG SysML, "SysML: OMG Systems Modeling Language." <https://www.omg.org/>, 2012, accessed: 2021-03-15. [Online]. Available: <https://www.omg.org/>
- [37] Amalthea, "Amalthea: An open platform project for embedded multicore systems." <http://www.amalthea-project.org>, 2013, accessed: 2021-03-15. [Online]. Available: <http://www.amalthea-project.org>
- [38] SPARC, "Robotics 2020 Multi-Annual Roadmap: SPARC Partnership for Robotics in Europe, February 2015, call 2 ICT24 – Horizon 2020." <http://www.eu-robotics.net/cms/upload/Multi-AnnualRoadmap2020ICT-24RevBfull.pdf>, 2015, accessed: 2021-03-15. [Online]. Available: <http://www.eu-robotics.net/cms/upload/Multi-AnnualRoadmap2020ICT-24RevBfull.pdf>
- [39] —, "SPARC: European SPARC Robotics Initiative." <http://sparc-robotics.eu/>, 2015, accessed: 2021-03-15. [Online]. Available: <http://sparc-robotics.eu/>
- [40] RobMoSys, "RobMoSys EU H2020 Project (2017-2020): Composable models and software for robotics systems - towards an eu digital industrial platform for robotics." <http://robmosys.eu>, 2017-2020, accessed: 2021-03-15. [Online]. Available: <http://robmosys.eu>
- [41] RobMoSysWiki, "RobMoSys EU H2020 Project Wiki," <http://robmosys.eu/wiki>, 2017-2020, accessed: 2021-03-15. [Online]. Available: <http://robmosys.eu/wiki>
- [42] W. L. Hürsch and C. V. Lopes, "Separation of Concerns," College of Computer Science, Northeastern University, Tech. Rep., 1995.
- [43] V. Kulkarni and S. Reddy, "Separation of concerns in model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 64–69, 2003.
- [44] D. Alonso, C. Vicente-Chicote, F. Ortiz, J.-A. Pastor Franco, and B. Alvarez-Torres, "V3CMM: A 3-view component meta-model for model-driven robotic software development," vol. 1, 01 2010.
- [45] S. Dhoubi, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *SIMPAR*, 2012.
- [46] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS component model: a model-based development paradigm for complex robotics software systems," in *SAC '13*, 2013.
- [47] V. Nagrath, "Software Architectures for Cloud Robotics : The 5 View Hyperactive Transaction Meta-Model (HTM5)," Ph.D. dissertation, 2015, thèse de doctorat dirigée par Mériaudeau, Fabrice et Morel, Olivier Informatique Dijon 2015. [Online]. Available: <http://www.theses.fr/2015DIJOS005>
- [48] A. Lotz, J. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, and C. Schlegel, "Towards a Stepwise Variability Management Process for Complex Systems – A Robotics Perspective," *Int. Journal of Information System Modeling and Design (IJISMD)*, vol. 5, pp. 55–74, 11 2014.
- [49] euRobotics, "EU Robotics Agenda 2020: EU Robotics Strategic Research, Innovation and Deployment Agenda 2020 on AI, Data and Robotics Partnership," 2020, accessed: 2021-04-25. [Online]. Available: <https://www.eu-robotics.net/cms/upload/downloads/ppp-documents/AI-Data-Robotics-Partnership-SRIDA-V3.0.pdf>
- [50] C. Schlegel, "Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach," Ph.D. dissertation, University of Ulm, 2004.
- [51] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel, "The SmartMDS Toolchain: An Integrated MDS Workflow and Integrated Develop-

ment Environment (IDE) for Robotics Software,” *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, pp. 3–19, 08 2016.

- [52] SRRC, “SmartMDS Services,” 2021, accessed: 2021-04-30. [Online]. Available: <https://robmosys.eu/wiki/modeling:metamodels:service>
- [53] —, “SmartMDS Communication Objects,” 2021, accessed: 2021-04-30. [Online]. Available: <https://robmosys.eu/wiki/modeling:metamodels:commobject>
- [54] —, “SmartMDS Communication Patterns,” 2021, accessed: 2021-04-30. [Online]. Available: <https://robmosys.eu/wiki/modeling:metamodels:commpattern>
- [55] C. Schlegel, “SMARTSOFT: Components and Toolchain for Robotics.” <http://www.servicerobotik-ulm.de/>, 2011, accessed: 2021-03-15. [Online]. Available: <http://www.servicerobotik-ulm.de/>
- [56] V. Nagrath, “SmartDBE Git: Eclipse features and plugins pertaining to SmartDBE additions to SmartMDS Toolchain.” <https://github.com/ServiceRobotics-Ulm/SmartDBE>, 2021, accessed: 2021-05-08. [Online]. Available: <https://github.com/ServiceRobotics-Ulm/SmartDBE>
- [57] SRRC, “Service Robotics Research Center Wiki,” 2021, accessed: 2021-04-30. [Online]. Available: <https://wiki.servicerobotik-ulm.de/tutorials:start>
- [58] —, “SmartMDS Collection of Applications, Software Components, (Domain) Models,” 2021, accessed: 2021-04-30. [Online]. Available: <https://wiki.servicerobotik-ulm.de/directory:collection>
- [59] SeRoNet, “SeRoNet: Eine Plattform zur arbeitsteiligen Entwicklung von Serviceroboter-Lösungen.” <https://www.seronet-projekt.de/>, 2017-2021, accessed: 2021-03-15. [Online]. Available: <https://www.seronet-projekt.de/>

AUTHORS’ PROFILE



Dr. Vineet Nagrath (M) is a researcher at the Service Robotics Research Centre, Ulm University of Applied Sciences (Technische Hochschule Ulm) in GERMANY. He received his dual doctoral degrees in 2015 from University de Bourgogne, FRANCE, and Universiti Teknologi Petronas, MALAYSIA in a Binational dual degree doctoral program (Software Architectures for Cloud Robotics - The 5 View Hyperactive Transaction Meta-Model for Agent-Oriented Cloud Robotic systems). He has earlier completed STIC MSc. in Computer Vision (MCV)(M1) and Erasmus Mundus Masters in Vision and Robotics (M2) from University de Bourgogne, FRANCE. His research focus is on Cloud-Connected Robots, Multi-Agent Systems, Model-Driven Software Engineering, and Machine Learning. Since 2017 he is contributing to the SmartMDS Toolchain for BMWi PAiCE SeRoNet and EU H2020 RobMoSys projects.

vineetnagrath.com



Prof. Dr. rer. nat. Christian Schlegel (M) is Professor for Real-Time and Autonomous Systems in the Computer Science Department of Technische Hochschule Ulm since 2004. He is Head of the Service Robotics Research Group. His research interests are in model-driven software and systems engineering for real-world cognitive systems such as service robotic and their application in industry 4.0. He is the technical lead of the EU H2020 RobMoSys project and the elected coordinator of the euRobotics Topic Group on Software Engineering,

System Integration, System Engineering. [servicerobotik-ulm.de](http://www.servicerobotik-ulm.de)

<http://www.servicerobotik-ulm.de/>