Reducing Domain Level Scenarios to Test Component-based Software

Oliver Skroch
Chair of Business Informatics and Systems Engineering, Universität Augsburg, Germany
Email: oliver.skroch@wiwi.uni-augsburg.de

Klaus Turowski
Chair of Business Informatics and Systems Engineering, Universität Augsburg, Germany
Email: klaus.turowski@wiwi.uni-augsburg.de

***Abstract*—Higher-order black box software tests against independent end user domain requirements has become an issue of increasing importance with compositional reuse of software artifacts. Recently, a general method was proposed that derives testable scenarios directly from a customer domain model by abstraction, reduction and inclusion for critical coverage [32]. The resulting linear (i.e. non-branching) scenarios are used as references to test suppliers' software specifications against. This paper presents the method in an overview and elaborates on the domain reduction step within the process for the generation of testable scenarios from a domain model. An example is provided which is non-fictitious on the domain side. Advantages of the method are an underlying clear business model, test oracles that are independent from the software development process, and validation results that are generated early in the development cycle, before the software itself is available.**

***Index Terms*—component, software, higher-order test, validation, scenario, domain, end user requirements**

I. Introduction

Software engineering is in the process of evolving from a craft to an industry and reuse is one decisive element that supports and propels this evolution. Reuse has even been described as "the only realistic approach" [20] to meet the needs of a software industry. Recently, further increasing needs for reuse have been listed among the top trends that will influence future software processes [5].

Compositional reuse is one of the fundamental software reuse technologies [4]. The approach is to reuse executable artifacts which are found in repositories, and compose them into larger applications [33]. Compositional reuse of black box business components is part of the overall concept of component-based business applications, where business components are described by multi-layered and semi-formal specifications, implement services from a business domain, and are envisaged to be traded on markets [35]. Such compositional reuse includes

- building software for reuse, by creating self contained, marketable, fully described black box artifacts on the supply side,
- building software from reuse, by composing larger applications from these executable stand-alone artifacts on the demand side, and
- trading the associated software artifacts or components on a market (possibly an internal market within a corporation).

In this environment, the demand side – customers and end users of component software – looks for useful software components and does not want to access source code but restricts to a black box view. The demand side focus therefore is on "higher-order" [22] compliance of domain level pragmatics and semantics, while mere formal and syntactical compliance is often perceived as technical precondition in the responsibility of the supply side.

Software component reuse with parts that can be looked up in catalogues and can then be integrated into large applications similar to electronic parts has been proposed already since long [17]. But non-trivial problems still complicate broad compositional software reuse in theory and in practice today. Among the problems on the demand side is the evaluation of available components against their more complex end user domain requirements: assuming that an offered component complies syntactically, it still needs to be tested if its pragmatics and semantics are useful for a specific domain automation purpose.

In traditional engineering disciplines, the importance of testing is well acknowledged because of a long history of experiences. In software engineering it is on the one hand known that software is fundamentally less reliable than traditional engineering products [28] and that building software will probably always be hard [7]. On the other hand the well-known notion of "good-enough

Based on "Validation of Component-based Software with a Customer Centric Domain Level Approach", by Oliver Skroch which appeared in the Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems 2007, Tucson Arizona, USA, March 2007. © 2007 IEEE.

software" [42] shows that we have to deal with a pragmatic view on quality aspects of software, in particular with large enterprise applications.

But also good enough software development can profit from testing, especially with enterprise-sized systems if errors are found efficiently and early in the development process. Firstly, it was shown that the effort for error correction grows markedly when the error is detected later. Secondly, the earlier errors are detected the more rectification alternatives are available. Thirdly, studies in science and projects in industry indicate that testing takes more than fifty percent of the effort even with non-safety critical software.

Software testing is even more important whenever prefabricated items such as components are reused. Firstly, a single component made for reuse must be more thoroughly tested than a component made to be used once because it is reused in combinations unknown at the time of development. Secondly, a system based on a configuration of multiple black box components from different suppliers must be more thoroughly tested as compared to large pre-integrated products. [19]

The distinction between technology based supplier testing and domain based customer testing is widely acknowledged, in particular with component-based software [12;40]. Recently, an approach was proposed specifically for component validation testing on the domain level of the demand side [32]. It is based on testable scenarios which are independently derived from an end user domain and become checked against reuse specifications from suppliers.

The rest of the paper presents and elaborates the method and is structured as follows. Section two of the paper sets out basic assumptions and presents the underlying business model. Section three introduces the approach in an overview, elaborates on the process reduction through an abstracted business domain, and finally applies the method in a small example, non-fictitious on the domain side. Section four elaborates on the current state of the art and on existing solutions, and delimits the contributions of the method. Section five summarizes and concludes this paper.

II. Basic Assumptions And Business Model

Dynamics and pragmatism of real life businesses demand good enough software which is useful to the customer, and therefore support a focus on higher-order domain tests. So our approach is based on the fundamental assumption that the final arbiter of software success is only the customer to whom the component software is useful or not. This most central assumption was stated already in 1979: "*A software error is present when the program does not do what its end user reasonably expects it to do.*" [22].

The end user domain is the area of intended application for the component-based software. While it usually lacks a fully formal definition or model, we still assume that customer test references from the application domain prevail over test oracles created with mere supplier knowledge from within the component software technology. Higher-order testing on the domain level, without the intention to change or reengineer components or their specifications, initially has as a goal to validate the suppliers' software for reuse and control on the demand side. The argument of assertive and independent consideration of the ontological domain and the supporting technologies can be founded in Ψ (psi) theory [9].

From an end user's domain point of view, it is favorable to test higher-order requirements independently and as early as possible. This supports the identification and assessment of components, if possible at best before the executable software itself is available. The necessary validation knowledge consists of testable business requirements that predefine what the right software solution is supposed to do, and it needs to be constructed.

Our construction approach is embedded into a clear business model assumption derived from the vision of industrialized compositional reuse for software engineering, which has been described in detail in [35]. Fig. 1 (notation: e3-value [13]) introduces the underlying business model assumption with the three actors: component supply, component demand and component market.

In the business model, suppliers create components for an anonymous market to satisfy an assumed demand or requirement on that market. These requirements can typically be acquired from discussions with individual clients but also could very well be entrepreneurial market assumptions. Software components offered to cover the requirements are technically mature and suppliers keep their source code undisclosed. They completely specify their components in black box style by fully defining the interfaces to convey the components' contracts (what the components do) but without disclosing their implementation details (how the components work) [18]. Specifications achieving this are multi-layered and semi-formal today. Respective specification approaches are proposed e.g. in [26;36] where contract levels and facts to be specified describe the external view onto the component for reuse. These component specifications can serve as black box description for reuse and are put into publicly available component specification libraries.

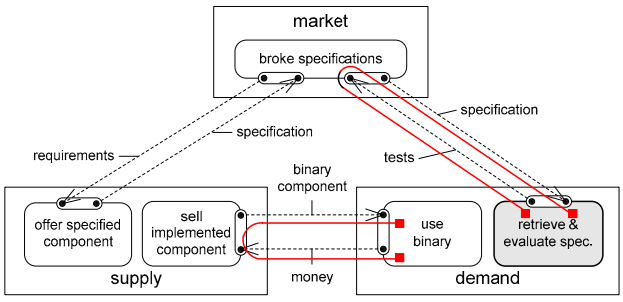

Figure 1. Business model assumption.

Component software users on the demand side want support and automation for their requirements and search a wide variety of library components for the right

software. The available components are found as specifications e.g. on the Internet. These semi-formal, multi-layered component reuse specifications represent the candidates offered by suppliers for domain testing. Customers query the black box functional specifications with specific predefined criteria, retrieve matches, and then evaluate the retrieved specifications in detail. Both retrieval and evaluation imply a comparison i.e. a test between reference features demanded and specification candidates offered.

Compositional reuse acknowledges the industrial segregation between a supply side offering components for reuse and a demand side requiring software built from reused and properly orchestrated parts. Such industry-style compositional reuse apparently requires advances to established software engineering methods, which also includes the testing stage. An important challenge for black box reuse at this point is how to derive reasonable specification retrieval and evaluation criteria, and that means: how to validate testable end user domain requirements against supplier specifications.

The associated testing may be classified as specification based or program based, and specification based testing can be divided into state based testing and black box testing [38]. The component paradigm of the described business model assumes that components are tested on the basis of their specifications, and restrict our approach to black box testing. It is acknowledged that good overall testing will be comprehensive and will employ a set of complementary methods in practice. It is also acknowledged that testing alone can not improve the quality of software, but early and expressive test results can improve decisions.

III. Constructing Linear Scenarios

A. ARIval Overview

A precondition for the validation of requirements is that these requirements are stated in testable terms. Fig. 2 (notation: activity diagram [24]) gives an overview on the ARIval (abstraction, reduction, inclusion, validation) method [32], where domain level scenarios are used to validate aspects of multi-layered component reuse specifications, if possible showing that the specified software works for the higher-order domain requirements.

To construct testable business requirements on the customer side, our first starting point is the observation that also for testing higher-order domain functionality, only a small subset of the full domain is actually relevant for the end users' intended automation with distinct effects on utilized system behavior.

The second starting point is the observation that some kind of domain model is usually available on the customer side, in many cases through prosaic business rules and process descriptions as semi formal or informal models, e.g. activity diagram, event driven process chain, Petri net, etc. Full or partial automation is required for the model, or parts of it, from ready-made software components.

Relevant parts of the model environment are first abstracted based on the well-known equivalence partitioning and boundary value analysis, which is described and used for program testing since the late 1970s [22]. This results in domain partition elements which are a discrete representation of the original continuous domain, with one representative element per partition.

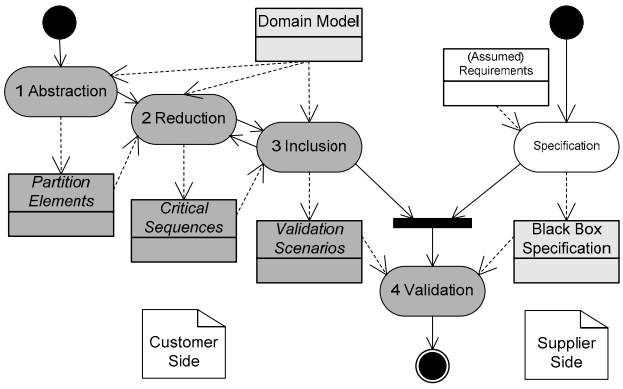

Figure 2. ARIval overview.

The abstracted elements are then reduced, by identifying reasonable and critical sequences. Complexity of typical requirements in real settings will lead to very many possible sequences at this point and prevent an exhaustive testing. This means that with each possible sequence of steps that requires automation on the domain side, and with the corresponding sequence of equivalence classes, a small number of critical sequences need to be selected from the very large number of all possibilities.

Selection criteria are domain centric and come from outside of the software engineering process. They include domain workflow and value flow considerations e.g. on frequency, criticality, financial or other risk, external visibility, etc. instead of software centric objectives such as coverage of all control statements in the source code. Furthermore, the sequences must not contain branching but make up linear paths in order to avoid quantitative evaluation problems during actual testing (cf. state explosion). To achieve this, a critical sequence with branches becomes de-branched until we have a number of linear sequences instead.

The abstracted and reduced domain part then contains value representatives in sequences, with each sequence linear and deemed critical by the customer for the intended application.

An inclusion will use the critical linear sequences to build scenarios, both within a domain part and across a number of different related domain parts, to cover the critical paths in their context as full business transaction flows. These scenarios must again not contain branching but make up linear paths. This can be guaranteed by constructing them accordingly, i.e. instead of a branching scenario we include two or more branch-free scenarios, until all resulting scenarios are linear at the end.

The method provides the possibility to re-iterate the reduction step, e.g. if certain sequences are found missing one can go back and establish them to be available for the construction of the respective end-to-end scenario. In this way each linear scenario is deliberately and consciously included into the validation step, or not. Inclusion criteria, again, are domain centric and are derived from considerations rooting in domain ontology instead of software technology, as described.

Finally, the actual testing will numerically check applicable parts of the reuse specifications from the supply side using all formally defined and branch-free critical validation scenarios as test cases.

Three basic coverage measures can be defined. Two start from the abstracted domain, which is an equivalent of the original domain. Reduction coverage measures the abstracted domain against reduced sequences requirements. Inclusion coverage measures reduced sequences against included scenarios. The third measure starts from the set of scenarios. Validation coverage measures a scenario's expected results against the actual validation success. The measures could be plain and weighted. The weighted coverage would scale on numeric scores given for each reduction criteria, inclusion criteria and scenario, e.g. by using a simple ranking.

Beneficiaries of the method are mainly customers and end users in the presented business model. The ARIVal method supports them in evaluating the many component specifications from repositories on the basis of their testable requirements, independently derived from their ontological domain, and before actual software is available.

B. Process Flow Transformation and Blocking

Through data abstraction, based on equivalence partitioning and boundary value analysis, we prepared a discrete data domain which is an equivalent representation of the complete and continuous original data domain. We now aim at the identification of an incomplete set of branch-free critical sequences through this abstracted, discrete domain model.

At the core is the reduction of the process domain. The approach is a double reduction: first, transformation and block building on process scheme level, and then numerical (de-)selection on the level of process instances (or, test sequences) in the simplified scheme. In this way, we deliberately resign from completeness twice. In other words, we first select the critical scheme parts from the overall process flow that need testing coverage. This leads to a simplified process scheme. The selected scheme parts that are deemed critical by the customer are at the same time numerically unfolded according to the abstracted domain model (i.e. all possible “traces” are listed that can be derived from the business rules). Now we can select a small number out of all possible numerical sequences through this simplified domain process scheme. The result is a small critical subset out of the very large set of all possible paths through the abstracted domain model.

Criteria to be used are based on aware stakeholder priority decisions, e.g. on business criticality of different process scheme parts and of different "traces" through the simplified model. This could be measured e.g. in terms of monetary value flow per path. If e.g. in a process scheme half of the revenues are generated within a certain small subset of maybe ten percent of the full scheme, and the other half of the revenue generation happens throughout the remaining ninety percent of the scheme, then apparently the smaller subset of the scheme is probably more important for the end-user testing. In this simplified example we could even already calculate a very simple priority value from the figures. The further elaboration of underlying stakeholder criteria would lead away from the scope of this paper. At this point we just need to take the diligent assumption that we are able to prioritize process scheme parts and process instances according to their business value.

From computability theory we can derive the fundamental process flow constructs “sequence”, “join” and “split” selection (joining pre-conditions, splitting post-conditions, also known as “selection”) and “iteration”, which are also described and used as starting point for workflow patterns definitions [37]. Process flow patterns use constructs ranging from these simple elements up to complex processing primitives. For each of the three basic constructs, we take workflow patterns from [37] and show how transformation and block building works for the basic construct (the notation used in the figures is UML activity diagram [24]). The full domain process flow can then be treated iteratively by treating the single basic constructs.

A *sequence* of process steps as shown in Fig. 3 is found in the basic workflow pattern “sequence”. It reflects the fundamental notion of an activity that is enabled after the completion of the preceding activity, and a common interpretation for the pattern is implication or causality.

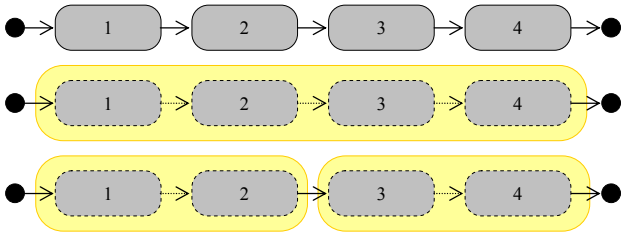

Figure 3. Sequence blocking.

As a linear sequence this basic construct is already in the form of our intended result, and we can – without further transformation – readily form a block unit by using any continuous subsequence of it; in Fig. 3, the second line shows a block built from the maximum subsequence, the third line shows an alternative block building. Each block can then be (de-)selected as a whole unit. This means that numerical test, and as a consequence also oracles, will be set at the block boundaries only; in line three of Fig. 3 before activity 1 and after activity 2, and before activity 3 and after

activity 4, but not, say, after activity 1. This implies that even when including the block unit, there will be no consideration of block internals. In the reduced, simplified process scheme, the internal structure of the block is hidden.

A *split* selection of the activities flow into multiple activities as shown in Fig. 4 is found in the basic workflow patterns "XOR-split" and "AND-split". The patterns reflects the essential notion of branching activities. A common interpretation for the XOR-split or switch pattern shown on the upper left side of the figure is decision. A common interpretation for the AND-split or fork pattern shown on the upper right side of the figure is parallel processing.

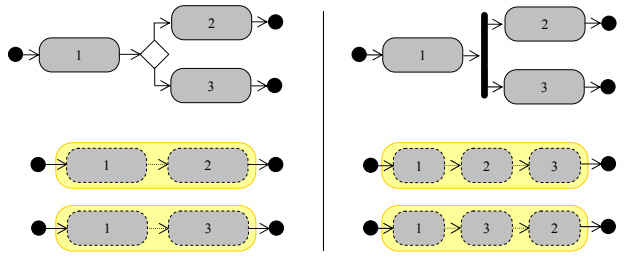

Figure 4. Split transformation and blocking.

For both split types, we transform the process scheme into a simpler scheme for blocking as shown in Fig. 4. On a binary XOR switch, as well as on a binary AND fork, two blocks encompass the construct, one block for each of the two subsequent steps within the scheme part. Splits with more than two following steps can be handled accordingly and result in more than two blocks. The internal block structures become hidden on the simplified scheme level. Numerical selection of the single "traces" in a subsequent step is less complex and establishes linear paths. Note that the concurrency aspect of the AND-split disappears, which seems appropriate for the intended testing against reuse specifications and without executable software.

A *join* selection of the activities flow from multiple activities as shown in Fig. 5 is found in the basic workflow patterns "XOR-join" and "AND-join". It reflects the essential notion of merging activities. A common interpretation for the XOR-join pattern shown on the upper left side of the figure is trigger. A common interpretation for the AND-join pattern shown on the upper right side of the figure is synchronization.

For both join types, we transform the process scheme into a simpler scheme for blocking as shown in Fig. 5. On a binary XOR trigger, as well as on a binary AND synchronization, two blocks encompass the whole construct, one block for each of the two preceding steps within the scheme constructs. Joins with more than two preceding steps can be handled accordingly and lead to more than two blocks. Again the internal block structures become hidden to the unfolding in the numerical reduction, when we select the "traces" in a subsequent step. Note also here that the synchronization aspect of the AND-join disappears, which again seems appropriate for the intended testing against reuse specifications and without executable software.

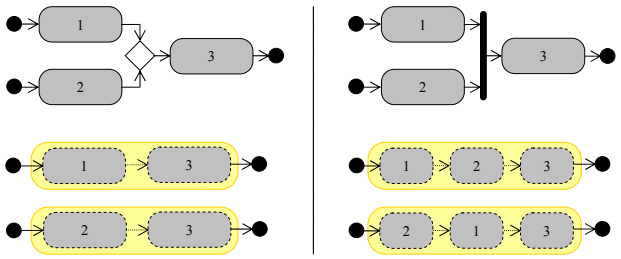

Figure 5. Join transformation and blocking.

An *iteration* in the activities flow as shown in Fig. 6 is found in the structural workflow pattern "arbitrary cycles". It reflects the notion of activities that can be done repeatedly. A common interpretation for repeated activities patterns is loop.

For an iteration construct in a real workflow, we transform the process scheme into a simpler scheme for blocking. We use the same approach as with the other constructs and unfold the iteration primitive into single linear paths. The number of possible paths is determined by the business rules ("loop conditions"). The number can be large, even in a non-theoretical workflow, even with an abstracted data domain and single value representatives per equivalence class (as produced from the preceding abstraction step).

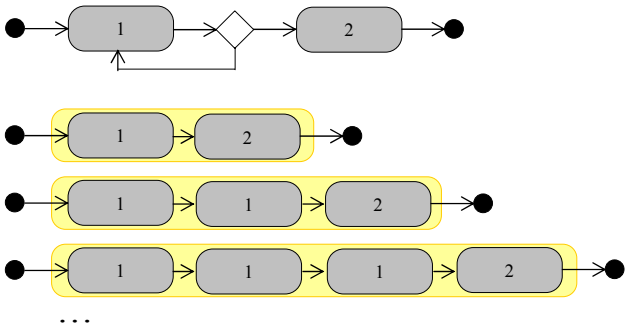

Figure 6. Iteration transformation and blocking.

Our approach is to bundle equivalence classes for iterations so that as many "traces" through the loop as possible fall into the same equivalence category. We start at the general and known approach to leave out sub-paths from the transformed iteration that are passed more than k times. We argue for our validation purposes that a sub-path that is included in a related larger path needs to be looked at only once, and so we set $k = 1$ (the example given later demonstrates the application of the idea). Together with the iterative sequence blocking in our approach, we still have the possibility to explicitly include also sub-paths that were identified as business critical within the loop, if they are included in a larger path (as suggested in the second and third line in Fig. 6). So we established a basis for selecting the critical passes that are needed for inclusion as validation scenarios.

Note that for our higher-order testing of reuse specifications we can omit unsolvable cases from information theory (cf. e.g. halting problem).

With the transformation and blocking procedures we can construct paths through the abstracted domain that are (i) part of the domain under consideration, (ii) critical for the customer and (iii) linear, without branches and without cycles. We call such a path a "Sunshine Path". Sunshine Paths can be serialized by construction, because they are a linear sequence of process steps, or transactions, which produce the same result as in the originating graph, if they were completely selected. They now go into the inclusion step as building blocks for end-to-end validation scenarios.

C. Example

The example is non-fictitious on the domain side and is taken from a large company's business rules and processes for the creation of credit items. Fig. 7 shows one function out of the process diagram and the relevant business rules for the "authorization level ok?" process step. A credit item has been recorded by an agent of the company at this stage. Now it needs formal release. Everyone involved in the process belongs unambiguously to a certain role, and all roles have limits for releasing (rel) a recorded credit item depending on its amount. If the credit amount is above the role's limit, it is not released by the role but instead submitted (sub) to the next superior role. Above a certain amount, any credit needs release by two different authorized roles (rel-s, rel). The two highest roles are entitled to release all credits. In the described process, credits that shall not be released remain in an undefined, or submitted, state.

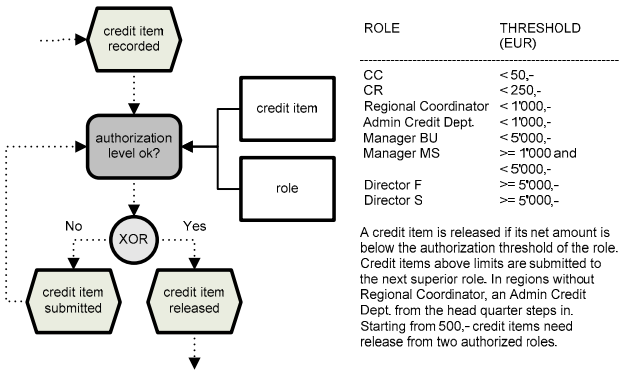

Figure 7. Domain model excerpt.

Abstraction maps the original domain model onto an equivalent domain model with defined discrete partitions and distinct value representatives per partition. The example results in seven partitions shown in Table I together with their values. If the analyzed customer domain section does not define any observable behavior, e.g. for partition P_1 in this example, then tests cannot be derived from this part of the domain model.

Further simplification of the scheme and its business rules by transformation and blocking is not necessary in the simple example. Reduction can readily identify the "traces" or, data sequences that are critical and reasonable for testing from the full set of possible sequences, from an end user validation point of view.

TABLE I.
PARTITIONS AND VALUES

Partition	Value	Partition	Value
$P_1 =]-\infty, 0]$	$e_1 = -1$	$P_5 = [500, 1000[$	$e_5 = 500$
$P_2 =]0, 50[$	$e_2 = 25$	$P_6 = [1000, 5000[$	$e_6 = 1000$
$P_3 = [50, 250[$	$e_3 = 50$	$P_7 = [5000, \infty[$	$e_7 = 5001$
$P_4 = [250, 500[$	$e_4 = 250$		

We restrict to demonstrating positive test sequences here, negative test sequences work according to the same principle. The iteration on the example can be reduced from an end user's business perspective to sequences starting at the least empowered call center (cc) role, which will subsequently cover also superior roles with suitable partition values (i.e. no explicit check for $k > 1$ in a first approach). This reduction results in Table II listing ten Sunshine Path sequences, the building blocks for critical business scenarios through the domain section. These sequences are now eligible for inclusion, also with critical sequences from other, interconnected domain parts, to build end-to-end branch-free business scenarios. The approach to connect sequences is the same as it was shown for the steps within a domain part. Joining two scenarios becomes possible by using the preceding scenario's output as the subsequent scenario's input. Inclusion criteria, again, are fully domain centric.

TABLE II.
"SUNSHINE PATH" SEQUENCES

Role	cca	cr	rc	acd	mbu	mms	df	ds
S_1	e_2 rel	-	-	-	-	-	-	-
S_2	e_3 sub	e_3 rel	-	-	-	-	-	-
S_3	e_4 sub	e_4 sub	e_4 rel	-	-	-	-	-
S_4	e_4 sub	e_4 sub	-	e_4 rel	-	-	-	-
S_5	e_5 sub	e_5 sub	e_5 rel-s	-	e_5 rel	-	-	-
S_6	e_5 sub	e_5 sub	-	e_5 rel-s	e_5 rel	-	-	-
S_7	e_6 sub	e_6 sub	e_6 sub	-	e_6 rel-s	e_6 rel	-	-
S_8	e_6 sub	e_6 sub	-	e_6 sub	e_6 rel-s	e_6 rel	-	-
S_9	e_7 sub	e_7 sub	e_7 sub	-	e_7 sub	-	e_7 rel-s	e_7 rel
S_{10}	e_7 sub	e_7 sub	-	e_7 sub	e_7 sub	-	e_7 rel-s	e_7 rel

To demonstrate how we check a specification artifact on the basis of the sequences from Table II, it is assumed that a software provider has specified and offered a fictitious `Comparator` software component. Next to other levels and facts, the behavior of this software artifact is described in OCL (Object Constraint Language) [23], and a `checkGE` service ("greater or equal") is defined according to Fig. 8. It is also specified for the `Comparator` component, on the respective layer of a multi-level reuse specification, that a `limits` relation maps a `value` to an `actor`.

To check the behavior specified in Fig. 8 against customer requirements given as critical sequences, the constraints from the supplier's specification are now numerically compared with one ore more branch-free scenarios. As described, such a scenario can be one path

through several subsequent critical customer sequences from interrelated domain parts that are assembled and validated together.

We assume in this example that our customer includes one single Sunshine Path, S_4 from Table II. So we can restrict our example to demonstrate this single sequence. In natural language, S_4 follows a recorded credit item of 250.- from a region without regional coordinator role. The credit item is (i) beyond the credit authorization limit of the call center role and therefore submitted to the customer representative role. It is (ii) beyond the credit authorization limit of the customer representative role and therefore submitted to the administrator credit department role. It is (iii) within the credit authorization limit of the administrator credit department role and released. Validation of this sequence is done by systematically walking through the OCL constraints from Fig. 8.

In step (i) the first two preconditions hold: `val` is 250.- and `act` is cca. The third precondition also holds: once the mapping table is set up with role descriptions and thresholds from the domain model, then cca will be found in the limits relation. If the preconditions hold as described, the specification's postcondition will evaluate (50 >= 250) and return false. The work flow can identify this with the meaning that the credit item is not released, and return to the "authorization level ok?" function with a "credit item submitted" state.

In step (ii) the first two preconditions hold: `val` is 250 and `act` is cr. As in the previous step the third precondition also holds for cr. If the preconditions hold as described, the specification's postcondition will evaluate (250 >= 250) and return true. The work flow can identify this with the meaning that the credit item is released, and continue to further parts of the domain model with a "credit item released" state.

In step (iii) the first two preconditions hold: `val` is 250 and `act` is acd. As in the previous steps the third precondition also holds for acd. If the preconditions hold as described, the specification's postcondition will evaluate (1000 >= 250) and return true. The work flow can identify this with the meaning that the credit item is released, and continue to further parts of the domain model with a "credit item released" state.

```
context Comparator::checkGE( val:Real,
                             act:String ):Boolean

pre:
( oclIsUndefined( val )=false )
  and
( oclIsUndefined( act )=false )
  and
  self.limits->
  exists( actor:String | actor=act )

post:
if self.limits->
     select( actor:String | actor=act ).value >= val
  then result=true   -- greater or equal
  else result=false  -- not (greater or equal)
endif
```

Figure 8. Behavioral specification artifact (OCL).

Thus, on the bottom line, validation of the `Comparator` component vs. S_4 using ARIval revealed a problem. While steps (i) and (iii) can be performed correctly by the specified software, in step (ii) the `Comparator` component fails check vs. the business rules. In the domain model and its critical sequence S_4, the credit item of 250.- is not released by a customer representative but instead submitted to be checked by the superior role. In the `Comparator` component, the validation shows that the credit item of 250.- is actually released by the customer representative role, which is inconsistent with the requirements from the domain model.

Possible consequences of this result could include looking for a `checkG` service ("greater") of the `Comparator` component, or changing the business rules slightly, or others. In any case the small but on the domain side non-fictitious validation example has shown that the proposed method gives an early hint at the necessity of a respective, aware decision and provides tangible support for it, without using any actual software.

IV. Related Work

Component software testing theory has become a large area of scientific research [38]. Important existing approaches with relation to our method have been selected and are shown in Table III to demarcate original contributions of the method.

TABLE III.
Related Approaches

Approach	**Component (Program)** verification	**Composition (Architecture)** ver. & val.	**Context (Domain)** validation
Built-in test technology	X		
Formal methods	X	(X)	
Scenario-/model-based testing	(X)	(X)	(X)
Specification matching	X		(X)
Tabular notation	X		
Test / composition languages	(X)	X	
Test input data sampling	X	(X)	
Test output oracles	X		
This Approach (ARIval)		(X)	X

Lines in Table III list the examined approaches which are further described below. Columns list three abstraction levels: component, composition and context. On component level formal program verification of single components with their interfaces is typical research focus. On composition level research from formal and less formal areas deals with architectures of several integrated components. On context level research focus is on the requirements side and less formal, concerned with system architectures in their socio-technical domain and business context. The availability of an approach for different abstraction levels is indicated in the cells. Our method's research contributions on the domain level or context

level are: embedding into a clear business model, independent domain based test oracles, and early domain level testing before software is available. The analyzed existing approaches don't seem to cover this.

A. Built-in Test Technologies

Built-in technologies for self-testing software components, in analogy to built-in tests from integrated circuits, have extensively been researched, e.g. in the Component+ project of the European Union [10]. Built-in tests come within the component, e.g. as additional test services, and are not intended to represent independent customer specific automation requirements but basic technical checks. Tests built into the component by their vendors are complementary to our domain centric axiom.

B. Formal Methods

Especially in formal model checking, plenty of verification approaches ("are we building the software right?") are discussed, among them the interesting domain reduction abstraction [8]. The method proposed here transfers some of the ideas to the domain validation ("are we building the right software?") viewpoint. But fully formal approaches for real components are prevented by computational effort with real systems in practice, decidability problems from computer theory, the absence of complete formal specifications, and the lack of a justifying business case or public interest. Also, formal verification can still be wrong. Formal verification methods provide valuable insight but in a practical sense don't apply to our complex domain level validation.

C. Scenario Based and Model Based Testing

Scenarios can be seen as special entities within the more general notion of a model. In model based testing, test references are generated from a model of the actual system. Many model based test approaches build upon the UML (Unified Modeling Language) today, and derive test references from UML diagrams [6;25]. Test references in existing approaches are built from artifacts within the component software development – models, design scenarios, etc. – and not from independent and unknown customer requirements as proposed here. Few if any approaches have yet addressed these model independency issues and its test implications, as does our method on the domain validation side.

D. Specification Matching

Existing approaches are based on fully formal language specifications, focus strongly on technical aspects, and are restricted to the matching of relatively simple functions [21;41]. Semi-formal matching methods from library science have also been described since long [29;31], and discussions exist to automatically extract classification attributes from natural language descriptions [16]. Further investigations include in particular relaxations of exact matching, and also contextual refinement theory [11]. Discussions started only recently that focus on more complex business domain perspectives for compatibility considerations of multi-layered specifications [43]. Our method goes beyond formal technical aspects and aims at checking specifications vs. higher-order requirements represented by domain level scenarios.

E. Tabular Notation

This approach aims at representing requirements fully formal by using a comprehensible, mathematically precise tabular notation of predicate logic for partial functions [27]. Domain requirements are successively translated into this tabular form, with promising first practical results [2]. Tabular notation seems very formal for "good enough" testing as intended in our method.

F. Test and Composition Languages

Similar to well known specification languages such as Z or OCL, special languages for testing and for composition have been proposed. One example on the testing side is TTCN 3 for test execution [14]. An example on the composition side is the Piccola calculus for formal component composition [1]. Test languages make implicit assumptions on their domains and their intended use, and have proven successful for testing software in their respective target areas. Architectural composition languages are formal and powerful but don't seem suitable for defining and evaluating actual test scenarios. Our method suggests a generic, widely applicable domain validation method without actual software but based on reuse specifications.

G. Test input data sampling

Exhaustive testing on all possible inputs is infeasible in general and inappropriate in particular for large real life enterprise applications. Hence an incomplete but appropriate test has to be determined. Existing approaches achieve this by sampling a domain of the input data according to fault hypotheses i.e. assumptions about which aspects or entities are error prone, allowing the test to reveal as many failures as possible with a minimum effort [3]. In our method, tests are generated not from fault hypotheses within the technological software system or its specification or models, but instead independently from the actual customer's ontological domain and its automation requirements which are unknown to, and detached from, the component software technology provider.

H. Test output oracles

The test oracle question [34;39] relates to outputs produced by a test: if the actual results differ from the expected results, did a proper test run produce wrong results revealing a software error, or were the expected results and/or the testing and/or basic assumptions wrong in the first place? Particular test outputs need careful analysis if the oracle grounds on the same model as the software [30]. Related issues can be observed in the controversial discussions of N-version programming in the 1980s. Sophisticated approaches such as e.g. [15] exist today. Our method instead sets priority to tests created independently from a software user, to deliver the independent oracle and the final judgment about an expected feature of a reused component.

V. Summary and Conclusions

Compositional reuse for industry style software production is an important approach pursued to master the ever increasing demands on software intensive systems. Testing black box software components from large repositories for their suitability to be reused in an actual end user situation is among the problems that complicate this approach. The associated validation activities are supported by the ARIval method, offering to the component demand side a domain centric component validation approach. The approach has some core advantages: it is derived from a clear business model assumption, sources test oracles from business domain requirements independent from the technological development process, and produces tangible results early, before the executable software is available, on the basis of suppliers' reuse specifications.

We demonstrated the principle in an example which is non-fictitious on the domain side. By constructing critical scenarios via abstraction, reduction and inclusion from a domain model, we obtain branch-free Sunshine Paths of automation sequences deemed validation critical on the domain level of the demand side. These scenarios represent references against which relevant levels from multi-dimensional supplier black box specifications can be checked very early in the compositional development process, and with oracles that are independent from this development process.

With our approach we support early and independent higher-order black box component software testing on the demand side in industrialized software processes. This can benefit software component customers through earlier and better testing within further decomposed division of work as required for industrialized software engineering processes.

References

[1] F. Achermann and O. Nierstrasz, "A calculus for reasoning about software composition," *Theoretical Computer Science*, vol. 331, pp. 367-396, 2-3 2005.

[2] R. L. Baber, D. L. Parnas, S. A. Vilkomir, P. Harrison and T. O'Connor, "Disciplined Methods of Software Specification: A Case Study," in International Symposium on Information Technology: Coding and Computing (ITCC 2005), 4.-6.IV.2005. Las Vegas, Nv, USA: IEEE Computer Society, 2005, pp. 428-437.

[3] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* New York, NY, USA: John Wiley & Sons, 1995.

[4] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, vol. 4, pp. 41-49, 2 1987.

[5] B. W. Boehm, "The Future of Software Processes," in Unifying the Software Process Spectrum: International Software Process Workshop (SPW 2005): Revised Selected Papers, 25.-27.V.2006. Beijing, China: Springer, 2006, pp. 10-24.

[6] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Journal of Software and Systems Modeling*, vol. 1, pp. 10-42, 1 2002.

[7] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, pp. 10-19, 4 1987.

[8] Y. Choi and M. Heimdahl, "Model Checking Software Requirement Specifications using Domain Reduction Abstraction," in 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6.-10.X.2003. Montreal, Canada: IEEE Computer Society, 2003, pp. 314-317.

[9] J. Dietz, *Enterprise Ontology: Theory and Methodology.* Berlin, Germany: Springer, 2006.

[10] H. Edler and J. Hörnstein, "Component+ Final report 1.1.," from http://www.component-plus.org/pdf/reports/Final report 1.1.pdf, download on 12.X.2005.

[11] C. J. Fidge, "Contextual Matching of Software Library Components," in 9th Asia-Pacific Software Engineering Conference (APSEC'02), 4.-6.XII.2002. Gold Coast, Australia: IEEE Computer Society, 2002, pp. 297-306.

[12] J. Z. Gao, H.-S. J. Tsao and Y. Wu, *Testing and quality assurance for component-based software.* Boston, Ma, USA: Artech House, 2003.

[13] J. Gordijn and H. Akkermans, "Designing and Evaluating E-Business Models," *IEEE Intelligent Systems*, vol. 16, pp. 11-17, 4 2001.

[14] J. Grabowski et al., "An introduction to the testing and test control notation (TTCN-3)," *Computer Networks*, vol. 42, pp. 375-403, 3 2003.

[15] O. Hummel and C. Atkinson, "Automated Harvesting of Test Oracles for Reliability Testing," in 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), 26.-28.VII.2005. Edinburgh, Scotland, UK: IEEE Computer Society, 2005, pp. 196-202.

[16] Y. S. Maarek, D. M. Berry and G. E. Kaiser, "An Information Retrieval Approach For Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering*, vol. 17, pp. 800-813, 8 1991.

[17] M. D. McIlroy, "Mass Produced Software Components," in Software Engineering: Report on a conference sponsored by the NATO Science Committee, 7.-11.X.1968. Garmisch, Germany: NATO Scientific Affairs Division, 1968, pp. 138-155.

[18] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, pp. 40-51, 10 1992.

[19] B. Meyer, "The Grand Challenge of Trusted Components," in 25th International Conference on Software Engineering (ICSE 2003), 3.-10.V.2003. Portland, Or, USA: IEEE Computer Society, 2003, pp. 660-667.

[20] H. Mili, F. Mili and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, vol. 21, pp. 528-562, 6 1995.

[21] A. Moormann Zaremski and J. M. Wing, "Specification Matching of Software Components," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 333-369, 4 1997.

[22] G. J. Myers, *The Art of Software Testing.* New York, NY, USA: John Wiley & Sons, 1979.

[23] Object Management Group, "UML 2.0. OCL Specification," from http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, download on 24.X.2006.

[24] Object Management Group, "Unified Modeling Language: Superstructure version 2.0," from http://www.omg.org/docs/formal/05-07-04.pdf, download on 31.III.2006.
[25] A. J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in The Unified Modeling Language: Beyond the Standard: Second International Conference (UML'99), 28.-30.X.1999. Fort Collins, Co, USA: Springer, 1999, pp. 416-429.
[26] S. Overhage, *Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UNSCOM Spezifikations-rahmen und Anwendung.* Dissertation, Universität Augsburg, 2006. In German.
[27] D. L. Parnas, "Predicate Logic for Software Engineering," *IEEE Transactions on Software Engineering*, vol. 19, pp. 856-862, 9 1993.
[28] D. L. Parnas, "Software Aspects of Strategic Defense Systems," in *Software Fundamentals: Collected Papers by David L. Parnas / edited by Daniel M. Hoffmann and David M. Weiss*, D. M. Hoffman and D. M. Weiss, Eds. Boston, Ma, USA: Addison-Wesley, 2001, pp 497-518.
[29] J. Penix and P. Alexander, "Efficient Specification-Based Component Retrieval," *Automated Software Engineering*, vol. 6, pp. 139-170, 2 1999.
[30] A. Pretschner and J. Philipps, "Methodological Issues in Model-Based Testing," in *Model-Based Testing of Reactive Systems: Advanced Lectures*, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, Eds. Berlin, Germany: Springer, 2005, pp 281-291.
[31] R. Prieto-Díaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, pp. 6-16, 1 1987.
[32] O. Skroch, "Validation of Component-based Software with a Customer Centric Domain Level Approach," in 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'07), Doctoral Symposium, 26.-29.III.2007. Tucson, Az, USA: IEEE Computer Society, 2007, pp. 459-466.
[33] C. Szyperski, D. Gruntz and S. Murer, *Component Software: Beyond Object-Oriented Programming.* London, England: 2nd Edition, Addison-Wesley, 2002.
[34] A. Turing, "Systems Of Logic Based On Ordinals," *Proceedings of the London Mathematical Society Ser. 2*, vol. 45, pp. 161-228, 1939.
[35] K. Turowski, *Fachkomponenten: Komponenten-basierte betriebliche Anwendungssysteme.* Aachen, Germany: Shaker, 2003. In German.
[36] K. Turowski, Ed. *Standardized Specification of Business Components*. Augsburg: GI Working Group 5.10.3 Component Oriented Business Application Systems, 2002.
[37] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, pp. 5-51, 1 2003.
[38] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto and W. E. Wong, "Component-Based Software: An Overview of Testing," in *Component-Based Software Quality: Methods and Techniques*, A. Cechich, M. Piattini and A. Vallecillo, Eds. Berlin, Germany: Springer, 2003, pp 99-127.
[39] E. J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, pp. 465-470, 4 1982.
[40] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, vol. 15, pp. 54-59, 5 1998.
[41] D. M. Yellin and R. E. Strom, "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 292-333, 2 1997.
[42] E. Yourdon, "When good enough software is best," *IEEE Software*, vol. 12, pp. 79-81, 3 1995.
[43] J. Zaha, "Automated compatibility tests for business related aspects of software components," in On the Move Federated Conferences (OTM 2004) PhD Symposium, 25.-29.X.2004. Larnaca, Cyprus: Springer, 2004, pp. 834-841.

Oliver Skroch was born in Frankfurt am Main, Germany in 1967. He holds a diploma degree in Business Sciences from Universität Augsburg, Germany. His major fields of study were production and logistics, operations research, statistics and mathematical methods.

He works world-wide for institutional investors, in high-tech and communications industries, in industrial and scientific R&D and in consulting. He was responsible for many large software engineering projects in technical and managerial functions. He is an independent consultant and expert witness in Augsburg, Germany. He gives lectures at the Chair of Business Informatics and Systems Engineering, Universität Augsburg, where he also works on his doctoral thesis. His research interests include software engineering for components & services.

Mr. Skroch is member of the IEEE Computer Society. He was repeatedly invited as reviewer and co-reviewer for conferences and journals, and as speaker at national and international industry and business events.

Klaus Turowski was born in Bruchsal, Germany in 1966. He holds a diploma degree in Industrial Engineering and Management from Technische Universität Karlsruhe, Germany and a doctoral degree in Business Information Systems from Universität Münster, Germany.

He had assignments in Germany at Technische Universität Darmstadt and Universität Konstanz, and as assistant professor at Universität Magdeburg. He was visiting professor at the University of the German Federal Armed Forces Munich and at the University of Tartu, Estonia. He holds the Chair of Business Informatics and Systems Engineering at Universität Augsburg and is the current dean of the Business Sciences faculty. His research interests include component & service engineering, inter-organizational systems and mobile commerce.

Prof. Dr. Turowski is spokesman for two working groups of the German Informatics Society (GI), founder of conference series, has presented his work at a variety of international conferences, and published articles in a number of journals.