The Journal of Systems and Software 216 (2024) 112148

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

BIT: A template-based approach to incremental and bidirectional model-to-text transformation ☆,☆☆

Xiao He [a], Tao Zan [b,*]

[a] School of Computer and Communication Engineering, University of Science and Technology Beijing, No. 30, Xueyuan Road, Haidian district, 100083, Beijing, China

[b] Longyan University, No. 1, North Dongxiao Road, Xinluo district, 364012, Longyan, China

ARTICLE INFO

Keywords:
Bidirectional transformation
Model-to-text transformation
Template language
Model-driven development
Round-trip engineering

ABSTRACT

Model-driven development is a model-centric software development paradigm that automates the development process by converting high-level models into low-level code and documents. To maintain synchronization between models and code/documents — which can evolve independently — this paper introduces BIT, a bidirectional language that can serve as a conventional template language for model-to-text transformations. However, a BIT program can function as both a *printer*, generating text by filling template holes with values from the input model, and a *parser*, putting parsed values back into the model. BIT comprises a surface language for better usability and a core language for formal definition. We define the semantics of the core language based on the theory of bidirectional transformation, and provide the translation from the surface to the core. We present the proof sketch of the well behavedness of BIT as a formal evidence of soundness. We also conduct three case studies to empirically demonstrate the expressiveness and the effectiveness of BIT. Based on the proof and the case studies, BIT covers the major features of existing template languages, and offers sufficient expressiveness to define real-world model-to-text transformations that can be executed bidirectionally and incrementally.

1. Introduction

Model-driven development (MDD) is a model-centric software development paradigm (Object Management Group, 2024; Brown, 2004; Rodrigues da Silva, 2015) that has been intensively studied and applied in both academia and industry over past decades (Umuhoza and Brambilla, 2016; Boussaïd et al., 2017; Akdur et al., 2018). In MDD, a software system is generally developed and maintained by (1) specifying the system models at the high abstraction level and then (2) transforming the models into some low-level artifacts, including low-level models, source code, and documents, using model-to-model transformation (Kahani et al., 2019) and model-to-text (M2T) transformation (Object Management Group, 2008).

M2T transformations are typically realized using *templates* (Syriani et al., 2018). A template, which consists of text literals, holes, and control directives, is a unidirectional transformation from models to text (e.g., code and documents). For example, Fig. 1 illustrates a simple template *generateJavaClass* written in Xtend (Anon, 2023h). This template generates a Java class from a UML class. Assuming that the input UML class (as shown in Fig. 1(b)) is provided, a snippet of Java code (as shown in Fig. 1(c)) will be generated.

In practice, it is inevitable for developers to manually modify and customize the generated code/documents (He et al., 2016). For instance, developers may modify the code as shown in Fig. 1(d), where field tel is deleted and field age is added. Consequently, the UML class (Fig. 1(b)) and the code become inconsistent.

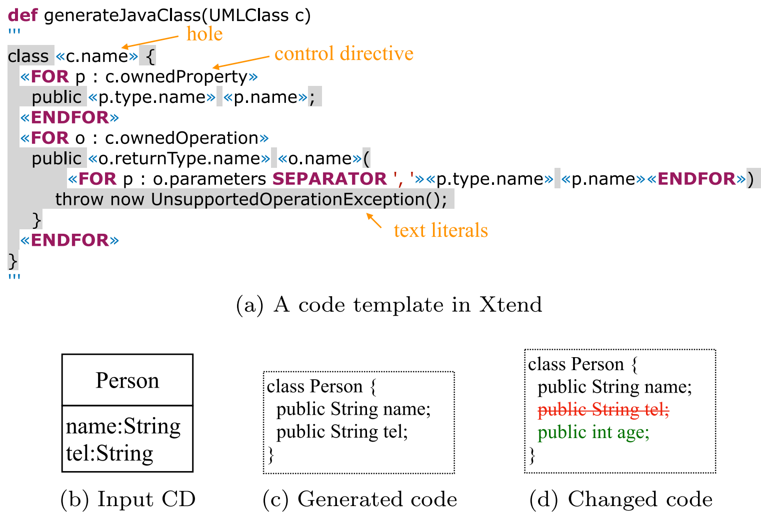
How to synchronize high-level models and derived artifacts to maintain their consistency has become a fundamental challenge in model-driven community. Numerous research efforts have been made on the synchronization over models (i.e., graph-like data structures) (Hermann et al., 2015; Giese and Wagner, 2009; Hermann et al., 2012; Orejas et al., 2020; Xiong et al., 2013; Macedo and Cunha, 2016; Samimi-Dehkordi et al., 2018; Buchmann et al., 2022; Boronat, 2023; He and Hu, 2018; He et al., 2022). Nevertheless, there are only a few generic solutions (Lemerre, 2023) for synchronizing models and text. A practical way of model-code synchronization, as employed in

☆ This work is funded by National Key Research and Development Program of China (No. 2023YFB3002903), Natural Science Foundation of Fujian Province for Youths (No. 2021J05230), Beijing Natural Science Foundation (No. 4192036).

☆☆ Editor: Prof. Raffaela Mirandola.

* Corresponding author.
E-mail addresses: hexiao@ustb.edu.cn (X. He), zan@lyun.edu.cn (T. Zan).

https://doi.org/10.1016/j.jss.2024.112148
Received 28 December 2023; Received in revised form 8 May 2024; Accepted 20 June 2024
Available online 27 June 2024



(a) A code template in Xtend

(b) Input CD (c) Generated code (d) Changed code

Fig. 1. A template example and its application.

many model-driven tools (e.g., Eclipse Modeling Framework (EMF) and Papyrus), is to develop a separate reverse engineering module, as a companion of the code generator, which can convert the text back to the original model. However, this practical solution has three significant limitations as follows.

1. It requires more development costs to implement both the code generator and the reverse engineering module.
2. A code generator and the corresponding reverse engineering module are expected to have consistent behaviors: if code C is generated from model M, the reverse engineering module should derive a model M' from C, such that M and M' are identical; if model M is reverse-engineered from code C, the code generator should produce code C' from M, such that C and C' are identical. Because they are *independently developed* and *algorithmically different*, ensuring their behavioral consistency is challenging.
3. This solution typically assumes that the textual data to be synchronized is equipped with a parser and a pretty printer. However, such a prerequisite does not always hold, particularly for plain text, documents, and even comments in source code.

Bidirectional transformation (BX) (Ko and Hu, 2017; Barbosa et al., 2010; Foster et al., 2005; Hu et al., 2008; Hidaka et al., 2010, 2013; Tran et al., 2020) can serve as the foundation of data synchronization. A BX program is a *single* specification that can be *consistently* evaluated in both forward and backward directions. Following the principles of BX, Yu et al. (2012) proposed a framework for model-code synchronization that facilitates bidirectional conversion between Java code and Ecore models. However, their approach is not generally applicable to other kinds of text, such as HTML pages, Graphviz images, and documents.

In this paper, we aim to introduce a novel template-based approach, called BIT, for model-text synchronization. BIT enables developers to write a single template that can be interpreted as a M2T transformation (known as a *printer*), similar to existing template engines. It can also be used to automatically derive a reverse engineering module (known as a *parser*) to reduce development costs. Furthermore, BIT ensures in theory that the derived printer and parser exhibit behavioral compatibility (i.e., they satisfy the round-trip properties). Specifically, we first propose a general-purpose template language for developers to specify M2T transformation. For better usability, our template language, which serves as a surface language, largely inherits the grammar of Xtend, with a few syntactic extensions to enable backward evaluation. Second, we design a core language, to which our template language can be translated. The core language consists of 5 primitive BXs and 8 combinators. A primitive BX tells how to parse/print a specific value according to a certain format, and a combinator allows us to combine smaller BXs into a larger one.

This paper focuses on the following two challenges that hinder the application of existing approaches to the synchronization between models and text.

1. Existing BX approaches are defined upon structured data (e.g., trees (Hu et al., 2008), relational databases (Tran et al., 2020), graphs (Hidaka et al., 2010, 2013), and models (He and Hu, 2018)). However, a string, which is a sequence of characters, is generally considered as unstructured. To address the unstructured nature of strings, we ask template developers to annotate each template hole with a lexical rule, so that the derived parser can determine the boundary of the string generated by the hole for the given input string. Furthermore, we adopt the mechanism of partial grammars (inspired by van Tonder and Le Goues, 2019) to analyze the structure of strings.
2. Existing BX approaches usually assume that BXs are pure functions. However, in our template-based bidirectional printing/parsing, some computations, such as local assignments and incremental parsing, require computational side effects. To handle these effects, we propose the concepts of *accumulative BXs* and *effect-binding BXs* to manage incremental parsing and local assignments, respectively.

The rest of this paper is structured as follows. Section 2 introduces the background information and presents a demonstration of our approach. Section 3 presents the detailed definitions of the surface language and the core language of BIT. Section 4 discusses the proof sketch of the well behavedness of the BIT semantics. Section 5 presents three case studies. Section 6 discusses the related work. The last section concludes the paper and future work.

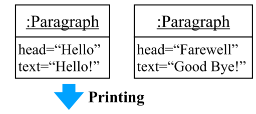
2. Background and demonstration

2.1. Bidirectional transformation

A bidirectional transformation (BX) is a program that bidirectionally converts between the source type S and the view type V. An asymmetric BX, written as $S \leftrightarrow V$, can be viewed as a pair (get, put) of functions. The forward transformation $get : S \to V$ generates a view value from the source, while the backward transformation $put : S \times V \to S$ updates the original source by taking the modified view into account.

```
1.  def someHTML(List<Paragraph> paragraphs)
2.  '''
3.    <html>
4.      <body>
5.        «var no = 0»
6.        «FOR p : paragraphs»
7.          «IF p.head != null»
8.          «no = no + 1»
9.            <h1>«no».«p.head.toUpperCase»</h1>
10.         «ENDIF»
11.         <p>
12.           «p.text»
13.         </p>
14.       «ENDFOR»
15.     </body>
16.   </html>
17. '''
```

(a) Xtend template




```
<html>
 <body>
  <h1>1.HELLO</h1>
  <p>
   Hello!
  </p>
  <h1>2.FAREWELL</h1>
  <p>
   Good Bye!
  </p>
 </body>
</html>
```

(b) Generated HTML

Fig. 2. An Xtend template and its generated HTML document.

A pair (get, put) of functions form a *well-behaved* BX iff. they satisfy the corresponding round-trip properties. For asymmetric BXs, the following round-trip properties must hold:

$$put\ s\ (get\ s) = s \qquad \text{(GetPut)}$$

$$get\ (put\ s\ v) = v \qquad \text{(PutGet)}$$

(GetPut) law states that updating the source s with the unmodified view generated from s should not cause any changes to s, while (PutGet) law states that if we perform the forward transformation immediately after the backward transformation with the view v, we should obtain the same v.

Consider a concrete example. Assume that

$$get_{head}\ [x_1, x_2, \ldots, x_n] = x_1$$
$$put_{head}\ [x_1, x_2, \ldots, x_n]\ x_1' = [x_1', x_2, \ldots, x_n]$$

The forward transformation extracts the head element x_1 of a source array $[x_1, x_2, \ldots, x_n]$ as the view value; the backward transformation simply replaces the head element of the original source array with the given view value x_1' to produce an updated array. It is easy to verify that both GetPut and PutGet laws hold, so the two functions form a well-behaved BX.

Bidirectional programming is a programming paradigm that enables developers to define a single specification from which a well-behaved BX program can be derived, thereby minimizing the development efforts. There are three basic approaches to bidirectional programming: the get-based, the putback-based, and the relational approach. The get-based approach (Xiong et al., 2013; Hidaka et al., 2010) derives the backward transformation *put* from the forward transformation *get*, while the putback-based approach (Ko and Hu, 2017; He and Hu, 2018; Tran et al., 2020) derives *get* from the backward transformation *put*; and the relational one (Hermann et al., 2015) derives both *get* and *put* from a set of consistency relations over the source and the view.

2.2. Xtend templates

Xtend is a dialect of Java that improves on many aspects of Java, such as extension methods, operator overloading, and template expressions. Xtend has been used in mobile development, Web development, and model-driven domain-specific language engineering. In particular, the template expressions in Xtend allow for readable string concatenation and text generation, which are frequently used for code/document generation.

Fig. 2(a) shows an Xtend template. In Xtend, templates are surrounded by triple single quotes ('''); template holes and control directives are placed within «and ». For example, «`p.head.toUpperCase`» in line 9 is a template hole, which is intended to replace the placeholder with the evaluation result of `p.head.toUpperCase` at runtime. As for control directives, Xtend templates support loops (e.g., lines 6–14), conditions (e.g., lines 7–10), and assignments (e.g., lines 5 and 8). Within an Xtend template, other templates may be invoked.

Xtend compiles the template in Fig. 2(a) into a Java method. If we input a list of paragraphs (each paragraph consisting of a head and a text field), the method generates HTML code by filling in the field values in the holes. For example, supposing that the input paragraphs are written in textual form[1] as [{head="Hello",text="Hello!"}, {head="Farewell",text="Good Bye!"}], the template will produce HTML code as shown in Fig. 2(b).

If we want to modify the generated text (e.g., we want to change "1.HELLO" in Fig. 2(b) to "1.GREETING") and keep the text consistent with the input data, we must go back to the input and locate the fields that affect the text fragment to be changed. After modifying the input, we must re-run the text generation to see if the text is updated as expected.

2.3. A taste of our approach

Fig. 3(a) shows the template defined in our BIT approach, which corresponds to the Xtend template in Fig. 2(a). A BIT template shares a similar syntax to an Xtend template, with the key difference being that in the BIT template, each template hole is annotated with a lexical rule that guides our approach in the parsing mode. For example, «`no | INT`» in line 10 indicates that this hole will be filled with a string that is produced by the expression `no` and conforms to lexical rule `INT`, where the rule is defined by regular expression -?[0-9]+. If the lexical rule is missing (e.g., the hole in line 13), then our tool implementation will try to infer a lexical rule.

A BIT template, e.g., Fig. 3(a), can function as a conventional template, generating the text depicted in Fig. 2(b) when supplied with the identical model. Nevertheless, BIT distinguishes itself from conventional template languages in the following aspects.

- Supposing that, after reading the generated HTML file, a user finds some typos and missing data, our approach allows for the direct modification to the generated text for correction. By bidirectonalizing the template, it can propagate text changes back to the input. As shown in Fig. 3(b), we change the text in Fig. 2(b) by alerting the content of the first h1, adding a new fragment "APPRECIATION", and adjusting the spaces. The derived parser reads the changed text and updates the input to [{head="Greeting", text="Hello!"}, {head="Farewell", text= "Good Bye!"}, {head="appreciation", text="Thanks!"}].

[1] For simplicity, we may represent a model, e.g., the one in Fig. 2(b), in a JSON-like format, which can be supported by our tool.

```
type Paragraph = {head:String, text:String}
template someHTML(paragraphs:[Paragraph])
'''
 <html>
  <body>
   «var no = 0»
   «FOR p:Paragraph IN paragraphs»
    «IF p.head != null»
    «no = no + 1»
     <h1>«no|INT».«p.head.toUpperCase»</h1>
    «ENDIF»
    <p>
     «p.text»
    </p>
   «ENDFOR»
  </body>
 </html>
'''
```

(a) BIT template example

```
<html>
 <body>
  <h1>1. GREETING</h1>
  <p>
   Hello!
  </p>
  <h1>2. FAREWELL</h1>
  <p>Good Bye!</p>
  <h1>3. APPRECIATION</h1>
  <p>
   Thanks!
  </p>
 </body>
</html>
```

Parsing

```
[{head="Greeting",text="Hello!"},
 {head="Farewell",text="Good Bye!"},
 {head="appreciation",text="Thanks!"}]
```

(b) Parsing modified text

Fig. 3. Demonstration of BIT template (colored background shows the changed text layout)

- Suppose that a user has adjusted the spacing in generated text for enhanced formatting and wishes to refresh the content through re-generation. Our approach facilitates incremental printing atop the old string. As shown in Fig. 4(a), if we print [{head="Modeling", text="UML"}, {head= "Programming", text="Java"}, {head= "Appreciation", text="Thanks!"}] based on the prior modifications depicted in Fig. 3(b), BIT updates the text with new values while retaining the original string's whitespace layout wherever feasible. Note that our approach keeps not just white-spaces but also non-whitespaces during incremental printing if specific directives (e.g., DEFAULT construct, see Section 3.1) are applied.
- Our approach further encompasses incremental parsing, as demonstrated in Fig. 4(b). The string for parsing mirrors that in Fig. 3(b). Yet, when an original value is introduced, which includes an extra paragraph with a head of "Some title", the parsing outcome diverges from what is seen in Fig. 3(b). Notably, the header of the third paragraph in the parsed result transforms to "Appreciation", instead of remaining as "appreciation", due to the capitalization of the first character "S" in the original head.

3. The BIT approach

The overview of our approach is depicted in Fig. 5. BIT is designed for the bidirectional transformation between models and text. Firstly, BIT provides a surface language, with which text templates can be specified. Secondly, the templates are translated into the core language of BIT, from which a model-to-text transformation (i.e., *print*) and a text-to-model transformation (i.e., *parse*) are automatically derived. Note that our approach bidirectionally converts between text with tree-like models, which can be defined by algebraic data types. For generality, the type system of BIT is not coupled with EMF ecosystem. The conversion between tree-like models with graph-like models (and EMF models) can further be achieved by existing BX approaches over models.

Section 3.1 introduces the surface language. Section 3.2 discusses the formal foundation, explaining the concepts of accumulative BXs and effect-binding BXs and defining the generic structure of a BIT primitive. Section 3.3 defines the core language that contains 5 primitives and 8 combinators. Section 3.4 describes how to translate the surface language into the core. Section 3.5 discusses some issues of BIT.

3.1. The surface language

BIT is a template-based approach for synchronizing models and text. BIT allows developers to define text templates, just like the ones in classical MDD. Subsequently, BIT derives a BX program, consisting both a printer and a parser, from these text templates. To facilitate the adoption of BIT, we have defined a surface language for developers, whose essential grammar is shown in Fig. 6.

For simplicity, the Type t in BIT can be a record type $\{f_1 : t_1, f_2 : t_2, \ldots\}$ (e.g., {name:String, age:int}), a tuple type $(t_1, t_2, \ldots)$ (e.g., (String, int)), a list type $[t']$ (e.g., [int]), or a primitive type (i.e., int, String, and boolean). It is possible to name a type in BIT, so that the type can be referred by this name. For example, in Fig. 3(a), Paragraph is defined as {head:String, text:String}.

The ValueLiteral v is a value of a specific type. It can be a record (e.g., {head="Greeting", text="Hello!"}), a tuple (e.g., ("Hello", 1)), a list (e.g., [1,2,3,4,5]), or a primitive value (e.g., "Hello", 1, true, false, and null).

We assume that the input value (i.e., the model to be synchronized) of a template be encoded as a record. For example, the input of the template in Fig. 3(a) is a record containing {paragraphs=[{head=..., text=...},{head=...,text=...}, ...].

Just like conventional programming languages, expressions (Expr) in BIT include basic arithmetic expressions (e.g., +, -), relational expressions (e.g., ==, !=, >, <), boolean expressions (e.g., &&, ||, !), instanceof expressions, path calls, and template calls (i.e., a CallExpr not occurring in a path call). We assume that every expression, except for the template call, is equipped with bidirectional semantics, as defined in existing BX languages (Xiong et al., 2013; Zhang and Hu, 2022; Zhang et al., 2023); while the semantics of template calls is specified in Section 3.3.

At first glance, the surface language appears to have a similar syntax to the Xtend template expressions. A BIT template (Template) starts with keyword template, followed by a template name and a parameter list. The body of a template is a TempFragment surrounded by '''. In brief, a TempFragment is a string with template holes (Hole) and control directives (Control), e.g.,

<h1>«p.head|ID»</h1>

A hole/control directive is a construct marked by ≪ and ≫. Like existing template languages, a hole specifies the dynamic content that will be filled during text generation, i.e., the result of the hole expression (e.g., p.head). BIT requires a hole to be annotated with a lexical rule (LexRule) to facilitate deriving a parser for that hole. A LexRule is a regular expression or a rule name bound to a regular expression, e.g., ID refers to [_a-zA-Z][_a-zA-Z0-9]*.

BIT supports common control directives, including loops and conditionals.

A loop (i.e., the FOR construct) is used to print values in a list. Fig. 7(a) shows a concrete loop example that aims to print a list of Strings. Line 3 defines an iterator variable i:String to enumerate the strings in list. For each string bound to i, line 5 prints it with the hole «i|ID». Similar to Xtend, we can specify a separator string, and starting/ending strings (see line 4). The separator string will be inserted automatically between two consecutive iterations; the starting and ending strings will be appended before and after the loop that has at least one iteration, respectively. For the template in Fig. 7(a),

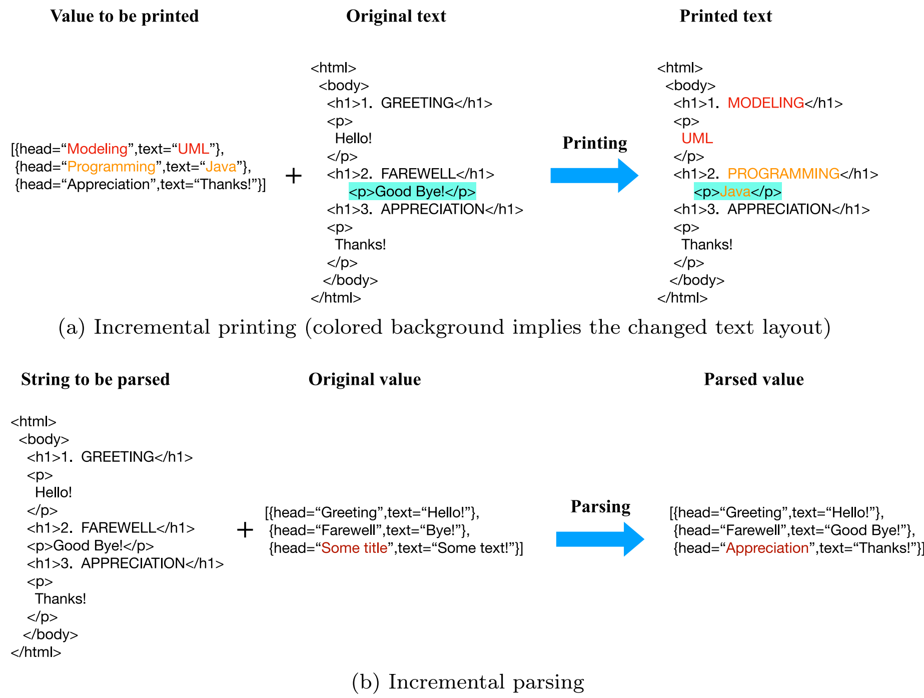


Fig. 4. Incremental printing and parsing in our approach.

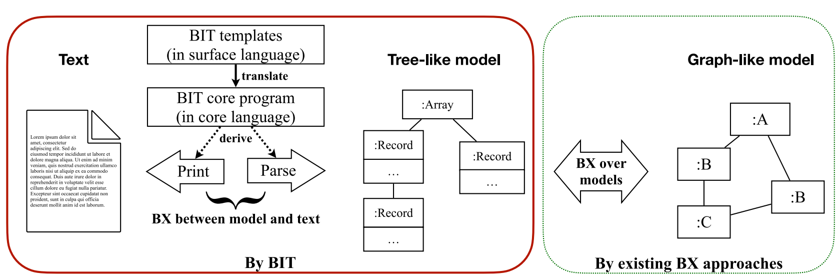


Fig. 5. Approach overview.

if list=["a","b"], then it yields string "[a,b]"; if list=["a"], then it prints out "[a]"; however, if list is empty, then it produces an empty string. In parsing mode, a loop repeatedly applies the body fragment to parse the input string, resulting a list of values, each parsed by the body. Subsequently, it updates the model using the list.

A conditional (i.e., the IF construct) prints different branches according to the branch condition. As shown in Fig. 7(b), this template aims to print a string v: if the string length is greater than 10 (line 3), then it prints the first 10 characters and appends "..." (line 4); otherwise, it prints the entire string (line 6). For instance, if v="abcdefghight", then the template selects the first branch and prints out "abcdefghig...". In theory, the ELSE branch is required. If the ELSE branch is missing in practice, then BIT will append a pseudo else-branch that prints nothing. In parsing mode, a conditional tries to parse the input string with different branches. It will select the branch that successfully consumes the greatest number of characters from the input, and subsequently updates the model to reflect the conditions met by the chosen branch.

BIT supports local variable definitions and assignments, which is an important feature in many template languages. For example, as shown in Fig. 7(c), line 3 defines a local variable v:int and initializes it with 0; line 4 assigns 1 to v so that in the rest of the template, v refers to 1, rather than 0. If we run this template, then we shall get a string "0 1". In parsing mode, BIT carefully keeps track of these assignments so as to correctly update the model.

BIT also supports the verbatim area, i.e., $\ll ! \ll \cdots \gg ! \gg$, which outputs any content within this area as plain text, ignoring other BIT directives.

In addition, BIT provides three extra control directives, namely, DEFAULT, UNORD, and FINAL, to enrich the bidirectional behavior of BIT templates.

The DEFAULT construct is used to print some default text. During parsing, it accepts a string that conforms to a lexical rule, even if the string is different from the default one. Consider the case of generating a method declaration in a Java interface. Fig. 7(d) shows a tiny example where the template generates a method m(int value) with an integer parameter whose default name is value. The DEFAULT construct in the template is responsible for printing "value" initially. Developers may change the parameter name arbitrarily (e.g., changing the method signature to m(int arg)). In parsing mode, the template acknowledges deviations in the parameter name from the default "value", and will not overwrite arg during incremental printing.

The UNORD construct generates a list of strings whose order may vary. For instance, Java modifiers (e.g., static and public) may occur in any order. UNORD is designed to handle this case, as shown in Fig. 7(e): when printing, if the original string is empty, it prints "static" and "public" sequentially; if the original string is not empty (e.g., "public static"), it keeps the original one; when parsing, both "static public" and "public static" are recognized and accepted.

```
TemplateUnit := TemplateList
    Template := template Name ( ParameterListComma ) ''' TempFragment '''
        Type := record types, tuple types, list types, and primitive types
   Parameter := VariableDecl
VariableDecl := Name : Type
TempFragment := TempLiteral ‖ TempLiteral HoleOrControl TempFragment
HoleOrControl := Hole ‖ Control
        Hole := «Expr | LexRule»
     Control := «IF Expr» TempFragment ElseBranch «ENDIF»
              ‖ «FOR VariableDecl IN Expr ForLits» TempFragment «ENDFOR»
              ‖ «DEFAULT | LexRule» TempFragment «ENDDEFAULT»
              ‖ «UNORD» TempFragment UnordFragList «ENDUNORD»
              ‖ «FINAL» TempFragment «ENDFINAL»
              ‖ «VAR VariableDecl = Expr» ‖ «Name = Expr» ‖ «!«···»!»
  ElseBranch := «ELSEIF Expr» TempFragment ElseBranch
              ‖ «ELSE» TempFragment ‖ ε
     ForLits := ForLit ForLits ‖ ε
      ForLit := SEPARATOR STRING ‖ BEFORE STRING ‖ AFTER STRING
UnordFragList := || TempFragment ‖ || TempFragment UnordFragList
        Expr := basic arithmetic, relational, and boolean expressions
              ‖ instanceof expressions
              ‖ Expr PathCall ‖ CallExpr ‖ ValueLiteral
    PathCall := . Name PathCall ‖ . CallExpr PathCall ‖ ε
    CallExpr := Name ( ExprListComma )
 TempLiteral := any char sequence that does not contain « and '''
     LexRule := regular expressions or predefined rule names
        Name := identifiers
```

Fig. 6. Essential grammar of the surface language.
Notation:SmallCap denotes non-terminals; SansSeri denotes terminal constants; ε means nothing; if unspecified, a non-terminal X-List (e.g., TemplateList) is expanded into ε or X X-List ‖ X, while X-ListComma is expanded into ε or X, X-ListComma ‖ X.

```
1.  template loopExample(list:[String])
2.  '''
3.  «FOR i:String IN list
4.     SEPARATOR "," BEFORE "[" AFTER "]"»
5.  «i|ID»
6.  «ENDFOR»
7.  '''
```

(a) Loop

```
1.  template branchExample(v:String)
2.  '''
3.  «IF v.length()>10»
4.  «v.substring(0,10)|ID»...
5.  «ELSE»
6.  «v|ID»
7.  «ENDFOR»
8.  '''
```

(b) Branch

```
1.  template varExample()
2.  '''
3.  «VAR v:int=0»
4.  «v|INT» «v=1» «v|INT»
5.  '''
```

(c) Assignment

```
1.  template defaultExample()
2.  '''
3.  m(int «DEFAULT|ID»value«ENDDEFAULT»);
4.  '''
```

(d) Default

```
1.  template unordExample(a:String)
2.  '''
3.  «UNORD»static«||»public«ENDUNORD»
4.  '''
```

(e) Unord

```
1.  template finalExample()
2.  '''
3.  int v«FINAL»;«ENDFINAL»
4.  '''
```

(f) Final

Fig. 7. Examples of control directives.

The FINAL construct outputs a cached string (obtained during parsing) before printing its body fragment. Consider the template of a variable declaration, as shown in Fig. 7(f). Initially, the template prints out "int v;". Because a variable may have an initializer, developers may change the declaration to "int v = 0;". The FINAL construct in line 3 tells the derived parser to skip all the characters after "int v" until it meets ";". In incremental printing mode, the construct preserves these characters before appending the final ";".

3.2. Formalization of bidirectional templates

To specify the semantics of BIT, we should formally define the function signatures of the printer and the parser that are derived from a BIT template, as well as the round-trip properties they must follow. Let us start from the trivial case that a parser and a printer can be defined as the following functions

$$parse : \mathbb{S} \rightarrow V \quad \text{(TrivialParse)}$$

$$print : V \rightarrow \mathbb{S} \quad \text{(TrivialPrint)}$$

where $\mathbb{S}$ denotes the string type and V is a certain value type, function *parse* consumes an input string and yields a value, and function *print* serializes a value to a string. We assume that there is a special string $\perp$ that denotes the *initially empty string*. In string calculation, $\perp$ is treated as "".

Such a simple signature does not support the major features of BIT, including incremental printing, incremental parsing, and local assignments. Our goal is to find an appropriate definition that enables the embedding of these features.

Incremental printing. As illustrated in Fig. 4(a), incremental printing allows for printing a value by rewriting an existing string. To achieve this, the *print* function must accept the original string as an additional input, so that it can determine which parts of the original string should be overwritten and which should be preserved. Hence, *print* must be declared as (IncPrint):

$$print : \mathbb{S} \times V \rightarrow \mathbb{S} \quad \text{(IncPrint)}$$

Obviously, (TrivialParse) and (IncPrint) can be viewed as a BX $\mathbb{S} \leftrightarrow V$.

Incremental parsing. When considering the feature of incremental parsing, which not only returns a value parsed from the string but also (incrementally) updates a model $\mathbb{M}$, the *parse* function must read a model and return an updated one. In other words, *parse* must be refined upon (TrivialParse) as (IncParse):

$$parse : (\mathbb{S}, \mathbb{M}) \rightarrow (V, \mathbb{M}) \tag{IncParse}$$

Unfortunately, (IncParse) and (IncPrint) cannot be combined into a BX because (IncPrint) does not use a model. As the model is read-only during printing, $\mathbb{M}$ can be considered as an additional view type that contributes to the transformation. Subsequently, (IncPrint) is redefined as (IncPrintM):

$$print : \mathbb{S} \times (V, \mathbb{M}) \rightarrow \mathbb{S} \tag{IncPrintM}$$

To combine (IncParse) and (IncPrintM) together, we propose a new kind of bidirectional transformations, namely, *accumulative BX* (αBX for short).

Definition 1 (*Accumulative BX*)**.** Given two functions $parse : (\mathbb{S}, \mathbb{M}) \rightarrow (V, \mathbb{M})$ and $print : \mathbb{S} \times (V, \mathbb{M}) \rightarrow \mathbb{S}$, they can be combined into an accumulative BX, written $l : \mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} V$, iff. they satisfy the following round-trip properties:

$$s \neq \bot \wedge parse(s, m) = (v, m') \implies print(s, (v, m')) = s \tag{1}$$

$$print(s, (v, m)) = s' \implies s' \neq \bot \wedge parse(s', m) = (v, m) \tag{2}$$

Just like (GetPut) and (PutGet) laws, properties (1) and (2) state that *parse* and *print* are mutually reversed. If $s = \bot$, *parse* effectively does nothing, and *print* generates a fresh string. Therefore, (1) does not have to hold in this case.

Example 3.1. Assume that $substr(s, a, b, c)$ returns a substring of the original string s, starting with the character at position a (inclusive) and ending with the character at position b (exclusive); if $a \geq b$, then it returns an empty string; if $len(s) < b$, then the resulting string will be padded with the character c to reach the expected length. Let $print_N$ be a function $\mathbb{S} \times (Int, \mathbb{S}) \rightarrow \mathbb{S}$, such that $print_N(s_0, (i, s_m)) = substr(s_m, 0, i, \#)$. Let $parse_N$ be a function $(\mathbb{S}, \mathbb{S}) \rightarrow (Int, \mathbb{S})$, such that $parse_N(s, s_m) = (len(s), s'_m)$, where $s'_m = s \mathbin{+\!\!+} substr(s_m, len(s), len(s_m), \epsilon)$. It is easy to verify that $print_N$ and $parse_N$ form a well-behaved αBX, where the model ($\mathbb{M}$) is also a string. Furthermore, in $print_N$, V (i.e., the integer) cannot be derived from $\mathbb{M}$.

αBX reflects the following bidirectional behavior: during parsing, a string s is consumed to produce the view v and update the model m into m'; during printing, the original string s is updated by considering the view v and the current model m. Obviously, (IncParse) and (IncPrintM) fit this specific behavior.

Special case. Considering the case when the updated model m' is computed by *putting* the view value v back to the original model m, we can define a model-value BX, $val : \mathbb{M} \leftrightarrow V$, and interpret αBX $l : \mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} V$ as follows:

- to compute $parse_l(s, m)$, l firsts converts s into v, and then updates m into m' by performing $m' = put_{val}(m, v)$;
- to compute $print_l(s, (v, m))$, l updates s into s' when $v = get_{val}(m)$.

This case actually requires that $\mathbb{M}$ and V be consistent in terms of the model-value BX. V is derivable from $\mathbb{M}$ and thus redundant. Formally, we propose a constructor $(*) : (\mathbb{M} \leftrightarrow V) \rightarrow (\mathbb{S} \leftrightarrow V) \rightarrow (\mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} Unit)$ to construct $(val * sl) : \mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} Unit$ from a model-value BX $val : \mathbb{M} \leftrightarrow V$ and a string-value BX $sl : \mathbb{S} \leftrightarrow V$, as follows:

$parse_{val*sl}(s, m) \equiv$ **do** $v \leftarrow get_{sl}(s)$ $m' \leftarrow put_{val}(m, v)$ **return** (unit, m')	$print_{val*sl}(s, (\text{unit}, m)) \equiv$ **do** $v \leftarrow get_{val}(m)$ $s' \leftarrow put_{sl}(s, v)$ **return** s'

where $Unit$ is the bottom type of all data types, which has one concrete value unit. Because $Unit$-typed arguments and return values can be ignored, a special αBX can also be regarded as $\alpha BX = \{parse : (\mathbb{S}, \mathbb{M}) \rightarrow \mathbb{M}, print : \mathbb{S} \times \mathbb{M} \rightarrow \mathbb{S}\}$.

Theorem 1. *For well-behaved* $val : \mathbb{M} \leftrightarrow V$ *and* $sl : \mathbb{S} \leftrightarrow V$, $(*)$ *ensures the well-behavedness of* $val * sl$.

For example, assume that $\mathbb{M}$ is a record type, $sl : \mathbb{S} \leftrightarrow Int$ converts a string and an integer bidirectionally, and val retrieves/stores a value from/to the field k of a record. When $s =$ "123" and $m = \{\text{k} = 5, \text{j} = 6\}$, (1) $get_{sl}(s) = 123$ and $put_{val}(m, 123) = m' \equiv \{\text{k} = 123, \text{j} = 6\}$, resulting in $parse_{val*sl}(s, m) = m'$; (2) $get_{val}(m') = 123$ and $put_{sl}(s, 123) =$ "123", so $print_{val*sl}(s, m') =$ "123".

Composability. So far, we assume that a *parse* function shall consume the entire input string. Nevertheless, a single parser may only parse a prefix of the input in practice, leaving the rest to the subsequent parsers. This fashion can be declared as a *prefixParse* function $\mathbb{S} \rightarrow (V, \mathbb{S})$. For better composability, we should integrate *prefixParse* into our formalization.

Firstly, we borrow the idea of many modern compilers that *a parser converts a string (i.e., code) into a pair of an internal representation* $\mathbb{T}$ *(e.g., concrete/abstract syntax trees) and the remaining string*, as outlined below:

$$parse : \mathbb{S} \rightarrow (\mathbb{T}, \mathbb{S}) \tag{SynParse}$$

(SynParse) (called *syntactic parser*) is similar to *prefixParse*, except that (SynParse) produces an internal representation $\mathbb{T}$, rather than a concrete value V. By straightforwardly inverting *prefixParse*, we obtain a *syntactic printer*

$$print : (\mathbb{T}, \mathbb{S}) \rightarrow \mathbb{S} \tag{SynPrint}$$

which prints the internal representation $\mathbb{T}$ to a string and joins it to the remaining string. We call (SynParse) and (SynPrint) a well-behaved $synBX$ $\mathbb{S} \leftrightsquigarrow \mathbb{T}$ iff. they make the following round-trip properties hold:

$$s \neq \bot \wedge parse(s) = (t, s_T) \implies print(t, s_T) = s \tag{3}$$

$$t \neq \bot \wedge print(t, s_T) = s \implies parse(s) = (t, s_T) \tag{4}$$

Property (4) means *parse* must *exactly* consume the prefix printed by $print(t, s_T)$.

Secondly, we redefine the special case of $\alpha BX : \mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} Unit$ into the *semantic BX* $semBX : \mathbb{T} \overset{\mathbb{M}}{\leftrightarrow} Unit$, which comprises (SemParse) and (SemPrint) as follows:

$$parse : (\mathbb{T}, \mathbb{M}) \rightarrow \mathbb{M} \tag{SemParse}$$

$$print : \mathbb{T} \times \mathbb{M} \rightarrow \mathbb{T} \tag{SemPrint}$$

In short, $semBX$ consumes/produces $\mathbb{T}$, rather than a string.

Finally, we propose a constructor $\circledast$ between $synBX$ and $semBX$

$$\circledast : (\mathbb{S} \leftrightsquigarrow \mathbb{T}) \rightarrow (\mathbb{T} \overset{\mathbb{M}}{\leftrightarrow} Unit) \rightarrow (\mathbb{S} \overset{\mathbb{M}}{\leftrightarrow} \mathbb{S})$$

$$
\begin{array}{lcll|lcl}
t & ::= & p & \text{i.e., primitives} & c & ::= & \mathsf{seq}(t_1, t_2) \\
 & \| & c & \text{i.e., combinators} & & \| & \mathsf{ite}(e, t_1, t_2) \\
e & ::= & \multicolumn{2}{l|}{\text{expressions}} & & \| & \mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \to t) \\
\rho & ::= & \multicolumn{2}{l|}{\text{regular patterns}} & & \| & \mathsf{scope}(s_b, s_a, t) \\
v & ::= & \multicolumn{2}{l|}{\text{variables}} & & \| & \mathsf{default}(\rho, t) \\
s & ::= & \multicolumn{2}{l|}{\text{string literals and } \perp} & & \| & \mathsf{unord}(t_1, t_2) \\
p & ::= & \multicolumn{2}{l|}{\mathsf{const}(s_c)} & & \| & \mathsf{final}(t) \\
 & \| & \multicolumn{2}{l|}{\mathsf{lex}(\rho, e)} & & \| & \mathsf{call}(t, v_1 = e_1, v_2 = e_2, \ldots) \\
 & \| & \multicolumn{2}{l|}{\mathsf{space}(s_w)} & \tau & ::= & s \parallel [\tau_1, \tau_2, \ldots] \parallel \mathsf{L}\ \tau \parallel \mathsf{R}\ \tau \\
 & \| & \multicolumn{2}{l|}{\mathsf{assign}(v, e)} & & \| & (\tau_b, [\tau_{r_1}, \tau_{r_2}, \ldots], \tau_a) \parallel \mathsf{C}\ \tau \\
 & \| & \multicolumn{2}{l|}{\mathsf{nop}} & & \| & \mathsf{D}\ \tau \parallel \mathsf{U}_i\ \tau \parallel\ < s_b, \tau, s_a >
\end{array}
$$

Fig. 8. Syntax of the core language.

which constructs a *composable* αBX $l_1 \circledast l_2 : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$ from a $synBX$ $l_1 : \mathbb{S} \leftrightsquigarrow \mathbb{T}$ and a $semBX$ $l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$, as follows:

$$
\begin{array}{l||l}
parse_{l_1 \circledast l_2}(s, m) \equiv & print_{l_1 \circledast l_2}(s, ((v, s_T), m)) \equiv \\
\quad \mathbf{do} & \quad \mathbf{do} \\
\qquad (t, s_T) \leftarrow parse_{l_1}(s) & \qquad (t, s'_{tail}) \leftarrow parse_{l_1}(s) \\
\qquad (v, m') \leftarrow parse_{l_2}(t, m) & \qquad t' \leftarrow print_{l_2}(t, (v, m)) \\
\qquad \mathbf{return}\ ((v, s_T), m') & \qquad s' \leftarrow print_{l_1}(t', s_T) \\
 & \qquad \mathbf{return}\ s'
\end{array}
$$

Theorem 2. *If $l_1 : \mathbb{S} \leftrightsquigarrow \mathbb{T}$ and $l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$ are well behaved, then $l_1 \circledast l_2$ is also a well behaved αBX.*

Local assignments. In a template language, a local assignment v=e can be viewed as a computational effect—it changes the binding of variable v to expression e. Assume $\mathbb{B}$ is the type of variable binding set and any $\beta : \mathbb{B}$ represents a set of variable bindings. Each variable binding has a form of $v \mapsto e$.

We can view a BIT template as a composition of local assignments and a special composable αBX. We define such a composition as an *effect-binding BX* (βBX for short). For any assignment v=e in a template, e is determined by constants, current variable bindings, and the model to be printed.

Definition 2 (*Effect-binding BX*)**.** An effect-binding BX, written $\mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, is a pair of *parse* and *print* functions

$$parse : (\mathbb{S}, \mathbb{M}, \mathbb{B}) \to (\mathbb{S}, \mathbb{M}) \times \mathbb{B}$$
$$print : (\mathbb{S}, \mathbb{B}) \times (\mathbb{S}, \mathbb{M}) \to (\mathbb{S}, \mathbb{B})$$

The parse function states that given a string s, a model m and a binding environment β, it returns the remaining string after parsing, the updated model and the updated binding environment. The print function states that given the original string s, environment β, the remaining string after parsing and model, it returns the updated string and updated environment.

It is well-behaved if it satisfies the following round-trip properties

$$s \neq \perp \wedge parse(s, m, \beta) = ((s_T, m'), \beta') \Rightarrow print((s, \beta), (s_T, m)) = (s, \beta') \quad \text{(ParsePrint)}$$

$$print((s, \beta), (s_T, m)) = (s', \beta') \Rightarrow parse(s', m, \beta) = ((s_T, m), \beta') \quad \text{(PrintParse)}$$

To construct a βBX, we propose the constructor

$$\odot : (\mathbb{B} \times \mathbb{M} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}) \to (\mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S})$$

for βBXs, such that for $r : \mathbb{B} \times \mathbb{M} \to \mathbb{B}$ and $pl : \mathbb{B} \to \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$,

$$
\begin{array}{l||l}
parse_{r \odot pl}(s, m, \beta) \equiv & print_{r \odot pl}((s, \beta), (s_T, m)) \equiv \\
\quad \mathbf{do} & \quad \mathbf{do} \\
\qquad l \leftarrow pl(\beta) & \qquad l \leftarrow pl(\beta) \\
\qquad (s_T, m') \leftarrow parse_l(s, m) & \qquad s \leftarrow print_l(s, (s_T, m)) \\
\qquad \beta' \leftarrow r(\beta, m') & \qquad \beta' \leftarrow r(\beta, m) \\
\qquad \mathbf{return}\ ((s_T, m'), \beta') & \qquad \mathbf{return}\ (s, \beta')
\end{array}
$$

where

- r is a binding update function that updates the variable bindings based on existing bindings and a model; for example, an assignment v=v+u can update a binding set $\{\mathsf{v} \mapsto a, \mathsf{u} \mapsto b\}$ to $\{\mathsf{v} \mapsto a + b, \mathsf{u} \mapsto b\}$;
- pl a binding-aware generator for αBX that generates an αBX based on given variable bindings, e.g., it generates «a+b+1 | INT» from an initial hole specification «v+u | INT» if the current bindings are $\{\mathsf{v} \mapsto \mathsf{a} + \mathsf{b}, \mathsf{u} \mapsto 1\}$.

Theorem 3. *If r and $pl(\beta)$ are well behaved for any β, then $r \odot pl$ is also a well-behaved βBX.*

3.3. Definition of the core language

The core language is designed to formally and precisely specify the semantics of BIT, into which the surface language can be translated. As shown in Fig. 8, the core language contains 5 BIT primitives and 8 BIT combinators. t stands for *template* (and *template fragment*); e and ρ denote a general *expression* (e.g., arithmetic, relational, boolean, and path call) and a *regular pattern*, respectively; v refers to a *variable*, which is also an expression; s denotes *string literals*; τ is the internal representation of parsed text, whose type is $\mathbb{T}$.

We assume every expression e is equipped with a bidirectional semantics. That is, we can interpret e as a BX. However, the bidirectional semantics of e is out of the scope of this paper, and please refer to Xiong et al. (2013), Zhang and Hu (2022), Zhang et al. (2023) for more details.

To define the semantics, the following helper functions are needed:

- $+\!\!+ : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ denotes string concatenation, e.g., "ab"++"12" = "ab12".
- $\mathsf{lookAt} : \mathbb{R} \times \mathbb{S} \to \mathbb{S} \times \mathbb{S}$ returns a prefix of the input string that matches the given regular pattern or $\perp$ if failed, where $\mathbb{R}$ denotes the type of regular patterns. For example, lookAt([0-9]+, "12ab") = ("12", "ab") and lookAt([0-9]+, "x12a") = ($\perp$, "x12a"). We also use lookAt to match a string constant because we can convert a string constant into a regular pattern.

Internal structure. $\tau : \mathbb{T}$ is the internal structure of the parsed text, produced by syn and consumed by sem. Different τs correspond to different primitives and combinators. τ can be a string s, a sequence $[\tau_{r_1}, \tau_{r_2}, \ldots]$, a L/R-labeled structure L τ/R τ for branches, a D-labeled structure for default fragments, a loop structure $(\tau_b, [\tau_{r_1}, \tau_{r_2}, \ldots], \tau_a)$, an unordered fragment structure $\mathsf{U}_i\ \tau$ (where i denotes the index of the body fragment of the unord primitive, which prints/parses τ), a template-call structure $\mathsf{C}\ \tau$, and a scope structure $\langle s_b, \tau, s_a \rangle$.

Formal structure. A BIT template is a βBX $\mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, which can be defined as the record type BIT:

$$\textbf{data}\ \mathsf{BIT} = \mathsf{BIT}\ \{syn : \mathbb{S} \rightsquigarrow \mathbb{T}, gSem : \mathbb{B} \to \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit, eff : (\mathbb{B}, \mathbb{M}) \to \mathbb{B}\}$$

where $gSem$ is a $semBX$ generator and eff is the binding update function.

- $eff(\beta, v \mapsto u)$: return $(\beta - \{v \mapsto x | v \mapsto x \in \beta\}) \cup \{v \mapsto resolve(\beta, u)\}$.
- $resolve(\beta, expr)$: return a new expression by substituting the free variables occurring in the expression $expr$ with their bindings in β. For example, $resolve(\beta, v_1 \times v_2) = (a+b) \times v_2$ if $\beta = \{v_1 \mapsto a+b\}$.

Given r : BIT, a βBX is built by $r.eff \odot (\lambda\beta \to r.syn \circledast r.gSem(\beta))$.

The structure of a BIT primitive can further be refined as a record type BIT′:

$$\textbf{data}\ \mathsf{BIT}' = \mathsf{BIT}'\ \{syn : \mathbb{S} \rightsquigarrow \mathbb{T}, sem : \mathbb{T} \leftrightarrow V, val : \mathbb{M} \leftrightarrow V, eff : (\mathbb{B}, \mathbb{M}) \to \mathbb{B}\}$$

Then, given record r' of BIT′, it can be converted into a record of BIT as follows BIT $\{syn = r'.syn, gSem = \lambda\beta \to \big(resolve(\beta, r'.val) * r'.sem\big), eff = r'eff\}$.

Primitives. This paper proposes 5 primitives, i.e., const, lex, space, assign, nop.

The primitive $\mathsf{const}(s_c)$ prints/parses a constant string s_c. It can be defined as the following structure:

$$\mathsf{const}(s_c) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if}\ s_I = \perp\ \textbf{then}\ (\perp, \perp) \\ \qquad\qquad \textbf{elif}\ s_c \mathbin{+\!\!+} s_T = s_I\ \textbf{then}\ (s_c, s_T)\ \textbf{else}\ \mathsf{error} \\ print(\tau, s_T) \triangleq \textbf{if}\ \tau = s_c\ \textbf{then}\ s_c \mathbin{+\!\!+} s_T\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if}\ \tau = s_c\ \textbf{then}\ \mathsf{unit}\ \textbf{else}\ \mathsf{error} \\ print(\tau, \mathsf{unit}) \triangleq \textbf{if}\ \tau = s_c \vee \tau = \perp\ \textbf{then}\ s_c\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ val = UnitBX, eff = IdEff \end{array} \right\}$$

where $UnitBX \equiv \{get(m) = \mathsf{unit}, put(m, \mathsf{unit}) = m\}$, $IdEff(\beta, m) \equiv \beta$, and error denotes a runtime exception which will abort execution if left uncaught.

The primitive $\mathsf{lex}(\rho, e)$ aims to print/parse the value of e according to a regular pattern ρ, considering e as $\mathbb{M} \leftrightarrow \mathbb{S}$. It is defined as follows.

$$\mathsf{lex}(\rho, e) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if}\ s_I = \perp\ \textbf{then}\ (\perp, \perp) \\ \qquad\qquad \textbf{elif}\ \mathsf{lookAt}(\rho, s_I) = (s, s_T)\ \textbf{then}\ (s, s_T) \\ \qquad\qquad \textbf{else}\ \mathsf{error} \\ print(\tau, s_T) \triangleq \textbf{if}\ \tau = s \wedge \mathsf{lookAt}(\rho, s \mathbin{+\!\!+} s_T) = (s, s_T) \\ \qquad\qquad \textbf{then}\ s \mathbin{+\!\!+} s_T\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if}\ \tau = s\ \textbf{then}\ s\ \textbf{else}\ \mathsf{error} \\ print(\tau, s) \triangleq s \end{array} \right. \\ val = e, eff = IdEff \end{array} \right\}$$

The primitive $\mathsf{space}(s_w)$ handles white spaces. In parsing, it consumes the prefix white spaces; in printing, it tries to preserve the existing spaces or prints s_w (s_w must be white spaces) if the original string is empty. Supposing that ρ_w is the regular pattern that matches white spaces, $\mathsf{space}(s_w)$ is defined as follows:

$$\mathsf{space}(s_w) \triangleq$$
$$\mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if}\ s_I = \perp\ \textbf{then}\ (\perp, \perp) \\ \qquad\qquad \textbf{elif}\ \mathsf{lookAt}(\rho_w, s_I) = (s, s_T)\ \textbf{then}\ (s, s_T)\ \textbf{else}\ \mathsf{error} \\ print(\tau, s_T) \triangleq \textbf{if}\ \tau = s \neq \perp \wedge \mathsf{lookAt}(\rho_w, s \mathbin{+\!\!+} s_T) = (s, s_T) \\ \qquad\qquad \textbf{then}\ s \mathbin{+\!\!+} s_T\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if}\ \tau = s \neq \perp\ \textbf{then}\ \mathsf{unit}\ \textbf{else}\ \mathsf{error} \\ print(\tau, \mathsf{unit}) \triangleq \textbf{if}\ \tau = \perp\ \textbf{then}\ s_w \\ \qquad\qquad \textbf{elif}\ \tau = s\ \textbf{then}\ s\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ val = UnitBX, eff = IdEff \end{array} \right\}$$

The primitive $\mathsf{assign}(v, e)$ changes the binding of v, rather than directly printing/parsing strings. It is defined as follows:

$$\mathsf{assign}(v, e) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \{parse(s_I) \triangleq (\perp, s_I), print(\perp, s_T) \triangleq s_T\}, \\ sem = \{parse(\perp) \triangleq \mathsf{unit}, print(\perp, \mathsf{unit}) \triangleq \perp\}, \\ val = UnitBX, eff = \lambda(\beta, m) \to update(\beta, v \mapsto e) \end{array} \right\}$$

Primitive nop does nothing in both printing and parsing:

$$\mathsf{nop} \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \{parse(s_I) \triangleq (\perp, s_I), print(\perp, s_T) \triangleq s_T\}, \\ sem = \{parse(\perp) \triangleq \mathsf{unit}, print(\perp, \mathsf{unit}) \triangleq \perp\}, \\ val = UnitBX, eff = IdEff \end{array} \right\}$$

Example 3.2. Fig. 9 shows some examples about BIT primitives. $\boxed{1}$, $\boxed{4}$, and $\boxed{7}$ demonstrate how to print from an empty string. $\boxed{2}$ and $\boxed{3}$ print constant string "a" onto the original strings, but $\boxed{3}$ fails because the original string does not start with "a". $\boxed{5}$ and $\boxed{6}$ print the values of v, but $\boxed{5}$ fails because v refers to "bb" that does not satisfy the lexical rule "a+". $\boxed{8}$ and $\boxed{9}$ demonstrate how space tries to preserve the original white-spaces as much as possible. $\boxed{\text{i}}$ to $\boxed{\text{x}}$ demonstrate the parsing behaviors. $\boxed{\text{ii}}$ and $\boxed{\text{v}}$ fail because the strings to be parsed do not match the lexical rules of the primitives.

Combinators. The core language has 8 combinators for complex behaviors.

The combinator $\mathsf{seq}(t_1, t_2)$ combines two template fragments t_1 and t_2 sequentially. Supposing that t_1 and t_2 are BIT records, $\mathsf{seq}(t_1, t_2)$ is defined by

$$\mathsf{seq}(t_1, t_2) \triangleq$$
$$\mathsf{BIT} \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_0) \triangleq \textbf{if}\ s_0 = \perp\ \textbf{then}\ (\perp, \perp) \\ \qquad\qquad \textbf{elif}\ parse_{t_i.syn}(s_{i-1}) = (\tau_i, s_i)\ \textbf{then}\ ([\tau_1, \tau_2], s_2) \\ \qquad\qquad \textbf{else}\ \mathsf{error} \\ print([\tau_1, \tau_2], s_2) \triangleq \textbf{if}\ s_{i-1} = print_{t_i.syn}(\tau_i, s_i) \\ \qquad\qquad \textbf{then}\ s_0\ \textbf{else}\ \mathsf{error} \end{array} \right. \\ gSem = \lambda\beta \to \\ \quad \left\{ \begin{array}{l} parse([\tau_1, \tau_2], m) \triangleq \textbf{do} \\ \quad m_1 \leftarrow parse_{t_1.gSem(\beta)}(\tau_1, m), \beta_1 \leftarrow t_1.eff(\beta, m_1) \\ \quad m_2 \leftarrow parse_{t_2.gSem(\beta_1)}(\tau_2, m_1) \\ \quad \textbf{assert}\ \beta_1 = t_1.eff(\beta, m_2) \wedge m_2 = parse_{t_1.gSem(\beta)}(\tau_1, m_2) \\ \quad \textbf{return}\ m_2 \\ print(\tau, m) \triangleq \textbf{do} \\ \quad [\tau_1, \tau_2] \leftarrow \textbf{if}\ \tau = \perp\ \textbf{then}\ [\perp, \perp]\ \textbf{elif}\ \tau = [\tau_1 \varepsilon, \tau_2 \varepsilon]\ \textbf{then}\ [\tau_1 \varepsilon, \tau_2 \varepsilon] \\ \quad \tau_1' \leftarrow print_{t_1.gSem(\beta)}(\tau_1, m), \beta_1 \leftarrow t_1.eff(\beta, m) \\ \quad \tau_2' \leftarrow print_{t_2.gSem(\beta_1)}(\tau_2, m) \\ \quad \textbf{return}\ [\tau_1', \tau_2'] \end{array} \right. \\ eff = \lambda(\beta, m) \to t_2.eff(t_1.eff(\beta, m), m) \end{array} \right\}$$

where **assert** throws error when the assertion predicate fails. The **assert** statement requires that t_2 does not break the consistency established by t_1.

Example 3.3. Consider a sequence $\mathsf{seq}(\mathsf{const}(\text{"a"}), \mathsf{lex}(\mathsf{INT}, n))$, where the evaluation of parse on the string "a1 b2" and model {n = 0} results in the remaining string " b2" and the value of n in the model is updated to 1.

$$parse_{\mathsf{seq}(\mathsf{const}(\text{"a"}),\mathsf{lex}(\mathsf{INT},n))}(\text{"a1 b2"}, \{n = 0\}, \{\}) = ((\text{"b2"}, \{n = 1\}), \{\})$$

If the value of n is changed to 2 in the model, the print function outputs "a2 b2".

$$print_{\mathsf{seq}(\mathsf{const}(\text{"a"}),\mathsf{lex}(\mathsf{INT},n))}((\text{"a1 b2"}, \{\}), (\{n = 2\}, \{\})) = (\text{"a2 b2"}, \{\})$$

Note that seq can be extended to combine more than two template fragments: $\mathsf{seq}(t_1, t_2, \ldots, t_n)$ is equivalent to $\mathsf{seq}(t_1, \mathsf{seq}(t_2, \mathsf{seq}(t_3, \ldots)))$.

Printing examples $print_l((s,\beta),(s_T,m)) = (s',\beta')$	Parsing examples $parse_l(s,m,\beta) = ((s_T,m'),\beta')$
1 $print_{\mathsf{const}(''a'')}((\bot,\beta),(''1'',m)) = (''a1'',\beta)$	**i** $parse_{\mathsf{const}(''a'')}(''ab'',m,\beta) = ((''b'',m),\beta)$
2 $print_{\mathsf{const}(''a'')}((''a2'',\beta),(''1'',m)) = (''a1'',\beta)$	**ii** $parse_{\mathsf{const}(''a'')}(''bb'',m,\beta) = \mathsf{error}$
3 $print_{\mathsf{const}(''a'')}((''b2'',\beta),(''1'',m)) = \mathsf{error}$	
4 $print_{\mathsf{lex}(''a+'',v)}((\bot,\{v \mapsto u\}),(''b'',\{u=''aa''\})) = (''aab'',\{v \mapsto u\})$	**iii** $parse_{\mathsf{lex}(''a+'',v)}((''aab'',\{\},\{\})) = ((''b'',\{v=''aa''\}),\{\})$
5 $print_{\mathsf{lex}(''a+'',v)}((\_,\{v \mapsto u\}),(\_,\{u=''bb''\})) = \mathsf{error}$	**iv** $parse_{\mathsf{lex}(''a+'',v)}((''aab'',\{v \mapsto u\},\{\})) = ((''b'',\{u=''aa''\}),\{\})$
6 $print_{\mathsf{lex}(''a+'',v)}((''ab'',\{\}),(''b'',\{v=''aa'',u=''aaa''\})) = (''aab'',\{\})$	**v** $parse_{\mathsf{lex}(''a+'',v)}((''cb'',\_,\_)) = \mathsf{error}$
7 $print_{\mathsf{space}(''\ '')}((\bot,\beta),(''b'',m)) = (''\ b'',\beta)$	**vi** $parse_{\mathsf{space}(''\ '')}(''\quad b'',m,\beta) = ((''b'',m),\beta)$
8 $print_{\mathsf{space}(''\ '')}((''\qquad '',\beta),(''b'',m)) = (''\qquad b'',\beta)$	**vii** $parse_{\mathsf{space}(''\ '')}(''b'',m,\beta) = ((''b'',m),\beta)$
9 $print_{\mathsf{space}(''\ '')}((''a'',\beta),(''b'',m)) = (''b'',\beta)$	
10 $print_{\mathsf{assign}(v,a+b)}((\_,\{\}),(''b'',\_)) = (''b'',\{v \mapsto a+b\})$	**xi** $parse_{\mathsf{assign}(v,a+b)}(''b'',m,\{\}) = ((''b'',m),\{v \mapsto a+b\})$
11 $print_{\mathsf{assign}(v,a+b)}((\_,\{a \mapsto u\}),(''b'',\_)) = (''b'',\{a \mapsto u, v \mapsto u+b\})$	**x** $parse_{\mathsf{assign}(v,a+b)}(''b'',m,\{a \mapsto u\}) = ((''b'',m),\{a \mapsto u, v \mapsto u+b\})$

Fig. 9. Examples of primitives.

$\mathsf{ite}(e,t_1,t_2)$ is the conditional combinator, corresponding to the IF construct in the surface language. In printing, it selects from t_1 and t_2 based on the branch condition e; in parsing, it chooses t_i to parse the input string if t_i consumes more characters than the other branch. ite is defined as follows:

$$
\mathsf{ite}(e,t_1,t_2) \triangleq
$$

$$
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot,\bot) \\
\quad (\tau_i, s_i) \leftarrow parse_{t_i.syn}(s_I) \quad s.t. \quad i=1,2 \\
\quad \textbf{return if } len(s_1) \le len(s_2) \textbf{ then } (\mathsf{L}\ \tau_1, s_1) \textbf{ else } (\mathsf{R}\ \tau_2, s_2) \\
print(\mathsf{L}\ \tau_1, s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow print_{t_1.syn}(\tau_1, s_T), (\tau_2, s_2) \leftarrow parse_{t_2.syn}(s') \\
\quad \textbf{assert } len(s_T) \le len(s_2) \\
\quad \textbf{return } s' \\
print(\mathsf{R}\ \tau_2, s_T) \triangleq \textbf{do}\\
\quad s' \leftarrow print_{t_2.syn}(\tau_2,s_T), (\tau_1,s_1) \leftarrow parse_{t_1.syn}(s')\\
\quad \textbf{assert } len(s_T) < len(s_1)\\
\quad \textbf{return } s'
\end{array}\right. \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{\begin{array}{l}
parse(\mathsf{L}\ \tau, m) \triangleq \textbf{do}\\
\quad l \leftarrow t_1.gSem(\beta), m_1 \leftarrow parse_l(\tau,m), m' \leftarrow put_e(m_1, true)\\
\quad \textbf{assert } m' = parse_l(\tau, m')\\
\quad \textbf{return } m'\\
parse(\mathsf{R}\ \tau, m) \triangleq \textbf{do}\\
\quad l \leftarrow t_2.gSem(\beta), m_2 \leftarrow parse_l(\tau,m), m' \leftarrow put_e(m_2, false)\\
\quad \textbf{assert } m' = parse_l(\tau, m')\\
\quad \textbf{return } m'\\
print(\tau,m) \triangleq \textbf{do}\\
\quad \textbf{if } get_e(m) = true \textbf{ then}\\
\qquad l \leftarrow t_1.gSem(\beta), \tau' \leftarrow \textbf{if } \tau = \mathsf{L}\ \tau_1 \textbf{ then } \tau_1 \textbf{ else } \bot\\
\qquad \textbf{return } \mathsf{L}\ print_l(\tau', m)\\
\quad \textbf{else}\\
\qquad l \leftarrow t_2.gSem(\beta), \tau' \leftarrow \textbf{if } \tau = \mathsf{R}\ \tau_2 \textbf{ then } \tau_2 \textbf{ else } \bot\\
\qquad \textbf{return } \mathsf{R}\ print_l(\tau', m)
\end{array}\right.\\
eff = \lambda(\beta,m) \rightarrow \textbf{if } get_e(m)=true \textbf{ then } t_1 eff(\beta,m) \textbf{ else } t_2 eff(\beta,m)
\end{array}\right\}
$$

The combinator $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$, corresponding to the FOR construct, prints each element using a list e_{arr} by loop body t with an iteration variable v, where s_s is the separator inserted between two consecutive iterations, while s_b and s_a are the strings inserted before and after all iterations.

To specify the semantics of $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$, we first convert s_s, s_b, and s_a to BIT primitives t_s, t_b, and t_a. If s_x ($x = s, b, a$) is a non-empty string, then $t_x = \mathsf{const}(s_x)$; otherwise, $t_x = \mathsf{nop}$. Assume that t_i denotes the ith-iteration of the loop body t whose iteration variable is renamed to v_i. For simplicity, we assume that there is no collision among variable names. For example, supposing that the loop body is $\mathsf{lex}(\rho, v)$ (where v is the iteration variable), then t_i is $\mathsf{lex}(\rho, v_i)$. Let $ls_0 \equiv \mathsf{nop}$ and $ls_n \equiv \mathsf{seq}(t_b, t_1, t_s, t_2, t_s, \ldots, t_s, t_n, t_a)$ ($n > 0$). The behavior of $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$ can be interpreted as a certain sequence ls_n. Formally, loop is defined as follows:

$$
\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t) \triangleq
$$

$$
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do}\\
\quad \text{let } n = \text{the largest integer s.t. } parse_{ls_n.syn}(s_I) \text{ succeeds}\\
\quad \textbf{if } n = 0 \textbf{ then return } (\bot, s_I)\\
\quad [\tau_b, \tau_1, \tau_{s_1}, \ldots, \tau_n, \tau_a] \leftarrow parse_{ls_n.syn}(s_I)\\
\quad \textbf{return } (\tau_b, [\tau_1, \tau_{s_1}, \ldots, \tau_n], \tau_a)\\
print(\tau, s_T) \triangleq \textbf{do}\\
\quad (n_l, \tau_l) \leftarrow \textbf{if } \tau = (\tau_b, [\tau_1, \tau_{s_1}, \ldots, \tau_n], \tau_a)\\
\qquad\qquad \textbf{then } (n, [\tau_b, \tau_1, \tau_{s_1}, \ldots, \tau_n, \tau_a]) \textbf{ else } (0, \bot)\\
\quad s' \leftarrow print_{ls_{n_l}.syn}(\tau_l, s_T)\\
\quad \textbf{if } parse_{ls_{n+1}.syn}(s') \text{ throws } \mathsf{error} \textbf{ then return } s'
\end{array}\right.\\
gSem = \lambda\beta \rightarrow \left\{\begin{array}{l}
parse(\tau, m) \triangleq \textbf{do}\\
\quad e' \leftarrow resolve(\beta, e_{arr})\\
\quad \textbf{if } \tau = (\tau_b, [\tau_1, \tau_{s_1}, \ldots, \tau_n], \tau_a) \textbf{ then}\\
\qquad ls \leftarrow ls_n, m' \leftarrow parse_{ls_n.gSem(\beta)}(\tau, m)\\
\qquad arr \leftarrow [get_{v_1}(m'), get_{v_2}(m'), \ldots, get_{v_n}(m')]\\
\quad \textbf{else } ls \leftarrow ls_0, m' \leftarrow m, arr \leftarrow []\\
\quad m\varepsilon \leftarrow put_{e'}(m', arr)\\
\quad \textbf{assert } m\varepsilon = parse_{ls.gSem(\beta)}(\tau, m\varepsilon)\\
\quad \textbf{return } m\varepsilon\\
print(\tau, m) \triangleq \textbf{do}\\
\quad e' \leftarrow resolve(\beta, e_{arr}), arr \leftarrow get_{e'}(m), n_a \leftarrow len(arr)\\
\quad \textbf{if } n_a = 0 \textbf{ then return } \bot\\
\quad \textbf{elif } \tau = (\tau_b, [\tau_1, \tau_{s_1}, \ldots, \tau_n], \tau_a) \textbf{ then}\\
\qquad \textbf{if } n \ge n_a \textbf{ then } \tau' \leftarrow [\tau_b, \tau_1, \tau_{s_1}, \ldots, \tau_{n_a}, \tau_a]\\
\qquad \textbf{else } \tau' \leftarrow [\tau_b, \tau_1, \tau_{s_1}, \ldots, \tau_n, \underbrace{s_s, \bot, \ldots, s_s, \bot}_{s_s \text{ occurs } n_a - n \text{ times}}, \tau_a]\\
\quad \textbf{else } \tau' \leftarrow [s_b, \underbrace{\bot, s_s, \ldots, s_s, \bot}_{s_s \text{ occurs } n_a - 1 \text{ times}}, s_a]\\
\quad m' \leftarrow m \cup \{v_1 = arr[0], \ldots, v_{n_a} = arr[n_a - 1]\}\\
\quad [\tau'_b, \tau'_1, \tau'_{s_1}, \ldots, \tau'_{n_a}, \tau'_a] \leftarrow print_{ls_a.gSem(\beta)}(\tau', m')\\
\quad \textbf{return } (\tau'_b, [\tau'_1, \tau'_{s_1}, \ldots, \tau'_{n_a}], \tau'_a)
\end{array}\right.\\
eff = \lambda(\beta, m) \rightarrow ls_n.eff(\beta, m)\\
\qquad \text{where } e' = resolve(\beta, e_{arr}) \wedge arr = get_{e'}(m) \wedge n = len(arr)
\end{array}\right\}
$$

Example 3.4. Consider the template defined in Fig. 3(a), which should be translated into the core language in the form of $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$. In this representation, e_{arr} refers to the input argument paragraphs, while the separators s_s, s_b, s_a are nop, v is p, and t is the body of the FOR-loop. Supposing that the input string s_I contains two Paragraphs (e.g., Fig. 2(b)), ls_n should be ls_2, i.e., $\mathsf{seq}(\mathsf{nop}, t_1, \mathsf{nop}, t_2, \mathsf{nop})$. Then, ls_2 will be used to print paragraphs to a string or parse a string to paragraphs. Let us illustrate the parsing process in detail. Firstly, in syn, $ls_2.syn$ is performed to parse s_I to an intermediate representation $(\mathsf{nop}, [\tau_1, \mathsf{nop}, \tau_2], \mathsf{nop})$, where τ_1 and τ_2 represents the intermediate representation generated by t_1 and t_2 (i.e., different iterations of t). Secondly, in $gSem(\beta)$, the expression e' computed by $resolve(\beta, e_{arr})$ still refers to paragraphs, since at the beginning of the loop, β should contain only one binding $\{no \mapsto 0\}$. Thirdly, $ls_2.gSem(\beta)$ is used to process the intermediate representation $(\mathsf{nop}, [\tau_1, \mathsf{nop}, \tau_2], \mathsf{nop})$. After processing τ_1, a new Paragraph will be created and assigned to a temporary iteration variable p_1 in the model. Meanwhile, as defined in seq, β will be updated to $\beta_1 = \{no \mapsto 1\}$ due to the local assignment in the loop. When parsing the hole concerning *no*, it is expected that the text is "1" because the value of *no* stored in β_1 is a constant integer 1. Similarly, after processing τ_2, the second Paragraph will be created and assigned to p_2. β_1 will also be updated to $\beta_2 = \{no \mapsto 2\}$. Fourthly, an array *arr* is constructed by fetching p_1 and p_2 from the current model. Finally, the model is further updated by using $put_{e'}$ and *arr*, resulting in an output model in which paragraphs is mapped to the two Paragraphs parsed out from s_I.

The combinator $\mathsf{scope}(s_b, s_a, t)$ is used to handle cases of balanced brackets (and comment delimiters). In source code, brackets must be correctly paired to recognize the code structure. For example, "(f())" must be parsed to "(", "f()", ")", i.e., the first "(" should be paired with the second ")", rather than the first one. Although a template language does not know the grammar of the printed text, our approach can be configured to handle these cases by specifying the opening and the closing tags (i.e., s_b, s_a). Then, $\mathsf{scope}(s_b, s_a, t)$ can recognize a text scope s_s, in which s_b and s_a are *balanced*, written $isBal(s_s, s_b, s_a)$. For example, $isBal(\text{"f[a+b[]]"}, \text{"["}, \text{"]"}) = true$, but $isBal(\text{"]+b[]["}, \text{"["}, \text{"]"}) = false$. The behavior of scope is defined as follows:

$$
\begin{array}{l}
\mathsf{scope}(s_b, s_a, t) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad \textbf{elif } s_I = s_b +\!\!+ s_s +\!\!+ s_a +\!\!+ s_T \wedge isBal(s_s, s_b, s_a) \textbf{ then} \\
\qquad (\tau', s_{s,T}) \leftarrow parse_{t.syn}(s_s) \\
\qquad \textbf{if } s_{s,T} = \text{""} \textbf{ then return } (<s_b, \tau', s_a>, s_T) \\
print(<s_b, \tau', s_a>, s_T) \triangleq \textbf{do} \\
\quad s_s \leftarrow print_{t.syn}(\tau', \text{""}) \\
\quad \textbf{assert } isBal(s_s, s_b, s_a) = true \\
\quad \textbf{return } s_b +\!\!+ s_s +\!\!+ s_a +\!\!+ s_T
\end{array}\right. \\
gSem = \lambda\beta \to \left\{\begin{array}{l}
parse(<s_b, \tau', s_a>, m) \triangleq parse_{t.gSem(\beta)}(\tau', m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad \tau' \leftarrow \textbf{if } \tau = <s_b, \tau\varepsilon, s_a> \textbf{ then } \tau\varepsilon \\
\qquad\qquad \textbf{elif } \tau = \bot \textbf{ then } \bot \textbf{ else } \mathsf{error} \\
\quad \textbf{return } <s_b, print_{t.gSem(\beta)}(\tau', m), s_a>
\end{array}\right. \\
eff = t.eff
\end{array}\right\}
\end{array}
$$

The combinator $\mathsf{default}(\rho, t)$, corresponding to the DEFAULT construct, generates a default string conforming to pattern ρ if the original string is empty, or preserves the original string if it is non-empty. It is defined as follows. Note that $\mathsf{default}(\rho, t)$ requires that t is not a default construct and does not contain local assignments (i.e., its binding update function must be $IdEff$).

$$
\begin{array}{l}
\mathsf{default}(\rho, t) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad (s_p, s_T) \leftarrow \mathsf{lookAt}(\rho, s_I) \\
\quad \textbf{assert } s_p \neq \bot \\
\quad \textbf{if } parse_{t.syn}(s_p) = (\tau', \bot) \textbf{ then return } (\mathsf{D}\ \tau', s_T) \\
\quad \textbf{else return } (s_p, s_T) \\
print(\tau, s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow \textbf{if } \tau = \mathsf{D}\ \tau' \textbf{ then } print_{t.syn}(\tau', s_T) \\
\qquad\quad \textbf{elif } \tau = s_p \textbf{ then } s_p +\!\!+ s_T \textbf{ else } \mathsf{error} \\
\quad \textbf{assert } \mathsf{lookAt}(\rho, s') = (s_p, s_T) \wedge s_p \neq \bot \\
\quad \textbf{return } s'
\end{array}\right. \\
gSem = \lambda\beta \to \\
\quad \left\{\begin{array}{l}
parse(\tau, m) \triangleq \textbf{if } \tau = \bot \textbf{ then } \mathsf{error} \\
\qquad\qquad\qquad \textbf{elif } \tau = \mathsf{D}\ \tau' \textbf{ then } parse_{t.gSem(\beta)}(\tau', m) \textbf{ else } m \\
print(\tau, m) \triangleq \textbf{if } \tau = \mathsf{D}\ \tau' \textbf{ then } \mathsf{D}\ print_{t.gSem(\beta)}(\tau', m) \\
\qquad\qquad\qquad \textbf{elif } \tau = \bot \textbf{ then } \mathsf{D}\ print_{t.gSem(\beta)}(\bot, m) \textbf{ else } \tau
\end{array}\right. \\
eff = t.eff = IdEff
\end{array}\right\}
\end{array}
$$

The combinator $\mathsf{unord}(t_1, t_2)$, corresponding to the UNORD construct, prints a string with t_1 and t_2. However, during parsing, it attempts to parse the string with $\mathsf{seq}(t_1, t_2)$ and $\mathsf{seq}(t_2, t_1)$. $\mathsf{unord}(t_1, t_2)$ requires that no matter in what order the binding update functions of t_1 and t_2 can be combined, the composite functions are equivalent. Supposing $ls_{i,j} = \mathsf{seq}(t_i, t_j)$, unord is defined as follows:

$$
\begin{array}{l}
\mathsf{unord}(t_1, t_2) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad ([\tau_{12,1}, \tau_{12,2}], s_{T,12}) \leftarrow parse_{ls_{1,2}.syn}(s_I) \\
\quad ([\tau_{21,2}, \tau_{21,1}], s_{T,21}) \leftarrow parse_{ls_{2,1}.syn}(s_I) \\
\quad \textbf{if } len(s_{T,12}) \leq len(s_{T,21}) \textbf{ then return } [\mathsf{U}_1\ \tau_{12,1}, \mathsf{U}_2\ \tau_{12,2}] \\
\quad \textbf{else return } [\mathsf{U}_2\ \tau_{21,2}, \mathsf{U}_1\ \tau_{21,1}] \\
print([\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j], s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow print_{ls_{i,j}}([\tau_i, \tau_j], s_T) \\
\quad \textbf{assert } (\_, s'_T) = parse_{ls_{j,i}}(s') \implies len(s_T) \leq len(s'_T) \\
\quad \textbf{return } s'
\end{array}\right. \\
gSem = \lambda\beta \to \\
\quad \left\{\begin{array}{l}
parse([\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j], m) \triangleq parse_{ls_{ij}}([\tau_i, \tau_j], m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad [\tau'_i, \tau'_j] \leftarrow \textbf{if } \tau = \bot \textbf{ then } print_{ls_{12}.gSem(\beta)}(\bot, m) \\
\qquad\qquad\qquad \textbf{elif } \tau = [\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j] \textbf{ then } print_{ls_{ij}}([\tau_i, \tau_j], m) \\
\quad \textbf{return } [\mathsf{U}_i\ \tau'_i, \mathsf{U}_j\ \tau'_j]
\end{array}\right. \\
eff = \mathsf{seq}(t_1, t_2).eff = \mathsf{seq}(t_2, t_1).eff
\end{array}\right\}
\end{array}
$$

Note that $\mathsf{unord}(t_1, t_2)$ can be generalized to $\mathsf{unord}(t_1, t_2, t_3, \ldots, t_n)$.

The combinator $\mathsf{final}(t)$, corresponding to the FINAL construct, preserves arbitrary characters in the original string before t can be applied; in parsing, it eats up the input characters until t can be applied to parse the remaining string. We define a helper function $until(parse, s)$ to find the first suffix of s that is parseable by a syntactic parser *parse* (written as $parse(s) \uparrow$) as follows

$$
until(parse, s) = (s_P, s_T) \quad \text{if } parse(s_T) \uparrow \wedge \forall s'_T \left(parse(S'_T) \uparrow \Rightarrow len(s'_T) < len(s_T)\right)
$$

$\mathsf{final}(t)$ is defined as follows.

$$
\begin{array}{l}
\mathsf{final}(t) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad (s_p, s_T) \leftarrow until(parse_{t.syn}, s_I), (\tau, s'_T) \leftarrow parse_{t.syn}(s_T) \\
\quad \textbf{return } ([s_p, \tau], s'_T) \\
print(\tau, s_T) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then } \mathsf{error} \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then} \\
\qquad s'_T \leftarrow print_{t.syn}(\tau', s_T), s' \leftarrow s_p +\!\!+ s'_T \\
\qquad \textbf{assert } until(parse_{t.syn}, s') = (s_p, s'_T) \\
\qquad \textbf{return } s'
\end{array}\right. \\
gSem = \lambda\beta \to \\
\quad \left\{\begin{array}{l}
parse(\tau, m) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then return } parse_{t.gSem(\beta)}(\bot, m) \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then return } parse_{t.gSem(\beta)}(\tau', m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then return } [\text{""}, print_{t.gSem(\beta)}(\bot, m)] \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then return } [s_p, parse_{t.gSem(\beta)}(\tau', m)]
\end{array}\right. \\
eff = t.eff
\end{array}\right\}
\end{array}
$$

Example 3.5. Consider the example in Fig. 7(f). Suppose the developer changes the output from "int v;" to "int v = 0;". Let us see how final(const(";")) works. In the function *syn.parse*, the input string s_I is the suffix after "int v". The *util* function returns a pair, where s_p is " = 0" and s_T is ";". In the function *gSem.parse*, it goes to the else branch by calling the *gSem.parse* function of const(";"). The noteworthy aspect lies in the printing orientation. In the function *gSem.print*, it recovers string " = 0", which is stored in s_p by going to the else branch. During *syn.print*, it concatenates s_p with s'_T which is ";".

Combinator $\mathsf{call}(t, v_1 = e_1, \ldots, v_n = e_n)$ denotes a template call whose target is t with actual argument e_i passed to formal parameter v_i. In brief, it prepares a new model by $\{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), \ldots, v_n =$

$get_{e_n}(m)\}$ first, and then delegates the conversion to t. The behavior of call is defined as follows.

$$
\mathsf{call}(t, v_1 = e_1, \ldots, v_n = e_n) \triangleq
$$

$$
\mathsf{BIT}\left\{
\begin{array}{l}
syn = \left\{ \begin{array}{l}
parse(s_I) \triangleq \textbf{if } s_I = \bot \textbf{ then } (\bot, \bot) \\
\qquad\qquad \textbf{else if } (\tau, s_T) = parse_{t.syn}(s_I) \textbf{ then } (\mathsf{C}\ \tau, s_T) \\
print(\tau, s_T) \triangleq \textbf{if } \tau = \bot \textbf{ then } print_{t.syn}(\bot, s_T) \\
\qquad\qquad \textbf{else if } \tau = \mathsf{C}\ \tau' \textbf{ then } print_{t.syn}(\tau', s_T)
\end{array} \right. , \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{ \begin{array}{l}
parse(\tau, m) \triangleq \textbf{do} \\
\quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), \ldots, v_n = get_{e_n}(m)\} \\
\quad \textbf{if } \tau = \mathsf{C}\ \tau' \textbf{ then} \\
\qquad m'_t \leftarrow parse_{t.gSem(\beta)}(\tau', m_t) \\
\qquad m_i \leftarrow put_{e_i}(m_{i-1}, get_{v_i}(m'_0)) \quad \text{where } i = 1..n, m_0 = m \\
\qquad \textbf{assert } get_{e_i}(m_n) = get_{v_i}(m'_t) \quad (i = 1..n) \\
\qquad \textbf{return } m_n \\
print(\tau, m) \triangleq \textbf{do} \\
\quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), \ldots, v_n = get_{e_n}(m)\} \\
\quad \textbf{if } \tau = \bot \textbf{ then return } \mathsf{C}\ print_{t.gSen(\beta)}(\bot, m_t) \\
\quad \textbf{else if } \tau = \mathsf{C}\ \tau' \textbf{ then return } \mathsf{C}\ print_{t.gSen(\beta)}(\tau', m_t)
\end{array} \right. \\
eff = \lambda(\beta, m) \rightarrow \textbf{do} \\
\quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), \ldots, v_n = get_{e_n}(m)\} \\
\quad \textbf{return } t.eff(\beta, m_t)
\end{array}
\right\}
$$

Note that when propagating the updates back to the original model, it requires no conflict in the merged result (see the assertion in *parse* of *gSem*).

3.4. Translation from surface to core

This subsection describes the translation from the surface language, designed for better usability, to the core language, whose semantics is formally defined. The basic idea of the translation is to map template fragments in the surface language onto primitives/combinators in the core language. It is not difficult to find a strong correspondence between the surface language and the core language if we compare their grammars in Fig. 6 and Fig. 8. For example, the FOR construct and the IF construct correspond to loop and ite, respectively; a hole corresponds to lex; a constant template literal corresponds to const; a verbatim areas is also interpreted as const.

For example, a template '''int «a | ID»()''' can straightforwardly be translated into a BIT BX t— seq(const("int"), lex(ID, a), const("()")). Note that there is a tailing space in "int ". This simple translation strategy has a limitation that makes the derived parser inflexible. For example, the derived parser of t accepts "int f()" but rejects "int   f ()".

The root cause is that the simple translation does not take the grammar of the generated text into account. If we know that the template generates a signature of a Java method, then we can translate the template into t'

seq(const("int"), space(" "), lex(ID, a), space(""), scope("(", ")", space("")))

by considering the white-space rules and bracket rules.

For simplicity, our approach does not require the full grammar of the generated text. Instead, we can define a *partial grammar* to guide the surface-to-core translation. The partial grammar is not intended to specify the syntactic constraints of the generated text, but is only used to enhance the derived parser. The *partial grammar* includes the following rules:

- **White-space rule** tells whether the white-spaces in the generated text can be *relaxed*, e.g., white-spaces can be added, changed, or removed whenever they may occur. If the rule is set, our approach uses space to handle white-spaces; otherwise, our approach will treat white-spaces as constants.
- **Operator rule** specifies the tokens (e.g., +, *) in template literals that must be considered as operators. If the rule and white-space rule are both set, then our approach will insert zero-width spaces around operators. In this way, template '''a+b''' accepts "a+b", "a +b", and "a + b".
- **Balanced bracket rule** tells what brackets should be balanced. If the rule is set, then our approach will scan template literals to match the scopes that start and end with balanced brackets (called *balanced scopes*).

We explain the surface-to-core translation by an example template

'''«a | ID»(){return 1+1;}'''

Firstly, we adopt the straightforward translation strategy and obtain an initial core program t_0 = seq(lex(ID, a), const("(){return 1+1}")). Secondly, supposing the balanced bracket rule tells (,), and {, } must be balanced, we scan const primitives in t_0 and detect balanced scopes. We rewrite t_0 to t_1 by extracting each balanced scope to a scope, as follows:

seq(lex(ID, a), scope("{", "}", nop), scope("{", "}", const("return 1+1")))

Thirdly, supposing the operator rule tells + is an operator, we rewrite t_1 into t_2 by tokenizing const primitives with operator +, as follows:

seq(lex(ID, a), scope("{", "}", nop), scope("{", "}",
  seq(const("return 1"), const("+"), const("1"))))

Finally, if the white space rule is set, we extract white-spaces from const primitives and insert zero-width spaces (let z = space("")) when necessary:

seq (lex(ID, a), z, scope("(", ")", z), space(""), scope("{", "}",
  seq(const("return"), space(" "), const("1"), z, const("+"), z, const("1"))))

3.5. Discussion

This subsection discusses a few issues related to our language design.

- **Bijectivity** A bijective transformation establishes a 1-to −1 mapping between the source and the target. BIT supports non-bijective transformations, i.e., different models may be mapped onto the same text, and vice versa. Take Fig. 2(a) as an example. Provided that the original string is empty, two models [{head="a",text="b"}] and [{head="A",text="b"}] will result in the same textual output. The semantics of default, unord, and final also support non-bijectivity.
- **Out-of-domain problem** A BX may encounter the out-of-domain problem. Take Fig. 3(b) as an example. If we changed the HTML by inserting <h2>Subtitle</h2> after <p>Good Bye!</p>, then the template in Fig. 3(a) will fail because the modified text is out of the domain of the transformation. The problem will also occur if we change the paragraph title within <h1>...</h1> to lowercase. Currently, BIT performs runtime checks and throws exceptions when the problem is detected. It will be our future work to improve the error handling strategy for this issue.
- **Ambiguity in parsing** As discussed by Zhu et al. (2020), the parsing process may contain ambiguity. For example, the same text "12345" can be syntactically parsed by either branch of «IF ...»1234«ELSE»123«ENDIF». At present, BIT always selects the branch achieving the largest string match. Zhu et al. (2020) also introduced other strategies for this problem.
- **Generality of BIT core** We believe that the core language of BIT can serve as a common BX foundation for other template languages, such as Acceleo and EGL. As demonstrated in Section 5.1, BIT covers the major features of existing template languages. It is possible to translate other template languages, other than Xtend, into the BIT core. It will be our future work to explore this potential application.

4. Well behavedness

In this paper, a BX is well behaved if it satisfies corresponding round-trip properties. This section discusses the round-trip properties of BIT BXs, which reflect the compatible behaviors of printers and parsers, as the *formal evidence of the soundness* of our approach.

4.1. Well behavedness of constructors

The formalization of our approach is built upon the constructors $*, \circledast, \odot$ defined in Section 3.2. Theorems 1, 2, and 3 state that these constructors preserve well-behavedness. The proof sketches are listed as follows.

Proof sketch of Theorem 1. Given well-behaved $val : \mathbb{M} \leftrightarrow V$ and $l : \mathbb{T} \leftrightarrow V$, we must prove $val * l$ satisfies properties (1) and (2) of $\alpha BX : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$. Because unit can be ignored from the definition of $*$, to prove (1) is equivalent to prove

$$v = get_l(t) \wedge m' = put_{val}(m, v) \implies v = get_{val}(m') \wedge s = put_l(t, v)$$

and to prove (2) is equivalent to prove

$$v = get_{val}(m) \wedge t' = put_l(t, v) \implies v = get_l(t') \wedge m = put_{val}(m, v)$$

Since val and l are well behaved (they satisfy (GetPut) and (PutGet)), the above two formulas hold. Thus, Theorem 1 holds.

Proof sketch of Theorem 2. Given well-behaved $l_1 : \mathbb{S} \leftrightsquigarrow V$ and $l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$, we must prove $l_1 \circledast l_2$ satisfies properties (1) and (2) of $\alpha BX : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$. To prove (1) is equivalent to prove

$$\begin{aligned}&(t, s_T) = parse_{l_1}(s) \wedge m' = parse_{l_2}(t, m)\\ &\quad \implies (t, s_T) = parse_{l_1}(s) \wedge t = print_{l_2}(t, m') \wedge s = print_{l_1}(t, s_T)\end{aligned}$$

and to prove (2) is equivalent to prove

$$\begin{aligned}&(t, s_T) = parse_{l_1}(s) \wedge t' = print_{l_2}(t, m) \wedge s' = print_{l_1}(t', s_T)\\ &\quad \implies (t', s_T) = parse_{l_1}(s') \wedge m = parse_{l_2}(t', m)\end{aligned}$$

Particularly, because l_1 and l_2 satisfy ParsePrint and PrintParse of $synBX$ and αBX, respectively, $m' = parse_{l_2}(t, m) \Rightarrow t = print_{l_2}(t, m')$ and $t' = print_{l_2}(t, m) \Rightarrow m = parse_{l_2}(t', m)$. Thus, Theorem 2 holds.

Proof sketch of Theorem 3. Given a binding update function $r : \mathbb{B} \times \mathbb{M} \to \mathbb{B}$ and a generator of αBX $pl : \mathbb{B} \to \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, we must prove $r \odot pl$ satisfies (ParsePrint) and (PrintParse) of $\beta BX : \mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$. Because the *parse* and *print* functions of a βBX always start from the same binding β, pl generates the same αBX from the same β. Thus, Theorem 3 holds straightforwardly.

4.2. Well behavedness of BIT primitives

BIT primitives are defined as BIT′ records. To prove a BIT primitive t is well behaved, we only need to prove $t.syn$, $t.sem$, and $t.val$ are well behaved. Afterwards, the constructors $*, \circledast, \odot$ shall ensure the well behavedness of t. We present the proof sketch of the well behavedness of BIT primitives as follows.

- Case const(s_c)—Since syn prints/parses the same constant s_c and sem maps s_c onto unit, it is easy to prove that they are well behaved. Besides, val is $UnitBX$. Accordingly, const(s_c) is well behaved.
- Case lex(ρ, e)—syn uses a regular pattern ρ to parse a string and to verify the string it prints out so it is well behaved. sem is equal to the BX, namely, REPLACE, defined in BiGUL (Ko and Hu, 2017). val, which is e, is assumed to be a well-behaved expression BX. Accordingly, const(s_c) is well behaved.
- Case space(s_w)—Its syn is very similar to that of lex, where space uses a regular pattern ρ_w for white-spaces. Its sem maps a non-$\perp$ string of white-spaces onto unit bidirectionally. Accordingly, space(s_w) is well behaved.
- Cases assign(v, e) and nop—It is easy to prove the two primitives are well behaved because their syn and sem functions actually do nothing.

4.3. Well behavedness of BIT combinators

BIT combinators are defined as BIT records. To prove a BIT combinator t is well behaved, we only need to prove $t.syn$ and $t.gSem(\beta)$ are well behaved. Afterwards, the constructors $\circledast, \odot$ shall ensure the well behavedness of t. We present the proof sketch of the well behavedness of BIT combinators as follows.

- Case seq(t_1, t_2)—syn applies $t_1.syn, t_2.syn$ in the forward order in parsing and in the reverse order in printing. Hence, syn should be well behaved if $t_1.syn, t_2.syn$ are well behaved. Similar to syn, $gSem$ applies $t_1.gSem, t_2.gSem$ in the forward order in parsing and in the reverse order in printing. Particularly, the assertion in $gSem(\beta)$ requires that t_1 is not affected by t_2. Hence, $gSem(\beta)$ is also well behaved.
- Case ite(e, t_1, t_2)—syn chooses $t_i.syn$ to parse the input string if $t_i.syn$ consumes longer prefix; while in printing, it chooses a branch according to the label (i.e., L/R) of the internal structure and asserts that the printed string cannot be consumed by the other branch. Hence, syn is well behaved. $gSem(\beta)$ chooses $t_i.gSem(\beta)$ according to the label of the internal structure and then enforces the branch condition e (which is also a well-behaved expression BX); while in printing, it chooses a branch according to the branch condition. Hence, $gSem(\beta)$ is well behaved.
- Case loop($e_{arr}, s_s, s_b, s_a, \lambda v \to t$)—Since we interpret the behavior of loop as seq, loop is well behaved if seq is so.
- Case scope(s_b, s_a, t)—In parsing, syn matches a scope in which s_b and s_a are balanced, and asks $t.syn$ to parse the inner content; while in printing, syn prepends s_b and appends s_a before and after the string printed by $t.syn$, in which s_b and s_a must be balanced. Hence, syn is well behaved. Since $gSem(\beta)$ delegates the conversion to $t.gSem(\beta)$, it is also well behaved. Accordingly, scope is well behaved.
- Case default(ρ, t)—If the input string can be parsed/printed by $t.syn$, then syn applies $t.syn$ to handle this string; otherwise, syn uses the regular pattern ρ to parse the string and to verify the string to be printed out. Hence, syn is well behaved. $gSem(\beta)$ delegates the conversion to $t.gSem(\beta)$ when the input is parsed/printed by $t.syn$; otherwise, $gSem(\beta)$ basically skips the conversion. It is easy to prove that $gSem(\beta)$ is also well behaved. As a result, default is well behaved.
- Case unord(t_1, t_2)—Similar to ite, syn of unord chooses from two pseudo branches seq(t_1, t_2) and seq(t_2, t_1). Particularly, syn assures that the string printed by one branch cannot be parsed by the other. Hence, syn is well behaved (see the proof of ite). Regarding $gSem(\beta)$, it chooses from the pseudo branches according to the labels of the internal structure, and then delegates the conversion to the branch it selects (and there is no branch switching that may happen in ite). It is easy to show that $gSem(\beta)$ is well behaved. Hence, unord is also well behaved.
- Case final(t)—ParsePrint of syn holds by definition; PrintParse of syn also holds because the assertion in $print$ ensures that the string printed can still be correctly parsed. Hence, syn is well behaved. $gSem(\beta)$ actually delegates the conversion to $t.gSem(\beta)$, so it is well behaved. Accordingly, final is well behaved.
- Case call($t, v_1 = e_1, \ldots, v_n = e_n$)—Due to the fact that call actually uses t to realize the conversion, it shall be well behaved.

5. Evaluation

In this evaluation, we focus on the following three research questions:

- **RQ1** Is BIT expressive in the specification of text generation templates?
  Rationale Compared with existing template languages, BIT is *unique* in the ability of bidirectional execution. To achieve bidirectionality, BIT compromises on its expressiveness. If the expressiveness is overly weakened, BIT will become a trivial language, unable to handle complex cases. Therefore, we must assess whether the expressiveness of BIT is sufficient to encompass the essential features of existing languages, ensuring that it remains capable of handling the most critical functionalities.
- **RQ2** Is BIT effective in bidirectionalizing model-to-text transformation?
  Rationale Model-to-text generation is frequently used in model-driven development. We wonder whether BIT can be applied to the bidirectionalization of existing model-to-text transformation programs?
- **RQ3** What are the merits and limitations of BIT compared with existing model-driven round-trip engineering approaches?
  Rationale Since several approaches to model-driven round-trip engineering already exist, we must assess the merits and limitations of BIT by comparing BIT with currently available methods.

To answer the research questions, we report three case studies.

- In the first case study (see Section 5.1), we focus on utilizing BIT to implement the example templates that are collected from the official tutorials of 7 template languages. This study aims to determine whether BIT covers the major features shared by existing template languages.
- In the second case study (see Section 5.2), we apply BIT to implement 12 example templates available on the Epsilon Generation Language (EGL) (Rose et al., 2008) playground website. Specifically, we try to answer whether BIT is capable of defining and bidirectionalizing text templates in the context of model-driven engineering.
- In the last case study (see Section 5.3), we leverage BIT to implement a code template originating from UMLLab, a real-world UML tool. This template generates a Java class from a UML *Class* element. Particularly, we compare BIT with existing model-code synchronization approaches to evaluate its effectiveness.

The details and implementation of BIT templates in the case studies can be found in He (2024).

5.1. Case study 1: language features coverage

The first case study focuses on **RQ1**, i.e., the *expressiveness*, by assessing to what extent BIT covers the major features of existing template languages.

Reference languages and their features. We chose 7 popular template languages as the *reference languages* of this study, including Velocity (Anon, 2023d), FreeMaker (Anon, 2023c), Xtend (Anon, 2023h), Acceleo (Anon, 2023b), Django (Anon, 2023e), Mustache (Anon, 2023f), and Nunjucks (Anon, 2023g), which cover different application domains(e.g., HTML page generation, code generation, configuration file generation, and document generation) and technical stacks (e.g., Java, JavaScript, and Python). The basic information on the seven template languages is listed in Table 1. The first two columns list the short and full names of the reference languages. The third column shows the application domains, which are claimed in the official documents. The fourth column lists URLs of the official tutorials.

Table 1
General information of the reference languages.

LID	Languages	Application	Tutorial URLs
V	Velocity	Web pages, XML, code, PostScript, document	https://velocity.apache.org/engine/devel/user-guide.html#what-is-velocity
F	FreeMaker	Web pages, e-mails, configuration files, code	https://freemarker.apache.org/docs/index.html
X	Xtend	General purpose	https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates
A	Acceleo	model-to-text generation	https://wiki.eclipse.org/Acceleo/Getting_Started
D	Django	Web pages	https://docs.djangoproject.com/en/4.1/intro/tutorial03/ https://docs.djangoproject.com/en/4.1/ref/templates/language/
M	Mustache	General purpose	http://mustache.github.io/mustache.5.html
N	Nunjucks	Web pages	https://mozilla.github.io/nunjucks/templating.html

Afterwards, we browsed through their official tutorials and summarized the major features of the reference languages.

Table 2 lists the language features and to what extent they are supported. The first two columns list the short and full names of template features. The third column briefly explains the meanings of the features. The fourth column lists the supported reference languages for each feature. The last column shows if BIT supports these features.

We found that the commonest features of template languages are conditionals (i.e., CD, supported by all reference languages), loops (i.e., LP, supported by all reference languages), template definition and calls (i.e., TD, supported by all reference languages), assignments (i.e., AN, supported by 6 reference languages), helper functions (i.e., HP, supported by 5 reference languages), verbatim output (i.e., VB, supported by 4 reference languages), and modularization (i.e., MD, supported by 6 reference languages). BIT supports CD (by IF construct and ite combinator), LP (by FOR construct and loop combinator), TD (by call expressions and call combinator), AN (by VAR construct and assign primitive), HP (by function calls), and VB (by verbatim construct and const primitive). BIT does not support MD, and our tool assumes that all the templates are defined in the same file. Nevertheless, MD is *irrelevant* to the expressiveness but is related to the maintainability. In theory, we can always copy all the templates, including the imported ones, to a single file. We assume that CD, LP, TD, AN, HP, and VB are the *essential features* of a template language.

The reference languages also have some minor features, such as template inheritance and overriding, whitespaces and escaping controls, filters, and protected areas, which are not commonly supported by most reference languages. BIT provides partial support for minor features. For example, the DEFAULT construct can be viewed as a restricted version of a protected area.

Benchmark set. We scanned the official tutorials of the reference languages and extracted their examples as our benchmark templates. We excluded an example if (1) it was incomplete (e.g., trivial single-line templates), (2) it was not intended to show the language features (e.g., it demonstrates the usage of library functions), or (3) it was a simplified/equivalent version of other examples. In total, we collected 57 benchmark templates, ranging from 3 to 44 LOCs, with an average of 8.3 LOCs. The supplemental file (He, 2024) includes all the benchmark templates. Finally, we use BIT to realize these templates to assess to what extent BIT can cover the essential features.

It is worth noticing that our benchmark templates are generally simple, as each typically focuses on a single language feature. A complex real-life template is a composition of multiple features. If we demonstrated that BIT covers these features, then BIT would be capable of handling complex templates.

Table 2
Features of template languages.

FID	Feature	Meaning	Supported By	BIT
CD	Conditionals	Selecting and executing a fragment based on a branch condition	V, F, X, A, D, M, N	Yes
LP	Loops	Repeating a fragment multiple times	V, F, X, A, D, M, N	Yes
TD	Template definition	Defining and calling a template	V, F, X, A, D, M, N	Yes
AN	Assignments	Assigning or binding a value to a variable	V, F, X, A, D, N	Yes
HP	Helpers	Defining and calling helper functions	F, X, A, M, N	Yes
VB	Verbatim	Output some text without interpreting the inner directives	V, F, D, N	Yes
MD	Modularization	Importing templates from other files	V, F, X, A, D, N	No
OTH	Others	Other minor features, such as template inheritance (X, A, D, N), whitespaces control (F, D, N), escaping control (F, N), filters (D, N), and protected area (A)		Partial

```
template django_case1(latest_question_list : [Question])
'''
«IF !latest_question_list.isEmpty()»
<ul>
«FOR question : Question IN latest_question_list»
 <li><a href="/polls/«question.id»/">«question.text»</a></li>
«ENDFOR»
</ul>
«ELSE»
<p>No polls are available.</p>
«ENDIF»
'''
```

(a) Example from Django

```
template freemaker_case8(animal : Animal)
'''
«IF animal.size == "small"»
 This will be processed if it is small
«ELSEIF animal.size == "medium"»
 This will be processed if it is medium
«ELSEIF animal.size == "large"»
 This will be processed if it is large
«ELSE»
 This will be processed if it is neither
«ENDIF»
'''
```

(b) Example from FreeMaker

```
template mustache_case1(name : String, value : int, in_ca : boolean, taxed_value : int)
'''
Hello «name | ID»
You have just won «value | INT» dollars!
«IF in_ca»
Well, «taxed_value | INT» dollars, after taxes.
«ELSE»
«ENDIF»
'''
```

(c) Example from Mustache

Fig. 10. Examples of benchmark templates implemented in BIT.

Table 3
Result of feature coverage.

LID	#Templates	CD	LP	TD	AN	HP	VB	MD	OTH
V	12(12)	4(4)	5(5)	3(3)	1(1)	–	1(1)	0(0)	–
F	15(12)	5(5)	5(5)	2(2)	1(1)	0(0)	1(1)	0(0)	4(1)
X	4(4)	3(3)	2(2)	0(0)	0(0)	0(0)	–	0(0)	1(1)
A	2(2)	0(0)	1(1)	0(0)	0(0)	1(1)	–	0(0)	1(1)
D	8(5)	3(3)	2(2)	0(0)	1(1)	–	1(1)	1(0)	2(0)
M	7(7)	4(4)	2(2)	1(1)	–	1(1)	–	–	–
N	9(5)	2(2)	2(2)	1(1)	1(1)	1(1)	0(0)	2(0)	2(0)

Result. Table 3 shows the results of feature coverage. The first two columns show the number of benchmark templates extracted from each reference language. The rest 8 columns list the numbers of templates that cover each language features. The numbers in brackets denotes the templates realized by BIT. For example, "15(12)" means there are 15 benchmark templates but BIT realized 12 of them. A cell of "0(0)" means that we did not find a complete example for this feature; a cell of "-" means that the language on the row does not support the feature on the column. Note that the sum of the rest 8 columns may be greater than the total number of templates in the second column because a template may cover multiple features.

According to Table 3, BIT successfully realized 47 out of the 57 benchmark templates. The lines of code for these BIT templates vary from 6 to 38 LOCs, with an average of 14.1 LOCs per template. Particularly, BIT handled all the templates demonstrating the essential features (i.e., CD, LP, TD, AN, HP, and VB). Accordingly, BIT covers all essential features and the expressiveness of BIT is comparable to existing template languages.

Fig. 3(a) is one benchmark template from Xtend (the original template is presented in Fig. 2(a)). Fig. 10 presents more examples:

- Fig. 10(a) is a template originated from Django, which generates a HTML fragment of an unordered question list.
- Fig. 10(b) is a template extracted from FreeMaker, which generates a constant sentence according to the value of the animal's size.
- Fig. 10(c) shows a template from Mustache, which a piece of text describing the dollars won by a person.

There are 10 benchmark templates which cannot be realized using BIT, including 3 templates concerning MD and 7 templates concerning OTH. As discussed above, BIT does not support MD because this feature is unrelated to language expressiveness. We further investigated the other 7 templates, and found that 2 templates demonstrate *nested macro* for FreeMaker, 1 template demonstrates *attempt-recover* for FreeMaker, 2 templates demonstrate *template extension* for Django and Nunjucks, 1 template demonstrates *asyncAll* for Nunjucks, and 1 template demonstrates *ifchanged* and *cycle* for Django.

For example, Fig. 11(a) depicts a FreeMaker template that contains *nested templates*. The macro definition specifies a template *border*, while the use of the template (i.e., <@border>...</@border>) fills the inner content to the <#nested> part. We think that these nested templates are similar, yet are not equivalent to, conventional template invocations. Additionally, Fig. 11(b) showcases a Nunjucks template that incorporates *asyncAll*, which serves as an asynchronous variant of traditional loop constructs.

Based on the result shown in Table 3, our answer to RQ1 is **yes**—BIT has sufficient expressiveness to specify text generation templates.

We tested each BIT template with some test cases to check whether it can correctly print/parse strings, bidirectionally and incrementally. All the BIT templates passed the tests, implying that our approach is functionally correct. Note that none of the reference languages can be bidirectionalized.

Table 4
General information about EGL templates and their BIT implementation.

TID	Functionality	Model	Text	LOCs (EGL)	LOCs (BIT)	Bidirectional
E1	Generate effort graphs	Project models	Graphviz	32	37	Yes
E2	Generate effort tables	Project models	HTML	27	39	Yes
E3	Generate task pie charts	Project models	HTML	31	43	Yes
E4	Generate LLM prompts	Image variant models	plain text	7	21	Yes
E5	Generate Java code from a state machine	State machine models	Java	47	53	Yes
E6	Generate a task list page	Task models	HTML	21	27	Yes
E7	Generate Graphviz image from a component model	Component models	Graphviz	89	70	Yes
E8	Generate language reports	Language models	HTML	38	27	Yes
E9	Generate SVG variants images	Palette models	SVG and HTML	57	79	Yes*
E10	Generate Java code from a Table model	table models	Java	82	68	Yes#
E11	Generate work breakdown diagrams	Work breakdown models	PlantUML	25	37	Partial
E12	Generate typed graphs	Type graph models	Graphviz	42	29	Partial

```
<#macro border>
  <p><#nested></p>
</#macro>
<@border>
content to be filled in the nested part
</@border>
```

(a) Nested macro in FreeMaker

```
<h1>Posts</h1>
<ul>
{% asyncAll item in items %}
  <li>{{ item.id | lookup }}</li>
{% endall %}
</ul>
```

(b) *asynchAll* in Nunjucks

Fig. 11. Examples of unsupported benchmark templates.

5.2. Case study 2: bidirectionalization of EGL programs

The second case study focuses on RQ2, specifically assessing effectiveness of BIT in bidirectionalizing model-to-text transformation programs. We collected 12 non-trivial EGL programs readily accessible on the EGL playground[2] as subjects of investigation. EGL is a *unidirectional* model-to-text generation language. We aim to use BIT to implement a *bidirectional* version of each of these subject programs.

Table 4 shows the general information about the subject programs and the corresponding BIT templates. The 12 subject programs fulfill a range of functionalities, encompassing the conversion of diverse models and the generation of various types of textual output. The sizes of the subject programs range from 7 to 89 lines of EGL code (41.5 LOCs on average); the sizes of BIT programs vary from 21 to 78 LOCs (44.2 LOCs on average).

We analyzed the bidirectionality of the BIT templates, as shown in the last column of Table 4. The BIT templates can be grouped into the following four categories.

- **Fully bidirectional** For E1–E8, we successfully developed equivalent BIT templates that exhibit full bidirectionality. A notable instance within this group is E1, and Fig. 12(a) presents the pivotal fragments of both the EGL and BIT templates. It is evident that the BIT version closely mirrors a direct translation of the EGL version.
- **Bidirectionalization with adjustment** For E9, we created a similar BIT implementation by adjusting the generation logic of the EGL program. As shown in Fig. 12(b), the EGL program first creates and appends the inverted combinations onto the group combination list (refer to the lines in yellow), and then generates all inverted combinations after the original combinations; however, to facilitate bidirectionality, the BIT version generates an inverted combination immediately following an invertible combination (refer to the lines in red). Obviously, the outputs of the two templates differ in the order of the generated SVG images but are isomorphic to each other.
- **Bidirectionalization with simplified expressions** For E9 and E10, we created bidirectional templates using BIT by simplifying the expressions in the EGL programs. Take E10 as an example. The EGL program uses `name.split("\.").last()` to generate a type name. Since BIT currently does not support `last()`, we simplified this expression to `name`. Note that this paper primarily focuses on the bidirectionalization of templates, rather than that of expressions. Extending the expressions supported in BIT will be our future work.
- **Partial bidirectionalization** For E11 and E12, we could only implement partially bidirectionalized versions using BIT because the EGL programs do not print all the data required to fully recover a model from text. Consequently, not all the text generated from a model can be modified. As shown in Fig. 12(c), this fragment of EGL code is partially bidirectionalizable because the `name`s of `task.partners` are not printed out (refer to the code in red). As a result, we cannot add new partners to a task by simply modifying the generated text. Note that the partial bidirectionalization is the intrinsic feature of these EGL programs, rather than a limitation of BIT.

For all the 12 EGL programs, BIT is able to define the corresponding bidirectionalized templates (with two partially bidirectionalized templates). Based on Table 4, our answer to RQ2 is **yes**—BIT is effective in bidirectionalizing model-to-text transformation programs.

We must emphasize that *one should not assume that every model-to-text transformation is bidirectionalizable*. In fact, while most template languages are Turing-complete, none of the bidirectional languages is. It is not difficult to define an unbidirectionalizable model-to-text transformation. For example, the following template fragment is ill-behaved when parsing "1"

```
«IF i> 0»«i|INT»«ELSE»1«ENDIF»
```

5.3. Case study 3: UML class templates

The third case study focuses on RQ3—assessing the merits and limitations of BIT by comparing it with existing approaches to model-code synchronization.

Baseline approaches. In this study, we choose UMLLab and a triple-graph-grammar-based (TGG-based) approach (Buchmann and Westfechtel, 2016) as the baseline approaches.

- UMLLab is a visual UML modeling tool. It provides a template-based approach to synchronizing UML models and Java code. In UMLLab, developers may define some code templates using

[2] EGL playground: https://eclipse.dev/epsilon/playground/

EGL version

```
digraph G {
        node[fontname="Arial",style="filled",fillcolor="azure"]
        edge[fontname="Arial"]

        [%for (p in Person.all){%]
        [%=p.getNodeId()%][label="[%=p.name%]"]
        [%}%]

        [%for (t in Task.all){%]
        [%=t.getNodeId()%][label="[%=t.title%]", fillcolor="wheat"]

        [%for (e in t.effort){%]
        [%=e.person.getNodeId()%]->[%=t.getNodeId()%]
[label="[%=e.percentage%]%"]
        [%}%]

        [%}%]
}
```

BIT version

```
template genEffortGraph(proj:Project)
'''
digraph G {
  node[fontname="Arial",style="filled",fillcolor="azure"]
  edge[fontname="Arial"]

  «FOR p : Person IN proj.people»
  «p.name.getPersonNodeId()|UUID»[label="«p.name|ID»"]
  «ENDFOR»

  «FOR t : Task IN proj.tasks»
  «t.getTaskNodeId()|UUID»[label="«t.title|ID»", fillcolor="wheat"]

  «FOR e : Effort IN t.effort»
  «e.person.getPersonNodeId()|UUID»->«t.getTaskNodeId()|
UUID»[label="«e.percentage|INT»"]
  «ENDFOR»

  «ENDFOR»
}
'''
```

(a) Full bidirectionalization

EGL version

```
[%
for (g in Group.all) {
  for (c in g.combinations.clone()) {
    if (c.invertible) {
      var inverse : new Combination;
      inverse.foreground = c.background;
      inverse.background = c.foreground;
      g.combinations.add(g.combinations.indexOf(c) + 1, inverse);
    }
  }
}
var combinationNumber = 0;
%]
…
[%for (c in g.combinations){
  combinationNumber ++;
  var class = "triangle" + Combination.all.indexOf(c);%]
<div style="padding:5px">
<h5>Option [%=combinationNumber%]</h5>
<svg … style="…;background-color:#[%=c.background.hex%]" …>
  …
</svg>
</div>
[%}%]
…
```

BIT version

```
template genSVGVariants(groups:[Group])
'''
…
«VAR combinationNumber : int = 0»
…
«FOR c : Combination IN g.combinations»
«combinationNumber = combinationNumber + 1»
«VAR cls : String = "triangle"+c.id»
<div style="padding:5px">
<h5>Option «combinationNumber|INT»</h5>
<svg … style="…;background-color:#«c.background.hex|HEX»"…>
  …
</svg>
</div>
«IF c.invertible»
«combinationNumber = combinationNumber + 1»
<div style="padding:5px">
<h5>Option «combinationNumber|INT»</h5>
<svg … style="…;background-color:#«c.background.hex|HEX»" …>
  …
</svg>
</div>
«ENDIF»
«ENDFOR»
</div>
…
'''
```

(b) Bidirectionalization with adjustment

```
[%for (task in wp.tasks){%]
*** [%=task.title%]
[%if(task.partners.notEmpty()){%][%=task.partners.collect(partner|"<color:" + partner.color + "><U+2B24></color>").concat(" ")%][%}%]
[%}%]
```

(c) Partially bidirectionalizable EGL code

Fig. 12. Examples of the bidirectionalization of EGL programs.

Xpand.[3] for code generation. Meanwhile, UMLLab can parse the source code to derive a UML model by interpreting the code templates as parsers. We choose UMLLab because, to the best of our knowledge, it is the only template-based round-trip engineering tool that is publicly accessible.

- The TGG-based approach (Buchmann and Westfechtel, 2016) combines code templates and TGG rules to facilitate model-code synchronization. For model-to-code transformation, it applies Acceleo templates to produce Java code from UML models. Conversely, for code-to-model transformation, it initially extracts ASTs (Abstract Syntax Trees) from Java code and subsequently applies TGG rules to synchronize these ASTs with UML models. We choose the TGG-based approach because it is a representative solution in MDE.

Subject synchronization. UMLLab has several built-in templates for code generation from UML models. We choose the *standard* template as the subject of this study. The basic idea of the template is as follows.

[3] The Xpand project: https://projects.eclipse.org/projects/modeling.m2t.xpand Note that Xpand is a unidirectional model-to-text transformation language and does not provide any support for bidirectional transformation.

- a UML Class is converted into a Java class;
- a UML Property is converted to a Java field, along with a getter method and a setter method (for single-valued Property only);
- an UML Operation is converted to a Java method with a default body when the Operation does not have a body code, or with the body code that is stored in the Operation element
- an UML Association is converted into two Java fields, as well as their getter/setter methods, within the source and the target classes of the Association, respectively;
- the conversion of UML interfaces is similar to that of UML Classes, except that it does not generate non-static field declarations and the body of all non-static methods.

Note that the subject is a classical synchronization in the model-driven community but is non-trivial. We must carefully deal with the conversion of modifiers, visibility flags, field types, Javadocs and comments, and method bodies.

Implementation of the subject using `bit`*.* The BIT implementation of the subject synchronization has 273 LOCs and 29 sub-templates. We briefly discuss how the synchronization is realized using BIT as follows.

First, as shown in Fig. 13(a), consider the template that generates a visibility flag for a Java element (e.g., a Java class/field/method)

```
template visibility(v : String)
'''
«IF v=="public" || v=="protected" || v=="private"»«v|VISIBILITY» «ELSEIF v=="package"»«ENDIF»
'''
```

(a) Visibility template

```
template comment(comments : [Comment])
'''
«IF !comments.isEmpty()»
/**«FOR c:Comment IN comments SEPARATOR " * ----------"»«FOR cl:String IN c.body.split("\n")»
 * «cl|COMMENTLINE»«ENDFOR»
«ENDFOR» */
«ENDIF»
'''
```

(b) Javadoc template

Fig. 13. Common templates.

```
template classifierForClass(c : Class)
'''
«comment(c.ownedComment)»
«visibility(c.visibility)»«IF c.isAbstract»abstract «ENDIF»↩
       «IF c.keywords.includes("static")»static «ENDIF»«IF c.isLeaf»final «ENDIF»class «c.name»
 «IF c.superType != null»extends «c.superType.typeString»«ENDIF»
 «IF !c.superInterfaces.isEmpty»implements ↩
       «FOR i : TypeRef IN c.superInterfaces SEPARATOR ","»«i.typeString»«ENDFOR»«ENDIF»
{
«FOR p : Property IN c.ownedProperty.filter((x:Property)->x.association==null)»
«attributeDeclarationForClass(p, c)»
... ««« the generation of the getter/setter methods for java field is omitted here
«ENDFOR»
«FOR p : Property IN c.ownedProperty.filter((x:Property)->x.association!=null && x.otherEnd!=null)»
«roleDeclarationForClass(p, c)»
... ««« the generation of getter/setter methods for association end is omitted here
«ENDFOR»
«FOR o : Operation IN c.ownedOperation»
«operationOfClass(o, c)»
«ENDFOR»
}
'''
```

Fig. 14. Class template (the simplified version).

from a UML visibility enumeration. The basic idea is to output the enumeration literal as a string when it is "`public`", "`protected`", or "`private`". However, when the enumeration is "`package`", the template outputs an empty string.

The second template, as shown in Fig. 13(b), generates a Javadoc for a Java element from a list of UML comments. A UML comment is a model element in UML that stores comments associated with other UML elements, e.g., UML Classes/Properties/Operations. Since a UML element may own multiple comments, the template prints a separator "`* ----------`" between any two comments. For each comment, it may also be multiline text. The trickiest part of this template is that it splits each comment by the line break character first and then prints each line with an extra prefix "`* `".

Fig. 14 shows the template for Java classes (we omit some details for simplicity). In the beginning, the template calls the Javadoc template and the visibility template. Afterwards, it performs several checks to produce modifiers, including `abstract`, `static`, and `final`. Subsequently, it outputs the class name and its super types. The class body is generated by printing its Properties, Association ends, and Operations. Note that in UML, if a Property, owned by a class, refers to an Association, then the Property represents an Association ends. In Fig. 14, we use some filters to choose between normal Properties and Association ends.

The template for generating a Java field declaration from a UML Property is shown in Fig. 15(a). Similar to the class template, it calls the visibility template and uses a complex branch structure to generates visibility flags and modifiers. Specifically, if the Property is both static and a leaf, then the visibility flag is generated by calling the visibility template; otherwise, it generates a private field. When the Property has a default value, the template also generates an initializer expression before the ending semi-colon.

Fig. 15(b) depicts the template responsible for generating field types, which is utilized in the field template. The basic idea of this template is to derive a type name based on the type and multiplicity of a given Property. However, if the Property type is unspecified, then it generates a default string type, accompanied by a comment stating `/*No type specified!*/`.

Fig. 15(c) presents the template that generates a getter method for a multi-valued Property. A generated getter will first check whether the corresponding field is initialized. If not, the getter will create an empty collection object to initialize the field before returning it. The template also distinguishes between static and non-static fields. For static fields, it generates the class name before the dot operator, whereas for non-static fields, it uses the `this` keyword.

The template for generating a Java field declaration from an Association end is shown in Fig. 16. The most interesting part of this template is that it generates a complicated Javadoc for the Association end to preserve the essential information about the Association.

There are also a few templates for the generation of type names, getter/setter methods, and Operations. Please refer to the supplemental file for the full details of the implementation.

Example synchronization. We tested the `BIT` implementation and Fig. 17 shows an example test. The UML model we used to perform the code generation contained three UML *Classes*: *Person*, *Student*, and *Course*. *Person* owned a *Property* called *name*; *Student* owned an *Operation*; there was an *Association* called *students_to_courses* between *Student* and *Course*. Then, we applied our `BIT` templates to generate

```
template attributeDeclarationForClass(p : Property)
'''
«IF p.isStatic && p.isLeaf»«visibility(p.visibility)»static final «ELSE»↩
    «IF p.isStatic»static «ENDIF»«IF p.isLeaf»final «ENDIF»private «ENDIF»↩
    «IF p.keywords.includes("transient")»transient «ENDIF»«typedMultiplicityElement(p)» «p.name»↩
    «IF !p.default.isEmpty()» = «p.default|CODELINE»«ENDIF»;
'''
```

(a) Field template

```
template typedMultiplicityElement(p : Property)
'''
«IF p._type==null»String/*No type specified!*/
«ELSE»«IF p.upper==-1»«container(p)»«ELSEIF p.upper==1»«_type(p._type, "void")»«ENDIF»«ENDIF»
'''
```

(b) Field type template

```
template toManyGetterForClass(p : Property, pc : Class)
'''
«visibility(p.visibility)»«IF p.isStatic»static «ENDIF»«container(p)» get«p.name.toFirstUpper()»()
{
  if («IF p.isStatic»«pc.name»«ELSE»this«ENDIF».«p.name» == null) {
    «IF p.isStatic»«pc.name»«ELSE»this«ENDIF».«p.name» = new «defaultContainerForProperty(p)»();
  }
  return «IF p.isStatic»«pc.name»«ELSE»this«ENDIF».«p.name»;
}
'''
```

(c) Field getter template

Fig. 15. Templates for field generation.

```
template roleDeclarationForClass(p : Property)
'''
«VAR o = p.otherEnd»
/**
 * <pre>
 *   «o.lower»..«o.upper»     «p.association.name»     «p.lower»..«p.upper»
 *
 *   «o._type.typeString» ------------------------«IF o.isNavigable»>«ELSE»-«ENDIF» «p._type.typeString»
 *
 *   «o.name»     «IF o.memberEndIndex == 0»>«ELSEIF o.memberEndIndex == 1»<«ENDIF»     «p.name»
 * </pre>
 */
«IF p.upper == -1»
«toManyDeclarationForClass(p)»
«ELSE»
«toOneDeclarationForClass(p)»
«ENDIF»
'''
```

Fig. 16. Association end template.

three Java classes from the model. The Java code was consistent with the UML model. After that, we modified the generated Java code and performed the parsing procedure to update the model. As shown in the bottom half of Fig. 17, we added a new field *favorites* in the Java class *Student* (the green lines). Finally, after parsing the modified code, we obtained an updated model which contains a new Association *favorite_courses* between *Student* and *Course*.

We also tested other code modifications, including (1) adding a new Java field corresponding to a UML *Property*; (2) deleting an existing Java field; (3) adding a new Java method; (4) deleting an exiting Java method; (5) modifying a method body; (6) adding a new Java class; (7) deleting an exiting Java class. BIT always can correctly synchronize the Java code with the UML model.

Comparison with umllab. The template language used in UMLLab is Xpand for model-to-code transformation. As a unidirectional template language, Xpand covers the major features of template languages, such as CD, LP, AN, TD, HP, and MD. It also provides built-in support for template fragments, template extension, lazy evaluation, aspect oriented programming, and model type systems.

UMLLab extends Xpand by providing it with a parsing semantics. With this extension, it is not surprising that UMLLab can also realize the subject synchronization since the subject is the built-in template. However, UMLLab does not base its printing/parsing semantics on the theory of bidirectional transformation. Consequently, UMLLab does not ensure the round-trip properties for user-defined templates. It may not successfully synchronize a model with the modified code, whereas BIT can. We explored and reported some failure cases as follows.

- **White-spaces matching** UMLLab treats the white-spaces in template literals as constant strings. When it parses code with a user-defined template, it requires that the white-spaces in the code should exactly match those in the template. Fig. 18(a) shows both BIT and Xpand versions of such a customized template. Thanking to the white-space rule in Section 3.4 and `space` primitive in Section 3.3, BIT can accepts strings such as
  `"Listener aListener = new Listener()"`
  `"Listener  aListener =  new Listener()"`
  However, UMLLab will only accept the first string.
- **Enforcing branch condition** UMLLab cannot properly enforce branch condition. When a template involves conditional structures, UMLLab may not be able to parse the string as expected. Fig. 18(b) presents an example, which generates a default value `"self"` when that of a Property is `"self"` but produces an empty

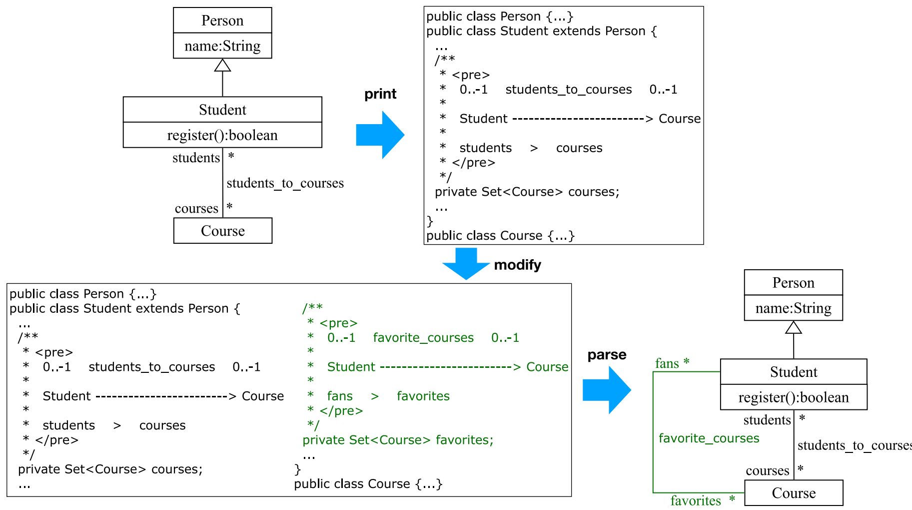


Fig. 17. Example execution (some code details are omitted).

```
BIT Version
template generateListener(p : Property)
'''Listener «p.name»Listener = new Listener();'''

Xpand Version
«DEFINE attribute (FailureStyleElement style, Classifier parent) FRAGMENT Listener FOR Property»
Listener «name»Listener = new Listener();
«ENDDEFINE»
```

(a) Failure case of white-space matching

```
BIT Version
template generateAttribute(p : Property)
'''«typedMultiplicityElement(p)» «p.name»«IF p.default=="self"» = "self"«ELSEIF p.default=="none"» = ""«ENDIF»;'''

Xpand Version
«DEFINE attribute (FailureStyleElement style, Classifier parent) FRAGMENT Declaration FOR Property»
«EXPAND typedMultiplicityElement FOR this» «name»«IF default=="self"» = "self"«ELSEIF default=="none"» = ""«ENDIF»;
«ENDDEFINE»
```

(b) Failure case of enforcing branch condition

```
BIT Version
template generateAttribute(p : Property)
'''«typedMultiplicityElement(p)» «p.name»«IF p.default!=null» = cast(«p.default»)«ENDIF»;'''

Xpand Version
«DEFINE attribute (FailureStyleElement style, Classifier parent) FRAGMENT Declaration FOR Property»
«EXPAND typedMultiplicityElement FOR this» «name»«IF default!=null» = cast(«default»)«ENDIF»;
«ENDDEFINE»
```

(c) Failure case of parsing complex string pattern

Fig. 18. Failure cases of UMLLab.

string when the default value in the model is "`none`". Assume that the Property is string-typed and has a name "a". Initially, its default value is "`self`". Both UMLLab and BIT will generate the following line of code

`String a = "self";`

If we change the code into `String a = "";`, BIT can successfully propagate the change back to the model by setting the default value of the Property to "`none`". However, UMLLab fails in the synchronization.

- **Parsing complex string patterns** We observed that UMLLab encounters difficulties when dealing with complex string patterns. As shown in Fig. 18(c), the templates generate a default value using the pattern

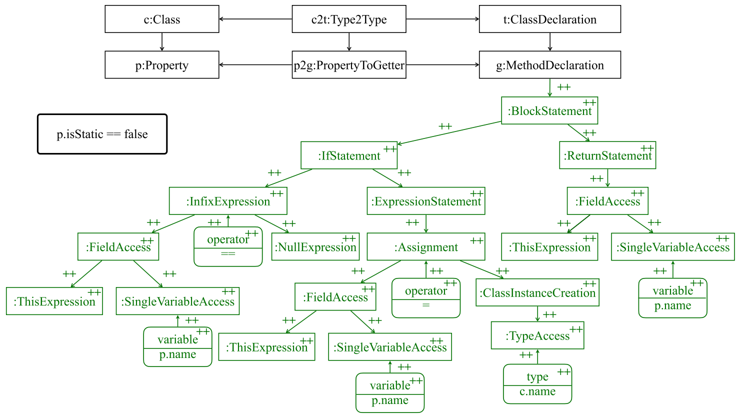
  `= cast(«p.default»)`

  which is the main difference from Fig. 15(a). BIT can print/parse this string pattern correctly. However, UMLLab cannot parse code according to the pattern. If we force UMLLab to use this template, then it generates

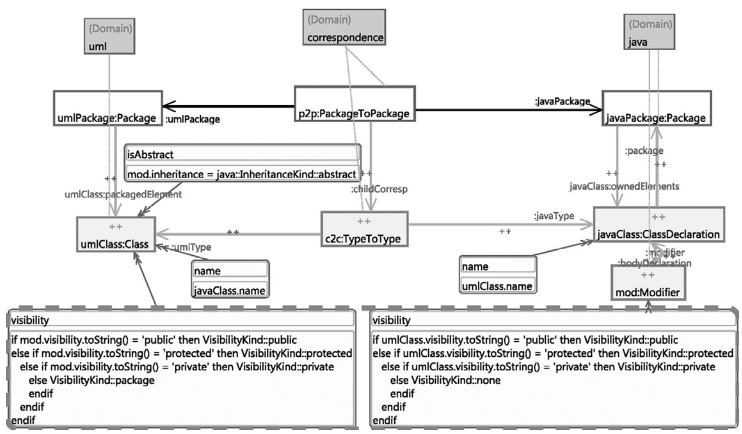
  `"= cast(...)"`
  `"= cast(cast(...))"`
  `"= cast(cast(cast(...)))"`, etc.

  after each synchronization, even though nothing is changed. In other words, UMLLab fails to ensure the round-trip properties.

Comparison with TGG-based approach. The TGG-based approach (Buchmann and Westfechtel, 2016) utilizes the state-of-the-art bidirectional (model) transformation technologies to facilitate model-code synchronization.



(a) A TGG rule for getter body



(b) A TGG rule containing redundancy expressions (adapted from [44])

Fig. 19. Deficiencies of the TGG-based approach.

We must emphasize that **TGG-based BX and `BIT` address two distinct problems**: the former is suitable for synchronizing structured data (e.g., graphs and models), while the latter focuses on synchronizing structured data with text.

Due to this difference, the TGG-based approach encounters the following deficiencies when it is applied to the subject synchronization.

- **Limited support for unstructured content** TGG rules must be defined between graphs. In theory, the TGG-based approach can only process the textual content that may be converted into a graph structure (e.g., an AST and a code graph). Hence, in Buchmann and Westfechtel (2016), a parser (specifically, MoDisco (Brunelière et al., 2014), a model discover) is used to parse Java code to an AST before further synchronization. If the textual content to be synchronized does not have a reasonable parser (e.g., it is plain text), then the TGG-based approach cannot be applied. Nevertheless, as demonstrated by case studies 1 and 2, `BIT` is capable of handling unstructured text (e.g., E4 in case study 2).
  Even in source code, there still are some unstructured parts, e.g., the Javadoc and comments. As shown in Fig. 16, the subject synchronization needs to analyze the text in Javadocs to update the model. As illustrated in Buchmann and Westfechtel (2016)[4], the TGG-based approach cannot handle plain text in Javadoc in an elegant manner.
- **High complexity of handling low-level code details** The TGG-based approach uses ASTs to specify the structure of the code, whereas `BIT` employs code templates. The employment of ASTs can lead to high complexity when handling low-level code details, such as method bodies. Take Fig. 15(c) as an example. The template outlines how a getter method body will be generated in an output-based way (Syriani et al., 2018). If we opt to realize the same template using TGGs, we would likely define a rule similar to the one depicted in Fig. 19. It is worth noting that this rule is still a simplified version, omitting numerous AST-related details. Furthermore, this rule is tailored specifically for the case where `p.isStatic==false`, so we also need another rule to handle the case where `p.isStatic==true`.
  In fact, as stated in Buchmann and Westfechtel (2016), the TGG-based approach does not consider low-level code details, such as method bodies and field initializers, because of the complexity. It concentrates on high-level class structures. Whereas, `BIT` is suitable for handling code details, and we believe that the two approaches complement each other.
- **Redundancy** When handling a value conversion, the TGG-based approach necessitates the definition of two distinct expressions for both forward and backward direction. As shown in Fig. 19(b), the TGG rule consists of two expressions within the dashed box to handle the conversion of visibility. This part is comparable to the

[4] Figure 7 in Buchmann and Westfechtel (2016) clearly shows this issue.

template shown in Fig. 13(a). Notably, a BIT template typically exhibits low redundancy because of its bidirectional semantics.
- **Consistency issue** As described in Buchmann and Westfechtel (2016), the TGG-based approach must work together with a model-to-code generator, which is used for printing. TGG rules are mainly used to propagate the changes from code to the model (i.e., parsing). It is therefore imperative that TGG-based rules must be consistent with the code generator to ensure that the printing and the parsing processes satisfy the round-trip properties. As discussed in Fischer et al. (2015), such consistency issues are the major challenge in developing BXs.
Nevertheless, BIT established a BX foundation for printing and parsing. By adopting BIT, the consistency issue can be mitigated.

After comparing BIT with UMLLab and the TGG-based approach, it is clear that BIT is unique as a well-defined template-based bidirectional model-text transformation language. The merits of BIT are summarized as follows.

- **Guarantee on round-trip properties** As is proved in Section 4, BIT provides a guarantee on the round-trip properties because of its foundation of bidirectional transformation.
- **Ability to handle unstructured text** BIT is designed for synchronizing models with text, without any assumption that the text follows a grammar or has an available parser.
- **Support for complex text patterns** BIT covers the major features of template languages, with its parsing semantics rigorously defined in Section 3.3. The syntax and semantics of BIT enable us to analyze a string using a template with complex text patterns.
- **Conciseness** BIT allows us to specify output patterns using templates, effectively concealing intricate and verbose abstract syntax.
- **Low redundancy** Based on the bidirectional semantics, our approach is capable of deriving a printer and a parser from a single specification of templates. There is no necessity to explicitly define a pair of the forward and backward transformations.

We are also aware that BIT owns a few limitations, as discussed as follows.

- **Order-sensitive** The parsing semantics of BIT is order-sensitive. Specifically, as illustrated in Fig. 14, the template adheres to a predefined order in which Java fields (derived from UML Properties) are parsed before Java methods (derived from UML Operations). Consequently, any rearrangement of these Java members would result in parsing failure, despite the fact that the occurrence order of Java members does not affect the meaning of a Java class. Note that *order-sensitive* is not always a harmful feature. For example, to parse a parameter list of a method, it is necessary to be order-sensitive.
- **Tree-like model** Presently, BIT only supports tree-like model structures. If a model to be synchronized is a graph, it must first be converted into a tree-like representation. After synchronizing text with the tree-like model, we can perform a bidirectional model transformation to synchronize the original model with its tree-like representation. Since real-world models are typically serialized into XML files, theoretically, it is not difficult to realize such a conversion. Additionally, existing bidirectional model transformation approaches, such as TGGs-based approaches, are capable of synchronizing a graph-like model with a tree-like model.
- **Left recursion** Parsers derived from BIT templates cannot handle left recursions. The following template shows an example containing left recursion:

```
template leftRec(a:int)
'''
«IF a>0»«leftRec(a-1)»+«a»«ELSE»0«ENDIF»
'''
```

Currently, our tool support cannot check left recursion statically. It will be our future work to investigate how to detect left recursions in templates by adopting existing techniques in the field of compilers.

6. Related work

6.1. Template languages and M2T transformation

Template languages, such as Velocity (Anon, 2023d), Freemaker (Anon, 2023c), Django templates (Anon, 2023e), Nunjucks (Anon, 2023g), and Mustache (Anon, 2023f), are widely used in modern Web engineering. In model-driven architecture, template languages, such as Acceleo (Anon, 2023b), Xtend templates (Anon, 2023h), and Java Emitter Template (Anon, 2023a), are also used to achieve model-to-text transformations (Rose et al., 2012). Specifically, Object Management Group proposed a standard template language, namely MOF Model to Text Transformation Language (Object Management Group, 2008), for code generation and document generation. State of the art template languages are unidirectional, which convert models/data into text/code. In this paper, we proposed BIT as an extension to Xtend templates to support the reverse conversion from text to models.

6.2. Model-model and model-code synchronization

In model-driven architecture, how maintain the consistency between high-level models and low-level models/code is a crucial problem (Hidaka et al., 2016; Diskin et al., 2016).

Most research efforts on this topic focused on the synchronization between models. Triple Graph Grammars (TGGs) were widely used to achieve model-to-model synchronization. Ehrig et al. (2007) discussed the issues of information preserving in the context of model synchronization. Hermann et al. (2015) proposed a formal foundation for bidirectional transformation using TGGs. Concrete implementations and optimizations of TGG-based model synchronization (Giese and Wagner, 2009; Hermann et al., 2012; Orejas et al., 2020) were also proposed. Xiong et al. (2013) proposed an ATL-based approach to model synchronization by defining a reverse semantics for ATL. Macedo and Cunha (2016) proposed a solver-based approach to bidirectional model transformation. Their basic idea is to convert ATL rules and QVT relations rules into Alloy constraints and then ask the Alloy solver to compute a synchronized model. EVL-Strace (Samimi-Dehkordi et al., 2018) is a model synchronization approach based on Epsilon Validation Language (EVL). It employs a trace model and EVL to achieve change propagation. Buchmann et al. (2022) proposed a layered framework for bidirectional model transformations combining declarative and imperative programming. Their approach achieved a high expressiveness and scalability. Boronat (2023) proposed EMF-Syncer which allowed for the synchronization between models and structured data. EMF-Syncer was highly scalable but limited to one-to-one mapping. Our previous work (He and Hu, 2018) on bidirectional model transformation proposed a putback-based language which enabled us to define a backward transformation from which a well-behaved BX can be derived. We also explored the synchronization between models and running systems (He et al., 2022).

There were a few research efforts on model-code synchronization. Buchmann and Westfechtel (2016) achieved the incremental round-trip engineering using TGGs. The model-to-code transformation is realized using Acceleo (Anon, 2023b) templates, while the code-to-model transformation is accomplished with the help of TGG rules. As discussed in Section 5.3, the TGG-based approach to model-code synchronization suffers several deficiencies, such as the limited support for unstructured content and the consistency issue.

Yu et al. (2012) proposed an approach to maintaining consistency between Ecore models and Java code. To synchronize an Ecore model m with current Java code c_u, their approach works as follows. First, generate Java code c_g from m using the built-in code generator of EMF. Second, convert c_g and c_u into ASTs t_g and t_u using a Java parser, respectively. Third, synchronize t_g with t_u by using GroundTram (Hidaka et al., 2010, 2013), a bidirectional graph transformation system, resulting in two updated ASTs t'_g with t'_u. Note that t'_g and t'_u are consistent at this point. Fourth, convert t'_g back to an Ecore model m' using the built-in model generator of EMF. Finally, serialize t'_u to Java code using a Java pretty-printer. This approach requires that a parser and a pretty-printer are available for the textual content to be synchronized. Thus, it cannot be applied to unstructured text.

Chivers and Paige (2009) proposed XRound, a reversible template language for the synchronization between models and XML-based documents. However, XRound cannot be used to define templates for non-XML generation, such as source code.

Lemerre (2023) proposed an approach RTL to reverse general-purpose templates. First, this approach derived a parser from a template to parse a string. Then, it used the concept of abstract interpretation and a constraint solver to determine a parsing tree and the value extracted out from the string. RTL was not built on the theory of bidirectional transformation so it did not have a well behaved bidirectional semantics. Neither, RTL did not support incremental printing/parsing.

6.3. Bidirectional transformation and bidirectional parsers

Bidirectional transformations (aka lenses) (Foster et al., 2005; Fischer et al., 2015; Hofmann et al., 2011) have been intensively studied in programming language community from the perspectives of theory, languages, and applications. Most existing BX approaches focused on the transformations over structured data (Ko and Hu, 2017), such as lists (Barbosa et al., 2010), trees (Foster et al., 2005; Hu et al., 2008), graphs (Hidaka et al., 2010, 2013), and relational databases (Tran et al., 2020). Zhang and Hu (2022), Zhang et al. (2023) proposed bidirectional live programming approaches for incomplete and object-oriented UI programs.

Bohannon et al. (2008) proposed Boomerang, the resourceful lenses for string data. Boomerang used regular patterns to tokenize strings into chunks and then synchronized these chunks with lenses for dictionaries and bidirectional regular transducers (aka string lens combinators). However, Boomerang is not template-based and cannot be used to reverse templates.

Cheney et al. (2017) discussed the principle of least change in the context of BX. The principle states that a BX should not make changes to the artifact beyond what is strictly necessary. They argued that a BX should follow this principle to provide *reasonable* behavior. We attempt to adhere to this principle by carefully designing the semantics of `BIT` in Section 3.3. For example, during printing, the primitive `space` will not change the original string if there are already some whitespace characters. Our future work will be to refine the semantics of `BIT` to ensure least change to both model and text.

There were also many research efforts on bidirectional parsers (Zhu et al., 2020; Matsuda and Wang, 2018b; Xia et al., 2019). Zhu et al. (2020) proposed BiYacc, a domain-specific language for the specification of mapping rules from abstract syntax to concrete syntax of a programming language. Afterward, BiYacc rules can be compiled into BiGUL (Ko and Hu, 2017) programs to achieve the synchronization between abstract syntax trees (ASTs) and concrete syntax trees (CSTs). Matsuda and Wang (2018b) proposed FliPpr, a system for deriving parsers from pretty-printers. FliPpr provided a surface language for describing a pretty printer of a language, which was further translated into an ugly printer by forgetting smart layouting mechanism. The ugly printer was then processed by their grammar-based inversion system (Matsuda and Wang, 2018a) to realize the parsing semantics. Similar to BiYacc and FliPpr, `BIT` also addresses the issue of consistent printing and parsing. The major difference is that `BIT` is a template-based approach while BiYacc and FliPpr are grammar-based. The core language of `BIT` is designed to handle template fragments, rather than grammar productions.

Xia et al. (2019) proposed the concept of *monadic profunctors* to define the behavior of bi-parsers (i.e., a pair of parse and print functions). In this way, bi-parsers can be combined in a monadic way. The concept of αBX can be regarded as an extension to the monadic profunctor—αBX adopts a value BX, rather than a *comap* function, to achieve more flexible bidirectional behavior.

Comby van Tonder and Le Goues (2019) was a template language for code matching and rewriting. In matching process, Comby interprets a code template and finds code fragments which match the structure of the template. In rewriting process, Comby generates code by filling the holes in the template. `BIT` is inspired by Comby to use partial grammars to guide the interpretation of templates. However, Comby was not designed for bidirectional transformation so it does not assure any round-trip properties.

7. Conclusions and future work

In this paper, we proposed `BIT`, a bidirectional template-based approach to incremental printing and parsing. We defined the surface language of `BIT` by extending conventional template language for better usability. We formally specified the semantics of `BIT` by means of the definition of the core language of `BIT`. The proof sketch of the well behavedness and three case studies were also presented to show the soundness and the expressiveness of our approach.

Regarding the future work, we plan to improve our approach in the following aspects. First, we will enhance `BIT` by adding more language features, such as modularization and template extension. Second, we will optimize the prototype implementation of `BIT` to improve its robustness and runtime efficiency. Finally, we will try to refine our approach by mapping the newly proposed string calculus (Crichton and Krishnamurthi, 2024) onto `BIT` core.

CRediT authorship contribution statement

Xiao He: Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization. **Tao Zan:** Writing – review & editing, Validation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have provided the link to my shared data/code in the paper.

Declaration of Generative AI and AI-assisted technologies in the writing process

Statement: During the preparation of this work the authors used ERNIE Bot 3.5 in order to grammar checks. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Acknowledgments

This work is funded by National Key Research and Development Program of China (No. 2023YFB3002903), Natural Science Foundation of Fujian Province for Youths (No. 2021J05230), Beijing Natural Science Foundation (No. 4192036).

References

Akdur, D., Garousi, V., Demirörs, O., 2018. A survey on modeling and model-driven engineering practices in the embedded software industry. J. Syst. Archit. 91, 62–82.

Anon, 2023a. Java Emitter Template Project, https://projects.eclipse.org/projects/modeling.m2t.jet.

Anon, 2023b. Acceleo project. https://eclipse.dev/acceleo/.

Anon, 2023c. Apache FreeMaker project. https://freemarker.apache.org.

Anon, 2023d. Apache Velocity Project. https://velocity.apache.org.

Anon, 2023e. Django templates. https://docs.djangoproject.com/en/4.2/topics/templates/.

Anon, 2023f. Mustache project. http://mustache.github.io.

Anon, 2023g. Nunjucks project. https://mozilla.github.io/nunjucks/templating.html.

Anon, 2023h. Xtend templates. https://eclipse.dev/Xtext/xtend/documentation/203_xtend_expressions.html#templates.

Barbosa, D.M., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C., 2010. Matching lenses: Alignment and view update. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10, Association for Computing Machinery, New York, NY, USA, pp. 193–204.

Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A., 2008. Boomerang: Resourceful lenses for string data. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '08, Association for Computing Machinery, New York, NY, USA, pp. 407–419.

Boronat, A., 2023. EMF-Syncer: scalable maintenance of view models over heterogeneous data-centric software systems at run time. Softw. Syst. Model..

Boussaïd, I., Siarry, P., Ahmed-Nacer, M., 2017. A survey on search-based model-driven engineering. Autom. Softw. Eng. 24 (2), 233–294.

Brown, A.W., 2004. Model driven architecture: Principles and practice. Softw. Syst. Model. 3 (4), 314–327.

Brunelière, H., Cabot, J., Dupé, G., Madiot, F., 2014. Modisco: A model driven reverse engineering framework. Inf. Softw. Technol. 56 (8), 1012–1032.

Buchmann, T., Bank, M., Westfechtel, B., 2022. BXtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language. J. Syst. Softw. 189, 111288.

Buchmann, T., Westfechtel, B., 2016. Using triple graph grammars to realise incremental round-trip engineering. IET Softw. 10 (6), 173–181. http://dx.doi.org/10.1049/iet-sen.2015.0125, arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2015.0125, URL https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2015.0125.

Cheney, J., Gibbons, J., McKinna, J., Stevens, P., 2017. On principles of least change and least surprise for bidirectional transformations. J. Object Technol. 16 (1), 3:1–3:31. http://dx.doi.org/10.5381/jot.2017.16.1.a3.

Chivers, H., Paige, R.F., 2009. XRound: A reversible template language and its application in model-based security analysis. Inf. Softw. Technol. 51 (5), 876–893.

Crichton, W., Krishnamurthi, S., 2024. A core calculus for documents: Or, lambda: The ultimate document. Proc. ACM Program. Lang. 8 (POPL), http://dx.doi.org/10.1145/3632865.

Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K., 2016. A three-dimensional taxonomy for bidirectional model synchronization. J. Syst. Softw. 111, 298–322.

Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G., 2007. Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (Eds.), Fundamental Approaches To Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 72–86.

Fischer, S., Hu, Z., Pacheco, H., 2015. The essence of bidirectional programming. Sci. China Inf. Sci. 58 (5), 1–21.

Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A., 2005. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, Association for Computing Machinery, New York, NY, USA, pp. 233–246.

Giese, H., Wagner, R., 2009. From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. 8 (1), 21–43.

He, X., 2024. Incremental and bidirectional template language for model-text synchronization. http://dx.doi.org/10.5281/zenodo.11124048.

He, X., Avgeriou, P., Liang, P., Li, Z., 2016. Technical debt in MDE: A case study on GMF/EMF-Based projects. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS '16, Association for Computing Machinery, New York, NY, USA, pp. 162–172. http://dx.doi.org/10.1145/2976767.2976806.

He, X., Hu, Z., 2018. Putback-based bidirectional model transformations. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 434–444.

He, X., Hu, Z., Meng, N., 2022. A theoretic framework of bidirectional transformation between systems and models. Sci. China Inf. Sci. 65 (10), 202103.

Hermann, F., Ehrig, H., Ermel, C., Orejas, F., 2012. Concurrent model synchronization with conflict resolution based on triple graph grammars. In: de Lara, J., Zisman, A. (Eds.), Fundamental Approaches To Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 178–193.

Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T., 2015. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Softw. Syst. Model. 14 (1), 241–269.

Hidaka, S., Asada, K., Hu, Z., Kato, H., Nakano, K., 2013. Structural recursion for querying ordered graphs. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP '13, Association for Computing Machinery, New York, NY, USA, pp. 305–318.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K., 2010. Bidirectionalizing graph transformations. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10, Association for Computing Machinery, New York, NY, USA, pp. 205–216.

Hidaka, S., Tisi, M., Cabot, J., Hu, Z., 2016. Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. 15 (3), 907–928.

Hofmann, M., Pierce, B., Wagner, D., 2011. Symmetric lenses. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, Association for Computing Machinery, New York, NY, USA, pp. 371–384.

Hu, Z., Mu, S.-C., Takeichi, M., 2008. A programmable editor for developing structured documents based on bidirectional transformations. High.-Order Symb. Comput. 21 (1), 89–118.

Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D., 2019. Survey and classification of model transformation tools. Softw. Syst. Model. 18 (4), 2361–2397.

Ko, H.-S., Hu, Z., 2017. An axiomatic basis for bidirectional programming. Proc. ACM Program. Lang. 2 (POPL).

Lemerre, M., 2023. Reverse template processing using abstract interpretation. In: Hermenegildo, M.V., Morales, J.F. (Eds.), Static Analysis. Springer Nature Switzerland, Cham, pp. 403–433.

Macedo, N., Cunha, A., 2016. Least-change bidirectional model transformation with QVT-R and ATL. Softw. Syst. Model. 15 (3), 783–810.

Matsuda, K., Wang, M., 2018a. Embedding invertible languages with binders: A case of the FliPpr language. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell. In: Haskell 2018, Association for Computing Machinery, New York, NY, USA, pp. 158–171.

Matsuda, K., Wang, M., 2018b. FliPpr: A system for deriving parsers from pretty-printers. New Gener. Comput. 36 (3), 173–202.

Object Management Group, 2008. MOF Model to Text Transformation Language, version 1.0. https://www.omg.org/spec/MOFM2T/1.0/About-MOFM2T/.

Object Management Group, 2024. Model Driven Architecture, http://www.omg.org/mda.

Orejas, F., Pino, E., Navarro, M., 2020. Incremental concurrent model synchronization using triple graph grammars. In: Wehrheim, H., Cabot, J. (Eds.), Fundamental Approaches To Software Engineering. Springer International Publishing, Cham, pp. 273–293.

Rodrigues da Silva, A., 2015. Model-driven engineering: A survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. 43, 139–155.

Rose, L.M., Matragkas, N., Kolovos, D.S., Paige, R.F., 2012. A feature model for model-to-text transformation languages. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. MiSE '12, IEEE Press, pp. 57–63.

Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C., 2008. The epsilon generation language. In: Schieferdecker, I., Hartman, A. (Eds.), Model Driven Architecture – Foundations and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–16.

Samimi-Dehkordi, L., Zamani, B., Kolahdouz-Rahimi, S., 2018. EVL+Strace: a novel bidirectional model transformation approach. Inf. Softw. Technol. 100, 47–72.

Syriani, E., Luhunu, L., Sahraoui, H., 2018. Systematic mapping study of template-based code generation. Comput. Lang. Syst. Struct. 52, 43–62.

Tran, V.-D., Kato, H., Hu, Z., 2020. Programmable view update strategies on relations. Proc. VLDB Endow. 13 (5), 726–739.

Umuhoza, E., Brambilla, M., 2016. Model driven development approaches for mobile applications: A survey. In: Younas, M., Awan, I., Kryvinska, N., Strauss, C., Thanh, D.v. (Eds.), Mobile Web and Intelligent Information Systems. Springer International Publishing, Cham, pp. 93–107.

van Tonder, R., Le Goues, C., 2019. Lightweight multi-language syntax transformation with parser parser combinators. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. In: PLDI 2019, Association for Computing Machinery, New York, NY, USA, pp. 363–378.

Xia, L.-y., Orchard, D., Wang, M., 2019. Composing bidirectional programs monadically. In: Caires, L. (Ed.), Programming Languages and Systems. Springer International Publishing, Cham, pp. 147–175.

Xiong, Y., Song, H., Hu, Z., Takeichi, M., 2013. Synchronizing concurrent model updates based on bidirectional transformation. Softw. Syst. Model. 12 (1), 89–104.

Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L., 2012. Maintaining invariant traceability through bidirectional transformations. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, IEEE Press, pp. 540–550.

Zhang, X., Guo, G., He, X., Hu, Z., 2023. Bidirectional object-oriented programming: Towards programmatic and direct manipulation of objects. Proc. ACM Program. Lang. 7 (OOPSLA1).

Zhang, X., Hu, Z., 2022. Towards bidirectional live programming for incomplete programs. In: 2022 IEEE/ACM 44th International Conference on Software Engineering. ICSE, pp. 2154–2164. http://dx.doi.org/10.1145/3510003.3510195.

Zhu, Z., Ko, H.-S., Zhang, Y., Martins, P., Saraiva, J., Hu, Z., 2020. Unifying parsing and reflective printing for fully disambiguated grammars. New Gener. Comput. 38 (3), 423–476.

Dr. Xiao He is an associate professor at University of Science and Technology Beijing (USTB). He obtained his Ph.D. in 2012 from Peking University. He was a visiting scholar in Groningen University from 2015 to 2016. His research interests include model-driven engineering, bidirectional transformation, domain-specific languages, and software testing.

Tao Zan received his B.S. degree from University of Science and Technology of China, China, in 2011 and his Ph.D. degree from the Graduate University for Advanced Studies, Japan, in 2016. From 2016, he is a founder of Ximen Yunqi Information Technology Co., LTD. Since 2020, he has been an associate professor at Longyan University. His research interests are in bidirectional transformation, voice coding, and software bug analysis.