Verifying Conceptual Domain Models with Human Computation: A Case Study in Software Engineering

Marta Sabou, Dietmar Winkler, Peter Penzerstadler, Stefan Biffl
Technical University of Vienna,
Favoritenstrasse 9-11, 1040 Vienna, Austria

Abstract

Conceptual domain models, such as taxonomies, knowledge graphs or Extended Entity Relationship (EER) diagrams are core to all information systems. The task of verifying the correctness of these models is of high interest to the knowledge and software engineering communities and attracted the first solution approaches using human computation. Yet, since these solutions are published within the boundaries of their communities, there is a lack of concerted work on this topic. As a first step to alleviate this status quo, we formalize the problem of verifying conceptual models and propose a generic approach (VeriCoM) to solve it with human computation techniques. We show how VeriCoM was applied in a software engineering use case focusing on verifying the correctness of an EER diagram against a system specification document. An evaluation of VeriCoM in a series of four workshops within one controlled experiment performed with a crowd of semi-experts lead to the identification of a set of defects with precision of 73% and a recall from a Gold Standard defect set of 63%.

Introduction

Information systems heavily rely on several *conceptual domain models* during their creation and operation. In *Software Engineering (SE)*, models, such as Extended Entity Relationship (EER) diagrams, diagram variants based on the Unified Modelling Language (UML) or Petri nets, play an important role in various software engineering life cycle phases (Brambilla, Cabot, and Wimmer 2012). Models can be used as foundation for building software artifacts and products, e.g., for code and test case generation, or detailed planning documents based on high level or abstract models. In *Knowledge Engineering* ontologies, taxonomies, and knowledge graphs are conceptual representations of the domain in which the information system is used and often act as schema for storing domain data (Pan et al. 2017).

The quality of the conceptual models underlying information systems is success-critical for later usage, because defects can have a major impact on the quality of the resulting artifacts (e.g., on code, test cases, or more detailed planning documents). Therefore, ensuring the quality and the correctness of these models is an important research topic, addressed in diverse research fields.

In Knowledge Engineering, *ontology evaluation* focuses on *"checking the technical quality of an ontology against a frame of reference"* such as a gold standard ontology, a representative domain corpus, a specification document, or general human knowledge (Sabou and Fernández 2012). Some evaluation tasks can be automatically performed: reasoning can assess the logical consistency of the model based on the semantics of the encoding language; lexical comparison can approximate domain coverage with respect to a corpus. Yet, a subset of model evaluation tasks require human input. This is the case of *ontology verification* which *"compares the ontology against the ontology specification document, thus ensuring that the ontology is built correctly (in compliance with the specification)"* (Sabou and Fernández 2012).

Similarly, in Software Engineering, besides automated model verification approaches based on language semantics, *Software Inspection (SI)* is a well-established approach (Aurum, Petersson, and Wohlin 2002) that supports defect detection of various document types early in the software life cycle by making use of human expertise from a group of inspectors (Fagan 1976). During software model inspection, inspectors check whether a conceptual model correctly and completely represents the content of a reference document (Laitenberger and DeBaud 2000).

However, the manual verification of conceptual models by experts is time-consuming, costly and faces several challenges even when following well-established procedures such as in software inspection, e.g., difficulty to cover the entire inspection object during typical sessions of two hours, high costs and time needed for coordinating the inspection group without dedicated tool support (Winkler et al. 2017c). In this context, human computation techniques promise, at a minimum, tool support and a better coordination of the model verification process as well as its outsourcing to crowds of layman or semi-experts.

Since the problem of verifying conceptual models is of high importance to at least two research communities, a number of approaches have successfully used human computation techniques to address this problem by enlisting crowds of experts or layman. However, because these approaches were mostly proposed within the boundaries of their own research areas, there is a lack of understanding

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Table 1: Overview of related work.

Paper	Evaluated Elements	Frame Of Reference	Selection	Defect Types
(Acosta et al. 2016) - Expert	Data Triples	Human Knowledge	Random/Manual	Defect Taxonomy
(Acosta et al. 2016) - Crowd	Data Triples	Human Knowledge	Random	3 or 2 defects
(Mortensen et al. 2015; 2016)	Subsumption Relations	Human Knowledge	Filtering	Binary
(Wohlgenannt et al. 2016)	Terms, Relations	Human Knowledge	None	Binary
(Sun et al. 2016)	Taxonomy	N/A	N/A	N/A
(Winkler et al. 2017a; 2017b)	EER Model	System Specification	Guided	Open ended

of their commonalities and differences. This hampers a concerted effort on addressing the model verification problem with human computation, the exchange of ideas between communities, the comparison of solutions and exchange of data e.g., through benchmarking. Therefore, our research question in this paper is:

RQ: How to propose a human computation based solution to the problem of conceptual model verification that is applicable across research areas?

To address our research question, we *generalize the problem of conceptual model verification* across the communities of knowledge and software engineering. To that end, we offer the following novel contributions:

A formalization of the conceptual model verification problem applicable across the fields of knowledge and software engineering which can serve as a bases for defining human computation-based solutions.

An approach for the **Veri**fication of **Co**nceptual **M**odels (**VeriCoM**) with human computation techniques that relies on the generic problem formalization. VeriCoM introduces the idea of splitting the complex task of verifying a model into individual tasks focused on verifying one model element at a time. Additionally, it guides workers towards identifying predefined defect types.

An evaluation of VeriCoM in a software engineering use case where we verified an EER diagram against a software specification document. We tested the approach in a series of four workshops within one controlled experiment with 53 semi-experts and found that it can lead to detecting defect sets with precision values of 73% and a coverage of over 60% over a manually identified defect set.

Next, we present related work, the generic problem formalization and the VeriCoM approach. We describe our software engineering use case and its experimental evaluation before concluding with lessons learned and future work.

Related Work

Representative work on evaluating conceptual models with human computation is shown in Table 1 and discussed next. In Knowledge Engineering, and especially in the *Semantic Web*, several knowledge management tasks require human contributions (Bernstein et al. 2014). Accordingly, a number of studies have successfully applied human computation to evaluating conceptual models specific to knowledge engineering, such as ontologies or knowledge graphs.

In (Acosta et al. 2016), authors report on evaluating the quality of triples from *Linked Data Knowledge Graphs*, namely *DBpedia* (Auer et al. 2007), by enlisting both experts and layman crowds. The goal is to identify quality issues frequent in DBpedia triples. During expert-sourcing, experts can opt for one of three strategies for selecting the data that they verify: (a) random suggestion of data, (b) working on data from a selected class, or (c) manual selection of data. Experts are asked to assign to each evaluated triple one quality issue from a pre-defined taxonomy of issues. For the crowdsourced experiments, data is selected randomly. In a first stage, workers select one of three possible quality issues that could apply for a given triple (i.e., value, link, datatype), then in a second phase they solve tasks in jobs dedicated to individual quality issues and specify whether a triple is correct/incorrect with respect to that issue.

Mortensen et al. (2015, 2016) report on using crowdsourcing for verifying the correctness of subsumption relations in large, domain-specific ontologies such as *SNOMED* and the *Gene Ontology*. Given the large size of these medical ontologies, their entire verification is not feasible. Instead, authors focus on evaluating non-asserted (i.e., derived), non-trivial and direct relations. The verification is performed on *CrowdFlower*, where a task contains the relation to be evaluated described in natural languages and the definition of concepts connected by that relation. Workers are asked to make a binary choice on the correctness of the relation and provide an explanation. For *SNOMED*, comparison with a baseline expert evaluation showed that crowds have comparable agreement rates with experts, and therefore can function as a scalable assistant in ontology engineering.

In order to bring crowdsourcing closer to the work of ontology engineers, the Protégé ontology editor (Musen 2015) was extended with a plugin that allows the crowdsourcing of ontology engineering tasks from within the editor (Wohlgenannt, Sabou, and Hanika 2016). Some of these tasks enable the evaluation of the correctness of ontology elements, namely: (a) the relevance of a term for a domain of interest or (b) the correctness of relations. The selection of the elements to be evaluated depends on the user of the plugin. The automatically generated crowdsourcing tasks collect binary decisions on the correctness of the model elements.

Differently from the approaches mentioned above, (Sun et al. 2016) aim to evaluate entire taxonomies in terms of how well they support user navigation. They evaluate the quality of the taxonomy as a function of the quality of the task that it enables (i.e., efficient navigation indicated by reduced time of navigation and number of clicks). Their approach is an example of a *task-based evaluation*, originating from the ontology evaluation area (Porzel and Malaka 2004).

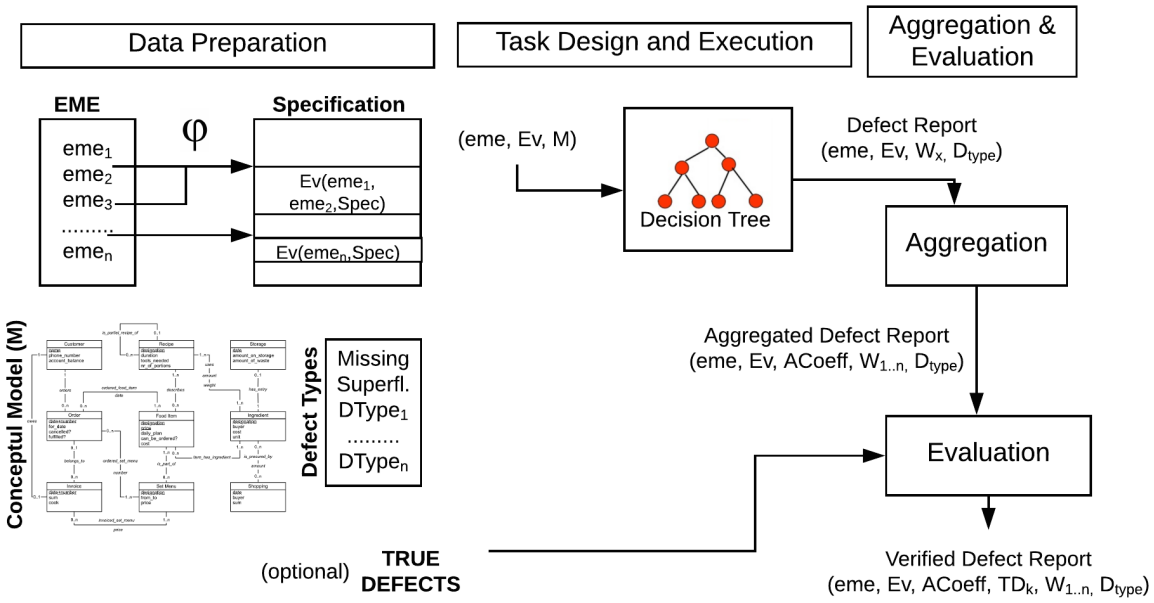


Figure 1: The VeriCoM Approach.

In *Software Engineering*, crowdsourcing techniques are wide-spread and support all stages of the software engineering life-cycle (LaToza and van der Hoek 2016), (Mao et al. 2017). However, work on the verification of software specific conceptual models is limited to our earlier work, in which we verified Extended Entity Relationship (EER) diagrams with respect to a systems specification (Winkler et al. 2017a; 2017b). The selection of the model elements to verify was guided by the content of the specification, and workers provided defects as part of open ended tasks, leading to overly complex aggregation.

We conclude that the problem of evaluating conceptual models with human computation has been addressed in both the knowledge and software engineering communities. Most approaches report on an evaluation where the frame of reference is generic human knowledge. Techniques for selecting the model elements to be evaluated differ across approaches and mostly resolve to selecting a subset of the model, often randomly. Most approaches collect binary decisions about the correctness of a model element. By formalizing the problem of verifying conceptual models at a generic level, we propose an approach which explores in more depth selection strategies and collecting multiple defect types.

Problem Formalization

The task of conceptual model verification with respect to a frame of reference is formally a function (γ) that applied to the model (M) and the frame of reference (FR) leads to the identification of a set of defects (D).

$$\gamma(M, FR) \; ! \; D \tag{1}$$

We consider a *conceptual domain model (M)* to be a collection of *model elements (me)*, $M = ME$. Let ME_1 ... ME_n be sets of model elements of different types, such that ($M = \lceil_n ME_n$). Minimally, M is the union of at least two model element sets, $M = ME_C \lceil ME_R$, where ME_C represents the concepts in a domain and ME_R their relations. M can be more complex, containing other types of model elements such as concept attributes, relation attributes, or instances in the domain.

For example, the EER model in Figure 2(a) captures the specification of a restaurant management system. It includes concepts such as *customer*, *order* or *invoice*, and relations between these concepts, for example, *(customer, orders, order)* or *(customer, owes, invoice)*. Concepts and relations can have attributes, for instance, *cutomer.name* is a concept attribute while *(recipe, uses, ingredient).amount* is an attribute of a relation.

The model is verified with respect to a *Frame of Reference (FR)*, which is a complementary knowledge source that incorporates (approximately) the same domain knowledge as the model. Examples of FRs are system specifications, document corpora representative for the domain but also general (common sense) knowledge. One type of frame of reference that is often used to support model verification consists of textual specifications ($Spec$). We will focus on verification tasks guided by such textual specifications.

Let an *expected model element (eme)* be a model element mentioned in $Spec$ and expected to be modelled in M. The collection of all emes (EME) represents the building blocks of the model to be verified (e.g., entities, attributes, relations, and relation attributes). The task of assigning each eme to a representative evidence from $Spec$ is:

$$\varphi(EME, Spec) \; ! \; EV_{EME;Spec} \tag{2}$$

The set EME overlaps, but must not be identical with the set of model elements in M. For example, M might contain entities that were not mentioned in the specifications and are therefore superfluous. Should the two sets be identical, that

would mean that the model completely addresses the specification. The intersection of the expected and the actually modelled model elements contains all those emes for which an equivalent model element me exists (i.e., the same as or a synonym of eme denoted with).

$$EME \setminus ME = \{eme \mid \exists me \in ME \wedge eme \quad me\} \quad (3)$$

We define a *relevant evidence for an* eme within a specification $Spec$ as an arbitrary long text chunk from $Spec$ where the eme is mentioned ($Ev_{eme;Spec}$). The evidence should contain information that is representative for describing the eme and the role it should play in the model M. Let $EV_{EME;Spec}$ represent the collection of $Spec$ evidences for all eme instances of EME.

The output of the model verification task ($\gamma(M, FR)$) is a set of *defects (D)* in the model M with respect to the $Spec$. Each defect $d \in D$ refers to whether and how one eme is modelled in model M. Defects can be of several types, depending on how the (expected) model element appears in $Spec$ and in M. Defects of type *Missing* should be declared when an eme is contained in the specification but not modeled in the model, and therefore, it refers to elements from the set $EME \cap ME$. Defects of type *Superfluous* refer to elements modelled in M that are not mentioned in $Spec$, that is $ME \cap EME$. Besides these defect types, other domain specific defect types might be identified as exemplified in our use case.

The VeriCoM Approach

Based on the formalization above, we propose a generic approach for **Veri**fying **Co**nceptual **M**odels (VeriCoM) with respect to a textual specification using human computation (see Figure 1). In VeriCoM, the EME set plays a central role in selecting the parts of the model that are inspected. The main stages of VeriCoM are:

1. Data Preparation starts with identifying the conceptual model M to be verified as well as a representative specification $Spec$ which can be used as a basis for the verification. Next, the focus of the verification needs to be decided, that is, which model element types will be verified (e.g., ME_C, ME_R).

1.1. Identification of EME. Within $Spec$, a set of mentioned model elements needs to be identified which are *expected* to appear in the model M. For shorter specifications this step can be performed manually, but larger specifications will require an automation of this step by relying on natural language processing techniques, crowdsourcing or a combination thereof. In other work, not reported in this paper, we have performed experiments of crowdsourcing this task and obtained encouraging first results (Sabou, Winkler, and Petrovic 2018).

1.2. Identification of $EV_{EME;Spec}$. For assigning each eme to a representative evidence from $Spec$, the function φ should lead to a set of evidences that are (a) representative for each eme; (b) small enough to be amenable for use within human computation tasks; and at the same time (c) capable of conveying the necessary context to the workers. For smaller specifications, this function can be implemented manually. For larger specifications, automatic techniques based on natural language processing should be developed. Here an important decision lies in how to automatically detect a relevant evidence(s) for a given eme. Some options are: (a) assigning the evidence where the eme is first mentioned; (b) assigning the evidence (e.g., a paragraph) where the eme is most often mentioned; or (c) instead of choosing a single representative evidence, selecting all evidences in which an eme is mentioned.

1.3 Definition of Defect Types. Beside the domain independent defect types defined during problem formalization, in this step defect types should be identified that are typical for the given domain. Being aware of concrete defect types enables creating task interfaces that can guide workers towards identifying these defect types, thus reducing the number of defects that need to be provide in free-text. This has a positive influence on the aggregation process as typed defects are easier to aggregate than free-text defects.

2.Task Design and Execution. For an eme and a relevant evidence from the specification, the task should enable the detection of various defect types. In Table 2 we depict the decision table underlying a proposed task design. For each eme and the corresponding evidence, the task design guides the worker's judgment process as follows (an example task interface based on this decision table is shown in Figure 2b):

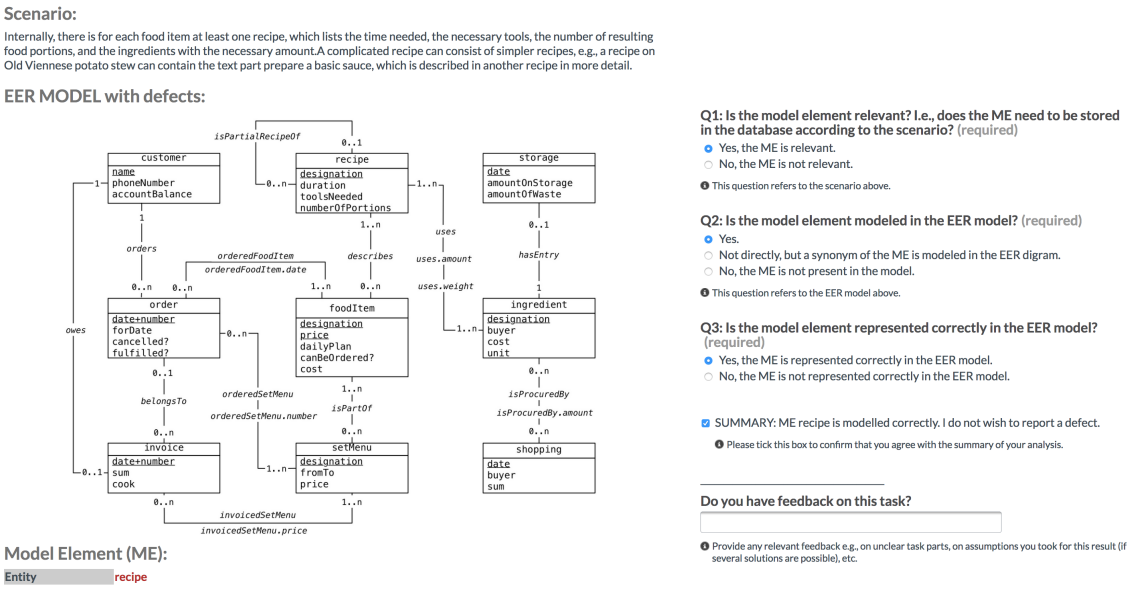
Relevance: First, the worker is asked to judge based on the evidence whether the eme is relevant and should be represented in the model.

Representation: Next the worker turns his attention to the model itself and locates the eme therein. We distinguish three cases: (a) when the eme or (b) a synonym thereof is represented in M; (c) when the eme is missing from M. If a non-relevant eme appears in M as itself or as a synonym then a *superfluous* defect has been identified. Non-relevant *eme*s that are not in M lead to the conclusion that the model is correct. If the eme was judged relevant but was not found in M, then a *missing* defect is registered. If however a relevant eme is in the model (as itself or a synonym), then it's correctness is judged next.

Correctness and Interpretation: Relevant emes (or their synonyms) appearing in M are evaluated on whether they are modelled correctly. If the modelling is found incorrect a defect should be reported either as free text or a typed defect specific to the application domain.

The output of the human computation task is a collection of individual *Defect Reports (DR)*. These are quadruples connecting an eme and its evidence to a defect type based on the judgment of individual workers (W_x): $DR(eme, Ev_{(eme;Spec)}, W_x, D_{type})$.

3. Aggregation. Defects reported for an eme by n workers are aggregated in order to identify a final defect type, using an output agreement strategy. For each defect type reported for that eme, we compute an agreement coefficient ($ACoeff$) as the inter-rater agreement on that defect type. The defect type with the highest $ACoeff$ (and, optionally, above a threshold value) is selected as the final defect type for the eme. If there is a tie between multiple defect types, then the aggregation labels this defect as *Undecided*. Aggregation for free-text defects must be performed manually



(a) (b)

Figure 2: Model verification task containing (a) the evidence scenario, the EER model and model element and (b) questions for guiding the verification process.

Table 2: Decision table underlying task design.

EME Relevant?	EME in Model?	EME Correct?	Interpretation
Relevant	In Model	Correct	Correct
		Not Correct	Defect
	In Model as Synonym	Correct	Correct
		Not Correct	Defect
	Not in Model	Defect: Missing	
Not Relevant	In Model	Defect: Superfluous	
	In Model as Synonym	Defect: Superfluous	
	Not in Model	Correct	

or within a follow-up human computation task. The output of aggregation are Aggregated Defect Reports denoted as: $ADR(eme, Ev_{(eme;Spec)}, ACoeff, W_{1::n}, D_{type})$.

4. Evaluation. The goal of this step is to evaluate the quality of the defect detection process in order to identify whether it is satisfactory or whether it needs to be improved. Ideally, the collected defects ADRs could be matched to a collection of *true defects (TDs)* (i.e., known defects) and recall and precision metrics could be computed. However, in the lack of such a gold standard, the resulting defect reports could be manually evaluated (this will shed light on the precision of the verification γ but not on its recall). Manual assessment is time consuming, therefore variations might be considered here: for example, only defect reports with high $ACoeff$ might be selected for manual evaluation. The output of this stage consists of verified defect reports, possibly aligned to a true defect (TD_k): $VDR(eme, Ev_{(eme;Spec)}, ACoeff, TD_k, W_{1::n}, D_{type})$

Software Engineering Use Case

In this section we exemplify how we adapted VeriCoM to a Software Engineering use case, where the correctness and completeness of an EER model needs to be checked with respect to a textual specification (i.e., a reference document).

1. Data Preparation. Our use case relies on the following elements[1]. *Spec* is a textual requirements specification describing typical processes in the context of a restaurant management system. The document is 3 pages long, is written in English and consists of an introduction section and 7 paragraphs each focusing on diverse aspects of the system to be designed, e.g., order management, cooking management, accounting. We refer to these paragraphs as *scenarios*. For instance, a scenario related to "food items and set menu management" is displayed at the top-right of Figure 2a. This requirements specification represents the *reference document* and was considered to be correct.

The *conceptual model* M under investigation is an EER diagram including 9 entities, 13 relationships, and 32 attributes. The model is displayed in Figure 2a. Formally, $M = M_E \; [\; M_R \; [\; M_{EA} \; [\; M_{RA} \; [\; M_{RM}$, where:

M_E is the set of all entities, e.g., *customer, order, invoice*;

M_R is the set of relations declared between entities, e.g., *(customer, orders, order)*;

M_{EA} is the set of all entity attributes, e.g., *customer.name*;

M_{RA} is the set of all relation attributes, e.g., *(customer, orders, order).date*;

[1]Use case data can be shared with interested researchers, but will not be made public to avoid disclosing defects to students taking part in future replication studies.

M_{RM} is the set of all relation multiplicities, e.g., *customer(1), orders, order(0..n))*.

The goal of the use case is to verify the modelling of all five model element types. The experiment team, i.e., the authors of this paper, included 21 defects into this model representing typical defect types in the context of software engineering, e.g., missing entities, incorrect attributes, and wrong/missing relations and relation multiplicities. These *true defects* are used for evaluating the performance of the VeriCoM approach.

1.1. EME identification was performed manually given the small size of the specification and lead to the identification of 120 emes. We also experimented with crowdsourcing eme identification and obtained encouraging results (Sabou, Winkler, and Petrovic 2018).

1.2. Identification of $EV_{EME;Spec}$ was also performed manually. For the use case at hand, we observed that the 7 scenarios of $Spec$ were self-contained and offered the right trade-off between being sufficiently succinct to be part of a task while still offering sufficient context to the workers. We therefore decided in using scenarios as representative evidence. For assigning an eme to a scenario we created an occurrence matrix between the elements of the EME and the seven scenarios. We assigned each eme to the scenario where it was mentioned the most often.

1.3 Definition of defect types. Two defect types that frequently surface in EERs are: (1) assigning the wrong key to an entity (keys are underlined entity attributes in Figure 2a); or (2) specifying an incorrect multiplicity for relations. Accordingly, for our use case we identified four defect types: *Missing, Superfluous, WrongKey, WrongRelM*. Defects other than these types are of type *Wrong*.

2.Task design and execution. We used *CrowdFlower* as a crowdsourcing engine. We designed a task where workers were given the model M, an eme and a relevant evidence for the eme ($Ev_{eme;Spec}$) and asked to verify the modelling of the eme (see top-part of the task interface in Figure 2a). Because the specification's 7 scenarios were used as evidence, tasks referring to the same scenario were grouped within one job to lower the cognitive overhead of switching between scenarios. The general description of the system to be designed (i.e., the introductory part of $Spec$) was provided as part of the task's instructions for overall context.

Workers are guided in the verification process by a set of questions which appear following the logic depicted in Table 2 (see bottom-part of the task interface in Figure 2b). When a defect of other type than *Missing* or *Superfluous* is reported, the interface will provide different questions depending on the type of the eme. Namely, if the eme is an entity attribute, the worker will be able to choose between the defect type *WrongKey* or specifying another defect as free text. If the eme is of type relation multiplicity, the interface allows selecting the predefined defect type *WrongRelM* or specifying another defect of undefined type (e.g., *Wrong*). For defects specified as free text a controlled language is provided. This controlled language is explained in the task instructions and reminded under the relevant input text-box.

When a node in the decision tree from Table 2 is reached, the *CrowdFlower* task interface automatically generates a *summary* of the evaluation based on the selections made. At this point the worker can either accept this summary or revisit his answers to change the final summary.

3. Aggregation. The aggregation of the judgments collected is performed for each *eme* following the algorithm of the general VeriCoM approach. If the final defect type is *Wrong* then the individual responses are manually checked to identify whether the workers identified the same or different defects.

4. Evaluation. We leveraged the available collection of true defects to evaluate the defect detection process. To that end, we matched the aggregated defect reports to the gold standard of true defects, based on eme and defect type.

Experimental Evaluation of VeriCoM

To evaluate the performance of VeriCoM, we set up a controlled experiment in a medium-scale classroom setting and followed the basic process approach according to Wohlin *et al.* (Wohlin et al. 2012). Because of laboratory space restrictions we split the experiment over 4 workshops following the same setup.

Experiment Setup

Participants were undergraduate students enrolled in a software quality assurance course at our university in fall 2017. We opted for an internal crowd of students as opposed to layman crowds for two reasons. Firstly, model verification requires specific software engineering knowledge (e.g., best practices about entity keys) that is not general human knowledge and we doubted it can be successfully collected in crowdsourcing platforms. Having said so, future work will investigate whether model verification could be split in ways that could also benefit from layman contribution in terms of general domain knowledge (e.g., about restaurants in our use case). Second, as students were learning about model verification as part of their course, this experiment allowed them to practice their skills and to get feedback on their overall performance in the style of learning analytics (Ferguson 2012). Participation in the experiment was voluntary and attracted 53 students over all workshops. The study participants had a qualification level equivalent to junior software engineering professionals. In each workshop participants were assigned randomly to two study groups.

Figure 3 presents the experiment setup replicated over four workshops. The setup contained four main stages:

1. A preparation stage (prior to the workshops) extended the data preparation described previously by setting up the *CrowdFlower* jobs. We split the data into 12 batches, with 10 emes per batch that had the same scenario as evidence. We created 12 jobs based on this data (one for each batch).
2. A tutorial (30 min) that explained the goal of the experiment and how to execute the experiment tasks (the *CrowdFlower* jobs). The content of the tutorial was also summarized in the job instructions.
3. The model verification task where workers identified and reported defects as part of executing the *CrowdFlower*

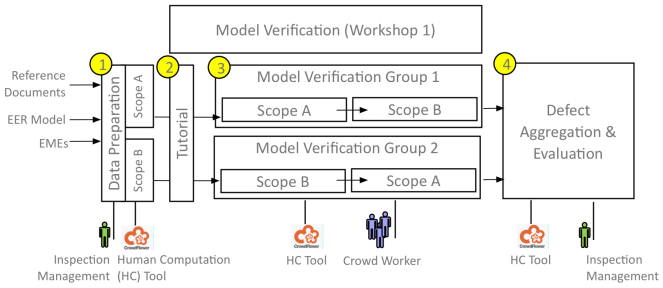


Figure 3: Experiment Setup.

Table 3: Overview of experimental results.

	WS1	WS2	WS3	WS4	WS1-4
Defect Reports	284	568	300	390	1542
Aggregated Defect Reports					
Undecided	19	13	32	13	3
NoDefect	78	78	71	80	89
Missing	12	7	8	9	7
Superfluous	3	7	2	3	5
Wrong	2	5	2	1	2(5)
WrongKey	2	3	0	1	2
WrongRelM	4	7	5	13	11
TotalDefects	23	29	17	27	27 (30)
Verified Defect Reports					
TruePositives	13	16	9	16	22
FalsePositives	10	13	8	11	8
MatchedTDs	10	12	8	12	14
Precision	57%	55%	53%	59%	73%
Recall	48%	57%	38%	57%	67%

jobs. Each student was randomly assigned to thee jobs. We applied a cross over design with the 2 student groups focusing on different parts of *Spec*, i.e., Scope A and Scope B. While group 1 focused on Scope A (max. 60 min) followed by Scope B (max. 60 min), group 2 executed the same tasks in a reverse order.

4. A defect aggregation and evaluation stage, executed by the inspection management according to the general process and as explained in the "Results" section. The experimental data was stored in a database that allowed automating the aggregation and parts of the evaluation.

Study Material included an experience questionnaire to capture previous experience of participants and feedback questionnaires with focus on feedback on individual tasks. Furthermore, we provided guidelines (as hardcopies) to guide the participants during their specific tasks. The overall experiment process was supported by an *Experiment Management System* hat holds relevant information for all groups, e.g., providing links to questionnaires and experiment tasks in the sequence to be solved. Study material includes a *Spec* and a *conceptual model M* (see details in the data preparation part of the software engineering use case).

Tool configuration. The participants received individual tasks for defect detection in different steps of the experiment process via *Crowdflower*. For more details please refer to the *Task Design* section.

Variables. Independent variables include the study treatment (i.e., the Scope of the inspection process), the set of *emes* (derived from the requirements specification), the set of seeded defects in the EER model (i.e., the true defects), defect types, and tool configurations. Dependent variables include the number of reported and true defects, precision, and recall of the model verification process.

Data Collection and Analysis. We collected experience and feedback questionnaire by using an online questionnaire in *Google Forms* and experimental data on defect detection in the *CrowdFlower* application. For data analysis we exported the *CrowdFlower* data in a relational database to run queries based on SQL statements.

Results

Table 3 sums up the data collected during the experiment runs over the four individual workshops (W1-W4), as well as during the entire experiment (W1-4). The first part of the table displays the number of defect reports (individual judgments) received, showing different numbers per workshop because the number of participants varied across workshops.

Analysis of aggregated defect reports. The number of *aggregated defect reports* is captured in the second half of Table 3. Workshops with more judgments tend to have less undecided results, although a clear correlation between the number of judgments and undecided results cannot be observed. All workshops lead to similar number of defects for each defect types. As shown in Figure 4, defects of type *Missing* and *WrongRelM* are identified most often, while defects of type *WrongKey* are the least frequent.

For the defects of type *Wrong*, often more than one defect was reported, especially when combining the data across workshops. For example, for *customer*, from the 19 judgments received during the four workshops of the experiment, 6 identified that *customer.contactAdress* is missing and 3 suggested that *customer.name* was a wrong key. In these cases, we decided to count an individual defect if it was suggested by at least 3 workers. In WS1-4, we had one defect covering 2 individual defects and another defect referring to 3 individual defects. Therefore, in Table 3 we denote this case with 2 (5). The total number of identified defects ranges between 17 and 30 across workshops.

Quality of the defect set. The total defect sets were evaluated with respect to a gold standard of 21 true defects (last part of Table 3).

True positives are defects that could be mapped to a true defect (i.e., they referred to the same eme and had the same type). Most of these mappings were obtained automatically and checked manually for correctness. The rest of the defects were aligned manually to the gold standard in the following three cases.

First, relation multiplicity defects were declared for $emes$ of type relation, instead of type relation multiplicity, thus hampering the automatic mapping.

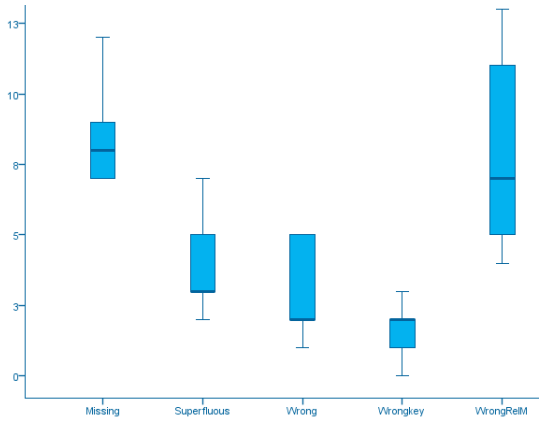


Figure 4: Defect types discovered in the workshops.

Second, we manually evaluated the defects of type *Wrong* to understand whether they are valid defects.

Third, some defect reports were declared for synonyms of the emes for which a true defect existed in the Gold standard. For example, true defect D22 specifies that *order.takeout?* is *Missing*. In our output we found this defect (which was matched automatically), as well as a second defect referring to the synonym *order.togo*. The second defect had to be mapped manually to D22. Since in some cases like this two defect reports were matched to the same true defect, the number of true positives is higher than the number of matched true defects.

The remaining defects are *false positives* and fall into the following three categories.

First, there are defect reports that are obviously wrong, e.g., reporting that an eme depicted in M is missing.

Second, superfluous defects reported cases when elements were in the model but were not mentioned in the supporting evidence. We found that all superfluous defects stem from the fact that the focus model element was not mentioned in the evidence text, but it appeared in M. This is therefore a side-effect of the data preparation process and not a failure of the workers. Having said that, we consider these cases as false positives with respect to the gold standard.

Third, some of the defect reports recommended alternative modeling approaches rather than reporting defects. E.g., for *setMenu.from_to* we received a defect of type *Wrong* with workers suggesting splitting this attribute into two attributes *setMenu.from* and *setMenu.to*. Such cases were also considered false positives with respect to the gold standard, although they yield interesting suggestions of improved modelling.

Based on the evaluation of the defect reports, we compute precision (proportion of *TruePositives* in *TotalDefects*) and recall (proportion of *MatchedTDs* over 21 true defects). Even with the strict decision of counting superfluous and alternative modelings as false positives, the precision of the defect set ranges from 53% to a maximum of 73% when combining the data of all workshops. The coverage of the gold standard (i.e., recall) ranged from 38% to 67%.

Coverage of true defects. Table 4 depicts which true defects were identified in each workshop. Six of the 21 defects were identified in all workshops (4 of type Missing and one of each of the other two types), while others were identified only in individual workshops or not at all (D37, D82, D61, D72). All defects of the type *WrongRelM* were identified in at least two workshops, while the identification of *WrongKeys* and *Missing* defect types proved more difficult.

The four defects which were not identified in any of the workshops include:

D37 which refers to the missing relation attribute *(invoice,invoicedFoodItem,foodItem).number*, and was only considered missing by 1 in 8 judgments. Workers might have been confused here since the entire relation in missing from the model and therefore reporting a missing attribute does in fact not make much sense.

D82 refers to the missing entity attribute *ingredient.category*. This eme was judged in conjunction with an evidence where it did not appear, therefore, workers judged it as not relevant and not in the model, which resulted in a *NoDefect* judgment. Therefore, incorrect input data lead to this defect not being identified.

The fact that *setMenu.designation* and *storage.date* were wrong keys (**D61, D72**), was only identified by 3 in 17 workers (5 times in 16 judgment respectively), probably because these terms do not appear per se in the specification, but are rather implied.

Limitations. This paper reports on a part of a controlled experiment with focus on defect detection and validation with human based computing approaches. The second aspect of the controlled experiment focuses on traditional pen-and-paper inspection as benchmark for precision and recall evaluation. However, in context of this paper, this aspect is out of scope. For the overall experiment design please refer to (Winkler et al. 2017a; 2017b).

Threats to validity. To address internal validity threats, we did not allow communication between inspectors during the study execution. The overall net duration was limited to 120 min. Individual breaks were allowed with break periods reported. Prior experience of participants was captured. We applied a random assignment of participants to the study groups. The experiment package was intensively reviewed by experts in pilot studies to identify and fix errors. Further, we executed a set of pilot runs to ensure the feasibility of the study design. To address external threats to validity, we selected a well-known application domain, a restaurant setting. We provided a tutorial to familiarize the participants with the method and technology.

Conclusions and Future Work

Motivated by the importance of the conceptual model verification for the knowledge and software engineering communities, as well as by the publication of the first approaches to solve this problem with human computation, in this paper we aim to generalize this problem across research areas in a

Table 4: Overview of True Defects identified in workshops.

	Defect Type:Missing									Defect Type:Wrong Key					Defect Type:Wrong Relation Multiplicity						
	D22	**D12**	**D21**	**D25**	**D31**	**D33**	**D34**	**D37**	**D82**	**D53**	**D92**	**D11**	**D61**	**D72**	**D14**	**D73**	**D26**	**D43**	**D23**	**D36**	**D42**
WS1	2	1	1	1		1	1			1	2				1	2					
WS2	2	1	1	1						2	1	1			2	1	1	1		1	
WS3	2	1	1	1	1					1						1		1			
WS4	2	1	1	1						1					2	1	2	1	2	1	1
WS1-4	2	2	1	1						2	1	1			2	1	2	1	2	1	1
Total	10	6	5	5	1	1	1	0	0	7	4	2	0	0	7	6	5	4	4	3	2

first effort to enable a more concerted work on this topic. To that end, we propose a first generic formalization of the verification problem and the VeriCoM approach for solving it with human computation, with particular focus on splitting the model verification task and guiding inspectors towards detecting a-priori defined defect types. We then illustrate the use of VeriCoM in a software engineering use case, where we achieve high precision of the defect set (73%) and a recall of 67% with respect to a manually curated set of true defects. We derive the following observations:

The use of $emes$ for splitting the verification of M promises to enable a good coverage of the model. Yet, this has a shortcoming for elements that are in the model but not in the specification ($M \cap EME$), as their verification is circumvented (no tasks are created to verify them). Choosing the intersection of model elements and expected model elements as basis for task splitting should be investigated. On a more generic level, if a $Spec$ is not available, model elements could be used for task splitting.

The right model element granularity is an open issue. We observed tendencies of workers to identify errors at the level of entities and relations rather than more fine-grained elements, such as attributes and multiplicities. An open research question therefore refers to the most suitable element granularity level for human computation.

The identification of a suitable evidence is crucial. Errors in this task introduced by our manual approach hampered the quality of the final defect set. Automating this task and experimenting with different approaches to it is important future work. More generally, for problem instances where a $Spec$ is not available, alternative evidence sources can be explored (for example, concept definitions derived from third party resources when verifying ontologies as shown by (Mortensen et al. 2015)).

Defect type-based guidance of the verification steps is a good middle ground between providing free-text defect reports that are hard to aggregate or constraining the choice of workers to binary inputs. Yet, in its current form, VeriCoM falls short of collecting *modelling alternatives* and distinguishing these from defects.

Future work will focus on extending and better formalizing the problem of conceptual model verification. For example, the decision table shown in Table 2 will be revised and extended to be conceptually grounded in research on the ontological issues of conceptual models (Wand and Weber 1993). We also aim to formally represent the verified model as well as our approach in terms of knowledge representation languages developed in the context of the Semantic Web research area (e.g., OWL [2]). This will enable the use of model checking engines (also called reasoners) for checking the syntactic correctness of the model and providing continuous feedback to workers during model verification (rather than only in a batch-mode, after the verification task). Such usage of the reasoning capabilities of Semantic Web technologies in Human Computation tasks is promising, but was only weakly addressed so far (Sabou et al. 2018).

In tandem with improving the formalization level of the approach, we plan on applying VeriCoM to other types of models or other problem cases too, e.g., from ontology engineering, in order to test and improve its generality.

In terms of the design of the task interface, while the question-based interface allows easy identification of defects, it might be vulnerable to question answering bias. Future work will therefore experiment with open-ended style interfaces to understand the degree of bias introduced otherwise.

We also plan to extend VeriCoM to assess worker reliability in order to identify potential low-performing workers. Finally, we are interested in investigating how to capture differences in modelling alternatives between workers by extending disagreement capturing mechanisms proposed for other task types (e.g., annotation) (Inel and Aroyo 2017).

Acknowledgments

We thank the 53 students that took part in the experiments. This work was performed in the context of the CitySPIN project, funded by FFG under grant number 861213.

References

Acosta, M.; Zaveri, A.; Simperl, E.; Kontokostas, D.; Flöck, F.; and Lehmann, J. 2016. Detecting Linked Data Quality Issues via Crowdsourcing: A DBpedia Study. *Semantic Web Journal*.

Auer, S.; Bizer, C.; Kobilarov, G.; Lehmann, J.; and Ives, Z. 2007. DBpedia: A Nucleus for a Web of Open Data. In *In 6th Int. Semantic Web Conference, Busan, Korea*, 11–15. Springer.

Aurum, A.; Petersson, H.; and Wohlin, C. 2002. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability* 12(3):133–154.

[2] https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

Bernstein, A.; Leimeister, J. M.; Noy, N.; Sarasua, C.; and Simperl, E. 2014. Crowdsourcing and the Semantic Web (Dagstuhl Seminar 14282). *Dagstuhl Reports* 4(7):25–51.

Brambilla, M.; Cabot, J.; and Wimmer, M. 2012. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering* 1(1):1–182.

Fagan, M. E. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.* 15(3):182–211.

Ferguson, R. 2012. Learning Analytics: Drivers, Developments and Challenges. *Int. J. Technol. Enhanc. Learn.* 4(5/6):304–317.

Inel, O., and Aroyo, L. 2017. Harnessing Diversity in Crowds and Machines for Better NER Performance. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I*, 289–304.

Laitenberger, O., and DeBaud, J.-M. 2000. An encompassing life cycle centric survey of software inspection. *Journal of systems and software* 50(1):5–31.

LaToza, T. D., and van der Hoek, A. 2016. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Softw.* 33(1):74–80.

Mao, K.; Capra, L.; Harman, M.; and Jia, Y. 2017. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software* 126:57–84.

Mortensen, J. M.; Minty, E. P.; Januszyk, M.; Sweeney, T. E.; Rector, A. L.; Noy, N. F.; and Musen, M. A. 2015. Using the wisdom of the crowds to find critical errors in biomedical ontologies: a study of SNOMED CT. *Journal of American Medical Informatics (JAMIA)* 22(3):640–648.

Mortensen, J. M.; Telis, N.; Hughey, J. J.; Fan-Minogue, H.; Auken, K. V.; Dumontier, M.; and Musen, M. A. 2016. Is the crowd better as an assistant or a replacement in ontology engineering? an exploration through the lens of the gene ontology. *Journal of Biomedical Informatics* 60:199–209.

Musen, M. A. 2015. The Protégé Project: A Look Back and a Look Forward. *AI Matters* 1(4):4–12.

Pan, J. Z.; Vetere, G.; Gomez-Perez, J. M.; and Wu, H., eds. 2017. *Exploiting Linked Data and Knowledge Graphs in Large Organisations*. Springer.

Porzel, R., and Malaka, R. 2004. A Task-based Approach for Ontology Evaluation. In *Proc. of ECAI 2004 Workshop on Ontology Learning and Population*.

Sabou, M., and Fernández, M. 2012. Ontology (network) evaluation. In Suárez-Figueroa, M. C.; Gómez-Pérez, A.; Motta, E.; and Gangemi, A., eds., *Ontology Engineering in a Networked World.* Springer. 193–212.

Sabou, M.; Aroyo, L.; Bozzon, A.; and Qarout, R. K. 2018. Semantic Web and Human Computation: the Status of an Emerging Field. *Semantic Web* 9(3):1–12.

Sabou, M.; Winkler, D.; and Petrovic, S. 2018. Expert sourcing to support the identification of model elements in system descriptions. In Winkler, D.; Biffl, S.; and Bergsmann, J., eds., *Software Quality: Methods and Tools for Better Software and Systems*, 83–99. Cham: Springer International Publishing.

Sun, Y.; Singla, A.; Yan, T.; Krause, A.; and Fox, D. 2016. Evaluating Task-Dependent Taxonomies for Navigation. In *In Forth AAAI Conference on Human Computation and Crowdsourcing (HCOMP)*, 229 – 238.

Wand, Y., and Weber, R. 1993. On the ontological expressiveness of information systems analysis and design grammars. *Information Systems Journal* 3(4):217–237.

Winkler, D.; Sabou, M.; Petrovic, S.; Carneiro, G.; Kalinowski, M.; and Biffl, S. 2017a. Improving model inspection with crowdsourcing. In *2017 IEEE/ACM 4th International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*, 30–34.

Winkler, D.; Sabou, M.; Petrovic, S.; Carneiro, G.; Kalinowski, M.; and Biffl, S. 2017b. Improving model inspection processes with crowdsourcing: Findings from a controlled experiment. In Stolfa, J.; Stolfa, S.; O'Connor, R. V.; and Messnarz, R., eds., *Systems, Software and Services Process Improvement*, 125–137. Cham: Springer International Publishing.

Winkler, D.; Wimmer, M.; Berardinelli, L.; and Biffl, S. 2017c. *Towards Model Quality Assurance for Multi-Disciplinary Engineering*. Cham: Springer International Publishing. 433–457.

Wohlgenannt, G.; Sabou, M.; and Hanika, F. 2016. Crowd-based ontology engineering with the uComp Protégé plugin. *Semantic Web* 7(4):379–398.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; and Wesslén, A. 2012. *Experimentation in software engineering*. Springer Science & Business Media.