# Tutorial of Bee

## 1 Introduction

Bee is a program synthesis framework that builds on the **data-actions** model. That is, a PBE task can be viewed as processing some **data entities** with repetitive **user actions**. For example, in a task to download hundreds of attachments from an email client, the data are the emails and attachments, and the user actions are the downloading operations on some attachments.

If the target PBE tasks match this model, developers can provide specifications of the data and actions to customize a PBE synthesizer on Bee.

Specifically, developers need to represent their data as relational tables so that Bee can correctly process them. Also, developers need to specify the user actions as program functions/methods. In this way, Bee can record these methods' invocations as examples, and invoke these methods after the target actions are generated.

In the followings, we would use a PBE task as running example to introduce how to create a synthesizer appropriate to the given tasks with Bee step by step (Section 2 to 5). We then briefly introduce the internals of Bee (Section 6) and discuss how to design more effective table schemas (Section 7).

## 2 Developing PBE for Email Client

Now imagine you are a developer of an email client and you want to support an auto-download function. To start with, consider the following specific tasks from the end-users of your email client.

**Task A**: From emails named with "assignment", save attachments to folders named by students' email addresses. Figure 1 shows three example emails. Two emails are from students and the three attachments included should be saved to corresponding target paths. For other irrelevant emails, no actions should be taken.

| Email Title | Sender | Attachment | Target Path |
|---|---|---|---|
| assignment1 | david@xxx.edu | project.zip | ~/downloads/A1/david |
| | | report.pdf | ~/downloads/A1/david |
| assignment1 | lucy@xxx.edu | assign1.zip | ~/downloads/A1/lucy |
| Discount Now | adv@yyy.com | discount.jpg | N/A |

Figure 1: Examples for task A.

**Task B**: Download bills sent from a bank to folders named by year-months. Figure 2 shows four example emails. Three emails are from the bank and the attached bills are saved to corresponding target paths. One email is irrelevant and no action is taken.

| Email Title | Sender | Date | Attachment | Target Path |
|---|---|---|---|---|
| Billing info | bill@boc.com | 2021-01-31 | bill.pdf | ~/bills/2021/Jan |
| Billing info | bill@boc.com | 2021-02-28 | bill.pdf | ~/bills/2021/Feb |
| Billing info | bill@boc.com | 2021-03-31 | bill.pdf | ~/bills/2021/Mar |
| Discount Now | adv@yyy.com | 2021-03-31 | discount.jpg | N/A |

Figure 2: Examples for task B.

Task A and Task B are two typical scenarios that can be automated by PBE. As a developer of the email client, you would like to add such a PBE feature to your software. With this PBE feature, not only Task A and Task B, but also other similar user tasks can be benefited. You may consider extending the PBE features for more scenarios in practice, but in this tutorial let's just focus on these two tasks.

Compared with developing PBE from scratch, using program synthesis framework like Bee is more convenient. Specifically, we need to supply Bee with the table representation of data (Section 3) and methods corresponding to the user actions (Section 4).

## 3 Modeling Data

We now consider what are the data in email client and design table schemas to represent them. An email has fields such as title, content, sender, receivers, datetime, attachments, to which email it is replied, and so on. These fields can all be modeled in relational tables, but in this example PBE we only model some of them. Readers can follow the approach and apply to other fields.

Since an email may include multiple attachments, we consider modeling attachments in a separate table. Figure 3 shows the table modeling of data in Figure 1. The first table holds basic email information (`string` type) plus an `Eid` to mark the identity of each email (`id` type). An `id` value can be understood as the address/pointer/reference of an object. The second table holds attachments' information. Each row (attachment) has a filename, a distinct `Fid` and the `Eid` of the email it belongs to.

| **Eid** | **Title** | **Year** | **Month** | **Sender** |
|---|---|---|---|---|
| e1 | assignment1 | 2021 | Mar | david@xxx.edu |
| e2 | assignment1 | 2021 | Mar | lucy@xxx.edu |
| e3 | Discount Now | 2021 | Mar | adv@yyy.com |

| **Eid** | **Fid** | **Filename** |
|---|---|---|
| e1 | f1 | project.zip |
| e1 | f2 | report.pdf |
| e2 | f3 | assign1.zip |
| e3 | f4 | discount.jpg |

Figure 3: Relational-table modeling of emails and attachments in Figure 1.

## 3.1 APIs for modeling data

To implement the above design, BEE provides a set of annotation-based APIs. For each table schema, you need to define a class, then mark it with the `[Entity]` annotation (using C#). If you are using Java or Python, the annotation should be `@Entity`. Below shows the declarations of two table schemas in C#.

```
[Entity]
public class EmailEntity {
    ...
}

[Entity]
public class AttachmentEntity {
    ...
}
```

Then, for each column in a table schema, you need to define a non-static method with no parameters, and mark the method with `[Field]`. The type of column is the same as the return type of the method. For example, the codes below composites an `Email` object and define the four string columns in the email table.

```
[Entity]
public class EmailEntity {
    private Email _email;
    EmailEntity(Email email) {
        _email = email;
    }

    [Field]
    public string Title() {
        return _email.Title;
    }

    [Field]
    public string Sender() {
        return _email.Sender;
    }

    [Field]
    public string Year() {
        return _email.SendTime.Year.ToString();
    }

    [Field]
    public string Month() {
        return _email.SendTime.Month.ToString();
    }

    ...
}
```

Regarding columns of `id` type, just define a method returning an object and mark the method with `[IdField]`. For example, the `Eid` method below returns an `Email` object, then the address/reference/pointer of this object would appear in the `Eid` column of this table.

```
[Entity]
public class EmailEntity {
    ...

    [IdField]
    public Email Eid() {
        return _email;
    }
}
```

## 4 Actions

In addition to the data, we also need to specify the user actions as program functions/methods, including the method names, parameters, and implementations. For example, to specify a download operation, we may just name the method as `Download`, and have two parameters (the attachment to download, and where it is downloaded to) like `Download(Attachment attachment, string targetPath)`. This implementation, by definition, should save the `attachment` to `targetPath`.

The action instances for the data in Figure 3 are shown in Figure 4. Note that there is a one-to-one correspondence between a set of actions, and their table representation.

| action | attachment | target path |
|---|---|---|
| download | f1 | ~/downloads/A1/david |
| download | f2 | ~/downloads/A1/david |
| download | f3 | ~/downloads/A1/lucy |

Figure 4: Example actions

### 4.1 APIs for action signature

To specify an action in C#, simply define a corresponding static method and mark it with `[Action]`. The codes below show the definition of download.

```
[Entity]
public class AttachmentEntity {
    ...

    [Action]
    public static void Download(Attachment attachment,
                                string targetPath) {
        attachment.SaveTo(targetPath);
    }
}
```

## 5 Connect Everything

After defining the table schemas and action signatures, let us see how to connect them to the framework and invoke the synthesis procedure. The codes below demonstrate how to supply example data/actions and do synthesis.

```
// prepared Email objects e1, e2, e3
// and Attachment objects f1, f2, f3, f4
...

var emailsTable = new DataParser<EmailEntity>()
    .ParseAsTable(new List<EmailEntity> {
        new EmailEntity(e1),
        new EmailEntity(e2),
        new EmailEntity(e3)
});
var attachmentsTable = new DataParser<AttachmentEntity>()
    .ParseAsTable(new List<AttachmentEntity> {
        new AttachmentEntity(e1, f1),
        new AttachmentEntity(e1, f2),
        new AttachmentEntity(e2, f3),
        new AttachmentEntity(e3, f4),
});

var recorder = ActionRecorder.GetInstance();
recorder.Start();
AttachmentEntity.Download(f1, "~/downloads/A1/david");
AttachmentEntity.Download(f2, "~/downloads/A1/david");
AttachmentEntity.Download(f3, "~/downloads/A1/lucy");
recorder.End();
var exampleActions = recorder.RecordedActions;

var synthesizer = new Synthesizer(new List<Table> {
                                      emailsTable,
                                      attachmentsTable },
                                  exampleActions);
var program = synthesizer.Synthesize(10000).First();
```

## 6 Internals of Bee

You may wonder how BEE works. With the data represented as relational tables and actions' signatures provided, BEE ships a language extended from SQL to synthesize programs that generate actions from data tables. A BEE program first takes as input the tables converted from the data, then performs transformations such as filtering rows, aggregating information (max, count, etc.) along columns and so on. In principle, developers do not have to bother handling these transformations, but should understand what transformations are available (Section 6.2) in order to adjust the table schema for representing data.

After transformation, the BEE program would map appropriate columns in the transformed tables to actions' arguments, and generate corresponding action instances. Finally, the generated actions can be used to replace manual repetitive work.

### 6.1 Synthesized Program

In order to transform the tables in Figure 3 to the table in Figure 4, the program should be able to: (1) filter assignment emails by title; (2) join the two tables on the two `Eid` columns; (3) select columns to map to the parameters of the download signature, where the destination is by performing string transformation on the sender column.

Listing 1 shows the synthesized program for the above PBE task. The three statements correspond to the above steps.

```
function(t_emails, t_attachs)
{
    t1 = Filter(t_emails, strContains(Title, "assignment"));
    t2 = Join(t_attachs, t1, t_attachs.Eid, t1.Eid);
    Yield(Download, t2, Fid, Mutate(concat(...), Sender));
}
```

Listing 1: Synthesized Program.

### 6.2 Grammar

Figure 5 shows the simplified grammar of the DSL in BEE. While developers do not need to modify this DSL, they can check the DSL's mechanism and adjust the data modeling and action signatures accordingly.

Overall, a program $p$ contains two consecutive subroutines, a transform program $p_t$ for transforming intermediate tables, and a mapping program

| | | | |
|---|---|---|---|
| Program | $p$ | $:=$ | $p_t\ p_m$ |
| Transform Program | $p_t$ | $:=$ | $s_t;\, p_t \ \mid\ \epsilon$ |
| Transform Statement | $s_t$ | $:=$ | $t$ = Filter $(t,\ \phi)$ |
| | | $\mid$ | $t$ = Join $(t_1,\ t_2,\ col_1^{\texttt{Id}},\ col_2^{\texttt{Id}})$ |
| | | $\mid$ | $t$ = GroupJoin $(t,\ col_{index},\ (\alpha,\ col)\ldots)$ |
| | | $\mid$ | $t$ = Order $(t,\ col,\ c_{start},\ c_{inv})$ |
| | | $\mid$ | $t$ = Order $(t,\ col,\ c_{start},\ c_{inv},\ col_{index})$ |
| Aggregation | $\alpha$ | $:=$ | max $\mid$ min $\mid$ sum $\mid$ avg $\mid$ cnt |
| Predicate | $\phi$ | $:=$ | $(\phi_1 \wedge \phi_2)\ \mid\ (\phi_1 \vee \phi_2)\ \mid\ \neg\phi$ |
| | | $\mid$ | $ps\ (col_1, col_2)\ \mid\ ps\ (col, c)$ |
| Mapping Program | $p_m$ | $:=$ | $s_m;\, p_m \ \mid\ \epsilon$ |
| Mapping Statement | $s_m$ | $:=$ | Yield $(c_{action}, t, \rho\ldots)$ |
| Projection | $\rho$ | $:=$ | $col\ \mid\ c\ \mid$ Mutate $(f,\ col\ldots)$ |

Figure 5: Grammar of Bee. $\epsilon$ refers to the empty literal, $f$ refers to a feature, $t$ refers to a table variable, *col* refers to a column name, $c$ refers to a constant value, *ps* refers to a predicate symbol

$p_m$ for mapping table rows to the action signatures. The Filter, Join and GroupJoin transform statements correspond to the where, join and group clauses in SQL, respectively. In addition, the order statement can add columns indicating the orders of the rows.

The mapping statement (Yield) can map either a column or a constant value or a mutation of some columns to the parameter of an action signature. The Mutate operator can generate columns by calculating integral arithmetics and string concatenation on existing columns.

## 7 Design Guidelines

Considering that there could be multiple table schemas for representing the same data, and that the principles in SQL modeling (for persisting data) may not be appropriate in Bee (for program synthesis), here we provide a few guidelines on how to design effective table schemas for Bee.

*Redundancy is not necessarily harmful.* Consider the table for emails in Figure 3. Note that the sender column do not meet the principle to reduce redundancy in SQL. In SQL, considering that a same sender could appear multiple times, we should use a separate table to store the addresses and use keys to connect them to the email table. However, the table nesting the sender column is valid in Bee, since data redundancy usually does not harm to synthesizing programs in Bee. Instead, embedding them in one table would save the step of joining tables in some cases, and thus is more efficient for program synthesis.

*You can represent the same data in multiple schemas.* For example, we use two schemas in our spreadsheet PBE tool built atop Bee: (a) a spreadsheet is divided into cells, where each cell corresponds to one row in the relational table, with columns such as text content, row number and row heading; (b) the original tabular form, where rows in spreadsheet are mapped to rows in relational table and so for columns. Different table schemas could affect the performance of synthesis procedure. For example, if we want to project all the cell texts to one column, schema (b) requires extra transformation because the cell texts exist in different columns, while schema (a) can be used as is. In other cases, schema (b) may outperform schema (a). Bee allows the coexistence of both schemas and would automatically find the one with higher efficiency in the synthesis process, so developers can start with any valid schema, then provide some other schemas to optimize the performance for special scenarios.