Surface Realization Architecture for Low-resourced African Languages

ZOLA MAHLAZA, Department of Informatics, University of Pretoria, South Africa, and Department of Computer Science, University of Cape Town, South Africa
C. MARIA KEET, Department of Computer Science, University of Cape Town, South Africa

There has been growing interest in building surface realization systems to support the automatic generation of text in African languages. Such tools focus on converting abstract representations of meaning to a text. Since African languages are low-resourced, economical use of resources and general maintainability are key considerations. However, there is no existing surface realizer architecture that possesses most of the maintainability characteristics (e.g., modularity, reusability, and analyzability) that will lead to maintainable software that can be used for the languages. Moreover, there is no consensus surface realization architecture created for other languages that can be adapted for the languages in question. In this work, we solve this by creating a novel surface realizer architecture suitable for low-resourced African languages that abides by the features of maintainable software. Its design comes after a granular analysis, classification, and comparison of the architectures used by 77 existing NLG systems. We compare our architecture to existing architectures and show that it supports the most features of a maintainable software product.

CCS Concepts: • **Computing methodologies** → **Natural language generation**; • **Software and its engineering** → **Software architectures;**

Additional Key Words and Phrases: Natural language generation, software architecture, low-resourced languages, surface realisation

ACM Reference format:
Zola Mahlaza and C. Maria Keet. 2023. Surface Realization Architecture for Low-resourced African Languages. *ACM Trans. Asian Low-Resour. Lang. Inf. Process.* 22, 3, Article 84 (March 2023), 26 pages.
https://doi.org/10.1145/3567594

1 INTRODUCTION

Natural Language Generation (NLG) systems are increasingly being deployed "in the wild" to support the generation of, among other things, financial reports, data-driven news stories, and product descriptions [23] in well-resourced languages, such as English and German. In recent years, we have also seen several efforts in building realization tools for languages that are either

This work was financially supported by the Hasso Plattner Institute for Digital Engineering through the HPI Research School at UCT and the National Research Foundation (NRF) of South Africa (Grant Number 120852).
Authors' addresses: Z. Mahlaza, Department of Informatics, University of Pretoria, Private Bag X20, Hatfield, Pretoria, Gauteng, South Africa, 0028, and Department of Computer Science, University of Cape Town, Private Bag X3, Rondebosch, Cape Town, Western Cape, South Africa, 7701; email: z.mahlaza@up.ac.za; C. M. Keet, Department of Computer Science, University of Cape Town, Private Bag X3, Rondebosch, Cape Town, Western Cape, South Africa, 7701; email: mkeet@cs.uct.ac.za.

2375-4699/2023/03-ART84 $15.00
https://doi.org/10.1145/3567594

low-resourced or have a low number of L1 speakers when compared to languages like English and German. Realization tools are responsible for only "converting abstract representations of sentences into the real text" [86, p. 49], whereas a full NLG system includes that and other modules that take care of other NLG tasks, such as content determination and sentence planning. For instance, there have been adaptations of the realization tool SimpleNLG [36] to Brazilian Portuguese [29] and Galician [34]. However, none of the efforts have focused on building support for **Niger-Congo B (NCB)** languages, a family of African languages. For those languages, there are a few attempts at building ad hoc realization components used in ontology verbalizers [16, 48], which have not made architectural design decisions a central component; hence, their maintainability is negatively affected. While implementation of the algorithms in software contributes to maintainability, it also depends on the properties of the underlying software architecture. Specifically, if certain maintainability properties, such as functional suitability [13], are not met, then the implementation is unlikely to be easy to maintain.

The languages in question are distinguished by a complex verb structure, noun classification, and an agreement system via morphemes. We will demonstrate these features using Byamugisha's [15] isiXhosa template for verbalizing the axiom $Teacher \sqsubseteq \exists teaches.Subject$. Their template produces the text *Wonke utishala nganye ufundisa isifundo nokuba sinye,* "Each teacher teaches at least one subject." The noun *Utishala,* "Teacher," belongs to class 1a of the 16 possible noun classes in isiXhosa. Noun class membership is governed by a variety of features [47]. The noun also governs the form of the verb *ufundisa,* "teaches," based on its class. If the plural form of it was used instead of the current form, then the verb would take the form *bafundisa,* "they teach"; if a dog (noun class 9) teaches its puppy, then the verb takes the form *ifundisa*; when relegating teaching to the computer (*ikhompyutha*, noun class 5), it is *lifundisa*. The prefix of the verb—*u-*, *ba-*, *i-*, and *li-* in the examples, respectively—is called the subject concord and is determined by the noun class of its governing noun. When forming the final verb, it is subject to phonological conditioning rules [64]. Similarly, the prefix of *sinye,* "one," the final word in Byamugisha's text, is governed by the noun *isifundo,* "subject." More details regarding verb structure can be found in, among others, [49, 52].

Similar to English, the noun-verb agreement can be managed in a "hacky" way: use *(u|ba)fundisa,* "teach(es)," where the brackets with divider are used to denote a choice between the two values for isiXhosa and an optional segment in the case of the English text. This becomes impractical when one must cater for some double-digit noun classes and even more so for transitive verbs where one must account for the verb's agreement with both its subject and object. Consider the same axiom not with "teaches" but the object property *-thanda,* "loves": the resulting verb in the generated text would be rendered as *(ndi|si|u|ba|i|li|a|si|zi|lu|bu|ku)(m|ba|wu|yi|li|wa|si|zi|lu|bu|ku|∅)thanda,* where the brackets sections are the possible values for the subject and object concords, respectively (∅ denotes the empty string). Of course, a computer can process this easily, but this cannot be given to humans as generated text to read. There are multiple such other grammatical complexities. Therefore, the NLG systems for generating NCB languages must combine template and grammar rules into so-called "grammar-infused templates" [65], such as [15, 66].

If there were a consensus architecture on what tasks should be in the surface realization module and how they should be organized, it possibly could be used as is or in an adapted form for Niger-Congo B languages, but no such architecture exists. The closest thing to a consensus is Reiter and Dale's three-step pipeline architecture [84, 86]. However, the **Reference Architecture for Generation Systems (RAGS)** [76] project showed that, in practice, researchers who choose the three-step architecture do not agree on where to place tasks between the modules [77]. For instance, while Reiter and Dale present the surface realization module as only being responsible

for ordering, several NLG systems also make lexicalization the module's responsibility. From an overview perspective, this means that the following gaps exist:

- Existing African language NLG systems do not use an architecture that yields realizers that are easy to maintain.
- There is no consensus on a maintainable surface realizer architecture within NLG that can be tweaked for NCB languages.

In this work, we fill the gaps found in existing work, with a direct focus on NCB African languages, by first conducting an extensive literature review and comparison of 77 contemporary surface realization systems. Instead of using the low-level tasks listed in [77, 86], we focus on five aspects: tactical decisions, structure encoding, structure induction, structure linearization, and candidate ranking. We chose to focus on these tasks based on our experience with the control issues and tight coupling of linearization rules by systems that we evaluated, including isiZulu verbalizer [51] and KPML [7]. The benefit of our approach vs. alternative approaches is that it emphasizes the use of formal specifications to improve the detection of inconsistencies and allow integration; it also emphasizes the reuse of the few limited resources and encourages modularity. We will elaborate on these decisions in Section 4. Our tasks differ from the ones used by [77, 86] in that they are more detailed and therefore enable a fine-grained analysis of surface realization.

Using this comparison of existing surface realizer modules, we identify their limitations for NCB languages and then proceed to the creation of a new architecture for surface realizers to support low-resourced languages, with particular emphasis on NCB languages that possess most of the maintainability characteristics specified in the software product quality model presented in [13] (namely modularity, componential re-usability, and analyzability). A maintainable surface realization architecture is important for any natural language but is amplified for low-resourced languages.

We compare our architecture against the 12 architectures used by the 77 reviewed NLG systems and demonstrate that our architecture supports the most maintainability features. Our architecture differs from existing ones as it introduces a pre-processing module that operates on sentential structures in order to introduce context-specific grammatical information; places the ranking module prior to linearization so that it can have access to explicit grammatical knowledge/annotations; uses an ontology to formalize templates and therewith allows analyzability; offers detection of logical inconsistencies in templates, sharing, and comparison of templates; and moves tactical decisions outside the realizer.

The remainder of the article is structured as follows: Section 2 discusses the current state of NLG for African languages, Section 3 discusses research that focuses on NLG system architectures, Section 4 zooms into surface realization to discuss our analysis of the current architectures used, Section 5 presents the new architecture, Section 6 compares the new architecture with existing architecture, Section 7 discusses the utility of using an ontology to capture template knowledge, Section 8 discusses a partial implementation of the architecture, and Section 9 concludes.

All supplementary material can be downloaded at https://github.com/AdeebNqo/ToCT/tree/main/OWLSIZ.

2 NLG FOR AFRICAN LANGUAGES

There are only two approaches that have been pursued for encoding the ordering structure in African language generation: the use of **computational grammar rules (CGRs)** and templates that may be combined with computational grammar rules to form what is called grammar-infused templates [65]. Grammar-infused templates differ from traditional templates in that they also include grammatical knowledge. A detailed description can be found in [65]. The grammar-only

approach has been pursued in the creation of a grammar engine for Arabic [1], but at the time of writing, it has not been used to create an NLG system. The templates, most of which are used together with CGRs, have been used for isiZulu language learning exercises [37] and ontology verbalization in Runyankore [16], isiZulu [50], and Afrikaans [87]. There are a few applications that support Southern African languages, but their surface realization approach, and their corresponding architectures, cannot be classified with certainty. For instance, AwezaMed relies on a controlled language,[1] but how they do this is not publicly available for inspection, only a general view of the system's architecture described in [69, 70].

NCB languages are a subset of African languages that are low-resourced. This refers to both the primary sources and the computational resources, as also demonstrated by the recent audit of South African NCB language resources [79]. As such, the development of computational grammar resources for the languages must be preceded by, or done in tandem with, linguistic documentation and analysis of the language's grammar. This is a labor-intensive task that is further complicated by the complexity of the language's grammar [17, 49, 64]. From a practical perspective, it is sensible to consider other techniques for the generation of text in these languages. Templates, on their own, have already been shown to be inadequate for these languages as a result of their complex agreement system (e.g., [50]), and unlike Machine Translation where researchers have mostly relied on religious, government, and phrasebook parallel corpora (e.g., [72, 82]), there has been no work that pursues fully data-driven solutions for NCB languages owing to a lack of training data. In general, the computational grammar rules, data-driven approaches, and templates on their own are not viable for generating text in Niger-Congo B languages since the languages are low-resourced and have complex grammar.

Grammar-infused templates can be created by hand in the same vein as simple templates, but they do not suffer the same limitations at handling grammatical complexity. In particular, the isiZulu and Runyankore verbalizers' [16, 48] patterns can capture some of the morphological agreement relations that exist between words in isiZulu and Runyankore. The patterns used in those verbalizers differ from simple templates with respect to two aspects: rules for specifying the source of a concord's value and rules for forming words from an ordered sequence of affixes. They are still limited because they only support a subset of the possible concords and duplicate the same morphological agreement rules across different templates, and their linearization algorithms are tightly coupled with the application domain (in casu, the verbalization of axioms). One cannot use them in a new NLG system that needs to generate a language whose features go beyond what they currently support.

The possible rise of the aforementioned grammar-infused templates, as is or otherwise, as a means for generating text in NCB languages raises questions about how best to organize the components of the realizer in order to prevent the tight coupling of grammar rules with the generation algorithm to allow resource reuse, to remove duplication of rules and hence lessen the manual effort, and to offer an explicit declaration of template concepts, relations, and constraints to ensure their analyzability.

3 EXISTING NLG ARCHITECTURES

Current work examining architectures in NLG focuses on entire systems and gives little attention to surface realizers. The names of the architectures, resulting from such work, are given in Table 1.

Existing architectures differ in modularization and organization of those modules, if they are present. For instance, systems that lack modules dedicated to specific tasks are called *integrated systems*, of which there are three types: planning-based, end-to-end, and classifier cascade

[1] https://repo.sadilar.org/handle/20.500.12185/536.

Table 1. Classification of Architectures Used by Complete NLG Systems (Based on [35, 88])

Integrated	**Modular**	
End-to-end	**Interleaved**	**Unidirectional**
	Interactive	What and How
Classifier cascades	Blackboard	Three-step pipeline
Planning based	Revision-based	Four-step pipeline
–	–	Generate and select

architectures. Planning-based architectures were mostly used by dated systems that viewed generation as a problem of searching for a sequence of actions, in some action-space, that can transform the initial state (i.e., the input) to some goal state (i.e., the output); end-to-end ones are very popular with extant deep learning systems (e.g., [32]); and classifier cascades are used in the uncommon NLG systems that treat generation as a classification task (e.g., [71]). While the planning-based methods and classifier cascades may have multiple parts, they are not modular in the same sense as NLG modularity as introduced in [86]: their actions or classifiers can "cut across the boundaries of many of the tasks that are normally encapsulated in the classic pipeline architecture, combining both tactical and strategic elements by viewing the problems of ['what to say'] and ['how to say it'] as part and parcel of the same set of operations" [35, p. 86]. Those two architectures, unlike end-to-end systems, are not black boxes, as it is possible to interpret the actions or classifiers. None of these three architectures have a separate module that transforms their internal specifications into natural language text; hence, it is impossible to isolate and analyze such a module's architecture and to reuse a module in another system. For instance, a modular approach allows the identification and resolution of problems pertaining to referring expression generation while keeping the other identifiable modules frozen, and that piece can be reused elsewhere as needed. Also, a modular approach reduces complexity by making the various dependencies between an NLG system's tasks visible (cf. a black-box approach that is not controllable).

Systems that use modular architectures are either interleaved or unidirectional and the only difference between them is the direction of data flow. Interleaved systems allow for a back and forth between modules to communicate errors detected by a downstream module back to the upstream module that caused them. These types of systems are rare, possibly due to the engineering difficulty associated with building them [88].

Modular architectures have an identifiable component that is responsible for surface realization. Such architectures can either have two (e.g., [57]), three (e.g., [86]), or four (e.g., [85]) modules. The two-step architecture starts by transforming the "input into a forest of possible expressions" [57], which are then fed into a ranking module. The widely used three-step pipeline architecture has document planning, micro-planning, and realization modules. The four-step architecture extends that by introducing a signal analysis and data interpretation module. The extent to which knowledge gained from one architecture can be transferred to other types of architectures is unknown, as there has been no examination and comparison of said components across the different architectures. To the best of our knowledge, the only architectural discussion surrounding the realizer is the distinction made between a tactical realizer vs. a grammar engine [36] and demonstration that there is no agreement on what low-level tasks should be in the surface realizer [77] for systems that use the three-step pipeline architecture. The difference between tactical generators and grammar engines is that the former "mak[e] appropriate linguistic choices given the semantic input [and] once tactical decisions have been taken, [they build] a syntactic representation, [apply] the right morphological operations, and [linearize] the sentence as a string" [36] while the latter focus "on the second of the two tasks, making no commitments as to how semantic inputs are

mapped to syntactic outputs" [36]. Both architectural analyses are limited because while Gatt and Reiter argue for a realizer that is responsible for "spell[ing] out the syntactic constraints of the language in which an utterance is to be generated" [91, p. 58], they do not make explicit what tasks are required to do so. Moreover, the architecture comparison by [77] only considers three-step architectures and it is high level with respect to the tasks it considers. For instance, there is no breakdown of the ordering task (i.e., "the choice of linear ordering of the elements of the text" [77, p. 4]) in order to identify differences in how realizers achieve it.

4 REVIEW OF SURFACE REALIZER ARCHITECTURES

We review existing surface realizers as they are found in multiple domains and eras in order to identify current and past architectural trends in finer detail. This is done to uncover knowledge that is useful for building maintainable architectures for NCB languages.

We compiled a list of 77 NLG systems and tools in preparation for the architecture analysis by extending the 54 verbalizers and tactical realizers found in [65] with 21 additional systems for a more comprehensive list. We added 9 grammar engines (SimpleNLG variations, GenDR [58], and JSrealB [78]), 12 data-driven NLG systems (10 of which are recent and 2 are dated), and 2 recent systems that use augmented template tools. The new systems were found by analyzing the work cited by the publications described in [65] and retrieving recent and relevant publications using Google Scholar. From this initial list, we manually read all the papers and then we removed systems whose paper(s) have insufficient details about the surface realizer's architecture to be able to determine how it works (e.g., because the paper's scope was different) and then manually determined the architectures used by the remaining systems.

4.1 Comparison Criteria and Results

The comparison focused on the following aspects:

- Structure selection: This is the tactical decisions component, which connects the surface realizer to the prior modules. It is responsible for making linguistic decisions given the semantic input (e.g., deciding on syntactical structure to use for a certain event type) [36]. We use the term "structure selection" instead of tactical decisions in order to improve clarity regarding the relationship of this task with the following three aspects (i.e., Structure encoding, induction, and linearization). Moreover, this task can be thought of as a kind of "structure determination"; however, we prefer to use the term "selection" to make clear that it involves the selection from an existing set of structures. Using SimpleNLG [36] as an example, the term "determination" may suggest that the NLG engineer is responsible for creating the English grammar rules for encoding sentential structure and using them at the same time. However, SimpleNLG already provides such rules, and the engineer is only selecting which ones are appropriate for certain semantic input to their system.
- Structure encoding: This is the method used to capture the sentential structure, which (1) captures the elements and position(s) of elements that will be inserted in order to form the final surface text and (2) specifies how the elements are to be ordered. For instance, some realizers use a template, while others may use a phrase structure tree.
- Structure induction: This refers to the method used to create the structures used for capturing sentences. These structures come in many forms. For instance, they could be phrase-structure trees or templates, among the many possible options. In some cases, the structures may be induced from examples, but they can also be induced/created manually by the NLG engineer based on their domain expertise.
- Structure linearization: This is the formation of text from some ordering structure. In the case of simple templates, it is equivalent to slot filling. In the case of grammar-infused

templates and grammar-only approaches, it may also include forming words from lemmas and traversing a tree in order to form the final surface texts.
- Ranking: The candidate filtering mechanism is responsible for selecting one sentence/structure out of many candidates' output sentences or sentential structures.

We have chosen the above tasks for analysis to zoom into overlooked issues and avoid limitations seen in architectures of previous generation systems. For instance, structure selection is important because analysis of the connection between the preceding module and the realizer provides us with an understanding for how to avoid the control issues present in dated wide-coverage tactical generators [36]. More precisely, such systems require a specialized input form since they make tactical decisions the responsibility of the realizer. As such, they offer no direct way for control over how "phrases are built and combined, inflectional morphological operations, and linearisation" [36, p. 91]. In addition, they allow us to scrutinize the placing of the linearization algorithm, and its combination with other aspects, *to avoid the need to* create an algorithm each time a new template is created (e.g., [16, 48]). Our analysis of the method used to encode the sentential structures is also important because some techniques (e.g., use of large-scale grammars) are not suitable for low-resourced languages. Lastly, we include the ranking task because it is intertwined with linearization in some NLG systems.

The aforementioned analysis, and an abstracting away of some details, resulted in 12 surface realizer architectures. The list of all systems considered for examination, excluding the ones filtered out, and their respective architectures are given in Table 2. The 12 architectures differ in the organization of the various modules and methods employed for structure induction, encoding, and linearization. We group these architectures into categories based on similarities in their organization of modules, which resulted in six categories that are illustrated in Figure 1. We have listed the architectures according to their categories in Table 3. The same table also lists the methods used for induction and linearization. While our use of the term "architecture" is in the usual sense, the term "architecture category" is not commonplace and we therefore will illustrate how to interpret Table 3 and Figure 1 using the system described by Lavoie and Rainbow [59]. It uses architecture 10 and, looking that up in column 2 of Table 3, it belongs to category AC4. Then, using Figure 1, it can be seen that its surface realizers only contain sentential structures and a structure linearization module. Table 3 also shows us that, unlike other architectures in the same category (e.g., arch. id 9), it relies on hand-coded sentential structures and uses grammar rules for linearization.

Categories AC2 and AC3 have modules labeled A and B. These represent the different ways that the two categories combine structure, linearization, and ranking and the various modules are listed in detail in Figure 2. To demonstrate, in category AC3 there is no ranking, and linearization either can be combined with selection in one module (A) or done separately (in A and B, respectively). In category AC2, structure linearization is either combined with selection in module A or with ranking in module B. Technically, we could have split these into separate architecture categories; however, doing so adds no explanatory value.

4.2 Architectures and Categories

In the following paragraphs, we discuss the architecture categories and provide examples of systems whose surface realizers belong to each of the categories. Our examples do not put emphasis on categories AC1–2 because they are only prevalent in dated NLG systems.

The AC1 category is used by so-called tactical generators; surface realizers are responsible for making linguistic decisions given semantic representations and applying linguistic rules to convert decided-upon syntactic structures to obtain surface text. This category of text generator was first identified by Gatt and Reiter [36] when they contrasted it to the notion of a grammar

Table 2. List of All Systems Whose Surface Realizers Were Classified and Their Corresponding Architecture Identifiers

Architecture id	Reference(s) to System
0	Aguado et al. [2], Bateman [7], Coch [22], Bouayad-Agha et al., Bouayad-Agha et al., [10, 11], Dongilli and Franconi [30], Lareau et al. [58]
1	Cimiano et al. [21]
2	Bohnet et al. [8]
3	Busemann [14]
4	Knight and Hatzivassiloglou [53], Langkilde [56]
5	Nakanishi et al. [81], White [95]
6	Angeli et al. [6], Kondadadi et al. [54]
7	Byamugisha et al. [16], Keet et al. [51], Lyudovyk and Weng [63]
8	Androutsopoulos et al. [4], Ang et al. [5], Camilleri et al. [18], Casteleiro et al. [19], Dannélls, Dannélls et al. [24, 25], Davis et al. [26], Davis et al. [28], Elhadad and Robin [33], Gruzitis et al. [39], Hewlett et al. [41], Hossain et al. [44], Jarrar et al. [45], Kaljurand and Fuchs [46], Liang et al. [60], Liang et al. [61], Lim and Halpin [62], McRoy et al. [75], Sanby et al. [87], Stenzhorn [89], Stevens et al. [90], Weal et al. [94]
9	Amith et al. [3], Bollmann [9], de Oliveira and Sripada [29], Dušek [31], Gatt and Reiter [36], Hielkema et al. [42], Kuanzhuo et al. [55], Mazzei et al. [73], Mazzei et al. [74], Molins and Lapalme [78], Ramos-Soto et al. [83], Vaudry and Lapalme [93]
10	Lavoie and Rainbow [59]
11	Mairesse et al. [68], Moryossef et al. [80], Wong [96]
12	Castro Ferreira et al.'s [20], Gubbala et al. [40], KnowledgeGraph Person verbaliser[2]

engine. In such an architecture, tactical decisions are conducted via rules and sentential structure encoding, and linearization is conducted via a large grammar. structure The grammar formalism varies across the various systems. An example of such a tactical generator, though it has other components such as an environment for creating grammars, is the KPML system [7] and other examples of surface realizers that rely on stratified theories such as **Meaning-Text Theory (MTT)** (e.g., [10, 58]). Such realizers take in some semantic structure and map them to some syntactic dependency using a variety of methods, and then said structures are converted, possibly through multiple steps, into the final surface form.

The AC2 category is like AC1 in that it makes use of a grammar for encoding sentential structure. However, it differs because it has a sentence ranking module. The AC2 architectures differ since some use a data-driven module and others use rules for structure linearization (see Table 3). An example is the generator in [56], which takes semantic input and uses it to create a forest via rules (a generation forest is a context-free representation that specifies all the possible sentences that can be generated). The forest's sub-trees are ranked and then linearized to produce the final sentence.

The AC3 category is like AC2, since one of its architectures (i.e., arch id 6) includes a ranking module. The category differs from AC2 as it relies on templates for structure encoding. An example

[2]https://github.com/m477301/KnowledgeGraphVerbalizer.

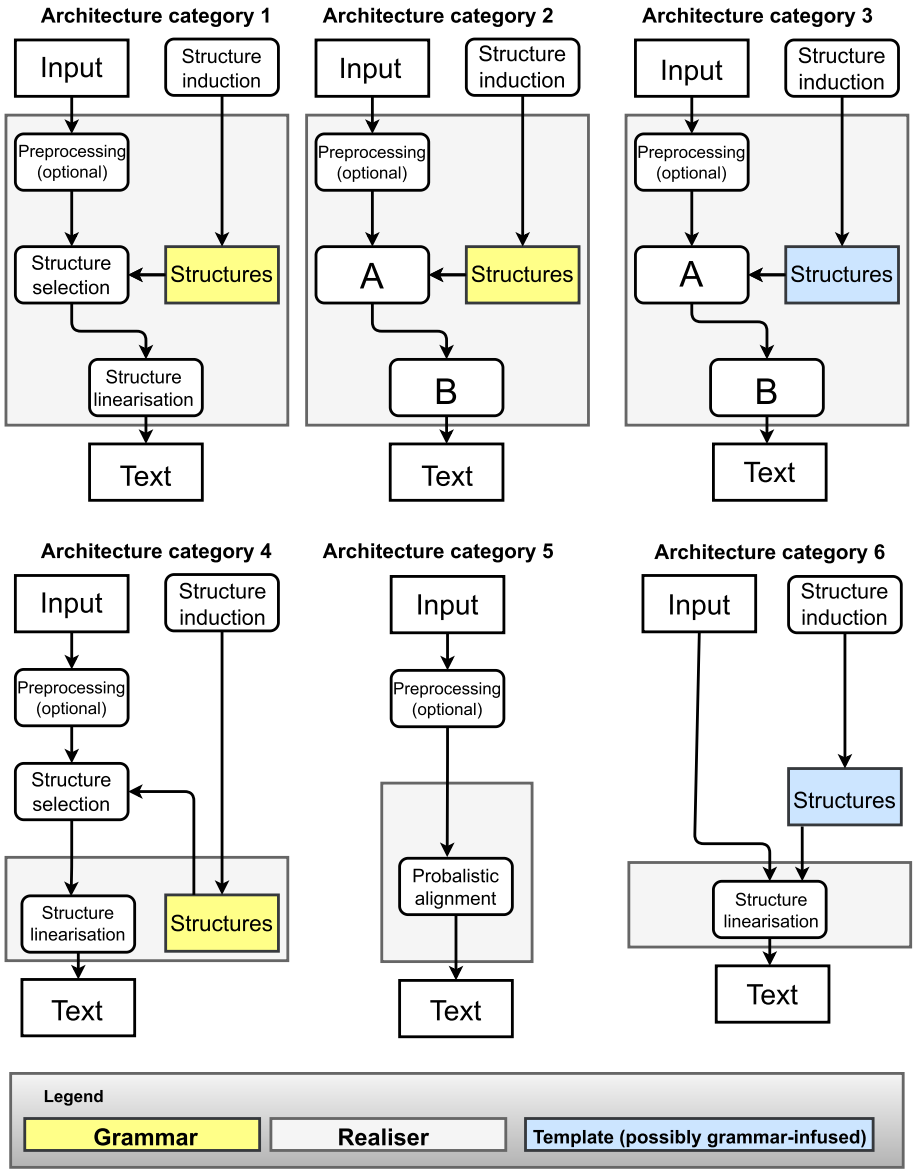

Fig. 1. Surface realizer architecture categories. The modules labeled A and B represent different combinations of structure selection, linearization, and ranking. For instance, in architecture 4 we have A = structure selection and B = structure linearization and ranking. The arrows symbolize the flow of data.

of such an NLG system is the template-based OWL verbalizer for Afrikaans [87], where the tactical decisions component maps the various axiom types that are supported to templates. For instance, axioms of the type SubClassOf(C1 C2) (in OWL functional syntax style), where C1 and C2 are classes, are mapped to a single template. This mapping is encoded via a Java rule that selects the following template whenever it encounters a subsumption axiom[3]:

[3]Taken from https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2015/sanby_todd.zip/index.html.

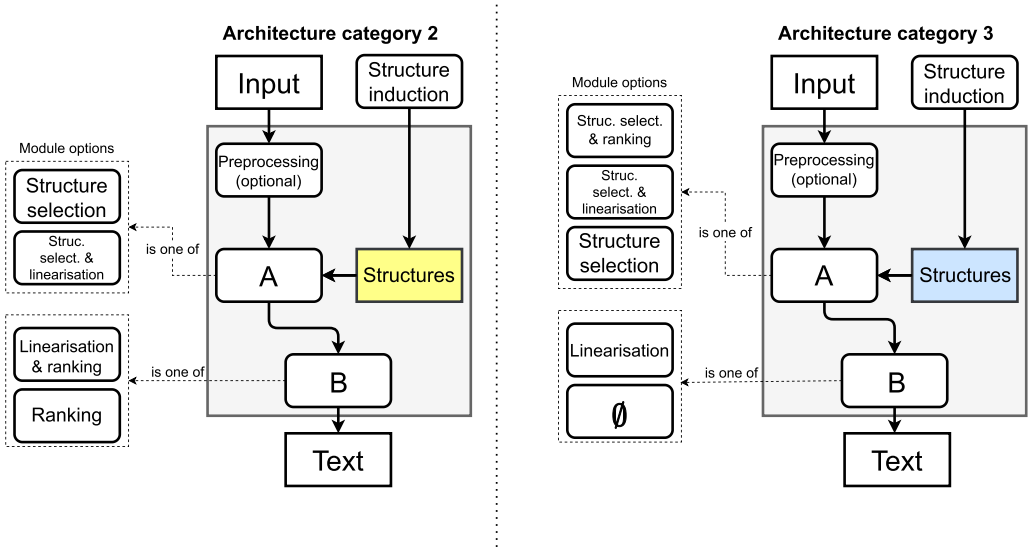

Fig. 2. A detailed view of the surface realizer architecture categories AC2 and AC3. The modules labeled A and B represent different combinations of structure selection (struc. sel.), linearization, and ranking. We use the symbol ∅ to denote a module that does not exist.

Table 3. Surface Realizer Architectures, Their Associated Categories, and the Methods They Use for Structure Induction and Linearization

Architecture Category	Architec-ture id	Structure Induction	Structure Linearization	Module A	Module B
AC1	0	Hand Coded	Grammar	-	-
AC1	1	Data driven	Rule-based	-	-
AC1	2	Data driven	Data driven	-	-
AC1	3	Hand Coded	Rule-based	-	-
AC2	4	Hybrid	Data driven	Structure sel.	Structure lin. + rank.
AC2	5	Hand Coded	Rule-based	Structure sel. + lin.	rank.
AC3	6	Data driven	Rule-based	Structure sel. + rank.	Structure lin.
AC3	7	Hand Coded	Rule-based	Structure sel. + lin.	∅
AC3	8	Hand Coded	Rule-based	Structure sel.	Structure lin.
AC4	9	Hand Coded	Rule-based	-	-
AC4	10	Hand Coded	Grammar	-	-
AC5	11	-	-	-	-
AC6	12	Hand Coded	Rule-based	-	-

Abbreviations: Structure ind. = Structure induction; structure lin. = Structure linearisation; rank. = Ranking.

```
<Constraint type="OWLSubClassOfAxiom">
<Text>Elke</Text>
<Object index="0"/>
<Text>is 'n</Text>
<Object index="1"/>
</Constraint>
```

When given SubClassOf(DOG, ANIMAL), the verbalizer will produce *Elke hond is 'n dier* ("Each dog is an animal"). The linearization of each template is done within Java; it involves filling in slot values and doing minor clean-up such as removing trailing spaces. The system described by [54] also uses this architecture, but it uses a data-driven module for ranking and selecting the best template to use for any provided input. The chosen template and processed input are then linearized using a rule-based module.

The AC4 category is used by most contemporary NLG systems. These systems rely on grammar engines such as SimpleNLG [36] or its adaptations for other languages such as Tibetan [55]. The engine provides reusable language-specific rules for creating syntactic structures and linearizing them to obtain well-formed text. These rules are either taken from an existing system (e.g., [36]) or developed from scratch (e.g., [55]) and stored within the realizer. The choice of which syntactic structure to use given the semantic input is not made by the realizer. Instead, it is left to the discretion of the engineer; i.e., the realizer makes no tactical decisions. Hence, Figure 1 shows structure selection as existing outside the realizer by using structures within the system. The input to such systems is a sentence plan (i.e., an underspecified syntactic structure) or the intermediate representation and a choice of syntactic structure that is to be created by the realization engine. Based on the input, control is given to the engine to create the final syntactic structure and apply all necessary linguistic rules. An example of a system that follows this architecture is [74], who feed **Content MathML (CMML)** expressions to the sentence planner, and it detects which expression categories are found in the input. For instance, when provided with the CMML expression for $\{x \mid x \leq 0\}$, the planner detects that it is made up of the *relational*, *(arithmetic, algebraic, set)*, and *conditional set*. Using the predefined rules for mapping each category to a syntactic structure, it creates a structure for the complex category (i.e., the *conditional set*) where the internal nodes capture the simple categories (i.e., the *relational* and *(arithmetic, algebraic, set)*), the child nodes capture lemmas, and edges capture dependency relations between the various nodes. This sentence plan is passed to the system's coordinator, which then builds a constituency-based syntactical structure using SimpleNLG-it [73] that corresponds to the input and then linearizes the tree to obtain text.

The AC5 category does not have a distinct module for making tactical decisions to determine a natural language's syntactic constraints; it has no explicit sentential structure encoding and linearization module. Instead, it makes use of a data-driven model (most contemporary systems use a deep neural network) to map some intermediate representation to natural language text. The first group of systems that belong to this category do not have an identifiable surface realization component, even though they are modular. For instance, Moryossef et al. [80] train a sequence-to-sequence model with attention for converting text plans to surface text using the OpenNMT toolkit. A text plan is a list of sentence plans where each sentence plan is a labeled directed graph where the nodes represent concepts and direction edges represent predicates. So, when the trained neural network is given the sentence plan [(John, residence, London), (England, capital, London)], it may generate "John lives in London, the capital of England" [80].

The only neural NLG system that has an identifiable surface realizer can be found in [20]. However, its surface realizer is not data driven. Instead, it is fed templates with abstract representations of verb phrases and noun phrases. These templates are produced by the preceding modules. Its architecture belongs to category AC6 and we will now introduce that category.

The AC6 architecture category is used by a few systems. The realizers of such systems are responsible for a single task, i.e., the linearization of structures. For instance, Castro Ferreira et al.'s [20] realizer takes a template with underspecified verb and noun phrases. These templates look like the following [20, p. 554]:

```
Massimo Drago VP[...] play for DT[...] the club SSD Potenza Calcio and
    his own club VP[...] be Calcio Catania . He VP[...] be currently VP
    [...] manage AC Cesena .
```

Listing 1. Example of a template found in Castro Ferreira et al.'s [20] pipeline neural system.

The realizer then uses rules[4] to resolve the verb and noun phrase values in order to generate the final string. Another system whose realizer abides by the organization seen in AC6 can be seen in Gubbala et al.'s [40] work.[5] Unlike Castro Ferreira et al.'s neural pipeline system, this one is not preceded by neural modules but it uses the RosaeNLG[6] library to generate text.

4.3 Appropriateness for NCB Languages

Architectures that belong to categories AC1/2 are inappropriate for, at least, NCB languages (and possibly others as well). The architectures are often used by domain-independent realizers that require the creation of an input specification for purposes of limiting the realizer's semantic space, since they make tactical decisions the responsibility of the realizer. Therefore, they require detailed input, hence making them difficult to easily accommodate in a new system [35, p.80]. This is affirmed by the low adoption of large-scale grammar-based realizers that use architectures AC1/2 (e.g., KPML and AlethGen/GL) in modern NLG systems.

Contemporary NLG systems use architectures that belong to categories AC3/4/5/6. AC3 architectures have been used by several NLG systems that support low-resourced languages, especially African languages. Most domain-specific systems designed for well-resourced languages make use of AC4 architectures when reliability is of concern. Recently, there has been an uptake in NLG systems that follow AC5 architectures for such languages. However, such systems are rarely used in areas where reliability is important since they sometimes exhibit hallucinations, incoherency, and degenerative repetitions [43]. These four architecture categories have various strengths; for instance, AC4 and AC6 move tactical decisions out of the realizer and leave minimal tasks; hence, they do not require restrictive input specifications. In addition, it has been demonstrated that it is possible to bootstrap an existing system for a related language if they use AC4 (see [16]).

Despite the benefits, architectures that belong in categories AC3/4/5/6 are currently also unsuitable for NCB languages with respect to maintainability, for the following reasons:

- AC4 relies solely on grammar engines for structure realization, but there is limited documented grammar for languages that are low-resourced; hence, AC4 is unsuitable. Once a full-scale grammar engine exists for such languages, it should be viable and possibly appropriate.
- The use of AC5 architectures is only appropriate for languages and domains where there exist parallel data-to-text resources to train statistical/neural models. However, this is also not available for isiXhosa and isiZulu—nor most other NCB languages—and therefore also not appropriate.
- The existing approaches that belong to category AC6 suffer from both the issues mentioned above.

The only categories that are promising for the NCB languages, in that the categories can form a basis for more suitable architectures, are AC3/4/6. This is partially motivated by the observation that all existing NLG systems for Nguni languages have used category AC3 architectures due to their practicality. The only limitation with AC3, for instance, is that it does not yield maintainable

[4] https://github.com/ThiagoCF05/DeepNLG/blob/master/realization.py.
[5] https://github.com/Alihussainladiwala/Citizen-Friendly-Report-of-diversitydatakids.org/.
[6] https://rosaenlg.org/.

and reusable software as it tightly couples the linearization rules with the actual templates and makes tactical decisions the responsibility of the realizer. For instance, consider the algorithm implementation[7] for the pattern that verbalizes negated simple existential quantification, given in Listing 2 (discussed afterward).

```
1  def nexist_zu(sub,op,super):
2      nc1m = find_nc(sub)
3      nc2m = find_nc(super)
4      nc2 = strip_m(nc2m)
5      pl = plural_zu(sub,nc1m)
6      ncp = look_ncp(nc1m)
7      qca = look_qca(ncp)
8      rc = look_relc(nc2)
9      qc = look_qce(nc2)
10     rt = find_rt(op)
11     negsc = look_negsc(ncp)
12     if rt[0] in 'aeiou':
13         negconjrt = negsc_vowel_vroot(rt,negsc)
14     else:
15         negconjrt = negsc + rt
16     return qca + ' ' + pl + ' ' + negconjrt + 'i' + ' ' + super + ' '
           + rc + qc + 'dwa'
```

Listing 2. Python implementation for verbalizing simple existential quantification in isiZulu (Source: [51]).

When the verbalizer is given the input $Umuntu \sqsubseteq \neg\exists feza.umsebenziOnqunyiwe$, it detects that the relevant lexical items are *umuntu,* "person"; *feza,* "achieve"; and *umsebenzi onqunyiwe,* "task" and invokes nexist_zu (line 1). It then identifies and processes the noun classes of the nouns associated with the sub and super classes (lines 2–4) and pluralizes the sub's noun and identifies the plural's noun class (lines 5–6). It then uses the retrieved noun classes to select the relevant quantitative and relative concords (lines 7–9). Subsequently, a negated verb is formed for the verb (lines 10–15). Finally, the algorithm combines them to form the sentence (line 16), in this case generating *Bonke abantu abafezi umsebenzi onquyiwe oyedwa,* "All humans do not achieve some task." Here, the grammar-infused template and the linearization rules used to generate text are interwoven in the parent function nexist_zu. In addition, the decision to choose an appropriate template given some input is also the responsibility of the realizer. This means that it is impossible to isolate the templates for re-use. This is a general problem of this approach and does not speed up progress for tools for NCB languages.

Technically, such systems can be re-designed into an AC6 architecture. The templates may be captured via a separate language (à la RosaeNLG's pug templates) and the choice of an appropriate template would be carried out by a separate module. The code in Listing 2 must be split into a template and linearization algorithm. A hypothetical template might be like the one shown in Listing 3, where concateList is a hypothetical extension of the pugjs[8] template language, which would order plain-text elements and intelligently introduce spaces between some special elements, and the value mixin is from RosaeNLG (lines 2–4, 6) but with parameters that are currently not

[7] https://github.com/mkeet/GENIproject/blob/master/isiZuluVerbaliser/OntologyVerbaliser_Zu/zulurulespw.py.
[8] https://pugjs.org/language/.

supported. Line 2 invokes value so that the template engine will generate the appropriate form of *onke,* "all," which depends on the noun class of the noun it is associated with, generating, e.g., the *Bonke* in the example above.

```
1 concateList
2         | +value('onke', {nc: getNC(sub)})
3         | +value(sub, {number: PLURAL})
4         | +value(op, {marker: NEGATIVE})
5         | #{super}
6         | +value('dwa', {nc: getNC(super)})
```

Listing 3. Hypothetical RosaeNLG template for simple existential quantification in isiZulu (summarized).

The hypothetical RosaeNLG is still a summarized version, however, because RosaeNLG is also not capable of processing the phonological conditioning required in line 2 to make the full word and it does not deal with the relative and quantitative concord in line 6 to complete the verbalization of the existential quantification (to generate the *oyedwa* in the example sentence from the *dwa* in the template). That is, an extended RosaeNLG or a new template language for NCB languages is needed to realize this.

The rules for linearizing the template can be moved to a separate module and invoked by some linearization code, like the hypothetical JavaScript code shown in Listing 4. It has a function for choosing the appropriate template when given an axiom (lines 3–7), a function for retrieving the template from a file and passing the lexical items for the axiom's elements (lines 9–17), and a reusable function for linearizing templates that use RosaeNLG's engine (lines 19–22).

```
1  const rosaenlgPug = require('rosaenlg');
2
3  function verbalise(axiom):
4      ...
5      if nExistAxiomMustBeUsed:
6          vals = getLexicalValues(axioms)
7          nexist_zu(vals[0], vals[1], vals[1])
8
9  function nexist_zu(someSubVar, someOpVar, someSuperVar):
10     return lineariseTemplate(
11     'template.pug',
12     {
13         language: 'zu',
14         sub: someSubVar,
15         op: someOpVar,
16         super: someSuperVar
17     });
18
19 function lineariseTemplate(templateName, params)
20         return rosaenlgPug.renderFile(
21                 templateName,
22                 params);
```

Listing 4. Hypothetical JavaScript code for using a RosaeNLG template to verbalize simple existential quantification in isiZulu.

The use of a declarative template language results in templates that are separate from the linearization rules, and they can be reused without the need to disentangle code and recreate the relevant rules.

A downside of such a declarative template language is the challenge of constraint checking to ensure that the template is well formed. This is also a challenge if one uses a task ontology to capture simple or grammar-infused templates [67]. In both ways of declaring templates, words and slots must be ordered and there must be at least one slot, and one is free to introduce additional constraints. However, since the template file will only specify the "what" and not the "how" (since it is declarative), this permits the use and reuse of invalid templates, which affects the viability of template re-use. This issue may be solved through the following methods:

- Best practice guidelines: one can collect pitfalls seen in template reuse over an extended period and create methodologies for template reuse that ensure that NLG engineers avoid those pitfalls. The disadvantage with this approach is that it can only lead to a solution after one has spent an extended amount of time observing pitfalls.
- Safeguards in system architecture: use the presented analysis of existing NLG architectures to introduce changes that ensure that the architecture has robust means for template constraint violation detection upfront. The challenge with this approach is that it requires some innovation regarding how to realize the error detection module and where in the architecture such a module must be added.

We choose the second option and introduce our architecture in the next section.

5 KNOWLEDGE-GUIDED ARCHITECTURE

We develop a new architecture by first specifying requirements, creating an architecture that meets the requirements, and then evaluating the result through a manual feature comparison.

5.1 Requirements

The requirements phase focuses on determining the high-level characteristics that should be in the architecture. These requirements were decided upon by the authors based on observations we have made on the evolution of realizer architectures in NLG, as discussed in the previous section, and the needs of the languages in question. The high-level design requirements are as follows:

- The sentential structure must be encoded via templates, which may be grammar infused, in order to support low-resourced languages. When necessary, it must be possible to enrich the grammar-infused templates with task/context-specific grammatical knowledge.
- Knowledge regarding grammar-infused templates must be incorporated via application ontologies as it allows use of semantic technologies for template creation, automated reasoning to detect inconsistencies, and comparison of template features. At first glance, it may seem like the approach of capturing templates is separate from a surface realizer, but it is not. The use of grammar-infused templates requires a principled approach to encoding the necessary grammatical and template knowledge. Just like there are formal theories for specifying grammars, a theory of templates is required in order to make the combination sound.
- Tactical decisions should not be the responsibility of the surface realizer. Like the proposal by SimpleNLG [36], tactical decisions should be left to the discretion of each NLG engineer. The advantage of this is that it avoids the control issues and restrictive input formalisms found in dated wide-coverage tactical generators. From an engineering perspective, this separation of concerns maximizes the reusability of the realizer while minimizing the complexity of its input.

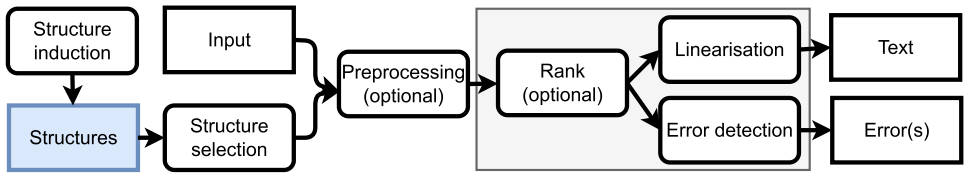

Fig. 3. Functional view of the knowledge-guided architecture to generate text from grammar-infused templates. The arrows symbolize the flow of data.

- Maintainability must be a key feature guiding the design of the architecture since template-based systems have been criticized in the past for being difficult to maintain. We are interested in modularity, componential re-usability, and analyzability.
- Statistical models should be usable as a means for the creation and selection of templates when generating text. It must be possible to also make use of the grammatical features afforded by grammar-infused templates to rank possible templates (cf. only using surface text). Moreover, the component for selecting a template given some input must not be necessarily data driven since the language for which one is building an NLG system may not have sufficient corpora to build a data-driven template selector.

It may seem counter-intuitive to rely on templates when building an architecture for reusable realizers, especially to individuals who are well versed in NLG for a language such as English. In line with van Deemter et al.'s argument, it is a viable option because while the adaptation of an existing system to a new domain may require template edits or the creation of new ones, "the underlying generation mechanism, [in the architecture] generally requires little or no modification" [92]. Moreover, the maintainability of templates must be weighed with the maintainability of CGRs in a case where there is little to no up-to-date grammar documentation or data driven in cases where training data is hard to come by. The goal of this approach is to rely on templates for generating task-specific text. However, these templates are used with limited, but possibly growing, CGRs that are designed to cater to general linguistic knowledge. Using a CGR-only approach would not be viable since there is none with large coverage. In addition, using templates with no additional grammar rules would also not suffice for the generation of human-friendly text, as demonstrated by the isiXhosa template(s) introduced in Section 1 for verbalizing the axiom $Teacher \sqsubseteq \exists teaches.Subject$.

5.2 Architecture Creation

Based on the above requirements and our analysis of existing architectures, we have created the architecture shown in Figure 3. The *tactical decisions* are moved out of the realizer like AC4. *Structure creation* is achieved via templates, possibly grammar infused. Moreover, for a faithful implementation of this architecture, template *knowledge* (i.e., concepts, relationships, and constraints) must be formalized in an ontology. Lastly, there is no prescription for the methods that can be used for structure induction so rules or *statistical models* can be used. Further, when generating text using an implementation of this architecture, there might be multiple candidate grammar-infused templates, as a result of needing variety for text to be generated. The *ranking* component selects an appropriate template from the candidates, which can be rule based or data driven. The selected template and its associated input are then passed to the linearization module for slot value insertion, application of the necessary grammar rules, and finally producing surface text.

The *preprocessing* and *ranking* modules are introduced and placed in a location that differs from the architectures presented in Table 2 and categorized in Table 3. This is done to satisfy the

requirements pertaining to adding task-specific grammatical information and using it to rank the possible templates. To demonstrate the usefulness of such features, we use Braun et al.'s [12] Safer-Drive system. The system was originally created to generate reports to make drivers aware and change their behavior of negative driving habits. One could extend the system for use by insurance companies so that they may offer their version of reports to drivers or to report to actuaries who set prices depending on driver behavior. The same templates could be used, but the grammatical voice would be different. For instance, templates for when a driver accesses the reports directly can be annotated in the active voice through the pre-processing module, but for the insurance company it would be annotated with passive voice. Consequently, when given multiple templates for verbalizing the message, say, drivingPeriod(distance, time), then the ranking module would select the appropriate template given the context; e.g., when generating text for the driver, it may generate "You drove 390 miles in 10 hours and 50 minutes during the last week" [12, p. 576] and for the insurance company it would then be "390 miles were driven in 10 hours and 50 minutes during the last week by X," where X is the name of the driver.

The error detection module is introduced to satisfy the requirement for detecting inconsistencies pertaining to template knowledge. To demonstrate this, consider the following Latvian template taken from [39] for parsing and verbalizing text for the axiom $Professor \sqsubseteq Teacher$: *Visi profesori ir (pasniedzēji | skolotāji),* "All professors are lecturers/teachers." In the template, the words *pasniedzēji* and *skolotāji* are specified to be synonyms. If one would create grammar-infused templates and use invalid synonymy relations in the templates, then the error detection module would detect such errors. Knowledge regarding the validity of such relations could be extracted from the Latvian Wordnet.[9] The error detection module is conceptualized to be useful for all grammar rules that are incorporated into templates, not just synonymy relations.

6 MAINTAINABILITY AND COMPARISON TO EXISTING WORK

In this article, we advance an *a priori* justification for maintainability. We are interested in the absence/presence of the following features: support for economical resource use (componential reusability), well-defined template concepts and relationships between the concepts (analyzability), and separation of surface realization tasks (modularity). We do not include the other two maintainability characteristics of [13], namely, modifiability and testability, since they require a posteriori justification and there is no implementation of the architecture that has been used over an extended period. Henceforth, we use the symbol $\mathcal{C}$ to refer to our operationalization of *C*omponential reusability, $\mathcal{O}$ to analyzability of the knowledge formalized in an *O*ntology, and $\mathcal{S}$ to modularization of the *S*urface realization tasks.

Table 4 shows each architecture's support of the maintainability features. We demonstrate them with the isiZulu verbalizer [51]. Analysis of the verbalizer's architecture (Arch. 7, Category AC3) shows that it supports reusable grammar rules for pluralization and other rules can be added in the same fashion (yes to $\mathcal{C}$). However, the algorithms used to generate text encode the selection of a specific template for each quantifier and their corresponding linearization rules. The verbalizer has a clear notion of what template concepts are allowed and how they relate to each other, even though the constraints are not explicitly declared. Moreover, the concepts can be extended when needed (e.g., [48]). Nonetheless, they are not formalized via an ontology (hence, no to $\mathcal{O}$ and $\mathcal{S}$). Capturing the verbalizer's concepts and relations using existing resources is possible and an ontology need not be created from scratch, since the **Task ontology for CNL Templates (ToCT)** [67] already exists.

[9] https://wordnet.ailab.lv/.

Table 4. Comparison of the New Architecture to AC5 and AC3

Architecture id(s)	Category	$\mathcal{C}$	$\mathcal{O}$	$\mathcal{S}$
Proposed architecture	-	●	●	●
0, 1, 3	AC1	○	-	●
2	AC1	○	-	●
4	AC2	●	-	○
5	AC2	●	-	○
7	AC3	●	○	○
8, 6	AC3	●	○	●
9, 10	AC4	●	-	●
11	AC5	-	-	-
12	AC6	●	○	●

Abbreviations: $\mathcal{C}$ = Computational grammar rule(s) reuse, $\mathcal{O}$ = Ontology formalised concepts and relationships, $\mathcal{S}$ = Separation of surface realisation tasks. Legend: Green = supported, Black = not supported, and Dash = not applicable.

Our architecture combines the strengths of categories AC1–6, as stated via our architecture requirements, and supports the most maintainability features. It is the best option for NCB languages since it supports the specified requirements and most of the maintainability requirements (our operationalization of [13]'s maintainability characteristics to ensure that they are sensitive to the needs of NCB languages). It is not necessarily the best choice for other languages, especially well-resourced languages, since it enforces the use of templates. In cases where one wants to avoid the use of templates, then architectures belonging to AC4/5/6 may be best. This is subject to availability of a grammar engine (AC4) and the toleration of degradation in textual coherence and correctness associated with probabilistic NLG models (AC5). If grammatical complexity is limited and hence either template without rules suffices or the complexity can be managed by introducing a handful of rules that are easy to manage, then AC6 may be appropriate. Given all that information, our architecture may still be used for such languages as it has no prescript regarding structure induction. This is meant to tackle the challenge of template creation and maintenance. As such, one can pursue data-driven techniques in the same vein as [54] for template creation.

As can be seen in Table 4, it is not possible to mix and match features from architectures in each category to satisfy all the maintainability features. For instance, using AC1 as an example, one cannot combine the best features of architectures 0, 1, 2, and 3 in order to satisfy $\mathcal{C}$, $\mathcal{O}$, and $\mathcal{S}$.

Compared to other architecture categories, excluding our proposed architecture, categories AC3/4/6 support the most features; hence, systems that abide by the architectures are somewhat easy to maintain. Our architecture, unlike all other existing architectures, supports all the features identified in Table 4 and this is primarily because none of the architectures support the use of ontologies to capture template knowledge.

7 SIGNIFICANCE OF TEMPLATE ONTOLOGY

We will demonstrate the utility of the ontology for grammar-infused templates as a means of facilitating detecting logical inconsistencies in templates, sharing, and comparison of templates cf. ad hoc approaches. We will use the demonstrative soccer templates and slot fillers given below and Sanby et al.'s two systems to demonstrate these benefits.

```
(a) Slot[player] Phrase[scored the third goal].
(b) Phrase[The goal at the 20th minute was]  AltPhrasing(Phrase([not
    offside]), Word([onside])).
(c) Phrase[The winning goal was the result of a header by] Slot[player].
(d) Key[player], Value[Aardvarkis]}
```

The templates above could be used to generate messages associated with the meaning representation goalEvent(scorer, time, goalLineStatus, goalNumber) in some system. In such a scenario, the ontology is used to declare the concepts to be used when creating a template (à la schema), for instance, to declare concepts such as Slot, Phrase, Value, and AltPhrasing. The ontology can also be used to specify constraints, for instance, to specify that templates must have at least one slot, i.e., $Template \sqsubseteq \geqslant 1\ hasPart.Slot$. It is the responsibility of the NLG engineer to create the templates such as (a), (b), and (c) together with the key and value pair (d) to be inserted in the templates. The templates can then be serialized using any number of formats (e.g., RDF/XML) and stored in a file.

7.1 Inconsistencies

The benefit of using an ontology instead of an XML schema definition language (or similar language) is automated reasoning. That is, the knowledge in the ontology (the TBox) can be combined with the templates (encoded as instances in the ABox) to create a knowledge graph that then can be sent to an automated reasoner (e.g., HermiT OWL Reasoner [38]). In the case of the soccer templates and their associated ontology, they can be loaded into the reasoner and checked if they are consistent. If the ontology includes the axiom restricting the number of slots in a template, then it will detect that template (b) is not a valid template, because it does not contain any slots. While automated reasoning is not a novel concept in general, it is novel within surface realization.

7.2 Template Comparison

Use of an ontology also facilitates sharing and comparing templates. For instance, the two systems found in [87] verbalize OWL's disjoint classes axioms via an XML template and a **Grammatical Framework (GF)** concrete grammar rule. The XML template is shown on the left and the GF grammar rule is given on the right:

```
1 <Constraint type="Disjoint">
2 <Text>'n</Text>
3 <Object index="0"/>
4 <Text>is nie 'n</Text>
5 <Object index="1"/>
6 <Text>nie</Text>
7 </Constraint>
```

```
DisjointClasses x y =                1
{s="'n "                             2
++x.s                                3
++"is nie 'n"                        4
++y.s                                5
++"nie"                              6
};                                   7
```

We can use a template ontology (e.g., Mahlaza and Keet's ToCT [67]) to show that the two templates given above are equivalent even though they use different methods for serialization. To demonstrate this equivalence, let us first explain the elements of the two templates; the XML template's *Text* object and the strings in the GF grammar rule are used to capture fixed texts only. The difference (and similarities) between the two templates can be seen in line 4 of both listings when they capture the fixed string *is nie 'n,* "is not." These portions from templates can be captured via ToCT's *Phrase* concept. The XML template's *Object* and GF's variables *x* and *y* are used as placeholders for different values that can be used. For instance, line 3 in both listings marks the position

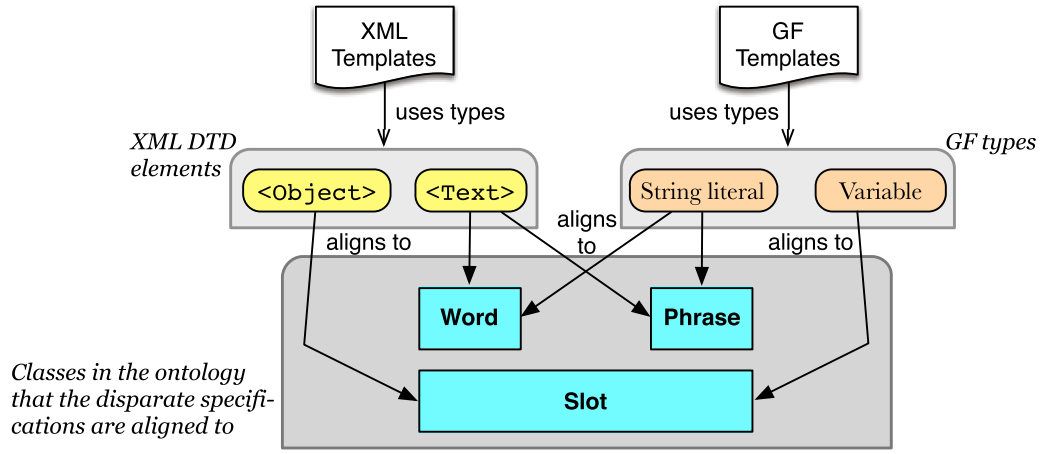

Fig. 4. Representation of the concepts used in the GF and XML templates and how they relate to each other.

that will be occupied by the name associated with first the class that participates in the Disjoint axiom. If the input is the axiom Disjoint(Animal Computer), then the position might be occupied by the noun "animal."

Analysis of the concepts used in the two types of templates shows that they have the same meaning and function, as demonstrated in Figure 4. Specifically, the XML's *Object* and GF's variables both refer to ToCT's *Slot* concept. The *Text* and GF's string literals are used to refer to ToCT's Unimorphic word and a special case of the Phrase concept.

In this example, when bells and whistles are removed, we are only left with fixed phrases and slots that are the same for both types of verbalizers. Thus, they can share the template, map it to one's preferred serialization, and avoid resource duplication. These capabilities are not possible in the other NLG/surface realizer architectures because they either do not use templates or use ad hoc template specifications.

8 REDESIGNING A TEXT GENERATOR

Since the proposed architecture is designed to be most[10] beneficial for NCB languages, we selected the most recent paper describing an NLG system that generates such a language and redesigned the architecture of its surface realizer. To the best of our knowledge, a system that fits this criterion is OWLSIZ [66]. It is built to generate questions from an OWL ontology.

The original system parses an ontology and feeds the supported axioms to a realizer that is responsible for structure selection, lexicalization, and linearization. The redesigned OWLSIZ uses the architecture shown in Figure 5.

After reading in the supported axioms, it selects the appropriate template(s) and chooses lexical items for the concepts found in the axiom. For instance, when generating a question from the axiom $dog \sqsubseteq animal$, OWLSIZ's template 1 is chosen and the **noun classes (NCs)** of $inja_{NC9}$, "dog," and $isilwane_{NC7}$, "animal," are resolved. The template is loaded from a file and, together with its associated slot fillers, sent to the surface realizer to produce *Ingabe yonke inja iyisilwane?*, "Is every dog an animal?"

[10]E.g., its reliance on templates that are used with grammar are a great benefit for low-resourced languages with complex grammar.

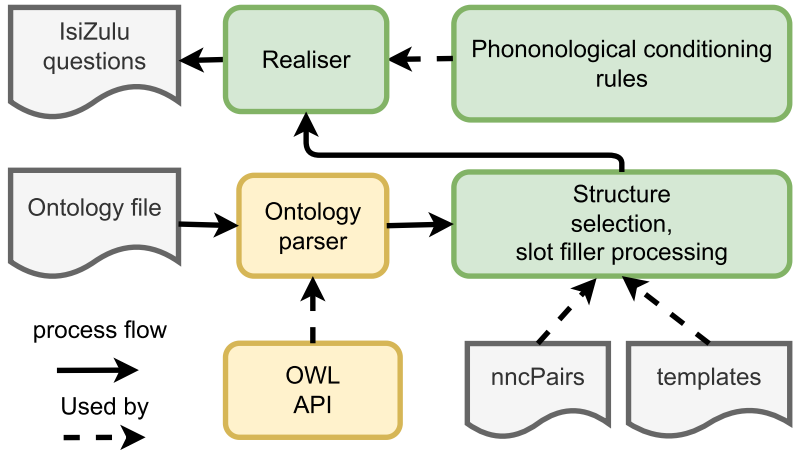

Fig. 5. Updated architecture used by OWLSIZ.

We developed an ontology for template specification that is used for the implementation-independent specification of the templates, thus enjoying the advantages of consistency checking. The redesigned verbalizer's error detection module was used to check the consistency of the templates and no inconsistencies were found. The updated realizer is not responsible for tactical decisions. Consequently, the realizer is reusable in other systems because system/area-specific tasks such as template selection are no longer its responsibility. Other existing isiZulu text generators (e.g., [48, 50]) could use the same realizer, provided their templates rely on the ontology. The redesigned architecture also ensures a separation of realization tasks cf. the original version, and therefore a change in one task can only introduce minimal disruption to other components. The code of the forked redesign is available as supplementary material.

9 CONCLUSIONS AND FUTURE WORK

We have presented a novel surface realizer architecture that moves tactical decisions out of the realizer; relies on templates (possibly grammar-infused) for encoding sentential structure; enforces the use of an ontology to formalize the templates' concepts, relations, and associated constraints; and supports candidate ranking via a possibly data-driven module. Unlike existing architectures, especially those used by NCB language generation systems, our architecture is designed to result in surface realizers that are easily maintainable since it supports our operationalization of componential re-usability, analyzability, and modularity. Specifically, by re-usability we refer to the architecture's support for economical resource use of grammar rules, by analyzability we refer to an architecture that relies on well-defined template concepts and relationships between the concepts, and by modularity we refer to its separation of the considered surface realization tasks. In summary, this article has the following findings when compared to existing work:

(1) The architecture supports the most features of a maintainable software system.
(2) It supports the detection of inconsistencies in the templates and allows integration across different template types.
(3) An existing tool's architecture can easily be updated to abide by the new architecture.

Our proposed architecture champions the use of an ontology for specifying the templates, for theoretically grounded reasons. However, some NLG engineers may not be familiar with ontologies, and this may negatively affect the adoption of our architecture. We plan to conduct

an empirical study to compare the ease and speed at which NLG engineers of different experience levels can specify templates using an ontology instead of using a framework such as Grammatical Framework or SimpleNLG for the same task. Future work is thus focused on ease of creation for engineers with different levels of expertise to increase chances of adoption, dovetailing with the currently presented work that is focused on maintainability.

REFERENCES

[1] Wael Abed and Ehud Reiter. 2020. Arabic NLG language functions, See [27], 7–14.

[2] G. Aguado, A. Bañón, J. Bateman, Socorro Bernardos, M. Fernández, Asunción Gómez-Pérez, Elena Nieto, A. Olalla, R. Plaza, and Antonio Sánchez. 1998. ONTOGENERATION: Reusing domain and linguistic ontologies for Spanish text generation. In *Workshop on Applications of Ontologies and Problem Solving Methods, 13th European Conference on Artificial Intelligence*. John Wiley and Sons, 10 pages.

[3] Muhammad Amith, Frank J. Manion, Marcelline R. Harris, Yaoyun Zhang, Hua Xu, and Cui Tao. 2017. Expressing biomedical ontologies in natural language for expert evaluation. In *Proceedings of the 16th World Congress on Medical and Health Informatics (MEDINFO'17): Precision Healthcare through Informatics (Studies in Health Technology and Informatics, Vol. 245)*, Adi V. Gundlapalli, Marie-Christine Jaulent, and Dongsheng Zhao (Eds.). IOS Press, 838–842.

[4] Ion Androutsopoulos, Gerasimos Lampouras, and Dimitrios Galanis. 2013. Generating natural language descriptions from OWL ontologies: The NaturalOWL system. *Journal of Artificial Intelligence Research* 48 (2013), 671–715.

[5] Wee Tiong Ang, Rajaraman Kanagasabai, and Christopher J. O. Baker. 2008. Knowledge translation: Computing the query potential of bio-ontologies. In *Proceedings of the Workshop on Semantic Web Applications and Tools for Life Sciences (SWAT4LS'08) (CEUR Workshop Proceedings, Vol. 435)*, Albert Burger, Adrian Paschke, Paolo Romano, and Andrea Splendiani (Eds.). CEUR-WS.org, 9 pages.

[6] Gabor Angeli, Percy Liang, and Dan Klein. 2010. A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP'10)MIT Stata Center, A meeting of SIGDAT, a Special Interest Group of the ACL*. Association for Computational Linguistics, 502–512.

[7] John A. Bateman. 1997. Enabling technology for multilingual natural language generation: The KPML development environment. *Natural Language Engineering* 3, 1 (1997), 15–55.

[8] Bernd Bohnet, Leo Wanner, Simon Mille, and Alicia Burga. 2010. Broad coverage multilingual deep sentence generation with a stochastic multi-level realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING'10)*, Chu-Ren Huang and Dan Jurafsky (Eds.). Tsinghua University Press, 98–106.

[9] Marcel Bollmann. 2011. Adapting SimpleNLG to German. In *Proceedings of the 13th European Workshop on Natural Language Generation (ENLG'11)*. Association for Computer Linguistics, 133–138.

[10] Nadjet Bouayad-Agha, Gerard Casamayor, Simon Mille, Marco Rospocher, Horacio Saggion, Luciano Serafini, and Leo Wanner. 2012. From ontology to NL: Generation of multilingual user-oriented environmental reports. In *Proceedings of the 17th International Conference on Applications of Natural Language to Information Systems (NLDB'12) - Natural Language Processing and Information Systems (Lecture Notes in Computer Science, Vol. 7337)*, Gosse Bouma, Ashwin Ittoo, Elisabeth Métais, and Hans Wortmann (Eds.). Springer, 216–221. https://doi.org/10.1007/978-3-642-31178-9

[11] Nadjet Bouayad-Agha, Gerard Casamayor, Simon Mille, and Leo Wanner. 2012. Perspective-oriented generation of football match summaries: Old tasks, new challenges. *ACM Transactions on Speech and Language Processing* 9, 2 (2012), 3:1–3:31.

[12] Daniel Braun, Ehud Reiter, and Advaith Siddharthan. 2018. SaferDrive: An NLG-based behaviour change support system for drivers. *Natural Language Engineering* 24, 4 (2018), 551–588. https://doi.org/10.1017/S1351324918000050

[13] BS ISO/IEC 25010:2011. 2011. *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*. Standard. British Standard Institution.

[14] Stephan Busemann. 1996. Best-first surface realization. *arXiv e-prints* (May 1996), 11 pages. arXiv:cmp-lg/9605010 [cs.CL].

[15] Joan Byamugisha. 2019. *Ontology Verbalization in Agglutinating Bantu Languages: A Study of Runyankore and Its Generalizability*. Ph.D. Dissertation. Department of Computer Science, University of Cape Town, South Africa.

[16] Joan Byamugisha, C. Maria Keet, and Brian DeRenzi. 2016. Bootstrapping a Runyankore CNL from an isiZulu CNL. In *Proceedings of the 5th International Workshop Controlled Natural Language (CNL'16) (LNCS, Vol. 9767)*, Brian Davis, Gordon J. Pace, and Adam Z. Wyner (Eds.). Springer, 25–36.

[17] Joan Byamugisha, C. Maria Keet, and Brian DeRenzi. 2016. Tense and aspect in Runyankore using a context-free grammar. In *Proceedings of the 9th International Conference on Natural Language Generation (INLG'16)*. Association for Computer Linguistics, 84–88.

[18] John J. Camilleri, Norbert E. Fuchs, and Kaarel Kaljurand. 2012. *ACE Grammar Library*. Technical Report D11.1. Multilingual Online Translation (MOLTO) project.

[19] Mercedes Argüello Casteleiro, Julio Des, Maria Jesus Fernandez Prieto, Rogelio Perez, and Stavros Lekkas. 2011. An ontology-based approach to natural language generation from coded data in electronic health records. In *Proceedings of the UKSim 5th European Symposium on Computer Modeling and Simulation (EMS'11)*, David Al-Dabass, Alessandra Orsoni, Athanasios A. Pantelous, Gregorio Romero, and Jesús Félez (Eds.). IEEE, 366–371.

[20] Thiago Castro Ferreira, Chris van der Lee, Emiel van Miltenburg, and Emiel Krahmer. 2019. Neural data-to-text generation: A comparison between pipeline and end-to-end architectures. *arXiv e-prints* (Aug. 2019), 12 pages. arXiv:1908.09022 [cs.CL].

[21] Philipp Cimiano, Janna Lüker, David Nagel, and Christina Unger. 2013. Exploiting ontology lexica for generating natural language texts from RDF data. In *Proceedings of the 14th European Workshop on Natural Language Generation (ENLG'13)*, Albert Gatt and Horacio Saggion (Eds.). Association for Computer Linguistics, 10–19.

[22] José Coch. 1996. Overview of AlethGen. In *Proceedings of the 8th International Natural Language Generation Workshop (INLG'96) - Posters and Demonstrations*. Association for Computer Linguistics, 25–28.

[23] Robert Dale. 2019. NLP commercialisation in the last 25 years. *Natural Language Engineering* 25, 3 (2019), 419–426.

[24] Dana Dannélls. 2012. On generating coherent multilingual descriptions of museum objects from Semantic Web ontologies. In *Proceedings of the 7th International Natural Language Generation Conference (INLG'12)*, Barbara Di Eugenio, Susan McRoy, Albert Gatt, Anja Belz, Alexander Koller, and Kristina Striegnitz (Eds.). Association for Computer Linguistics, 76–84.

[25] Dana Dannélls, Mariana Damova, Ramona Enache, and Milen Chechev. 2012. Multilingual online generation from semantic web ontologies. In *Proceedings of the 21st World Wide Web Conference (WWW'12) (Companion Volume)*, Alain Mille, Fabien Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab (Eds.). ACM, 239–242.

[26] Brian Davis, Ramona Enache, Jeroen van Grondelle, and Laurette Pretorius. 2012. Multilingual verbalisation of modular ontologies using GF and lemon. In *Proceedings of the 3rd International Workshop on Controlled Natural Language (CNL'12) (Lecture Notes in Computer Science, Vol. 7427)*, Tobias Kuhn and Norbert E. Fuchs (Eds.). Springer, 167–184.

[27] Brian Davis, Yvette Graham, John D. Kelleher, and Yaji Sripada (Eds.). 2020. *Proceedings of the 13th International Conference on Natural Language Generation (INLG'20)*. Association for Computational Linguistics.

[28] Brian Davis, Ahmad Ali Iqbal, Adam Funk, Valentin Tablan, Kalina Bontcheva, Hamish Cunningham, and Siegfried Handschuh. 2008. RoundTrip ontology authoring. In *Proceedings of the 7th International Semantic Web Conference (ISWC'08) (Lecture Notes in Computer Science, Vol. 5318)*, Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan (Eds.). Springer, 50–65.

[29] Rodrigo de Oliveira and Somayajulu Sripada. 2014. Adapting SimpleNLG for Brazilian portuguese realisation. In *Proceedings of the 8th International Conference on Natural Language Generation (INLG'14)*. Association for Computer Linguistics, 93–94.

[30] Paolo Dongilli and Enrico Franconi. 2006. An intelligent query interface with natural language support. In *Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference*, Geoff Sutcliffe and Randy Goebel (Eds.). AAAI Press, 658–663.

[31] Ondřej Dušek. 2017. *Novel Methods for Natural Language Generation in Spoken Dialogue Systems*. Ph.D. Dissertation. Institute of Formal and Applied Linguistics, Charles University, Prague.

[32] Ondrej Dusek, Jekaterina Novikova, and Verena Rieser. 2020. Evaluating the state-of-the-art of end-to-end natural language generation: The E2E NLG challenge. *Computer Speech & Language* 59 (2020), 123–156.

[33] Michael Elhadad and Jacques Robin. 1996. An overview of SURGE: A reusable comprehensive syntactic realization component. In *8th International Natural Language Generation Workshop (INLG'96) - Posters and Demonstrations*, Donia Scott, John Bateman, Guy Lapalme, David McDonald, Cécile Paris, and Keith Linden (Eds.). Association for Computer Linguistics, 1–4.

[34] Andrea Cascallar Fuentes, Alejandro Ramos-Soto, and Alberto José Bugarín Diz. 2018. Adapting SimpleNLG to Galician language. In *Proceedings of the 11th International Conference on Natural Language Generation (INLG'18)*. Association for Computer Linguistics, Tilburg University, 67–72.

[35] Albert Gatt and Emiel Krahmer. 2018. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research* 61 (2018), 65–170.

[36] Albert Gatt and Ehud Reiter. 2009. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG'09)*, Emiel Krahmer and Mariët Theune (Eds.). Association for Computer Linguistics.

[37] Nikhil Gilbert and C. Maria Keet. 2018. Automating question generation and marking of language learning exercises for isiZulu. In *Proceedings of the 6th International Workshop on Controlled Natural Language (CNL'18) (Frontiers in Artificial Intelligence and Applications, Vol. 304)*, Brian Davis, C. Maria Keet, and Adam Wyner (Eds.). IOS Press, 31–40.

[38] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. 2014. HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning* 53, 3 (Oct. 2014), 245–269.

[39] Normunds Gruzitis, Gunta Nespore, and Baiba Saulite. 2010. Verbalizing ontologies in controlled Baltic languages. In *Human Language Technologies - The Baltic Perspective - Proceedings of the 4th International Conference (Baltic HLT'10) (Frontiers in Artificial Intelligence and Applications, Vol. 219)*, Inguna Skadina and Andrejs Vasiljevs (Eds.). IOS Press, 187–194.

[40] Anil Gubbala, Ali Hussain Ladiwala, Urja Naik, and Priyanka Cornelius. 2021. *Citizen Friendly Report of diversity-datakids.org*. Technical Report. San Josè State University.

[41] Daniel Hewlett, Aditya Kalyanpur, Vladimir Kolovski, and Christian Halaschek-Wiener. 2005. Effective NL paraphrasing of ontologies on the semantic web. In *Proceedings of the ISWC 2005 Workshop on End-user Semantic Web Interaction (CEUR Workshop Proceedings)*. CEUR-WS.org, 9 pages. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-172/.

[42] Feikje Hielkema, Chris Mellish, and Peter Edwards. 2007. Using WYSIWYM to create an open-ended interface for the semantic grid. In *Proceedings of the 11th European Workshop on Natural Language Generation (ENLG'07)*, Stephan Busemann (Ed.). Association for Computer Linguistics, 69–72.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations (ICLR'20)*. OpenReview.net, 16 pages. https://openreview.net/forum?id=rygGQyrFvH.

[44] Bayzid Ashik Hossain, Gayathri Rajan, and Rolf Schwitter. 2019. CNL-ER: A controlled natural language for specifying and verbalising entity relationship models. In *Proceedings of the 17th Annual Workshop of the Australasian Language Technology Association (ALTA'19)*, Meladel Mistica, Massimo Piccardi, and Andrew MacKinlay (Eds.). Australasian Language Technology Association, 126–135.

[45] Mustafa Jarrar, C. Maria Keet, and Paolo Dongilli. 2006. *Multilingual Verbalization of ORM Conceptual Models and Axiomatized Ontologies*. Technical Report. Starlab, Vrije Universiteit Brussel, Belgium.

[46] Kaarel Kaljurand and Norbert E. Fuchs. 2007. Verbalizing OWL in attempto controlled English. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions (CEUR Workshop Proceedings, Vol. 258)*, Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia (Eds.). CEUR-WS.org, 10 pages.

[47] Francis Katamba. 2014. Bantu nominal morphology. In *The Bantu Languages*, Derek Nurse and Gérard Philippson (Eds.). Routledge, Chapter 7, 103–120.

[48] C. Maria Keet and Langa Khumalo. 2016. On the verbalization patterns of part-whole relations in isiZulu. In *Proceedings of the 9th International Conference on Natural Language Generation (INLG'16)*. Association for Computer Linguistics, 174–183.

[49] C. Maria Keet and Langa Khumalo. 2017. Grammar rules for the isiZulu complex verb. *Southern African Linguistics and Applied Language Studies* 35, 2 (2017), 183–200.

[50] C. Maria Keet and Langa Khumalo. 2017. Toward a knowledge-to-text controlled natural language of isiZulu. *Language Resources and Evaluation* 51, 1 (2017), 131–157.

[51] C. Maria Keet, Musa Xakaza, and Langa Khumalo. 2017. Verbalising OWL ontologies in isiZulu with Python. In *The Semantic Web: ESWC 2017 Satellite Events (LNCS, Vol. 10577)*, Eva Blomqvist, Katja Hose, Heiko Paulheim, Agnieszka Lawrynowicz, Fabio Ciravegna, and Olaf Hartig (Eds.). Springer, 59–64.

[52] Langa Khumalo. 2007. *An Analysis of the Ndebele Passive Construction*. Ph.D. Dissertation. University of Oslo, Norway.

[53] Kevin Knight and Vasileios Hatzivassiloglou. 1995. Two-level, many-paths generation. *CoRR* (1995), 9 pages. arXiv:cmp-lg/9506010.

[54] Ravi Kondadadi, Blake Howald, and Frank Schilder. 2013. A statistical NLG framework for aggregated planning and realization. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL'13), Volume 1: Long Papers*. Association for Computer Linguistics, 1406–1415.

[55] Zewang Kuanzhuo, Li Lin, and Weina Zhao. 2020. SimpleNLG-TI: Adapting SimpleNLG to Tibetan, See [27], 86–90.

[56] Irene Langkilde. 2000. Forest-based statistical sentence generation. In *6th Applied Natural Language Processing Conference (ANLP'00)*. Association for Computational Linguistics, 170–177.

[57] Irene Langkilde-Geary. 2002. An empirical verification of coverage and correctness for a general-purpose sentence generator. In *Proceedings of the International Natural Language Generation Conference*. Association for Computational Linguistics, 17–24.

[58] François Lareau, Florie Lambrey, Ieva Dubinskaite, Daniel Galarreta-Piquette, and Maryam Nejat. 2018. GenDR: A generic deep realizer with complex lexicalization. In *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC'18)*, Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Kôiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga (Eds.). European Language Resources Association (ELRA), 3018–3025.

[59] Benoit Lavoie and Owen Rainbow. 1997. A fast and portable realizer for text generation systems. In *5th Applied Natural Language Processing Conference (ANLP'97)*. Association for Computer Linguistics, 265–268. https://doi.org/10.3115/974557.974596

[60] Fennie Liang, Robert Stevens, and Alan L. Rector. 2011. OntoVerbal-M: A multilingual verbaliser for SNOMED CT. In *Proceedings of the 2nd International Workshop on the Multilingual Semantic0 Web (CEUR Workshop Proceedings, Vol. 775)*, Elena Montiel-Ponsoda, John P. McCrae, Paul Buitelaar, and Philipp Cimiano (Eds.). CEUR-WS.org, 13–24.

[61] Shao Fen Liang, Robert Stevens, Donia Scott, and Alan L. Rector. 2011. Automatic verbalisation of SNOMED classes using ontoverbal. In *Artificial Intelligence in Medicine - 13th Conference on Artificial Intelligence in Medicine (AIME'11), Proceedings (Lecture Notes in Computer Science, Vol. 6747)*, Mor Peleg, Nada Lavrac, and Carlo Combi (Eds.). Springer, 338–342. https://doi.org/10.1007/978-3-642-22218-4

[62] Shin Huei Lim and Terry A. Halpin. 2016. Automated verbalization of ORM models in malay and mandarin. *International Journal of Information System Modeling and Design* 7, 4 (2016), 1–16.

[63] Olga Lyudovyk and Chunhua Weng. 2019. SNOMEDtxt: Natural language generation from SNOMED ontology. In *Proceedings of the 17th World Congress on Medical and Health Informatics (MEDINFO'19): Health and Wellbeing e-Networks for All (Studies in Health Technology and Informatics, Vol. 264)*, Lucila Ohno-Machado and Brigitte Séroussi (Eds.). IOS Press, 1263–1267.

[64] Zola Mahlaza. 2018. *Grammars for Generating isiXhosa and isiZulu Weather Bulletin Verbs*. MSc. Thesis. Department of Computer Science, University of Cape Town, South Africa.

[65] Zola Mahlaza and C. Maria Keet. 2020. Formalisation and classification of grammar and template-mediated techniques to model and ontology verbalisation. *International Journal of Metadata, Semantics and Ontologies* 14, 3 (2020), 249–262.

[66] Zola Mahlaza and C. Maria Keet. 2020. OWLSIZ: An isiZulu CNL for structured knowledge validation. In *Proceedings of the 3rd International Workshop on Natural Language Generation from the Semantic Web (WebNLG+'20)*. Association for Computational Linguistics (Virtual), 15–25.

[67] Zola Mahlaza and C. Maria Keet. 2021. ToCT: A task ontology to manage complex templates. In *Proceedings of the Joint Ontology Workshops 2021 Episode VII: The Bolzano Summer of Knowledge co-located with the 12th International Conference on Formal Ontology in Information Systems (FOIS'21), and the 12th International Conference on Biomedical Ontologies (ICBO'21) (CEUR Workshop Proceedings, Vol. 2969)*, Emilio M. Sanfilippo, Oliver Kutz, Nicolas Troquard, Torsten Hahmann, Claudio Masolo, Robert Hoehndorf, Randi Vita, Maria M. Hedblom, Guendalina Righetti, Dusan Sormaz, Walter Terkaj, Tiago Prince Sales, Sergio de Cesare, Frederik Gailly, Giancarlo Guizzardi, Mark Lycett, Chris Partridge, Oscar Pastor, Daniel Beßler, Stefano Borgo, Mohammed Diab, Aldo Gangemi, Alberto Olivares Alarcos, Mihai Pomarlan, Robert Porzel, Ludger Jansen, Mathias Brochhausen, Daniele Porello, Pawel Garbacz, Selja Seppälä, Michael Grüninger, Amanda Vizedom, Damion M. Dooley, Robert Warren, Hande Küçük-McGinty, Matthew Lange, Alsayed Algergawy, Naouel Karam, Friederike Klan, Franck Michel, and Ilaria Rosati. (Eds.). CEUR-WS.org, 9 pages. http://ceur-ws.org/Vol-2969/paper40-FoisShowCase.pdf.

[68] François Mairesse, Milica Gasic, Filip Jurcícek, Simon Keizer, Blaise Thomson, Kai Yu, and Steve J. Young. 2010. Phrase-based statistical language generation using graphical models and active learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10)*, Jan Hajic, Sandra Carberry, and Stephen Clark (Eds.). Association for Computer Linguistics, 1552–1561.

[69] Laurette Marais, Johannes A. Louw, Jaco Badenhorst, Karen Calteaux, Ilana Wilken, Nina van Niekerk, and Glenn Stein. 2020. AwezaMed: A multilingual, multimodal speech-to-speech translation application for maternal health care. In *IEEE 23rd International Conference on Information Fusion (FUSION'20)*. IEEE, 1–8. https://doi.org/10.23919/FUSION45008.2020.9190240

[70] Laurette Marais and Laurette Pretorius. 2020. Exploiting a multilingual semantic machine translation architecture for knowledge representation of patient data for Covid-19. In *Proceedings of the 1st Southern African Conference for Artificial Intelligence Research (SACAIR'20)*. Springer, 264–279. https://sacair.org.za/wp-content/uploads/2021/01/SACAIR_Proceedings-MainBook_vFin_sm.pdf#page=279.

[71] Tomasz Marciniak and Michael Strube. 2004. Classification-based generation using TAG. In *Proceedings of the 3rd International Conference on Natural Language Generation (INLG'04) (LNCS, Vol. 3123)*, Anja Belz, Roger Evans, and Paul Piwek (Eds.). Springer, 100–109.

[72] Laura Martinus and Jade Z. Abbott. 2019. A focus on neural machine translation for african languages. *arXiv e-prints* (June 2019), 11 pages. arXiv:1906.05685 [cs.CL].

[73] Alessandro Mazzei, Cristina Battaglino, and Cristina Bosco. 2016. SimpleNLG-IT: Adapting SimpleNLG to Italian. In *Proceedings of the 9th International Conference on Natural Language Generation (INLG'16)*. Association for Computer Linguistics, 184–192.

[74] Alessandro Mazzei, Michele Monticone, and Cristian Bernareggi. 2019. Using NLG for speech synthesis of mathematical sentences. In *Proceedings of the 12th International Conference on Natural Language Generation*. Association for Computational Linguistics, 463–472.

[75] Susan W. McRoy, Songsak Channarukul, and Syed S. Ali. 2000. YAG: A template-based generator for real-time systems. In *Proceedings of the 1st International Conference on Natural Language Generation - Volume 14 (INLG'00)*. Association for Computational Linguistics, 264–267.

[76] Chris Mellish, Mike Reape, Donia Scott, Lynne Cahill, Roger Evans, and Daniel Paiva. 2004. A reference architecture for generation systems. *Natural Language Engineering* 10, 3–4 (2004), 227–260.
[77] Chris Mellish, Donia Scott, Lynne J. Cahill, Daniel S. Paiva, Roger Evans, and Mike Reape. 2006. A reference architecture for natural language generation systems. *Natural Language Engineering* 12, 1 (2006), 1–34.
[78] Paul Molins and Guy Lapalme. 2015. JSrealB: A bilingual text realizer for web programming. In *Proceedings of the 15th European Workshop on Natural Language Generation (ENLG'15)*, Anja Belz, Albert Gatt, François Portet, and Matthew Purver (Eds.). Association for Computer Linguistics, 109–111.
[79] Carmen Moors, Ilana Wilken, Karen Calteaux, and Tebogo Gumede. 2018. Human language technology audit 2018: Analysing the development trends in resource availability in all South African languages. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT'18)*. ACM, 296–304.
[80] Amit Moryossef, Yoav Goldberg, and Ido Dagan. 2019. Step-by-Step: Separating planning from realization in neural data-to-text generation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19), Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 2267–2277.
[81] Hiroko Nakanishi, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic models for disambiguation of an HPSG-based chart generator. In *Proceedings of the 9th International Workshop on Parsing Technology (IWPT'05)*, Harry Bunt, Robert Malouf, and Alon Lavie (Eds.). Association for Computational Linguistics, 93–102.
[82] Evander Nyoni and Bruce A. Bassett. 2021. Low-resource neural machine translation for Southern African languages. *arXiv e-prints* (April 2021), 8 pages. arXiv:2104.00366 [cs.CL].
[83] Alejandro Ramos-Soto, Julio Janeiro Gallardo, and Alberto José Bugarín Diz. 2017. Adapting SimpleNLG to Spanish. In *Proceedings of the 10th International Conference on Natural Language Generation (INLG'17)*, José M. Alonso, Alberto Bugarín, and Ehud Reiter (Eds.). Association for Computational Linguistics, 144–148.
[84] Ehud Reiter. 1994. Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *Proceedings of the 7th International Workshop on Natural Language Generation (INLG'94)*. Association for Computer Linguistics, 9 pages.
[85] Ehud Reiter. 2007. An architecture for data-to-text systems. In *Proceedings of the 11th European Workshop on Natural Language Generation (ENLG'07)*, Stephan Busemann (Ed.). Association for Computational Linguistics, 97–104.
[86] Ehud Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press, Cambridge, England.
[87] Lauren Sanby, Ion Todd, and C. Maria Keet. 2016. Comparing the template-based approach to GF: The case of Afrikaans. In *Proceedings of the 2nd International Workshop on Natural Language Generation and the Semantic Web (WebNLG'16)*, Claire Gardent and Aldo Gangemi (Eds.). Association for Computational Linguistics, 50–53.
[88] Koenraad De Smedt, Helmut Horacek, and Michael Zock. 1993. Architectures for natural language generation: Problems and perspectives. In *Trends in Natural Language Generation, An Artificial Intelligence Perspective, 4th European Workshop (EWNLG '93), Selected Papers (Lecture Notes in Computer Science, Vol. 1036)*, Giovanni Adorni and Michael Zock (Eds.). Springer, 17–46. https://doi.org/10.1007/3-540-60800-1
[89] Holger Stenzhorn. 2002. XtraGen - A natural language generation system using XML and Java-Technologies. In *The 2nd Workshop on NLP and XML (NLPXML@COLING'02)*. Association for Computer Linguistics, 8 pages.
[90] Robert Stevens, James Malone, Sandra Williams, Richard Power, and Allan Third. 2011. Automating generation of textual class definitions from OWL to English. *Journal of Biomedical Semantics* 2, S-2 (2011), S5.
[91] Elke Teich. 1999. *Systemic Functional Grammar in Natural Language Generation*. Cassell, London, UK.
[92] Kees van Deemter, Mariët Theune, and Emiel Krahmer. 2005. Real versus template-based natural language generation: A false opposition? *Computational Linguistics* 31, 1 (2005), 15–24.
[93] Pierre-Luc Vaudry and Guy Lapalme. 2013. Adapting SimpleNLG for bilingual English-French realisation. In *Proceedings of the 14th International Conference on Natural Language Generation (ENLG'13)*. . Association for Computer Linguistics, 183–187.
[94] Mark J. Weal, Harith Alani, Sanghee Kim, Paul H. Lewis, David E. Millard, Patrick A. S. Sinclair, David De Roure, and Nigel R. Shadbolt. 2007. Ontologies as facilitators for repurposing web documents. *International Journal of Human-Computer Studies* 65, 6 (2007), 537–562.
[95] Michael White. 2006. CCG chart realization from disjunctive inputs. In *Proceedings of the 4th International Natural Language Generation Conference (INLG'06)*, Nathalie Colineau, Cécile Paris, Stephen Wan, Robert Dale, and Anja Belz (Eds.). Association for Computer Linguistics, 12–19.
[96] Yuk Wah Wong. 2007. *Learning for Semantic Parsing and Natural Language Generation Using Statistical Machine Translation Techniques*. Ph.D. Dissertation. Department of Computer Sciences, University of Texas at Austin, Texas.

Received 18 August 2021; revised 22 September 2022; accepted 4 October 2022