ABSTRACT
Dependencies among software input parameters make test case generation difficult. One possible solution is using an input space model, which allows the generation of all valid (satisfied all dependencies) and only valid test combinations. For large number of dependencies, automatic creation of the model is necessary. For this purpose, the paper presents MIST – a new tool for Modeling Input Space for Testing. MIST implements a new algorithm based on subgraph splitting and creates a software input space graphical model, where paths through the graph represent sets of valid test cases. The paper considers MIST design, implementation, and usage issues. MIST inputs and outputs are illustrated for a case study of a fragment of a nuclear power plant control system. Integration of MIST with other tools, in particular, the JUMBL tool, is addressed. This allows automatic test case generation for statistical testing.

KEY WORDS
Software testing, software tools, input space, dependencies among parameters.

1. Introduction

Testing software programs with a large number of input parameters and/or large number of possible parameter values requires special approaches. When parameters are independent, various combinatorial methods are often used [1, 2]. However, when there are dependencies among input parameters, it can be difficult to use many of these methods. Some approaches to this situation have been considered [3] but none of them provides a complete solution, which is satisfactory from the practical point of view.

A dependency between two parameters can be usually described in “if-then” statements. For example, software for booking air tickets from New-York airports can have the parameters DESTINATION with values \{Chicago, London, Los-Angeles, Paris, Toronto\} and FLIGHT_TYPE with values \{International, Domestic\}. Then these two parameters have the following dependency: if FLIGHT_TYPE=International then DESTINATION∈\{London, Paris, Toronto\}; if FLIGHT_TYPE=Domestic then DESTINATION∈\{Chicago, Los-Angeles\}. In other words, existence of a dependency means that some combinations of input parameters are invalid, for example, the combination FLIGHT_TYPE=International and DESTINATION=Chicago.

For complex software systems, one parameter can participate in several dependencies at the same time, and can appear in the “if” part of some dependencies and in the “then” part of others. Combinations of various dependencies can be quite complicated and make test case generation difficult. One possible solution is to use an input space model, which allows the generation of all valid (satisfied all dependencies) and only valid test combinations. An algorithm of creation of such a graph model has been recently suggested by the first author of this paper [4, 5].

When there are only a few dependencies, the algorithm can be implemented manually. However, for large numbers of dependencies, automatic creation of the model is necessary. For this purpose, the paper presents MIST – a new tool for Modeling Input Space for Testing. The paper is organized as follows: In section 2, we briefly review the algorithm of model creation and then consider the design, implementation, and usage of the MIST tool. Section 3 addresses a case study of a MIST application for modeling a fragment of the input space of a nuclear power plant control system. Integration of the MIST tool with existing tools for the test case generation is considered in Section 4. Section 5 provides conclusions and directions for further work.

2. The MIST Tool

2.1 Purposes of the MIST Tool

The MIST tool is designed for automatically creating a graphical model of software input space and using this model for test case generation. The MIST input information contains a list of software parameters with their values and descriptions of dependencies among the
parameters in the “if-then” terms. An output of MIST is a
direct graph, which has two special nodes, “ENTER” and
“EXIT”. All other nodes are associated with software
parameters, and edges are associated with parameter
values. In the case of dependencies among parameters,
several nodes can be associated with one parameter. A
simple example of the model with three parameters, x, y,
z, and one dependency between x and z is shown in Fig.
1. Here x has values \{a, b, c\}, y has values \{5, 10\}, and
the values of z are \{50 – 100\}, while the dependency is
the following: if x=a then z \in \{50 – 75\}; if x \in \{b, c\}
then z \in \{76 – 100\}.

\[
\text{Figure 1. Dependency between x and z}
\]

Each path through the graph represents a group of valid
test cases, i.e., combinations of parameter values, which
satisfy all dependencies. All valid combinations are
covered by paths, and no paths exist for any invalid
combination. The graph has a Markov property [6]: a path
from any node to “EXIT” does not depend on how this
node was reached. This property enables using MIST
together with existing tools, which generate test cases
from Markov chain models.

2.2 Algorithm

For input space modeling, MIST implements a new
algorithm based on subgraph splitting [4, 5]. The
algorithm starts from a simple linear graph, which reflects
input parameters and their values without dependencies.
Then the algorithm repeats the same cycle for each
dependency, adding new nodes into the graph to reflect
the dependency. The cycle includes three steps:

- Splitting the subgraph
- Annotating input and output edges of subgraphs
- Eliminating dead nodes and edges

For dependency between parameters, x and y, at the first
step, the algorithm splits the subgraph, which includes all
nodes for parameters between x and y, including x but
excluding y. The number of new subgraphs equals the
number of “if” statements in the description of the
dependency between x and y.

At the second step, new annotations (parameter values)
are calculated for the input and output edges of the
subgraphs. A new annotation is an intersection of the old
annotation, and the set from the corresponding “if” part
(for input edges) or “then” part (for output edges) of the
dependency. For some edges, this intersection can be
empty. These edges are considered as “dead” and should
be eliminated at the third step of the algorithm cycle.
After elimination, some nodes can remain without input
or output edges. Such nodes are also considered as dead
and should be eliminated.

MIST only works directly with dependencies between
two parameters (“one-on-one” dependencies). Other
possible types of dependencies are “one-on-many,”
“many-on-one,” and “many-on-many.” An approach on
how to reduce these types to the “one-on-one” type using
“derived parameter” is considered in [7]. It allows
applying MIST for dependencies among many
parameters.

2.3 Design and Implementation

The implementation of the MIST tool has been done in
Java. NetBeans has been used as an Integrated
Development Environment (IDE) for the entire
development process. The operating cycle of MIST is
illustrated in Fig. 2. First, the parameter values and
dependencies between the parameters are entered into the
tool. Then MIST executes a cycle for each dependency:
splits a subgraph, annotates input edges, eliminates dead
nodes and input edges, annotates output edges, and
eliminates dead nodes and output edges.

MIST then generates two forms of outputs, which
describe the model in graphical and text formats.
Graphical format helps in better visualization of the
model, whereas text format is helpful in integrating MIST
with external tools for test case generation. In particular,
the text format uses TML (The Modeling Language [8]),
which is an input language for the JUMBL (Java Usage
Model Builder) tool [9]. JUMBL was developed in the
Software Quality Research Laboratory at the University
of Tennessee, and is used for testing complex systems
based on Markov chain usage models [10]. This is
especially useful when a large amount of test cases needs
to be generated for statistical testing [11]. Integration of
MIST with JUMBL is described in Section 4.

MIST supports two types of input values: enumerations
(separate values for string and numerical values) and
intervals for integer values. MIST has been developed as
thirteen Java classes. The class diagram is shown in Fig.
3. The main classes of MIST are:

- ContinuousValues.java – deals with the range of
inputs entered. It supports intervals as values.
(Base generic class of parameter values).
- DateProvider.java – contains parameters and the
dependencies among them.
• Dependency.java – manages all the dependency-based operations, such as splitting a sub-graph, annotating input and output edges, and elimination of dead nodes and edges. This class handles the generation of TML form of output.
• DependencyGraph.java – constructs a graphical form of output.
• EnumValues.java – handles the inputs with values of an enumeration type.
• IntegerValues.java – handles the inputs consisting of numerical values.

2.4 MIST Usage

To start working with MIST, the user needs to enter information about the parameters and their dependencies. To enter the input information, the user fills in two tables: parameters (their names and values) and dependencies (correspondences between parameter values). Fig. 4 illustrates filling in the table for parameters. MIST prompts the user to enter the parameter type, followed by the name and values. For parameters of the integer type, values can be entered as an interval, for example, 1 – 100. For all other types of parameters (real numbers, strings, characters), values should be entered as a list of separate values.
Once the input information is entered, the user can run MIST. No more interactions are necessary. MIST output information in TML format and in the graphical format is illustrated in Section 3 for the case study example.

### 3. Case Study

Consider using MIST for a fragment of a nuclear power plant control system. The manual step-by-step application of the subgraph splitting algorithm for this system was considered in [4], so we address only MIST inputs and outputs here. The system has six input parameters: three technological parameters and three warning signals.

The technological parameters are reactor capacity ($30\% \leq N \leq 100\%$), pressure in the first circuit ($50 \text{ kg/cm}^2 \leq P \leq 300 \text{ kg/cm}^2$), and coolant temperature ($200^\circ C \leq T \leq 400^\circ C$). Warning signals are low pressure (WP - less than 145 kg/cm$^2$), high temperature (WT - more than 270$^\circ C$), and high reactor capacity (WN - equal or more than 50%). The values of all warning signals are either ON or OFF.

There are five dependencies among these parameters. Two dependencies arise because only the most safety important situations are considered, namely the simultaneous decrease of the pressure and increase of the temperature. Dependencies reflect two sets of operational limits depending on the reactor capacity:

- if $N \geq 75$ then $P \leq 150$; if $N < 75$ then $P \leq 140$.
- if $N \geq 75$ then $T \geq 260$; if $N < 75$ then $T \geq 280$.

Other tree dependencies reflect the nature of the warning signals:

- if $P \leq 145$ then WP = ON; if $P > 150$ then WP = OFF.
- if $T \geq 270$ then WT = ON; if $T < 270$ then WT = OFF.
- if $N \geq 50$ then WN = ON; if $N < 50$ then WN = OFF.

Fig. 6 shows the MIST inputs (tables with information about parameters and dependencies).
If a user would like to have separate models for each dependency, it is necessary to run MIST immediately after the information on the dependency is entered. Running MIST when the tables in Fig. 6 are completed gives the final model, which reflects all five dependencies. Fig. 7 shows the MIST outputs for the model in TML format and Fig. 8 shows them for the graphical format.
4. Integration with Other Tools

For many situations, software parameters possess some values more frequently than other ones. For example, the parameter FLIGHT_TYPE (mentioned in Section 1) could take the value Domestic in 70% of cases and the value International in 30%. Our model can be easily extended for such situations, adding probabilities for each arc of the graph. When probabilities are not shown directly (as in the case study in Section 3), we assume that all values are equiprobable. In both cases, when probabilities are and are not present, the model produced by MIST can be considered as a special acyclic type of a Markov chain. It makes it possible to integrate MIST with any existing tool, which uses Markov chains, for test case generation.

In particular, MIST includes an interface with JUMBL [9], a tool for analyzing, generating, and executing test cases from Markov chain models. The output file of MIST in TML format can be directly used as input data for JUMBL. Creating an input model is the most complicated, and often the manually realized part of working with JUMBL. In our case, MIST does it automatically. In this sense, MIST and JUMBL compliment each other and can be used as a joint system for modelling and test case generation from an input space with dependencies among parameters.

MIST users can choose the main options of JUMBL directly from the MIST menu (Fig. 9). The full performance capabilities of JUMBL are available via a command line.

Figs. 10 to 13 provide examples of the JUMBL outputs for the case study model from Section 3.
Via the MIST menu, the user has access to the JUMBL results of a model analysis, including statistics for the model as a whole (Fig. 10), and detailed statistics related to each node, arc, and stimulus (Fig. 11 for node statistics). This includes, for example, node and arc occupancy (relative frequency), probabilities that nodes and arcs will occur in a particular test case, and so on. JUMBL results for the test record analysis are also available. Fig. 12 shows a fragment from the arc statistics and Fig. 13 gives a test case generated by JUMBL.

5. Conclusion

MIST is a new tool for modeling software input space with dependencies among input parameters. To use MIST, a user just needs to provide information on values of software input parameters and describe dependencies among parameters in a simple tabular form. MIST automatically creates a model of the input space in two formats: graphical and TML text format. Both formats can be used for manual or automatic test case generation.

The graphical format provides visualization of all feasible combinations of input values. Each path through the graph corresponds with the set of valid test cases, and all valid test cases are covered. TML format is a simple text description of the same graph and is useful for integration of MIST with other tools for automatic test case generation. In particular, MIST provides an interface for working with the JUMBL tool (for which TML is an input language), and so can be used for statistical testing.
The advantage of MIST is that the tool automates the most sophisticated part of software test case generation and turns the other parts of this process into routine tasks. This paper is the first presentation of MIST. Though MIST has been successfully used for several case studies, our plans include further practical approbation and improvement of the tool. One of the possible extensions is managing different types of dependencies, including those for many parameters.

References