Random generation of BIP Models from Architectural Templates

BIN LU
RISD, I&C, EPFL
June, 2014

Abstract

BIP is a component-based language for modeling and programming complex systems. The goal of this project is to develop a tool which can randomly generate BIP models from architectural templates. Concretely, we use a sample BIP file and a template file to randomly generate new models. After reading this report, we will learn how to define a template file and how to use the tool.

1. Introduction

BIP (Behavior, Interaction, Priority) is a component-based language for modeling and programming complex systems.

- Behavior is specified by a set of components.
- Interaction defines the synchronization or data transmission between components.
- Priority is used for resolving conflicts between interactions or for defining interaction schedule policies

Atomic component is the simplest component in BIP model, whose behavior is expressed by automata. Compound component consists of a set of components (atomic or compound), connectors and priorities. With compounds, we can build a hierarchical model.

The goal of this project is to generate BIP models randomly. For two main reasons, we need to achieve random generation. First, testing various verification tools of the BIP framework requires sufficiently large and random models. Second, given an architectural template, we might assume that any model satisfying the template would have the structural properties, which are described in the template. But with sufficiently large number of components, we might discover that such assumption is wrong. However, manually generating large quantity of different models or models with considerable components is not feasible.

To solve the problem, we have developed a tool that can randomly generate certain number of models from architectural templates. Meanwhile, if we set restriction on this process, the randomly generated models will have sufficiently large number of components.

This report is structured as follows. In Section 2, we provide the necessary BIP background. In Section 3, we explain how the architectural templates are specified, based on a sample BIP model and a template file. In Section 4, we describe the structure of the program and how to use the tool. In Section 5, we test the tool on two user cases, which shows how to use the tool concretely. In the last section, we make the conclusion on the project.

2. BIP Background

The following presentation is borrowed from [1]

The BIP framework uses connectors to specify possible interactions between components and priorities to select among possible interactions. Interactions express synchronization constraints between the composed components' activities, and priorities filter possible interactions to steer system evolution toward meeting performance requirements. The combination of interactions and priorities is the source of BIP's expressive power. It defines a clean, abstract concept of architecture separate from behavior.

Atomic components are finite-state automata or Petri nets extended with data and ports. Ports are action names that can be associated with data and used for interactions with other components. States denote control locations where components wait for interactions. A transition is an execution step, labeled by a port, from one control location to another. Each transition has an associated guard and action—respectively a Boolean condition and a function defined on local data. In BIP, complex data and their transformations are written in C/C++.

A transition can be executed if its guard evaluates to true and some interaction involving its port is enabled. The execution is an atomic sequence of two micro steps: first, execution of the interaction involving the port, which is a synchronization between several components with possible data exchange, followed by execution of the action associated with the transition.

Composite components (or Compound Components) are defined by assembling constituent components (atomic or compound) using connectors.

Connectors define relationships between ports of interacting components. They represent sets of interactions—that is, nonempty sets of ports that must be jointly executed. Within a connector, an interaction can occur in two situations: when all involved ports are ready to participate (strong synchronization) or when a port triggers the interaction without waiting for other ports (broadcast). The valid interactions within connectors are formally defined by algebraic expressions on ports using a binary fusion operator and a unary typing operator [2]. Typing associates connector ends (ports or connectors) to synchronization types: trigger (active port, initiates broadcast) or synchron (passive port). Moreover, every connector interaction is associated with a guard and a data transfer function. An

interaction can be executed only when its guard is true. Its execution consists of transferring the data and then, notifying the components involved in the interaction.

Priorities for choosing between simultaneously enabled interactions within a BIP component are defined as rules, each consisting of a pair of interactions associated with a condition. When the condition holds and both interactions of the corresponding pair are enabled, only the one with higher-priority can be executed.

3. Specification of Architectural Templates

The architectural template consists of two parts: a sample BIP model file and a template file. The sample BIP model is manually generated according to the architecture template. The template file defines how to generate the random part.

Specification of the template file is provided in the Appendix.

3.1 Randomly Generated Part

In the project, we assume the model is not hierarchical, which means one model only contains one Compound Type. Also, Compound Type does not contain priorities. Besides Compound Type, Port Type, Atomic Type and Connector Type can also be generated randomly.

Port type has parameters that can be randomly generated--concretely, the number, name or type of parameters.

Atomic Type has five parts which can be randomly generated. They are parameters, variables, ports, states (including initial states), and transitions (including initial transition).

Connector type has four parts which are connected ports, triggers, variables and interactions

We also need to set restrictions on how to generate each part. For example, in the second user case (see 5.2), the number of Clients type components and the number of ClientInterface type components should be equal. To achieve this, we divide a model into several groups. In the same group, every generated part has certain similarity. Parts belonging to different groups are independent. In some cases, one part may belong to multiple groups. For example, in the second user case (see 5.2), the connectors between ClientInterface and ServerInterface belong to two groups.

3.2 Template File

The template file is organized hierarchically. On the same level, there can be several blocks and each block ends with "end". On the first level, we have four types of blocks: Compound Type, Atomic Type, Connector Type, and Port Type.

Specification of the template file is provided in the Appendix.

3.2.1 Compound Type

- "Compound"--it indicates the start of a compound type block. Following "Compound", there should the name for the new compound type. Optionally, the old type can be specified after the string “from”. A "Compound" block have one "RandomGroup" block, several "RandomComponent" blocks and several "RandomConnector" blocks.

  Example: Compound:new_root from old_root
  or
  Compound:new_root

  - "RandomGroup"--It tells how many groups there are in the model and what the unique name for each group is. The group name should be wrapped in "#". Names are separated by comma.

    Example: RnadomGroup:#group1#,#group2#

  - "RandomComponent"--Optionally, it can be followed by a list of integers, indicating which components in the sample BIP model should be modified (the components are counted from 0). All components listed here should have the same pattern. If the compound type is completely generated from template file, the list should be empty. Otherwise, it shouldn't. If the compound type is partly generated from the sample BIP model and some component in the original compound type is not mentioned in any "RandomComponent" block, this component will be copied from the sample model (in other words, "RandomComponent" only need to describe the randomly generated components). A "RandomComponent" block has several attributes: “name”, “group”, “type”, “parameter“

    Example: RandomComponent:0,1,2
    or
    RandomComponent:

- “name”--it must be included. And if “group” attribute is not empty, all the group name should be parts of the name.

  Example: name:s#id0#
  group:#id0#
  or
  name:s#id0#_#id1#
  group:#id0#,#id1#
  or
  name:s
  group:

  In the name, there could be numerical calculation expression, wrapped in “#”.

  Example: name:s#(#id0#+1)%id0#
  group:#id0#

  In this example, id0 (not wrapped in “#”) can be seen as a global variable, which is associated with a random number.

- “group”--it is an optional attribute. It is followed by a list of group names(wrapped in “#”). The names are separated by comma. Actually, each group name is associated with a random number and “group” can be regarded as “for” loop(or nested “for” loop), so that the block can be generated repeatedly according to a pattern.

- “type”--if the atomic type of this component is the same as the original component, “type” is not needed. Otherwise, “type” should be included and followed by the new atomic type name.

  Example: type:new_server

- “parameter”--In one compound type, there will be none or several parameter blocks. If the compound type is completely generated from template file, “parameter” is followed by nothing and each parameter block represents one set of parameters which will be added to the component in the same order as the “parameter” block. Otherwise, “parameter” is followed by a list of integers which indicates the position of the parameters in original components. These parameters can be added to the model according to the same pattern. Parameters which are not covered by any “parameter” block will be copied from original component(in other word, “parameter” only need to describe the modified parameters). Parameter block have two attributes: “group”, “value”

Example: parameter:0
or
parameter:

- “group”--the same as “Compound”-”RandomComponent”-”group”
- “value”--it is an optional attribute.
  It can be integer
  or
  boolean
  or
  numerical calculation expression, wrapped in “#”, which is the same as that in “Compound”-”RandomComponent” -”name”. The expression should be evaluated to integer.

  Example: group:#id0#
  value:#id0#
  or
  group:#id0#
  value:true

- “RandomConnector”--almost the same as "RandomComponent". Except that, in “RandomConnector”-”parameter”, there is no “value” attribute but “atomic” and “port”. And the way to use these two attributes is the same as “Compound”-”RandomComponent”-”name”.

3.2.2 Atomic Type

The basic idea of describing how to randomly generate atomic type is the same as compound type.

- “Atomic”--same as “Compound”.
  - “RandomParameter”--None, one or several such blocks. It is followed by a list of integers or nothing
    - “name”--Must have
    - “group”--Optional
    - “type”--Optional
  - “RandomVariable”--None, one or several such blocks. It is followed by a list of integers or nothing
    - “group”--Optional
    - “name”--Must have

    - “type”--Optional
- “RandomPort”-None, one or several such blocks. It is followed by a list of integers or nothing
    - “group”--Optional
    - “type”--Optional. It is followed by the name of the port type.
    - "isExport”--Optional. It is followed by “true” or “false”.
    - “name”--Must have
    - “variable”--None, one or several such blocks. It is followed by a list of integers or nothing
        - “name”-Must have
        - “group”-Optional
- “RandomInitialState”--Optional. It is followed by the state name
- “RandomState”--None, one or several such blocks. It is and followed by a list of integers or nothing.
    - “group”--Optional
    - “name”--Must have
- “RandomInitialTransition”--None, one or several such blocks. It is followed by nothing.
    - “action”--None, one or several such blocks. It is followed by a list of integers or nothing. The list is the position of actions in the original composite/single action, which needs to be modified.

      NB: In the transition, the action could be single assignment action, single function call action or composite action which includes several assignment actions or function call actions.

      In the project, only assignment action can be randomly generated. If the left side of the assignment action is generated randomly, it could only be some variable. If the right side of the assignment action is generated randomly, it could be some variable, parameter, boolean, integer or numerical calculation expression (evaluated to integer).
        - “type:AssignmentAction”--Must have and Must be at the first line.
        - “group”--Optional.
        - “target”--Optional. It is the left side of the assignment action
        - “value”--Optional. It is the right side of the assignment action

      Example： action:0,1
      type:AssignmentAction
      group:#id1#
      target:server#id1#
      end

- “RandomTransition”--None, one or several such blocks. It is and followed by a list of integers or nothing
    - “group”--Optional
    - “port”--Optional.
    - “sourceState”--Optional. The start state of a transition
    - “targetState”--Optional. The end state of a transition
    - “guard”--Optional.

      NB: Guard is an expression that can be evaluated to a boolean value. In the project, only unary expression and binary expression can be generated randomly. If it is an unary expression and the operand is generated randomly, the operand could be some variable, some parameter, boolean, integer or numerical calculation expression(evaluated to integer). It is same for the left or right operand of a binary expression.

      If guard is a unary expression

        - “type:UnaryExpression”--Must have and Must be at the first line.
        - “operator”--Optional. It can be followed by “!’ or nothing
        - “operand”--Optional.

      example: guard:
      type:UnaryExpression
      operand:server#id1#
      end

      If guard is a binary expression

        - “type:BinaryExpression”--Must have and Must be at the first line
        - “operator”--Optional. It can be followed by “!=”, “==”, “>”, “<”, “>=”, “<=”
        - “left”--Optional.
        - “right”--Optional

    - “action”--Optional. This block is the same as “RandomInitialTransition” -”action”

3.2.3 Connector Type

When using template file to generate a connector type, the description should be complete which means the generated connector type should be correct according to BIP language.

- “Connector”--Optional. It is followed by an name or nothing
  “RandomPort”--At least have one such block. “RandomPort” is followed by nothing.
  - “name”--Must have.
  - “group”--Optional.
  - “type”--Must have. It indicates the type of the port
- “Triggers”--Optional. It is followed by a list of integers. The integer indicates the ports generated by which “RandomPort” block will be the triggers.
- “RandomVariable”--None, one or several such blocks. It is followed by nothing
  - “group”--Optional
  - “name”--Must have
  - “type”--Must have
- “Interaction”--None, one or several such block. It is followed by nothing.
  - “group”--Optional
  - “triggers”--Must have. It is followed by a list of integers or strings. If it is string, it gives the name of the port. If it is an integer, it indicates the ports generated by which “RandomPort” block will be the triggers.
  - “guard”--Optional. It is the same as “RandomTransition”-”guard”
  - “upAction”--Optional. It is the same as “RandomInitialTransition”-”action”
  - “downAction”--Optional. It is the same as “RandomInitialTransition”-”action”

Example:

```
Connector:Singleton
        RandomPort:
                name:p#id1#
                group:#id1#
                type:ThirdType
        end
        RandomPort:
                name:q
                type:ThirdType
        end
        Triggers:
        Interaction:
                triggers:q,0
                upAction:
                        type:AssignmentAction
                        group:#id1#
                        target:p#id1#.x
                        value:q.x
                end
        end
end
```

Result:
```
connector type Singleton(ThirdType p0, ThirdType p1, ThirdType p2, ThirdType q)
        define  p0 p1 p2 q
        on q p0 p1 p2
            up {
              p0.x = q.x;
              p1.x = q.x;
              p2.x = q.x;
             }
 end
```

3.2.4 Port Type

When using template file to generate a port type, the description should be complete, which means the generated port type should be correct according to BIP language.

- "Port"--Optional. It is followed by an name
    - "RandomParameter"--None, one or several such block. It is followed by nothing.
        - "group"--Optional
        - "name"--Must have
        - "type"--Must have

3.3 Other Information

In the template file, we only need to describe the randomly generated part.

If random generated parts have names, such as the name of variables or components, we need to declare them explicitly in the template file.

Since the "parser" for the template file is very weak, any unexpected or redundant character will result in incorrect output.

If one part in the original BIP file is not randomly generated but changed in the template file, the result will be changed. In other word, the template file will be checked first.

The name or group of each part should be consistent with other parts, such that the generated BIP model is correct.

If the atomic type is not randomly generated (the same as that in the sample BIP file), we just generate a new atomic type and copy every part from the original atomic type. Otherwise, we need to parse the template file to get the description for this atomic type, generate the modified parts and copy

unmodified parts from the original atomic type. The connector type is generated the same way. There is one difference between generating atomic type and generating the connector type. Atomic type can be generated completely from original type, partly from original type and partly according to the template file or completely according to the template file. But for connector type, it can only be generated completely from original type or completely from the template file. While generating the atomic type, we also need to add port type to the model. The compound type can be generated in three ways as atomic type. Port type can be generated in two ways as connector type

4. Implementation

4.1 Structure of the Program

The whole program is developed with Java and can be separated into four parts:

- Parse the Sample BIP File -- Parsing the sample BIP file is simply done by calling the function TransformationFunction.ParseBIPFile(String file) in source2source.BIPTransformation package. We get the Compound Type after parsing. Then we get components and connectors from Compound Type and their corresponding types.

- Parse the Template File -- Parsing the template file is done by reading lines in the file, finding the exactly matched string for each part and storing them in corresponding classes. Each multi-lines block should be ended with “end”.

- Assemble the Model -- Before generating the BIP model, each part is assembled using Java classes. It is because different parts are generated differently depending on source files. Manipulating the Java classes is much easier than the BIP model.

- Generate the BIP code -- After getting the necessary information for each component, we will use Meta-model to generate the BIP model. All the transforming functions are in *BIPTranform* class. Part of the functions are borrowed from the project in [3]. But the original functions are dedicated to a particular user case. We need to expand *BIPTranform* class such that the tool can generate BIP models more generally.

  The factories in Meta-model that are used in this project include (The following part is borrowed from [3]):

  - Interactions factory – this is mainly concerned with the Interactions layer and is used to generate interactions, components, connectors and port parameters for the connectors.

  - Actions factory – this is used to generate do{...} code blocks to be executed along transitions. The actions can be simple or composite.

- Behaviours factory – this is used to generate all of the entities that are included in the behavior specification such as atoms, variables, ports, Petri nets and transitions.

- Expression factory – this is used to generate entities that are related to expressions, such as FunctionCallExpressions, literals, data parameter references and binary expressions.

4.2 Run the Program

To run the program, call

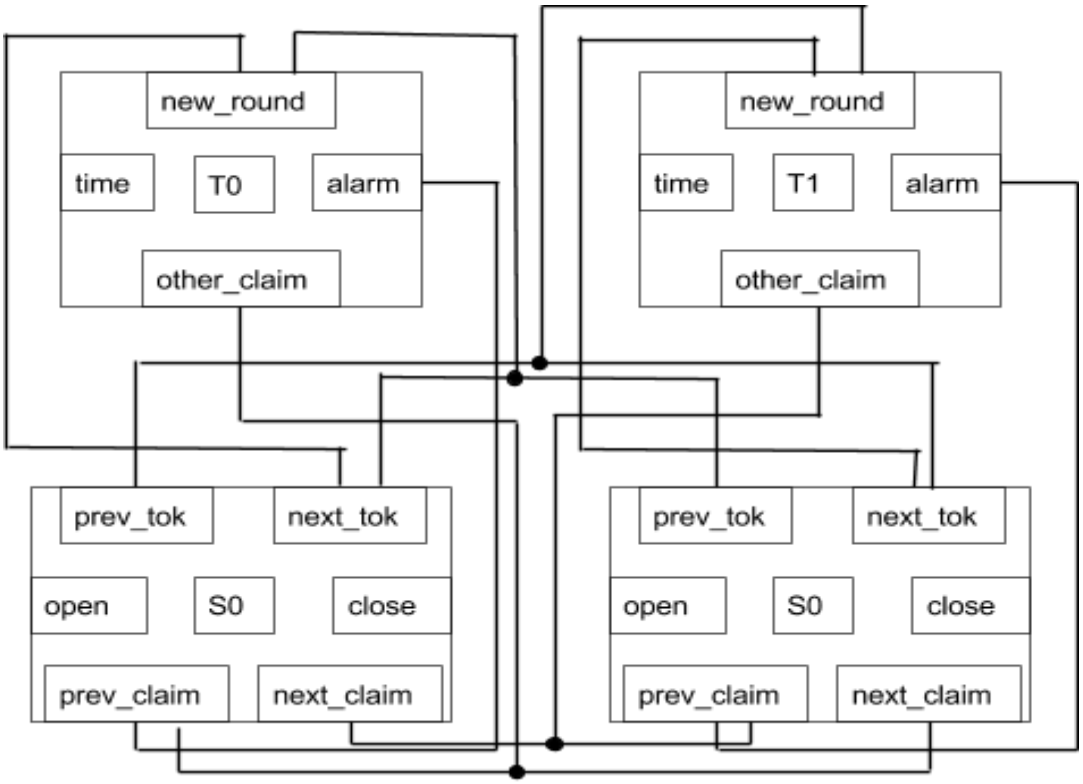
> BIPRandomBuilder.createModels(String sampleFile, String templateFile, String outPutFolder, String root, Integer numOfModels, Integer restriction)

sampleFile: location for the sample BIP file
templateFile: location for the template file
outPutFolder: output folder location for the generated BIP model files
root: the compound type name
numOfModels: specify the number of generated models, which should be larger than 0
restriction: set the least number of components, which should be larger than 0

5. Case studies

5.1 Test Example 1-Election

5.1.1 Architecture of the Model



5.1.2 Template File

In this model, only the compound type can be generated randomly. So, there is only one group #id0#. Components can be categorized into two classes:

- "component Station s**#id0#**(**#id0#**)"
- "component Timer t**#id0#**(**#id0#**, 1)"

Connectors can be categorized into seven classes:

- connector Synchro c**#id0#**(s**#id0#**.next_tok, s#(**#id0#**+1)%id0#.prev_tok, t**#id0#**.new_round)
- connector Loss l**#id0#**(s**#id0#**.next_tok, t**#id0#**.new_round)
- connector ClaimTransmit ct**#id0#**(s**#id0#**.next_claim, s#(**#id0#**+1)%id0#.prev_claim, t**#id0#**.other_claim)
- connector AlarmTransmit al**#id0#**(t**#id0#**.alarm, s**#id0#**.prev_claim)
- connector Singleton sgtime**#id0#**(t**#id0#**.time)
- connector Singleton so**#id0#**(s**#id0#**.open)
- connector Singleton sc**#id0#**(s**#id0#**.close)

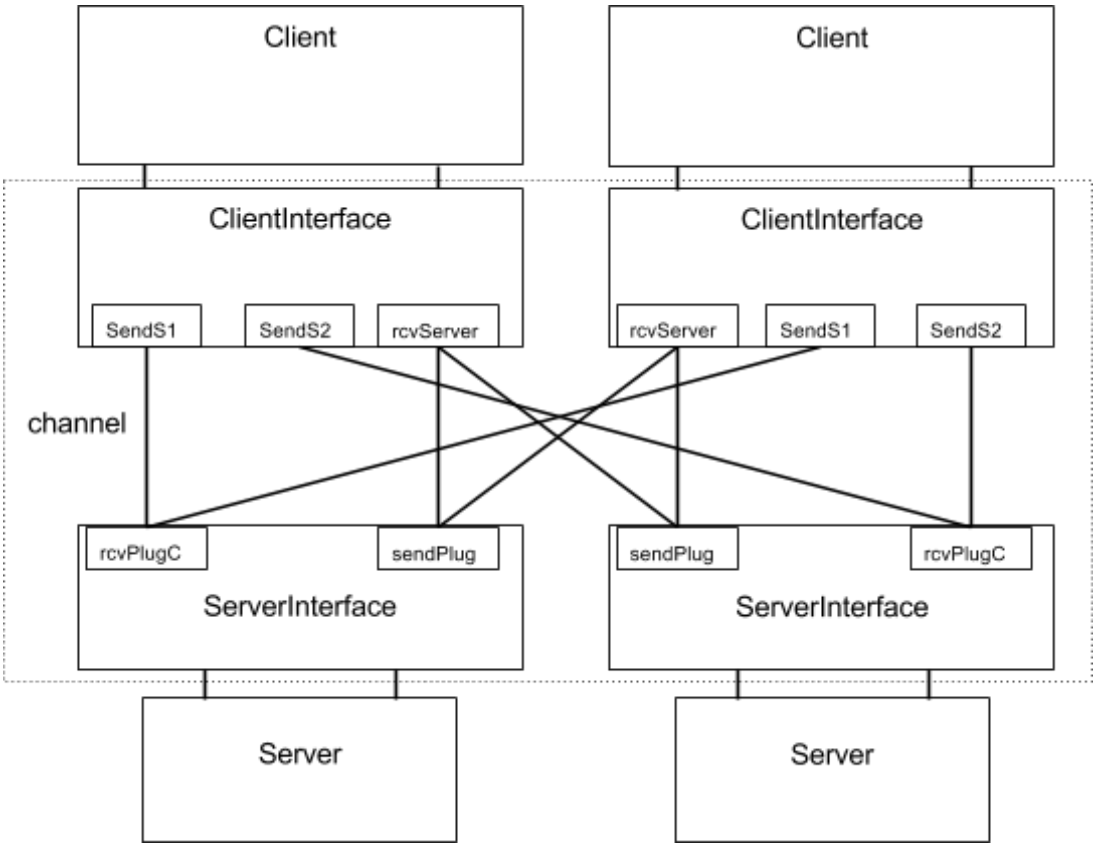
5.1.3 Time Measurement

	10	100	1000
Time/s	8.799	124.261	9759.403(around 162 minutes)

The difference is guaranteed by assigning different random numbers to groups in each iteration. The name of each generated file is associated with those random numbers. At first, we assume that checking whether a model has existed would take most of the processing time, which leads to time growing faster than the number of generated models. But actually, when the number of models gets larger, the components in one model become more, which results generating a single model takes a major time.

5.2 Test Example 2-Quorum protocol

5.2.1 Architecture of the Model



5.2.2 Template File

In Quorum protocol, the number of Clients and the number of Servers can be generated randomly. So there are two groups #id0#, #id1#. The number of ClientInterfaces should be equal to #Client. The number of ServerInterface should be equal to #Server. Also, we need to modify the atomic type for ClientInterface, such that the number of SendS#id1# ports will be equal to the number of servers. The connectors between ClientInterface and ServerInterface belong to group #id0# and group #id1#.

The randomly generated part in compound type Root:

component CustomerPlugQuorum plug**#id0#**(**#id0#**)
component plugConnector plugC**#id0#**(**#id0#**)
component ServerConnector serverC**#id1#**(**#id1#**)
component Server s**#id1#**(**#id1#**)
connector Singleton sendToClient**#id0#**(plug**#id0#**.sendToClient)
connector Singleton switchB**#id0#**(plug**#id0#**.switchB)
connector Singleton ticker**#id0#**(plug**#id0#**.ticker)
connector Singleton loose**#id1#**(serverC**#id1#**.loose)
connector Singleton msglost**#id1#**(serverC**#id1#**.msglost)

connector SendFromPlugToPlugConnector PlugToPlugConn**#id0#**
(plug**#id0#**.sendToChannel,plugC**#id0#**.rcvPlug)

connector SendFromPlugConToServerCon plugConn**#id0#**Toserv**#id1#**Conn
(plugC**#id0#**.SendS**#id1#**,serverC**#id1#**.rcvPlugC)

connector SendFromServerConnToServer serverConnToServer**#id1#**
(serverC**#id1#**.sendServer,s**#id1#**.RcvFromChannel)

connector SendFromServerToServerC server**#id1#**ToserverC**#id1#**
(s**#id1#**.SendToServerC,serverC**#id1#**.rcvServer)

connector ServerCPlugC serverC**#id1#**ToplugC**#id0#**
(serverC**#id1#**.sendPlug,plugC**#id0#**.rcvServer)

connector PlugCToPlug plugC**#id0#**Toplug**#id0#**
(plugC**#id0#**.sendClient,plug**#id0#**.recieveFromServer)

The randomly generated part in atomic type plugConnector:

```
data bool server#id1#
export port ThirdType SendS#id1#(clientId,proposedValue)
initial to start_rcvFServer do {server#id1#=false;clientId=id;}

on SendS#id1# from sendServer_rcvFServer to sendServer_rcvFServer provided(!server#id1#)
do {server#id1#=true;}

on SendS#id1# from sendServer_sendTClient to sendServer_sendTClient provided(!server#id1#)
do {server#id1#=true;}
```

5.2.3 Time Measurement

	10	100
Time/s	12.582	3023.902

6. Conclusion

In the project, we use template file to define how to generate models randomly. An alternative approach is to build the model entirely in Java using the meta-model. In the second approach, we can manipulate each part in BIP model as a Java class, which provides more flexibility for random generation. But the obvious disadvantage is that mapping each part from BIP to Java is a laborious job. Also, errors cannot be prevented while coding in Java. Using template file is relatively easier and error-proof. Though temporarily the tool only works for certain models, for example, randomly generated actions can only be assignment action, it can be easily generalized.

References

[1] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.H. Nguyen and J. Sifakis. Rigorous component-based system design using the BIP framework. IEEE Software 28(3):41-48(2011).
[2] S. Bliudze and J. Sifakis, “Causal Semantics for the Algebra of Connectors,” Formal Methods in System Design, vol. 36, no. 2, 2010, pp. 167–194.
[3]Translation of java BIP Specifications of a Use-Case into BIP Code, Ehab Otman

Appendix

Template File Grammar **:**

<syntax> := {<port_type> “\n”|<atomic_type> “\n”|<connector_type>“\n”|<compound_type> “\n”}

<port_type> := “Port:” <type_name> “\n”
{<port_par> “\n”}
“end”

<port_par> := <tab> “RandomParameter:” “\n”
<name2> “\n”
<type2> “\n”
[<group2> “\n”]
<tab>“end”

<atomic_type> := “Atomic:” <type_name> [“ from “ <type_name>] “\n”
{<atomic_par> ”\n” | <atomic_var> ”\n” | <atomic_port> “\n” | <atomic_state> “\n” | <atomic_initstate> “\n” | <atomic_inittrans> “\n” | <atomic_trans> “\n”}
“end”

<atomic_par> := <tab> “RandomParameter:”[<int>{ “,” <int>}]“\n”
<name2> “\n”
[<type2> “\n”]
[<group2> “\n”]
<tab>“end”

<atomic_var> := <tab> “RandomVariable:”[<int>{ “,” <int>}] “\n”
<name2> “\n”
[<type2> “\n”]
[<group2> “\n”]
<tab>“end”

<atomic_port> := <tab> “RandomPort:”[<int>{ “,” <int>}] “\n”
<name2> “\n”
[<type2> “\n”]
[<group2> “\n”]
[<export> “\n”]
[<atomic_port_var> “\n”]
<tab>“end”

<export> := <tab> <tab>“isExport:”<boolean>

<atomic_port_var> := <tab> <tab> “variable:”[<int>{ “,” <int>}] “\n”
<name3> “\n”
<group3> “\n”
<tab> <tab>”end”

<atomic_state> := <tab> “RandomState:”[<int>{ “,” <int>}] “\n”
<name2> “\n”
[<group2> “\n”]
<tab>“end”

<atomic_initstate> := <tab> “RandomState:” <name> “\n”

<atomic_trans> := <tab> “RandomTransition:”[<int>{ “,” <int>}] “\n”
[<group2> “\n”]
[<atomic_trans_port> “\n”]
[<atomic_trans_startstate> “\n”]
[<atomic_trans_endstate> “\n”]
[<guard> “\n”]
{<action> “\n”}
<tab>“end”

<atomic_trans_port> := <tab> <tab>”port:”<name>
<atomic_trans_startstate> := <tab> <tab>”sourceState:”<name>
<atomic_trans_endstate> := <tab> <tab>”targetState:”<name>
<guard> := <tab> <tab>”guard:” “\n”
((<tab> <tab> <tab>”type:UnaryExpression” “\n”
[<u_operator> “\n”]
[<operand> “\n”]) |
(<tab> <tab> <tab>”type:BinaryExpression” “\n”

```
                [<b_operator> “\n”]
                [<left_operand> “\n”]
                [<right_operand> “\n”]))
                <tab> <tab>”end”
<u_operator>    :=      <tab> <tab> <tab> “operator:” (“” | “!”)
<operand>       :=      <tab> <tab> <tab> “operand:”(<int> |<boolean> | <expression> | <name>)
<b_operator>    :=      <tab> <tab> <tab> “operator:”(“==” | “!=” | “>” | “>=” | “<” | “<=”)
<left_operand>:=        <tab> <tab> <tab> “left:”(<int> |<boolean> | <expression> | <name>)
<right_operand>:=       <tab> <tab> <tab> “right:”(<int> |<boolean> | <expression> | <name>)
<action> :=     <tab> <tab>”action:”[<int>{ “,” <int>}] “\n”
                <tab> <tab> <tab>”type:AssignmentAction” “\n”
                [<group3> “\n”]
                [<target> “\n”]
                [<value> “\n”]
                <tab> <tab>”end”
<target> :=     <tab> <tab> <tab> “target:” <name>
<value> :=      <tab> <tab> <tab> “value:” (<int> |<boolean> | <expression> | <name>)

<atomic_inittrans> :=   <tab> “RandomInitialTransition:” “\n”
                        {<action> “\n”}

<connector_type> :=     “Atomic:” <type_name> “\n”
                        <connector_port> ”\n”{<connector_port> ”\n” | <connector_var> “\n” |
                        <connector_trigger> ”\n” | <connector_interaction> “\n”}
                        “end”
<connector_port> :=     <tab> “RandomPort:” “\n”
                        <name2> “\n”
                        <type2> “\n”
                        [<group2> “\n”]
                        <tab>“end”
<connector_var> :=      <tab> “RandomVariable:” “\n”
                        <name2> “\n”
                        <type2> “\n”
                        [ <group2> “\n”]
                        <tab>“end”

<connector_trigger>:=   <tab> “Triggers:”[<int>{ “,” <int>}] “\n”
<connector_interaction>:=       <tab>”Interaction:”“\n”
                                [ <group2> “\n”]
                                <connector_interaction_trigger> “\n”
                                [<connector_interaction_guard> “\n”]
                                {<up_action> “\n”}
```

{<down_action> “\n”}
<tab>“end”

<connector_interaction_trigger>:= <tab><tab>”triggers:”[(<int>|<name>){ “,” (<int>|<name>)}]

<connector_interaction_guard> := <tab> <tab>”guard:” “\n”
((<tab> <tab> <tab>”type:UnaryExpression” “\n”
<u_operator> “\n”
<operand> “\n”) |
(<tab> <tab> <tab>”type:BinaryExpression” “\n”
<b_operator> “\n”
<left_operand> “\n”
<right_operand> “\n”))
<tab> <tab>”end”

<up_action> := <tab> <tab>”upAction:” “\n”
<tab> <tab> <tab>”type:AssignmentAction” “\n”
[<group3> “\n”]
<target> “\n”
<value> “\n”
<tab> <tab>”end”
<down_action> := <tab> <tab>”downAction:” “\n”
<tab> <tab> <tab>”type:AssignmentAction” “\n”
[<group3> “\n”]
<target> “\n”
<value> “\n”
<tab> <tab>”end”

<compound_type> := “Compound:” <type_name> [“ from “ <type_name>] “\n”
<compound_group> ”\n” {<compound_component> ”\n” |
<compound_connector> “\n”}
“end”

<compound_group> := <tab>”RamdonGroup:”<group>”\n”
<compound_component> := <tab> “RandomComponent:”[<int>{ “,” <int>}] “\n”
<name2>”\n”
[<type2> “\n”]
[<group2> “\n”]
{<compound_component_par> “\n”}
<tab>“end”
<compound_component_par>:= <tab><tab> “parameter:”[<int>{ “,” <int>}]“\n”
[<group3>“\n”]

[<tab><tab><tab>”value:”(<int>|<boolean>|<expression>)”\n”]
<tab><tab> “end”

<compound_connector> := <tab> “RandomConnector:”[<int>{ “,” <int>}] “\n”
<name2>”\n”
[<type2> “\n”]
[<group2> “\n”]
{<compound_connector_par> “\n”}
<tab>“end”

<compound_connector_par>:= <tab><tab> “parameter:”[<int>{ “,” <int>}]“\n”
[<group3>“\n”]
[<tab><tab><tab>”atomic:”(<int>|<boolean>|<expression>)”\n”]
[<tab><tab><tab>”port:”(<int>|<boolean>|<expression>)”\n”]
<tab><tab> “end”

<type_name> := {<character>}
<name> := {<character>|(“#”<expression>”#”)}
<name2> := <tab> <tab>”name:”<name>
<name3> := <tab> <tab> <tab>”name:”<name>
<type2> := <tab> <tab>
<group_name>:= {<character>}
<group> := [“#”<group_name>”#”{“,” “#”<group_name>”#”}]
<group2> := <tab> <tab>“group:”<group>
<group3> := <tab> <tab> <tab>“group:”<group>
<tab> := “ “
<int> := {<digit>}
<boolean> := “true” | “false”
<expression> := < factor > { ("+" | ”-”| “*” | “/” | “^” | ”%”)<factor > }
factor := <group_name> |
(“#” <group_name> “#”) |
<int> |
(“(“ <expression> “)”)

<letter> = "A" | "B" | "C" | "D" | "E" | "F" | "G” | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | “a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o “ | “ p” | “q” | “r” | ”s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”
<digit> = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<character> = <letter> | <digit> | “_” | “-” | “.”