Combining Techniques to Verify Service-based Components

Pascal André[1], Christian Attiogbé[1] and Jean-Marie Mottu[1]
[1]*AeLoS Team LINA CNRS UMR 6241 - University of Nantes, France*
{firstname.lastname}@univ-nantes.fr

Keywords: Component; Service; Model-driven Development; Contract; Proof; Model-Checking; Test

Abstract: Early verification is essential in model-driven development because late error detection involves a costly correction and approval process. Modelling real life systems covers three aspects of a system (structure, dynamics and functions) and one verification technique is not sufficient to check the properties related to these aspects. Considering Service-based Component Models, we propose a unifying schema called multi-level contracts that enables a combination of verification techniques (model checking, theorem proving and model testing) to cover the V&V requirements. This proposal is illustrated using the Kmelia language and its COSTO tool.

1 Introduction

The model-driven development (MDD) strength resides in delaying the implementation concerns to focus more on the abstract models which decrease the system specification complexity. Since (abstract) Platform Independent Models (PIM) are the starting points for MDD, we need to trust them. Late error detection involving a costly correction (and approval) process, an early verification activity is essential.

Despite the details of Platform Specific Models (PSM) are omitted, the complexity of verification and validation (V&V) remains important when the PIM elements cover the three usual aspects of a system (structural, dynamic and functional). Accordingly one verification technique does not suffice to check the properties related to these different aspects.

We address the issue of verifying multi-aspect models from the practitioner's point of view. We consider Service-based Component (SbC) Models (Crnkovic and Larsson, 2002; Beek et al., 2006) that promote the (re)use of components and services coming from third party developers to build new systems. The success of the large-scale development of SbC depends on the correctness of the parts before assembling them. SbC software systems are designed at different specification levels: *service interactions* (messages exchange), *functional contracts* (pre/post conditions), service *behaviour* (labelled transition systems (LTS) with computation statements). Establishing their correctness is complex and requires the use of various verification techniques.

We propose a method based on multi-level contracts where the properties are classified before being verified with the combination of appropriate techniques. Contracts act as a glue between the structure levels, properties and V&V techniques. Classifying the properties enables us to select the adequate technique to cover the V&V requirements; *model checking, theorem proving, model testing*. The interaction properties are verified using model checking; the consistency properties are checked using theorem proving and the behaviour conformance against the functional contract is checked using a specific model testing technique. We assume a modelling language which is formal enough to specify SbC elements and contracts. We experiment this method on an embedded system using the Kmelia modelling language and its associated COSTO toolbox (André et al., 2010).

Applying the proposed method increases confidence in the SbC models early in the development process: they are correct and embed passed test where contracts are reinforced. Thus the method helps to obtain correct software as soon as possible and allows one to apply thereafter advanced development techniques such as *agile* ones (thanks to the qualified test cases and data we constructed) or *Design-by-Contract* techniques (thanks to the used contracts).

In the remaining of the article, we sketch the Service-based Component model in Section 2; multi-level contracts are introduced in Section 3. Section 4 describes the combination of V&V techniques. We illustrate the proposed method and framework with the Kmelia/COSTO toolbox in Section 5. Section 6 discusses related works and we conclude in Section 7.

2 Service-based Component Models

In Service-based Component (SbC) models, a functionality is implemented by the services provided by some components. Provided services are not necessarily atomic calls and may possess a complex behaviour, in which other services might be needed (called). These needs are either satisfied internally by other services of the same component, or specified as required services in the component's interface. The required services can then be bound to provided services from other components, which might also require others, and so on. A provided service needs all its direct and indirect dependencies satisfied in order to be available for use. Modelling languages, such as UML2, AADL, rCOS or Sofa (Rausch et al., 2008), can be used to specify SbC systems. As a case study, we model a simplified version of a platoon of vehicles using the SCA notation (OSOA, 2007).

Figure 1 shows a small architecture composed of a *driver* and two *vehicle* components. Each component has a configuration service conf (used when instantiating the component), a main service run to activate the vehicle behaviour and services to give their position and speed. The computeSpeed service reads the vehicle's state and the run and conf services assign values to the vehicle's state. Auxiliary services like stop which interrupts a vehicle, have been omitted for simplicity. We extend the SCA notation to make explicit the component's state (its variables) and the service calling, reading, and writing.

The service run of Vehicle calls computeSpeed which requires pilotspeed and pilotpos services. We consider only the speed and the position (X axis only) of the vehicles. The vehicles are designed to follow their predecessor (which they consider to be their pilot) except the first one which follows a component taking the role of the driver. The driver is assumed to be a special kind of vehicle that controls its own values according to a target position. Each running vehicle can compute its own speed by considering its current speed and position, its predecessor's position and speed and a safety distance with its predecessor.

3 Multi-level Contracts

According to (Meyer, 2003), *a Trusted Component is a reusable software element possessing specified and guaranteed property qualities*. The notion of contract is helpful to model various kind of correctness properties. But it should be made precise and extended to cope with the expressiveness of the SbC models. The properties, *e.g. interoperability*, are classified at different *requirement level (RL)*:

1. *Static*: the compatibility of interface signatures (names and types); does a component give enough information about its interface(s) in order to be (re)usable by others?
2. *Architectural*: the availability of the required components and services, the correctness of the linked component interfaces;
3. *Functional*: do the components do what they must do? These correctness properties may be checked both on each component and on the component assemblies and compositions.
4. *Behavioural*: the correct interaction between two or more components which are combined. The properties depends on the interaction model features: sequential vs. concurrent, call vs. synchronisations, synchronous vs asynchronous, pair vs. multipart communication, shared data, atomic/structured actions...
5. *Quality of service*: the non-functional requirements (time, size...) are fulfilled. Note that this level will not be detailed in this paper.

A **multi-level contract** (Messabihi et al., 2010) is a contract defined at different SbC *structure levels (SL)* (service, component, assembly, composition) according to different expected *requirement levels*. This vision of contracts provides a convenient framework to master both the incremental construction of SbC and the verification of multi-aspect properties by combined techniques. Table 1 summarises the crossing of the structure levels properties with the requirement levels (RL).

RL	Structure level			
	service	**component**	**assembly**	**composite**
1	type checking	type checking	service signature compatibility (ssic)	ssic
2	well-formedness	service accessibility	service structure consistency (sstc)	sstc
3	functional correctness	component consistency	service compliance (sco)	sco
4	behavioural consistency	protocol correctness	behavioural compatibility (bhc)	bhc

Table 1: Multi-level Contracts and Properties

Multi-level contracts are useful to define interoperability levels between different SbC languages. *e.g.* a Corba component with IDL interfaces can be compatible with components defined with other SbC models at the first level only. We detail now the main properties of each structure level.

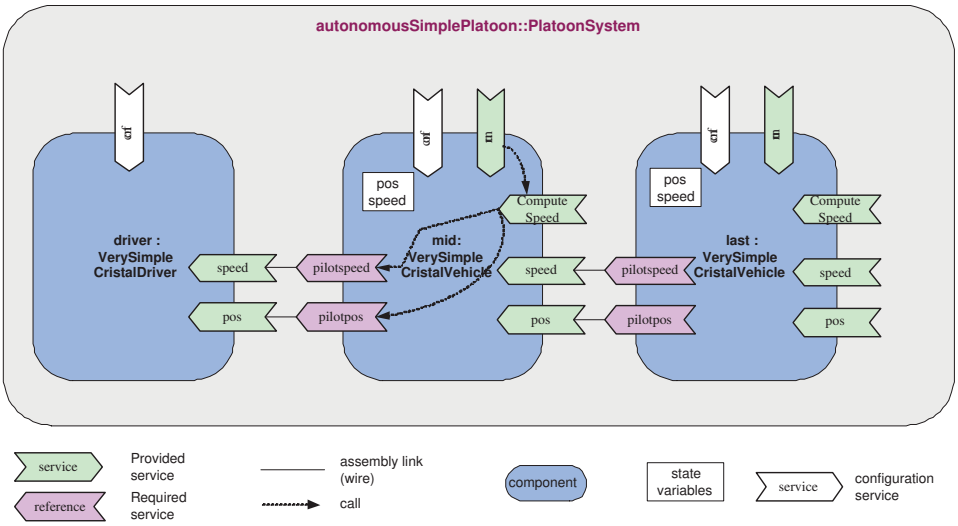

Figure 1: Component model of the Platoon system

Service contract It expresses that the service terminates in a consistent state. This contract deals mainly with two properties.

- The *behavioural consistency* property states that the execution of the service actions does not lead to inconsistent states (such as deadlock).
- The *functional correctness* property expresses that a service achieves what it is supposed to do. The functional correctness of a service of a component is defined using the Hoare-style specification (Pre-condition, Statement, Post-condition) where Statement is the service behaviour. This property should be checked with respect to the requirements of the owner component.

Component contract The component is confidently reusable. It is ensured with three main properties.

- The *service accessibility* property states that the services defined in the interface of a component are available. This is related to intra-component traceability of service dependency.
- The *component consistency* property states that the invariant properties of the component are preserved by all the services embodied in the component. Considering that a component equipped with services is *consistent* if its properties are always satisfied whatever the behaviour of the services is, one can set a consistency preservation contract between the services and their owner component to ensure that property.
- The *protocol correctness* property expresses that the order in which the services are to be invoked by clients is correct with respect to the rules given by the services' specification. A component protocol is defined here as the set of all the valid sequences of service invocations.

Assembly contract In an assembly, made of linked trusted components, each component will contribute to the well-formedness of the links by requiring or ensuring appropriate assertions: this is the coarse-grained contract. The link establishes a client/supplier relation. The assembly contract covers correctness properties with four requirement levels:

- The first level deals with *service signature compatibility* among the services of the interfaces of the assembled components. The service call should respect the service signature. The signature matching between the involved services of component interfaces covers at least name resolution, visibility rules, typing and subtyping rules.
- The second level deals with *service structure consistency* of the assembled components. Assuming that services can be composed from other (sub)services, connecting services is possible only if their structures are compatible (but not necessary identical).
- The third level deals with *service compliance* of assembled components. If the services use a Hoare-like specification, post-conditions relate to their pre-conditions (Zaremski and Wing, 1997). The caller pre-condition is stronger than the called one. The called post-condition is stronger than the caller's one. Each part involved in the assembly should fulfil its counterpart of the contract.
- The fourth level deals with *behavioural compatibility* between the linked services of the assembled components. It ensures the correct interaction between two or more components which are combined through their services.

Composite contract It is similar, up to specific expressions, to the one of assemblies.

4 Combining V&V Techniques

Modelling and V&V are mutually dependent during design. As depicted in Figure 2, multi-level contracts are set during the specification activities and checked during the formal analysis activities. The structure levels are represented here by columns. The design workflow is presented as a whole but the activities can be performed iteratively in any order.

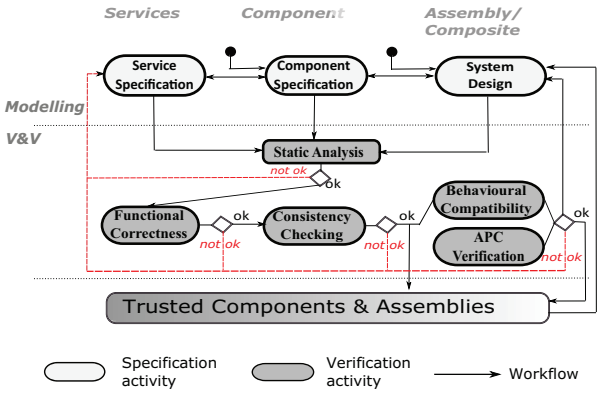

Figure 2: Integrated process for design verification

From a practical point of view, the specifier would switch from one activity to another according to a customised methodology, inspired from top-down or bottom-up approaches, with a component or system orientation. For example the specifier may work only at the service and component levels (the left part of Figure 2) to deliver *off the shelf* components.

Modelling: making contracts explicit Modelling includes three activities: software system design (assembly/composition), software component specification and service specification. In a top-down approach, the *system design* activity starts first. It defines the system as a collection of interacting subsystems and components. If components or assemblies that match the requirements already exist on the shelf, they can be directly integrated in the system design. Otherwise, the *component specification* activity will produce the new component(s). Once the component structure is established, the detailed *service specification* activity proceeds. The main concern is that the contracts must be explicitly written at each level in order to be checked.

V&V : checking contract properties The models produced during the specification are analysed by checking the contracts properties. The verification process iterates on five V&V activities as depicted in Figure 2, each activity refers to the contracts properties of Section 3.

1. The *Static Analysis* ***(SA)*** activity checks the syntactic correctness at all levels, the service accessibility of the component level, and the *static interoperability* of the assembly level, which itself covers the service signature compatibility and the service structure consistency.
2. The *Functional Correctness* ***(FC)*** activity checks the *behavioural consistency* property at the service level and a part of the *protocol correctness* property at the component level.
3. The *Consistency Checking* ***(CC)*** activity covers the *component consistency* property at the component level.
4. The *Behavioural Compatibility* ***(BC)*** activity checks the *behavioural consistency* property at the service level, a part of the *protocol correctness* property at the component level and the *behavioural compatibility* at the assembly level.
5. The *Assembly/Promotion contracts* ***(APC)*** verification activity checks the *service compliance* of the assembled components at the assembly level and the composite level.

Table 2 overviews how each technique contributes to a verification activity of multi-level contracts.

	Static Analysis	**Theorem Proving**	**Model Checking**	**Model Testing**
SA	types structures			
FC	See details in Section 4.4			assertions oracle
CC		assertions invariant		
BC			deadlock liveness	
APC		refinement aggregation		

Table 2: Multi-level contracts and verification techniques

Next sections provide insights on these techniques.

4.1 Structural Correctness by Static Analysis

The static analysis checks the structural correctness of models. It includes the syntax analysis, the type checking and the verification of well-formedness rules (WFR). For example, the service dependency satisfaction WFR states: *to be executable, all the services called (directly or indirectly) by a service must be available.* The checking algorithm of verification

is specified here using the Z notation (Spivey, 1992), which is a concise formal description. We consider only a part of it, the abstract definition of types for components, services, state spaces. Let *Composition* be a specification of components, services and compositions where $\mathbb{P}\, S$ is the power set of S, $X \leftrightarrow Y$ is the set of relations from X to Y and $X \nrightarrow Y$ is the set of partial functions from X to Y.

$[COMP, SERV, STATE]$ //the basic sets
$Composition \mathrel{\widehat=}$
$[components : \mathbb{P}\, COMP; states : COMP \nrightarrow STATE;$
$services : SERV \nrightarrow COMP; interface : SERV \nrightarrow COMP;$
$provided, required : \mathbb{P}\, SERV; intrequires : SERV \leftrightarrow SERV;$
$extrequires : SERV \leftrightarrow SERV; composite : COMP \nrightarrow COMP;$
$alink : SERV \nrightarrow SERV; plink : SERV \nrightarrow SERV \mid ...]$

The service dependency is the closure (denoted with $^+$) of the *requires* relations restricted (denoted with $\rhd$) to the provided services (*provided*), while taking into account the assembly and promotion links ($alink, plink$). Note that the closure should preserve the component encapsulation.

$\forall Composition;\ dependency : SERV \leftrightarrow SERV \bullet$
$dependency = (((intrequires \cup extrequires)^+$
$\rhd provided) \cup alink \cup plink)^+$

If the system is ready to run, its basic dependency is valid if there are no unsatisfied services *i.e.* $dependency = \emptyset$. This constraint is too strong when working with an incomplete architecture, so we restrict the dependency to the target provided services (the *source*), which are the services under test ($source \lhd dependency = \emptyset$). If the source must belong to the root of the system component, we add $service(source) \in (components \setminus \mathrm{dom}\, composite)$.

Building a test architecture is equivalent to applying a sequence of architectural transformations, also defined by a Z operation. The operation precondition ensures the preservation of the *Composition* system invariant.

$Transformation \mathrel{\widehat=} [\Delta Composition;$
$newComp? : SystemComponents$
$composite? : COMP \nrightarrow COMP; nalink? : SERV \nrightarrow SERV$
$ralink? : SERV \nrightarrow SERV; nplink? : SERV \nrightarrow SERV$
$rplink? : SERV \nrightarrow SERV; plink? : SERV \nrightarrow SERV \mid ...]$

A sequence of architecture transformations $T_1 \fatsemi ... \fatsemi T_n$ is valid if there are no unsatisfied required services ($required' \lhd dependency' = \emptyset$).

4.2 Consistency by Theorem Proving

Theorem proving techniques are helpful to prove the Component Consistency (CC) and the Assembly/Promotion Contract (APC).

The demonstration process (Figure 3) consists in writing model transformations to the target prover language and proving the theorems using the associated proof support. Some expertise in the prove environment is usually required.

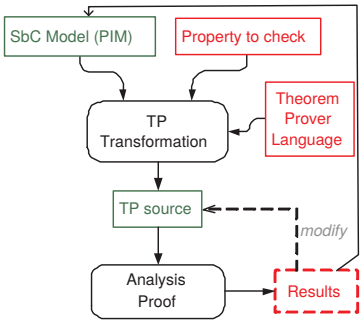

Figure 3: Theorem proving process overview

Component consistency (CC) At the component level, we have to check the *Invariant consistency vs. pre/post conditions* for its observable features (a kind of read-only visibility) and its non-observable features. Powerful tools like Atelier-B[1] and Rodin[2] are appropriate to prove that kind of property with high level data types. The difficulty is to transform in an adequate source for the target language and prover.

Assembly/Promotion Contract (APC) At the assembly level, we have to check the *Assembly Link Contract correctness*; this ensures that the contract for a required service is compliant with the one of the provider linked to it, up to data and message mappings. Based on a service assembly link, the main issue is to decide whether the provided service matches with the required service it is linked to. The matching condition is: *the pre-condition of required service Req is stronger than the one of provided service Prov and the post-condition of Req is weaker than the one of Prov*. In term of B proof obligations this property is viewed as: the provided service refines the required service. The transformation is applied for each link.
At the composite level, we have to check the *Promotion Link Contract correctness*; this ensures that the contract for a promoted service is compliant with the one of the original provider linked to it, up to data and message mappings. In term of B proof obligations this property is viewed as: the provided service refines the promoted required service and the promoted required service refines the base required service. Actually these are strong conditions but lighter alternatives are detailed in (André et al., 2010).

[1] http://www.atelierb.eu/
[2] http://rodin-b-sharp.sourceforge.net

4.3 Behavioural Compatibility by Model Checking

Model checking techniques are helpful to prove the *Behavioural compatibility (BC)*. At this stage we assume that services are neither atomic nor executed as transactions. Checking the behavioural compatibility means that services can synchronize and exchange data with other services without any troubles and terminate (Yellin and Strom, 1997; Attie and Lorenz, 2003; Bracciali et al., 2005). It often relies on checking the behaviour of a (component-based) system through the construction of a finite state automaton. To avoid state explosion problems (Attie and Lorenz, 2003) we work with peer services instead of the whole assembly. Ensuring dynamic behavioural compatibility of communicating processes is a property usually checked by model checkers.

The checking process (Figure 4) consists in writing model transformations to target languages (one per verification tool) and proving the properties using the dedicated model checker (Spin, Uppaal, CADP...). Depending on the model checker, the properties can be defined separately from the model (*e.g.* temporal logics) or not and a transformation may be needed for a single property. The verification process is improved when the result of a property verification is re-injected at the model level. Note that if the SbC formalism is very different from the target language, the transformation is difficult and an expertise in the target language is required to prove the properties.

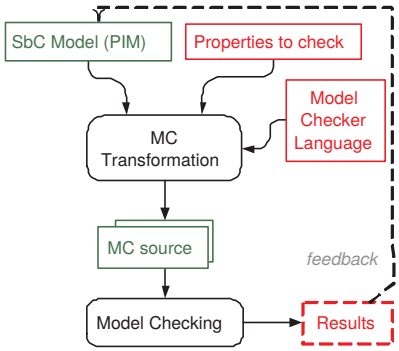

Figure 4: Model checking process overview

4.4 Functional Correctness by Model Testing

The basic idea of *Functional correctness (FC)* is to *evaluate* all paths of a service behaviour and to determine whether it is compliant with the post-condition or not. This is a non-trivial problem similar to the one of model-checking a program. As soon as the modelling language includes high level data types and computation statements (*e.g.* loops) the provers reach their limits to prove. Model testing is used here to supply the missing verifications of automatic and interactive provers. In (André et al., 2013) we argued for early testing at the model level to detect platform independent errors without melting them with implementation errors. Indeed plunging the model in a middleware decreases the testability and is often a burden to the V&V process. The model testing (not model-based testing) process consists in building a test application from a *test intention* (a test goal with data definitions and an oracle expression) and run it on test cases (Figure 5). It reduces the test complexity and improves both the application and test evolutivity.

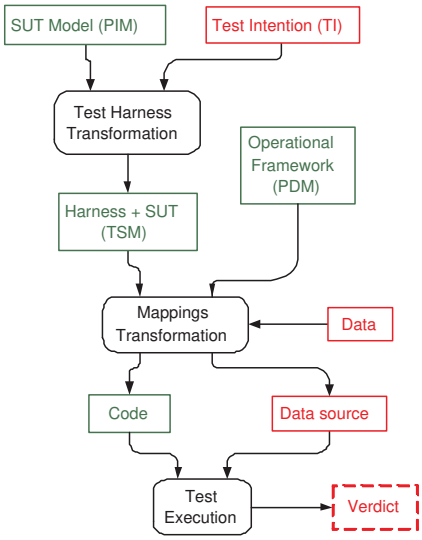

Figure 5: Testing process overview

A tool must assist the tester in managing the way the test data can be provided: some of them by the configuration, other ones by mocks, and the oracle by a test driver. To achieve this, the tool can:

- select a subset of the System Under Test (SUT) model according to a test intention;
- check if the Test Specific Model (TSM) is satisfying properties to be a SbC application: no bad connections right, no missing data or services;
- bind required services to mocks provided by libraries;
- check the TSM consistency and completeness in regards to its test intention (it may be improved/-completed during the test harness building);
- generate a test component including the *test case* services ;
- launch the test harness with several test data values sets and to collect the verdicts.

5 Experimentations

We experimented the above ideas through the Kmelia language and the related COSTO toolbox.

Modelling with the Kmelia language

Kmelia is an abstract formal component model dedicated to the specification and development of correct components (André et al., 2010; André et al., 2010). A Kmelia component system is an assembly of components, which can themselves be composite. A component is a container of services; it is described with a state space constrained by an invariant. A service describes a functionality; it is more than a simple operation; it has a pre-condition, a post-condition and a behaviour described with a labelled transition system (LTS). Moreover a Kmelia service may give access to other (sub)services. The behaviour supports communication interactions, dynamic evolution rules and service composition. Kmelia is supported with an Eclipse-based analysis platform called COSTO (see Figure 6). The tool and the case study material are available at `http://costo.univ-nantes.fr/`.

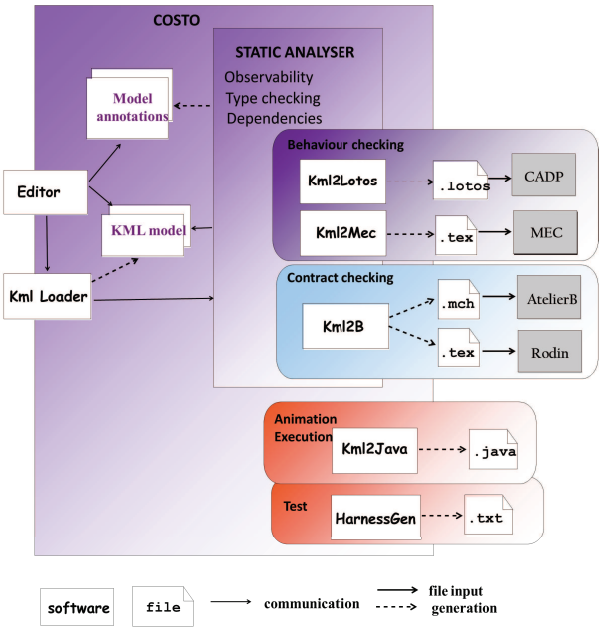

Figure 6: COSTO Tool Architecture

Using Kmelia, the platoon system elements (vehicles and driver) are components assembled through their services. Figure 1 illustrates the design of the *platoonSyst* assembly in the spirit of Kmelia: a composite component including the component assembly, which is statically defined over the three components initialized by a internal service of a composite. Each component provides an initialisation service (used when assembling), a main run to activate the vehicle behaviour and a stop service to interrupt or to end the vehicle. The driver and the vehicles are designed similarly with a *run* main (asynchronous) service. The goal to reach belongs to the driver state space. The vehicles require their predecessor position *pilotpos* and speed *pilotspeed* to update their own state. The *start* and *stop* services model the system environment actions.

Combined verifications with COSTO

We illustrate the above combination of verification techniques on services at different specification levels (service contract, interactions, behaviour) in Figure 6.

Structural correctness by static analysis The structural properties (such as syntax, correctness, consistency, accessibility, observability rules...) are checked during the compilation of the Kmelia specification by COSTO (*cf.* Figure 7).

Consistency by theorem proving We developed a series of plugins named `Kml2B` in the Figure 6 to extract B specifications. For each Kmelia component K we build an (Event-)B model called C, its state space is extracted from the component's one. The provided services srv_i in K are translated into srv_i operations within the C model. The extracted specification is imported and checked in Atelier-B or Rodin. The B tools enables the verification of invariant consistency at the Kmelia level.

CC At the component level, we check the *Invariant consistency vs. pre/post conditions* for both the observable features (a kind of read-only visibility) of it and the non-observable features.

APC At the assembly and the composite levels, each service link, up to data and message mappings, leads to a refinement relation and a related proof obligation.

In the case of the computeSpeed service, the Atelier-B generated seven proof obligations. At first attempt four of them were automatically proved. The three others could not be proved because the original Kmelia specifications was insufficiently precise and complete: parameter ranges, over ranged speed values, missing speed assignment. Once corrected in Kmelia model and updated in the B specifications, the seven PO were proved correct with a level-1 automatic proof.

Behavioural compatibility by model checking We developed a couple of plugins named `Kml2Mec` (resp. `Kml2Lotos`) in the Figure 6 to extract finite state machines (resp. processes) specifications. For each assembly link, a corresponding MEC (or LOTOS) spec-

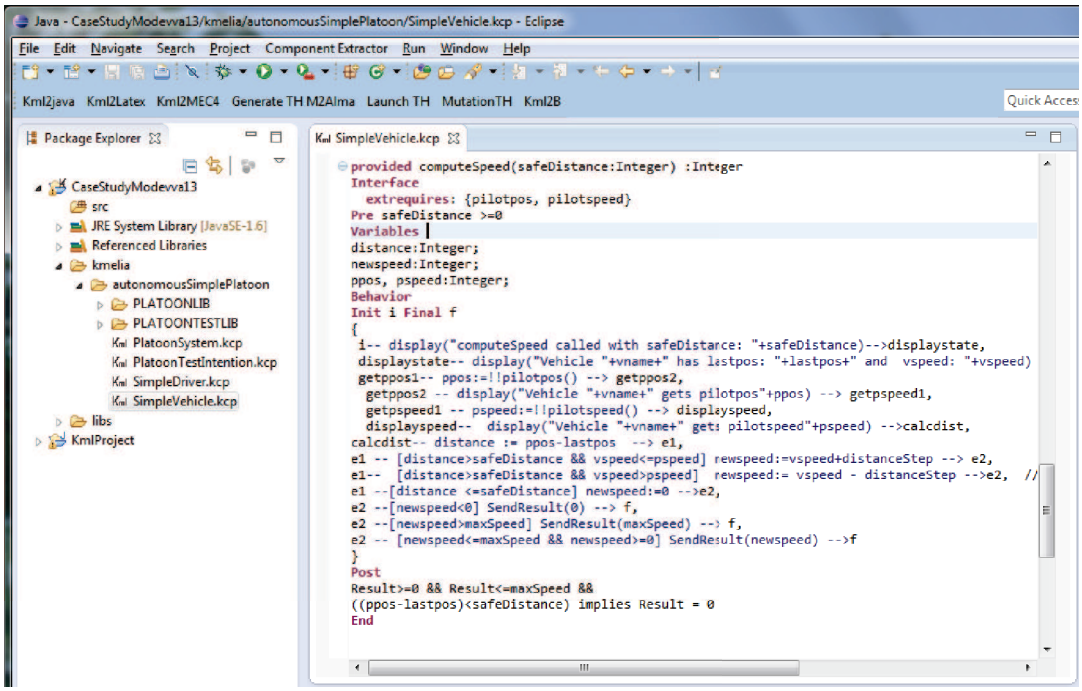

Figure 7: Specification of service computeSpeed

ification is generated that includes the synchronisations of the communications. The promotion links are gateways for the communications and need no specific proof. The translation details are given in (Attiogbé et al., 2006). The verification is achieved using model-checking techniques provided by existing tools (Lotos/CADP[3] and MEC[4]). The advantage of MEC is that it preserves the finite state machine (FSM) structure of Kmelia services, so we could develop a plugin to interpret the result of the model checking.

To prove the *Functional correctness (FC)* we first tried model checkers but they could not support high level data and functions. We then investigated B tools, including *ProB* a model checker for B. We had to turn back to more appropriate tools because B tools needed additional material to prove loop invariants and *ProB* was not powerful enough. We also investigated the Key tool (Beckert et al., 2007). Key accepts JML specifications as input; in order to prove properties of Java programs. The idea was to transform finite state machine (FSM) statements into Java-like statements however the distance between FSM-based models and procedural-based models is still to important to write transformation without side effects also the communications are interpreted as method call in plain Java, a communication framework is required. A more pragmatic way was to turn to testing techniques.

Functional correctness by model testing We developed a Model Testing Tool (named COSTOTest) as specified in section 4.4. The test process is illustrated on the computeSpeed service in the mid platoon vehicle. Its specification is given in the Figure 7. The result of the computeSpeed service depends on several data: the recommended safe distance from the pilot (previous vehicle), the position and speed of the current Vehicle and the position and speed of the pilot. This is represented by the test intention of Listing 1. For each test intention, a test harness (TSM) is elaborated during an iterative building process. As an example, Figure 8 represents a component application for testing the service computeSpeed in the mid Vehicle. The test and the corresponding oracle are encapsulated into a testComponent tc, and a Mock component has replaced the Driver to offer better control. The last Vehicle has not been selected here because it is not needed to test the computeSpeed service of the mid Vehicle, but a more complex architecture could have been retained. The service testcase1 of testComponent contains a simple computeSpeed call and oracle evaluation. Every data is obtained by using abstract functions in the model that are mapped to concrete data providers.

In the following we will detail the process that allows us to create test applications like the one we presented in Figure 8. The testing process is a sequence

[3] http://www.inrialpes.fr/vasy/cadp/
[4] http://altarica.labri.fr/wiki/tools:mec_4

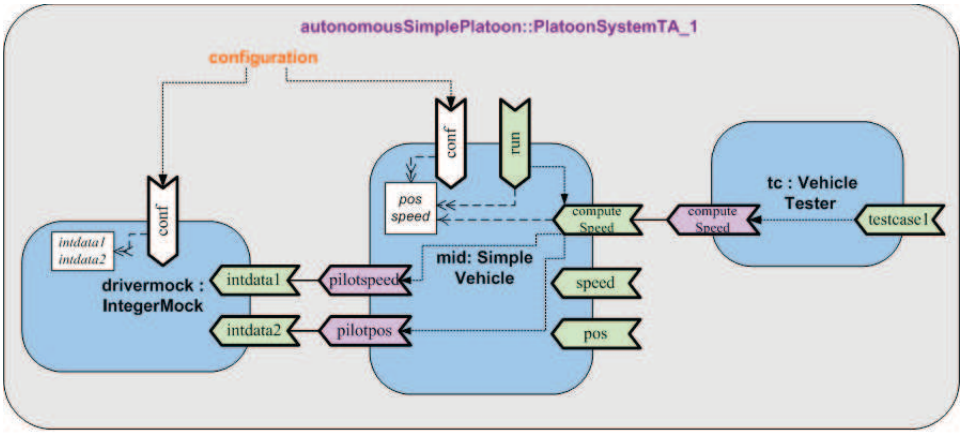

Figure 8: Test architecture for service computeSpeed

of model transformations which successively merge models, integrating features into them, as illustrated in Figure 5. The input System Under Test is a PIM of the SbC and a test intention is also a model described in *cf.* Listing 1. The process is made of two successive model transformations which return an executable code of the *test harness*.

Listing 1: Test intention for computeSpeed service

```
TEST_INTENTION PlatoonTestIntention
DESCRIPTION "test of the service computSpeed,
             covering control flow graph"
USES {PLATOONTESTLIB}
INPUT VARIABLES
  lastpos:Integer;
  vspeed:Integer;
  safeDistance:Integer;
  pilotpos:Integer;
  pilotspeed:Integer;
OUTPUT VARIABLES
  speed:Integer;
  oracledata:Integer;
 ORACLE
  speed=oracledata
```

The first model transformation is a model-to-model transformation. It builds the *test harness* as an assembly of selected parts of the SUT with test components (mocks, test driver), and returns a *Test Specific Model* (TSM). It is semi-automatic transformation: the test intention is provided by the tester and COSTOTest asks her/him to make choices that are selected on the basis of static analysis of the PIM. During this first step, the aim for the tester is to build a harness such as the one illustrated in the bottom of Figure 8.

The second transformation is a model-to-code transformation; COSTOTest generates the code to simulate the behaviour of the harness, then it merges the harness with a *Platform Description Model* (PDM) to get code (Java code in this case). It can be executed, because the model of the components describes the behaviour of the services, in the form of communicating finite state machines. The test data and test oracle providers are designed in the PDM, thanks to the input "Data". A "data source" is generated, it is an XML file, with a structure corresponding to the test intention, that should be fulfilled with concrete values by the tester.

Finally, *the test execution* consists in setting the test data and then "run" the test harness component. COSTOTest proposes interactive screens to enter all the data values into the XML file generated by the second model transformation. The tester can also provide the test data values in a CSV file which is transformed into the XML file. We consider the test of computeSpeed service, covering its control flow graph to generate test data. We create 45 test cases and run them getting the verdicts. The data source XML file will also store the verdicts (*cf.* Figure 9).

6 Related Work

The combination of formal verification and testing is not new but the way they are combined varies with the verification goals (Bousse, 2013), *e.g.* hybrid approaches for functional verification (Bhadra et al., 2007).

Many works that combine tests and proofs use finite state machines dialect as modelling DSL (Constant et al., 2007; Falzon and Pace, 2012; Artho et al., 2005). In the spirit of Model Based Testing (MBT), the authors focus on conformance checking and the goal is to generate test cases from a formal specification to check whether an implementation conforms

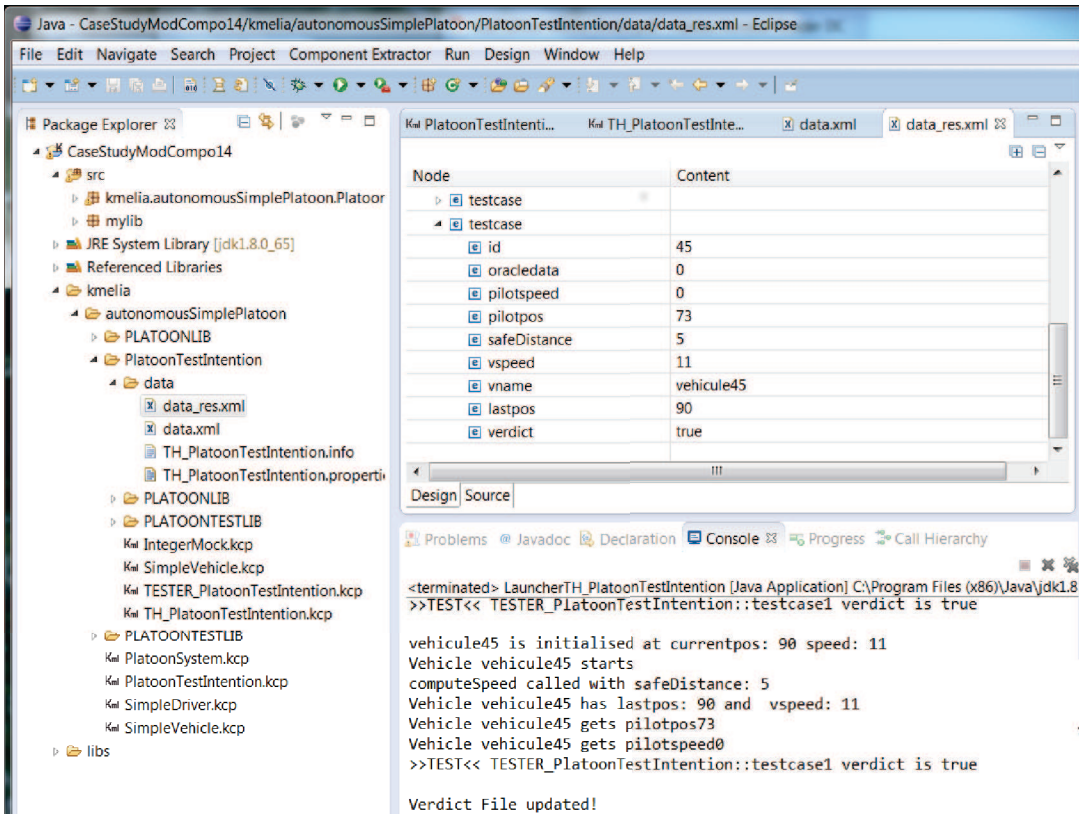

Figure 9: Test harness assignments: verdict stored in the XML file

to the model (Constant et al., 2007) or to monitor runtime verification (Falzon and Pace, 2012). Herber et al. generate conformance tests to complete the model-checking of SystemC designs (Herber et al., 2009). Conversely Dybjer et al. use testing to avoid the effort of costing proofs (Dybjer et al., 2004), their method interweaves proving steps and testing steps while usually the proofs are done first on the model. Similarly Sharyginal and Peled use testing (with PET) prior to the actual verification (with SPIN) and during the evaluation of counter examples (Sharygina and Peled, 2001), testing is thus a kind of heuristics to reduce the state space explosion. However their objective is not to get a correct-by-construction code but check whether the C++ code is correct by translating to model (reverse-engineering). Also no tool support is provided for the translations.

We can get inspired by the above techniques but none is a direct answer to our goal which is (i) centred on the verification of correctness, (ii) at the model level, (iii) for heterogeneous models and properties (structure, dynamics, functions) that supposes some completeness. As mentioned by E. Bousse at the beginning of his PhD in (Bousse, 2013), the question is "How to perform effective V&V on such complex and potentially heterogeneous models?". He mentioned several pitfalls: the V&V tools have limited application fields, low expressive power, low scalability, low integrability, semantic gap between domains... Consequently one pivot language cannot catch all the aspects. Several years later, he proposed a transformation based approach to align SysML with B (Bousse et al., 2012) that managed to prove safety properties. The alignable subset of SysML is covered but the problem stays open for the unaligned aspects.

We are convinced that the solution solution is a collaborative approach for model testing instead of an unifying approach. The forthcoming question is what makes the glue between the heterogeneous aspects. A possible answer is the concept of contract because it has the same underlying semantics that crosses the approaches, especially those related to services. A contract is the agreement between clients and providers and the interesting point is that it includes clauses that can focus on the heterogeneous aspects (rights and duties, quality of service...) (Beugnard et al., 1999). The notion of multi-level contract that we promote here can be an unifying paradigm for the functional contracts of Meyer (Meyer, 2003) or the behavioural contracts (Acciai et al., 2013; Fenech et al., 2009).

Contracts are a basis for property verification as well as for testing oracles (Le Traon et al., 2006). We agree with Dwyer and Elbaum that noted the risk of focusing on individual techniques (Dwyer and Elbaum, 2010) and Table 2 defines a way to characterise their property-behaviour coverage.

Contracts and services have been studied in the context of service composition. From a service composition point of view *e.g.* BPEL, the behavioural aspect is preeminent (ter Beek et al., 2007). Considering only the formal models, composition is mainly based on automata, Petri nets and process algebra, as illustrated by the orchestration calculus of Mazzara and Lanese (Mazzara and Lanese, 2006); therefore the contracts focus mainly on dynamic compatibility. Conversely the contracts (in the sense of *design-by-contract*) are taken into account in (Milanovic, 2005) (using abstract machines) but not the dynamic behaviour. Kmelia handles both aspects. In (Brogi, 2010), the contract is supported at four levels (signature, quality of service, ontology, behaviour) but none of them handle the functional contract. The service concept is a key one. The component architecture (SCA) approaches (Ding et al., 2008) emphasize the service concept, like Kmelia does; but unfortunately contract features are not introduced yet in SCA. Testing LTS behaviours is performed in (Schätz and Pfaller, 2010). The authors customize component testing at the level of component in a system use. Our framework also allows to customize the testing through the definition of the testing perimeter and the selection of mock services, then it applies the same kind of tests with a mutation analysis. In (Lei et al., 2010), the authors target robustness testing of components using rCOS. Their CUT approach involves functional contracts and a dynamic contract (protocol). Our approach does not target robustness, but the mutation analysis exploits the kind of errors of (Lei et al., 2010) (bad call sequence / invalid parameter) in a more systematic manner.

7 Conclusion

Reusability and composability belong to the foundations of service and components systems and their confidence must be ensured at early stages of the design of systems, by verification and validation techniques. In practice, to face this challenge, one must combine several techniques and the notion of multi-level contracts including the right/duty clauses on the orthogonal aspects of a system (structure, dynamic and functional behaviour) seems a promising unifying paradigm. We experimented these idea with the Kmelia language that enables one to specify service-based component systems where the service interfaces are equipped with contracts and service behaviours are defined with communicating finite state machines. Each level of the contract is checked by adequate tool support including model checking, theorem proving and model testing.

The current state of the proposal requires additional work and tool improvement. The additional work concerns the specification and verification of quality of service, related to the non-functional properties. New language primitives have to be implemented to specify additional constraints on time and resources. Related V&V techniques have to be experimented. The main issues on tool improvement concern platform facilities and abstraction because the verification stages require expertise in domain specific provers. At best, the modeller would need to know the proof techniques but not the proof tools. This is mainly the case with model checking and testing where the GUI can hide the implementation level but additional work has to be done for provers.

REFERENCES

Acciai, L., Boreale, M., and Zavattaro, G. (2013). Behavioural contracts with request-response operations. *Sci. Comput. Program.*, 78(2):248–267.

André, P., Ardourel, G., Attiogbé, C., and Lanoix, A. (2010). Using assertions to enhance the correctness of kmelia components and their assemblies. *ENTCS*, 263:5 – 30. Proceedings of FACS 2009.

André, P., Ardourel, G., and Messabihi, M. (2010). Component Service Promotion: Contracts, Mechanisms and Safety. In *7th International Workshop on Formal Aspects of Component Software(FACS 2010)*, LNCS. to be published.

André, P., Mottu, J.-M., and Ardourel, G. (2013). Building test harness from service-based component models. In *proceedings of the Workshop MoDeVVa (Models2013)*, pages 11–20, Miami, USA.

Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., and Washington, R. (2005). Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234.

Attie, P. and Lorenz, D. H. (2003). Correctness of Model-based Component Composition without State Explosion. In *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*.

Attiogbé, C., André, P., and Ardourel, G. (2006). Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer.

Beckert, B., Hähnle, R., and Schmitt, P. H., editors (2007). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag.

Beek, M., Bucchiarone, A., and Gnesi, S. (2006). A survey on service composition approaches: From industrial standards to formal methods. In *In Technical Report 2006TR-15, Istituto*, pages 15–20. IEEE CS Press.

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *Computer*, 32(7):38–45.

Bhadra, J., Abadir, M. S., Wang, L.-C., and Ray, S. (2007). A survey of hybrid techniques for functional verification. *IEEE Des. Test*, 24(2):112–122.

Bousse, E. (2013). Combining verification and validation techniques. In *Doctoral Symposium of ECMFA, ECOOP and ECSA 2013*, page 10, Montpellier, France.

Bousse, E., Mentr'e, D., Combemale, B., Baudry, B., and Takaya, K. (2012). Aligning sysml with the b method to provide v&v for systems engineering. In *Model-Driven Engineering, Verification, and Validation 2012 (MoDeVVa 2012)*, Innsbruck, Austria.

Bracciali, A., Brogi, A., and Canal, C. (2005). A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54.

Brogi, A. (2010). On the Potential Advantages of Exploiting Behavioural Information for Contract-based Service Discovery and Composition. *Journal of Logic and Algebraic Programming*.

Constant, C., Jéron, T., Rusu, V., and Marchand, H. (2007). Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574.

Crnkovic, I. and Larsson, M., editors (2002). *Building Reliable Component-Based Software Systems*. Artech House publisher.

Ding, Z., Chen, Z., and Liu, J. (2008). A rigorous model of service component architecture. *Electr. Notes Theor. Comput. Sci.*, 207:33–48.

Dwyer, M. B. and Elbaum, S. (2010). Unifying verification and validation techniques: Relating behavior and properties through partial evidence. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 93–98, New York, NY, USA. ACM.

Dybjer, P., Haiyan, Q., and Takeyama, M. (2004). Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025.

Falzon, K. and Pace, G. J. (2012). Combining testing and runtime verification techniques. In Machado, R. J., Maciel, R. S. P., Rubin, J., and Botterweck, G., editors, *Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers*, volume 7706 of *Lecture Notes in Computer Science*, pages 38–57. Springer.

Fenech, S., Pace, G. J., Okika, J. C., Ravn, A. P., and Schneider, G. (2009). On the specification of full contracts. *Electr. Notes Theor. Comput. Sci.*, 253(1):39–55.

Herber, P., Friedemann, F., and Glesner, S. (2009). *Combining Model Checking and Testing in a Continuous HW/SW Co-verification Process*, pages 121–136. Springer Berlin Heidelberg, Berlin, Heidelberg.

Le Traon, Y., Baudry, B., and Jézéquel, J.-M. (2006). Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586.

Lei, B., Liu, Z., Morisset, C., and Li, X. (2010). State based robustness testing for components. *Electr. Notes Theor. Comput. Sci.*, 260:173–188.

Mazzara, M. and Lanese, I. (2006). Towards a unifying theory for web services composition. In Bravetti, M., Núñez, M., and Zavattaro, G., editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 257–272. Springer.

Messabihi, M., André, P., and Attiogbé, C. (2010). Multilevel contracts for trusted components. In Cámara, J., Canal, C., and Salaün, G., editors, *WCSI*, volume 37 of *EPTCS*, pages 71–85.

Meyer, B. (2003). The Grand Challenge of Trusted Components. In *Proceedings of 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society.

Milanovic, N. (2005). Contract-based web service composition framework with correctness guarantees. In Malek, M., Nett, E., and Suri, N., editors, *ISAS*, volume 3694 of *Lecture Notes in Computer Science*, pages 52–67. Springer.

OSOA (2007). Service component architecture (sca): Sca assembly model v1.00 specifications. Specification Version 1.0, Open SOA Collaboration.

Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors (2008). *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, Heidelberg.

Schätz, B. and Pfaller, C. (2010). Integrating component tests to system tests. *Electr. Notes Theor. Comput. Sci.*, 260:225–241.

Sharygina, N. and Peled, D. A. (2001). A combined testing and verification approach for software reliability. In Oliveira, J. N. and Zave, P., editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 611–628. Springer.

Spivey, J. M. (1992). *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall.

ter Beek, M., Bucchiarone, A., and Gnesi, S. (2007). Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10.

Yellin, D. and Strom, R. (1997). Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333.

Zaremski, A. M. and Wing, J. M. (1997). Specification matching of software components. *ACM Transaction*

on Software Engeniering Methodolology, 6(4):333–369.