# SOFL Specification Animation with Tool Support

Mo Li[1(✉)] and Shaoying Liu[2]

[1] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
mo.li.3e@stu.hosei.ac.jp
[2] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

**Abstract.** Formal specification animation is a very useful technique for verification and validation. It provides the end users and field experts with an intuitive way to observe the operational behaviour of software system described by the formal specification. Several tools have already been built to support animations of specifications written in different formal languages. In this paper, we describe the design of a tool that can support the animation of specification written in Structure Object-oriented Formal Language (SOFL). The animation strategy underlying the tool uses system functional scenario as a unit and data as connection among independent operations involved in one system functional scenario. A system functional scenario defines a behaviour that transforms input data into the output data through a sequential execution of operations. It is the target of animation. In the animation, data is used to connect each operation in a specific scenario. The data can be provided by the user or generated automatically. It will help the user and the developer to understand the system. We explain the whole animation process step by step and present a prototype at the end of the paper.

**Keywords:** Formal method · Specification · Animation · Tool

## 1 Introduction

Formal specification animation is an effective technique for specifications verification and validation. The purpose of animation is to provide an intuitive way for the end user and domain expert to monitor the states of a behaviour so that they do not need to read the formal specification which is fulfilled with mathematical or logical formulas. Usually, a tool is required to support the animation

This work has been conducted as a part of "Research Initiative on Advanced Software Engineering in 2012" supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA).

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2013, LNCS 8332, pp. 118–131, 2014.
DOI: 10.1007/978-3-319-04915-1_9, 

process since it faces a lot of challenges in practice and one of the challenge is that formal specification is complex and difficult to understand. A tool is needed to hide the complexity and derive the intuitive information for the user. Several tools have been built to support animation of specification written in different formal languages, such as ANGOR [1], B-Model animator [2], and ZAL animation system [3]. Most of them require a translation from a formal specification language to an executable programming language, so that the animation can be done automatically. But the translation may imposes many restrictions to the style of the specifications written in a formal notation and this would bring inconvenience to the developer.

In this paper, we describe the design of a tool for SOFL [4] specification animation. There is no further restrictions to the language or style of specification. In the tool, the end user or field expert can animate the specification on the Conditional Data Flow Diagram (CDFD, a part of SOFL specification) directly, and we think it is a very good way to connect user, developer and formal specification. The animation strategy underlying the tool uses system functional scenario as a unit and data as connection among independent operations involved in one system functional scenario. System functional scenario is used as unit of an animation since it presents a specific behaviour of the system, and the behaviour is the target or object of animation. In order to animate the entire specification, all of the potential behaviours, or system functional scenarios, should be animated. Usually, a system functional scenario is defined as a sequence of operations that process a group of input data to a group of output data. Each operation in scenario is connected by intermediate data, and it is the reason why we use data as connection in animation.

One of the advantages of using data for animation is that the end user can observe each behaviour by monitoring the states of it. The states of behaviours are presented as the input and output data of each operation involved. The tool allows users to select providing the data themselves or letting the data generated automatically. If users select to provide the data for animation, they usually provide the most typical data of the system. Meanwhile, the users need to guarantee that the data they provide satisfies the pre- and post-conditions. If users want to let the data generated automatically, the data generation method would generate the data that satisfies the pre- and post-conditions. But the generated data may not present the specific circumstance users want to animate. We will introduce how to animate a scenario step by step in Sects. 2 and 3.

The prototype is implemented on the bases of a framework that is built to help developers specifying SOFL specification. The framework includes the editors of informal specification, formal specification, and CDFD, etc. All the specifications are well organized and stored in the well formatted files under this framework. It provides a fundamental to further utilization of the formal specifications. The prototype is now a part of the framework, and it contains functions of deriving system functional scenario and animating a specific scenario. Section 4 describes the framework and prototype in details.

The remainder of this paper is organized as fellows. Section 2 describes the animation strategy and the animation process. Section 3 briefly introduces the specific methods of system functional scenarios derivation and data generation used in the tool. The framework for specifying SOFL specifications and the prototype is described in Sect. 4. Section 5 gives a brief overview of related work. Finally, in Sect. 6 we conclude the paper and point out future research directions.

## 2 Animation Strategy and Process

As mentioned previously, we use the system functional scenario as the basic unit in animation. Each system scenario presents a specific function of the system. Since the goal of a formal specification animation is to validate the potential behaviours of the specification, we suggest that every possible system functional scenario defined in the specification should be animated. A system scenario defines a specific kind of operational behaviour of the system through a sequential executions of operations. And a specific operational behaviour is usually presented to end users as a pair of input and output, that is, given an input, the result of a behaviour of the system results in a certain output. The definition of a system functional scenario is detailed in Definition 1.

**Definition 1.** *A **system functional scenario**, or **system scenario** for short, of a specification is a sequence of operations $d_i[OP_1, OP_2, ..., OP_n]d_o$, where $d_i$ is the set of input variables of the behaviour, $d_o$ is the set of output variables, and each $OP_i (i \in \{1, 2, ..., n\})$ is an operation.*

The system scenario $d_i[OP_1, OP_2, ..., OP_n]d_o$ defines a behaviour that transforms input data item $d_i$ into the output data item $d_o$ through a sequential execution of operations $OP_1, OP_2, ..., OP_n$. Actually, other data items are used or produced within the process of executing the entire system scenario but not being presented. For example, the first operation $OP_1$ in the system scenario receives the input data item $d_i$ and produces a data item, which is the input data item of operation $OP_2$. Operation $OP_2$ cannot be executed without the output data item of $OP_1$. We call these data items *implicit data items*. In order to show the behaviour of system step by step in an animation, the implicit data items in system scenario should be presented explicitly. When presenting implicit data items explicitly is necessary, we use $[d_i, OP_1, d_1, OP_2, d_2, ..., d_{n-1}, OP_n, d_o]$ to present a system scenario, where $d_1$ indicates the output data item of $OP_1$ or input data item of $OP_2$.

For example, Fig. 1 shows the CDFD of a simplified ATM. There are only to functions included in this simple specification: *withdraw* and *show balance*. Based on the definition of system functional scenario, five scenarios defined in the specification is listed as follows:

- {*withdraw_com*}*[Receive_Command, Check_Password, Withdraw]*{*cash*}
- {*withdraw_com*}*[Receive_Command, Check_Password, Withdraw]*{*err2*}
- {*withdraw_com*}*[Receive_Command, Check_Password]*{*err1*}

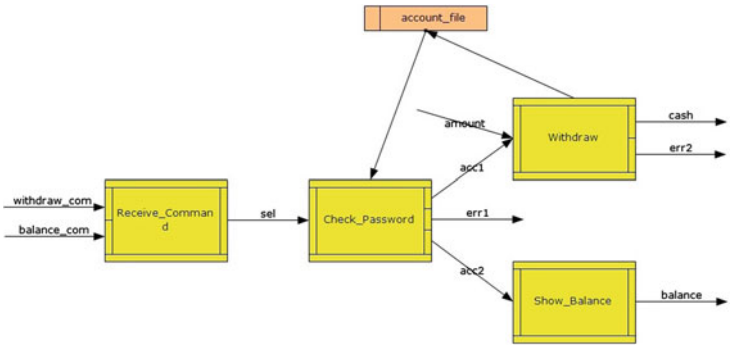


**Fig. 1.** CDFD of a simple ATM

– $\{balance\_com\}[Receive\_Command,\ Check\_Password]\{err1\}$
– $\{balance\_com\}[Receive\_Command,\ Check\_Password,\ Show\_Balance]\{balance\}$

Such a system functional scenario clearly describes how the final output data is produced by a sequence of processes based on the input data. Note that the input data and output data indicate the sets of variables when talking about the concept of system scenario or a specific system scenario of an behaviour. Under the context of animation, the input and output data are instances rather than concepts. They indicate sets of values of the input and output variables.

To animate a specific system scenario, the data is used to connect each operation involved in the scenario. Since the data is restricted by the pre- and post-conditions of process, the data present a real environment of the behaviour. The user and experts can observe the behaviour by monitoring the data. There are two ways to collect the data for animation. The first way is to let the user providing the data. The advantage of this way is that the data provided by user is usually the most typical data. The provided data presents the circumstance that the user cares about. Meanwhile, the user have to guarantee that the data satisfies the pre- and post-condition, otherwise the data would be meaningless. The alternative way of collecting data is to generate data automatically. The generation method does not require translating the formal specification to any executable program. One of the obvious disadvantage of the method is that the generated data may not present the specific circumstance users want to animate. No matter which way the user use, the data provide a concrete point of views of the behaviour.

When collecting input and output data for a single process, the operation functional scenarios of the process have to be extracted first. By operation functional scenario, we mean an predicate expression derived from the pre- and post-condition of a process, which precisely defines the relation of a set of input and output data. Liu first gives a formal definition of operation functional scenario [5] and we repeat it here to help the reader understand the rest of this paper.

```
process Check_Password(id: nat0, sel: bool, pass: nat0)
                acc1: Account | err1: string | acc2: Account
ext     rd Account_file
pre     true
post    (exists![x: Account_file] | ((x.id = id and
                                      x.password = pass) and
        (sel = true and acc1 = x or
        sel = false and acc2 = x)))
        or
        not(exists![x: Account_file] | (x.id = id and
              x.password = pass)) and
              err1 = "Reenter your password or insert
                    the correct card"
end_process
```

**Fig. 2.** Specification of process "Check_Password"

**Definition 2.** *Let* $OP(OP_{iv}, OP_{ov})[OP_{pre}, OP_{post}]$ *denote the formal specification of an operation* $OP$*, where* $OP_{iv}$ *is the set of all input variables whose values are not changed by the operation,* $OP_{ov}$ *is the set of all output variables whose values are produced or updated by the operation, and* $OP_{pre}$ *and* $OP_{post}$ *are the pre and post-condition of operation* $OP$*, respectively.*

**Definition 3.** *Let* $OP_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee ... \vee (C_n \wedge D_n)$*, where each* $C_i (i \in \{1, ..., n\})$ *is a predicate called a "**guard condition**" that contains no output variable in* $OP_{ov}$ *and* $\forall_{i,j \in \{1,...,n\}} \cdot i \neq j \Rightarrow C_i \wedge C_j = false$ *;* $D_i$ *a "**defining condition**" that contains at least one output variable in* $OP_{ov}$ *but no guard condition. Then, a formal specification of an operation can be expressed as a disjunction* $(\sim OP_{pre} \wedge C_1 \wedge D_1) \vee (\sim OP_{pre} \wedge C_2 \wedge D_2) \vee ... \vee (\sim OP_{pre} \wedge C_n \wedge D_n)$*. A conjunction* $\sim OP_{pre} \wedge C_i \wedge D_i$ *is called an **operation functional scenario**, or **operation scenario** for short.*

Note that we use $\sim x$ and $x$ to represent the initial value before the operation and the final value after the operation of external variable $x$, respectively. The decorated pre-condition $\sim OP_{pre} = OP_{pre}[\sim x/x]$ denotes the predicate resulting from substituting the initial state $\sim x$ for the final state $x$ in pre-condition $OP_{pre}$. We treat a conjunction $\sim OP_{pre} \wedge C_i \wedge D_i$ as an operation functional scenario because it defines an independent behaviour: when $\sim OP_{pre} \wedge C_i$ is satisfied by the initial state (or intuitively by the input data), the final state (or the output data) is defined by the defining condition $D_i$.

The reason why we need the operation functional scenario is that the definition of SOFL allows the process to receive more than one exclusive input or output data. In a specific system functional scenario, only one pair of input and output data of each process is involved. For instance, the process "Check_Passwork" of the simple ATM system in Fig. 1 is formally defined in the specification shown in Fig. 2. There are three operation functional scenario contained in this specification.

1. $(\exists x \in Account\_file \cdot x.id = id \wedge x.password = pass) \wedge sel = true \wedge acc1 = x$
2. $(\exists x \in Account\_file \cdot x.id = id \wedge x.password = pass) \wedge sel = false \wedge acc2 = x$
3. $\neg(\exists x \in Account\_file \cdot x.id = id \wedge x.password = pass) \wedge err1 =$ *"Reenter your password or insert the correct card"*

If the system functional scenario {balance_com}[Receive_Command, Check_Password, Show_Balance]{balance} is under animation, there is no doubt that the second operation functional scenario should be selected to collect input and output data for the process "Check_Passwork".

After the introduction of our animation strategy, we would describe how to apply the strategy in practice. For a given formal specification, the following stages supply a procedure for systematically performing the animation.

**Stage 1.** *Derive all possible* ***system scenarios*** *from the formal specification.*

Different methods are used to derived system functional scenario from formal specification written in different formal languages. For example, for a formal specification language containing graphic specification, the system scenarios can be derived based on the topology of the graph. On the other hand, for a formal specification language that does not contain graphic specification, the system scenarios can be derived based on the data dependency among operations. For SOFL specification, we derive system functional scenarios from the topology of the CDFD. We will introduce the specific automatic approach for deriving all possible system scenarios in next section.

**Stage 2.** *Let* $d_i[P_1, P_2, ..., P_n]d_o$ *be a selected* ***system scenario****. Derive related* ***operation scenarios*** *of each* $P_i (i \in \{1, 2, ..., n\})$ *from its specification and get a set of* ***operation scenarios*** $\{OS_1, OS_2, ..., OS_n\}$*, where* $OS_i$ *is the related* ***operation scenario*** *of* $P_i$*.*

According to our animation strategy, only one system scenario should be selected to animate each time. For any selected system scenario, the related operation scenario of each operation involved should be derived from the formal specification. Since an operation scenario of an operation defines an independent relation between its input and output under a certain condition, only one operation scenario of an operation can be involved in the selected system scenario. Therefore, the second stage of entire animation process should be deriving related operation scenarios.

As the start point of a system functional scenario, the input data of the first process in the system scenario should be collected first. Actually, the input data of the first process is the only input that need to be collected from the user or generated by data generation method. The input data of the following processes is the output data of the previous processes.

**Stage 3.** *Let* $\sim S^1_{pre} \wedge C^1_i \wedge D^1_i$ *be the related operation scenario of the first operation* $P_1$ *in the selected system scenario. The input data should be collected and satisfy the predicate expression* $\sim S^1_{pre} \wedge C^1_i$*.*

The input data generated in Stage 3 is actually the input of the selected system scenario. It can be used as bases to collect output data of $P_1$. The output data collected should satisfy the predicate expression $\sim S^{1*}_{pre} \wedge C^{1*}_i \wedge D^1_i$, which can be created by applying the input data to predicate expression $\sim S^1_{pre} \wedge C^1_i \wedge D^1_i$. The output data of $P_1$ then be used as input data of $P_2$ to collect output data of $P_2$. Repeating this procedures, the output data of entire system scenario can be generated eventually. This idea is reflected in Stage 4.

**Stage 4.** *Use the input data generated in Stage 3 and the operation scenarios derived in Stage 2 to generate the output data for each operation and entire system scenario.*

So far, all of the data involved in a selected system scenario has been generated. And the behaviour can be simulated by using the data involved in it. The end users and field experts can monitor the states of the behaviour during its execution, and analyse whether the specification meets their requirement.

**Stage 5.** *Repeat Stage 2 to Stage 4 until all the* ***system scenarios*** *derived in Stage 1 are animated.*

Animating all possible behaviours of the system is required by our animation strategy. The process of animating a specific behaviour should be repeated until all of the potential behaviours have been animated.

## 3 Design of the Tool

According to our animation strategy, there are two steps in the animation process. The first step is to derive all possible system functional scenario, and the second step is to collect input and output data for all involved processes of a specific system functional scenario. In this section, we first introduce the method of deriving system functional scenario briefly, and then describe how to collect input and output data under the two ways mentioned above.

### 3.1 Deriving System Scenario

We choose deriving functional scenarios from a CDFD because it is extremely difficult to automatically generate scenarios from a process specification directly. Form the introduction of SOFL, we know that process specification uses mathematical notation to define processes in a module, and it is not designed to present the architecture or logic between different processes. If deriving functional scenarios from process specifications, it requires parsing the entire specification to determine which two processes are connected. It is obviously not a cost-effective approach. On the contrary, deriving scenarios from the CDFD will be more cost-effective, because the CDFD is specifically designed to describe the relations among the processes, and we can derive functional scenarios based on the topology of CDFD.

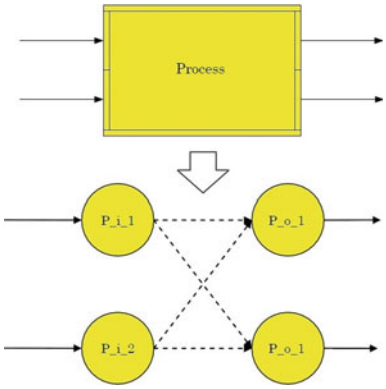


**Fig. 3.** The decomposition of a process

In order to generate all possible scenarios, the first step is to decompose the CDFD. This can be realized by decomposing every process in the CDFD. Each process can be decomposed into a corresponding graph. Figure 3 illustrates the decomposition of a process. In the original CDFD, the port list on the left side of a process is input port list, in which each input port is ordered from top to bottom, we use a number to label each port. The output ports of a process are labelled in the same way. In the corresponding graph, each node represents one port of the process, one input port or output port.

Each node in the corresponding graph has a name, consisting of three parts: the first character of the process's name, the identification of input or output port, and the port number. Different nodes are connected by two kinds of edges, solid edges and dotted edges. The solid edge represents the data flow in original CDFD and the dotted edge represents the mapping relationship inside the process. Like the data flows connecting two different processes in CDFD, the solid edges connect one input port node and one output port node that belong to different processes. Contrasts to the solid edges, the dotted edges represent the implicit relation in process. We use dotted edges to explicitly present this kind of relation because the ports that belong to the same port list are exclusive. If a process has more than one input port and output port, we need to find all possible combinations between its input ports and output ports. So that we can find all possible functional scenarios. The dotted edges represent such possible combinations or relations. In practice, just one dotted edge in each process can be valid each time. It means at each time process receiving and sending data from the input port and output port which are concerned by the valid dotted edge.

By using the decomposition method, one CDFD can be decomposed to a graph that contains only input port nodes and output port nodes. The nodes in the decomposed graph is linked by solid and dotted edges. The process of deriving all possible system functional scenarios can be realized as finding all possible paths in the decomposed graph. For the space sake, we will not explain this method further. The details of this method is included in [8].

### 3.2 Collecting Data from Users

The process of collecting data from users is the easiest way of collecting data. It can be separated into two steps. First is let the user provide data, and second is to check whether the provided data satisfy the related operation functional scenarios. The two steps are usually mixed in practice. For example, in the beginning of an animation, users first provides the input data for the first process, $P_1$, in the selected system functional scenario. Then, the users should check whether the input data they provide satisfy the predicate expression $\sim S^1_{pre} \wedge C^1_i$, a part of the related operation functional scenario of process $P_1$. If the input data satisfy the expression, the users can provide output data of $P_1$ based on the input data. Otherwise, the users should provide the input data again. Once the users provide the output data of $P_1$ successfully, the output data of $P_1$ will be used as the input data of the second process, $P_2$. This procedure will continue until the output of the entire system functional scenario are successfully provided.

To facilitate the users to check whether the data they provide satisfy a specific predicate expression, we use fault tree in our tool to help the users to do the judgement. We first apply the provided data to the predicate expression, and than decompose the expression using a tree structure, Each node in this tree presents an atomic expression. Since the fault tree technique is well known, we do not do any further explanation here.

### 3.3 Generating Data Automatically

The automatic data generation method underlying our tool is first introduced in [6]. The advantage of this method is that there is no need to translate the formal specification to any executable program. Here we give a brief introduction of the principle of the method.

As described previously, an *operation scenario* is expressed as a conjunction $\sim OP_{pre} \wedge C_i \wedge D_i$. To derive input data based on the *operation scenario*, it must be decomposed first. The decomposing process is divided into following two steps:

- **Step 1: Eliminate Defining condition.** The defining condition $D_i$ is eliminated first since the execution of program only requires input values. The input data generation depends on the pre-condition and guard condition, and defining condition usually do not provide the main information for input data generation. The conjunction after eliminating defining condition is $\sim OP_{pre} \wedge C_i$, called "*testing condition*".
- **Step 2: Convert to disjunctive normal form.** The remainder of the *operation scenario* is translated into an equivalent disjunctive normal form (DNF) with form $P_1 \vee P_2 \vee ... \vee P_n$. A $P_i$ is a conjunction of atomic predicate expressions, say $Q^1_i \wedge Q^2_i \wedge ... \wedge Q^m_i$.

Let $Q(x_1, x_2, ..., x_w)$ be one of the atomic predicate expressions $Q^1_i, Q^2_i, ..., Q^m_i$ mentioned previously. The variables $x_1, x_2, ..., x_w$ is a subset of all the input

variables. The values for the input variables involved in each atomic predicate expression Q can be generated using a set of algorithms that deals with the following three situations, respectively. Here we are using variables of numerical types as examples for convenience.

**Table 1.** Input data generation algorithm

| No. of Algorithms | $\ominus$ | Algorithms of test case generation for $x_1$ |
|---|---|---|
| 1 | $=$ | $x_1 = E$ |
| 2 | $>$ | $x_1 = E + \Delta x$ |
| 3 | $<$ | $x_1 = E - \Delta x$ |
| 4 | $\leq, \geq, \neq$ | Similar to above |

- **Situation 1:** If only one input variable is involved and $Q(x_1)$ has the format $x_1 \ominus E$, where $\ominus \in \{=, <, >, \leq, \geq, \neq\}$ is a relational operator and $E$ is a constant expression, using the algorithms listed in Table 1 to generate test cases for variable $x_1$.
- **Situation 2:** If only one input variable is involved and $Q(x_1)$ has the format $E_1 \ominus E_2$, where $E_1$ and $E_2$ are both arithmetic expressions which may involve variable $x_1$, it is first transformed to the format $x_1 \ominus E$, and then apply Situation 1.
- **Situation 3:** If more than one input variables are involved and $Q(x_1, x_2, ..., x_w)$ has the format $E_1 \ominus E_2$, where $E_1$ and $E_2$ are both arithmetic expressions possibly involving all the variables $x_1, x_2, ..., x_w$, first randomly assigning values from appropriate types to the input variables $x_2, x_3, ..., x_w$ to transform the format into the format $E_1 \ominus E_2$, and then apply Situation 2.

Note that if one input variable $x$ appears in more than one atomic predicate expression, it needs to satisfy all the expressions in which it is involved.

To define the generated input data precisely, we use a set of states of input variables, called *input case*, to present the one-to-one correspondence between each input variable and its value. The *input case* is denoted as $I_c$ and defined as follows.

**Definition 4.** *Let* $OP_{iv} = \{x_1, x_2, ..., x_r\}$ *be the set of all input variables of operation* $OP$ *and* $Type(x_i)$ *denotes the type of* $x_i (i \in \{1, 2, ..., r\})$. *Then* $I_c = \{(x_i, v_i) | x_i \in OP_{iv} \wedge v_i \in Type(x_i)\}$. *If* $(x_i, v_i) \in I_c$, *we write* $I_c(x_i) = v_i$.

After automatically collecting the input data, the next step is to generate output data. The same algorithm can be used. The similar process is described as follows.

- **Step 1: Verify the given input data.** For any given input case $I_c$, evaluate the predicate $(\sim OP_{pre} \wedge C_i)[I_c(x_i)/x_i]$, which is the result of substituting the variable $x_i$ with the value of variable $x_i$ in the *testing condition* $\sim OP_{pre} \wedge C_i$. The result of true means that the given input data can be processed by the operation, and the output data can be produced based on the input data.

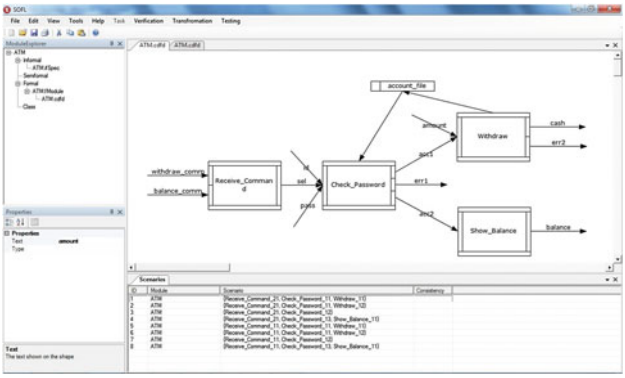


**Fig. 4.** Scenario explorer

- **Step 2: Substitute input variables.** Substitute the input variables in *operation scenario* with the corresponding values in $I_c$, and get a new predicate $\sim OP^*_{pre} \wedge C^*_i \wedge D^*_i = (\sim OP_{pre} \wedge C_i \wedge D_i)[I_c(x_i)/x_i]$, which merely contains output variables.
- **Step 3: Convert to disjunctive normal form.** Translate the conjunction $\sim OP^*_{pre} \wedge C^*_i \wedge D^*_i$ into an equivalent disjunctive normal form (DNF) with form $P^*_1 \vee P^*_2 \vee ... \vee P^*_n$. A $P^*_i$ is a conjunction of atomic predicate expressions, say $Q^{*1}_i \wedge Q^{*2}_i \wedge ... \wedge Q^{*m}_i$.
- **Step 4: Generate values for output.** For each $Q^{*j}_i (j \in \{1, 2, ..., m\})$, use the algorithms explained in previous subsection to generate values for output variables.

Similar to the definition of *input case*, we define *output case* to present output variables and their generated values formally.

**Definition 5.** *Let* $OP_{ov} = \{y_1, y_2, ..., y_k\}$ *be the set of all input variables of operation* $OP$ *and* $Type(y_i)$ *denotes the type of* $y_i (i \in \{1, 2, ..., k\})$. *Then* $O_c = \{(y_i, v_i) | y_i \in OP_{ov} \wedge v_i \in Type(y_i)\}$. *If* $(y_i, v_i) \in O_c$, *we write* $O_c(y_i) = v_i$.

## 4 Framework and Prototype

The prototype is implemented on the bases of a framework that is built to help developers specifying SOFL specification. The framework includes the editors of informal specification, formal specification, and CDFD, etc. All the specifications are well organized and stored in the well formatted files under this framework. It provides a fundamental to further utilization of the formal specifications. Now, the prototype has been a component of the framework. In this section, we introduce some functions that related to the animation.

The implementation prototype is corresponding to the two steps. The first step is to derive system functional scenario. Figure 4 shows the snapshot of

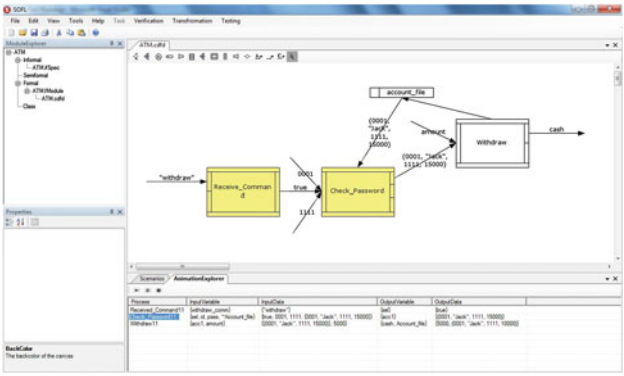


**Fig. 5.** Animation board

**Table 2.** Operation scenarios involved in the selected system scenario

| Process | Operation scenario |
|---|---|
| Received_Command$_{11}$ | $withdraw =$ "$withdraw$" $\wedge sel = true$ |
| Check_Password$_{11}$ | $x.id = id \wedge x.password = pass) \wedge sel = true \wedge acc1 = x$ |
| Withdraw$_{11}$ | $amount \leq \sim x.balance \wedge x.balance$ $= \sim x.balance - amount \wedge cash = amount$ |

derivation. The CDFD is the system shown in Fig. 1. Here we chose the first scenario {withdraw_com}[Receive_Command, Check_Password, Withdraw]{cash} to animate. The corresponding operation functional scenario is listed in Table 2.

Figure 5 shows the snapshot of animation. The data collected is listed at the lower part of the window. Each row shows the input and output data for a single process. For example, the first row list the input and output of process "$Receive_Command_{11}$". The CDFD in the center of the window shows the mid-step of animation, and the process "$Check_Password_{11}$" is under animated.

## 5 Related Work

Formal specification animation is an effective technique for the communication between users and developers. Tool support animation can make such communication easier. In this section, we introduce some existing work on specification animation.

Liu and Wang introduced an animation tool called SOFL Animator for SOFL specification animation [7]. It provides syntactic level analysis and semantic level analysis of a specification. When performing animation, the tool will automatically translate the SOFL specification into Java program segments, and then use some test case to execute the program. In order to provide reviewers a graphic presentation of the animation, SOFL Animator uses Message Sequence Chart (MSC) to present the simulation of the operational behaviours.

MSC is also adopted in other animation approach as a framework to provide a graphical user interface to represent animation. Stepien and Logrippo built a toolset to translate LOTOS traces to MSC and provide a graphic animator [9]. The translation is based on the mappings between the elements of LOTOS and MSC. Combes and his colleagues described an open animation tool for telecommunication systems in [1]. The tool is named as ANGOR, and it offers an environment based on a flexible architecture. It allows animating different animation sources, such as formal and executable language like SDL and scenario languages like MSC.

## 6 Conclusions and Future Work

In this paper, we describe a tool to support SOFL specification animation. The animation strategy is introduced first and then the prototype is presented. Comparing to other existing tool, the advantage of our work is that there is no requirement to translate formal specification to executable program. This means there is no further request for the developers about the style of specification. We provide two ways for users to collect data for animation. Each way shows different aspects of the system to users. We think the data can give the users a concrete point of view, and help them to understand the behaviours of the system. But only the animation is not enough to validate the formal specification, in the future, we hope to combine the inspection technique with animation. The inspection contains a list of question and require the users to think rather than observation only. We hope the combination of these two techniques can make validation more effective and efficiency.

## References

1. Combes, P., Dubois, F., Renard, B.: An open animation tool: application to telecommunication systems. Int. J. Comput. Telecommun. Netw. **40**(5), 599–620 (2002)
2. Waeselynck, H., Behnia, S.: B model animation for external verification. In: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, pp. 36–45 (1998)
3. Morrey, I., Siddiqi, J., Hibberd, R., Buckberry, G.: A toolset to support the construction and animation of formal specifications. J. Syst. Softw. **41**(3), 147–160 (1998)
4. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer, Heidelberg (2004). ISBN 3-540-20602-7
5. Liu, S., Nakajima, S.: A Decompositional approach to automatic test case generation based on formal specification. In: Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement, pp. 147–155 (2010)
6. Li, M., Liu, S.: Automated functional scenarios-based formal specification animation. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), pp. 107–115. IEEE CS Press, Hong Kong (2012)
7. Liu, S., Wang, H.: An automated approach to specification animation for validation. J. Syst. Softw. **80**, 1271–1285 (2007)

8. Li, M., Liu, S.: Automatically generating functional scenarios from SOFL CDFD for specification inspection. In: 10th IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 18–25 (2011)
9. Stepien, B., Logrippo, L.: Graphic visualization and animation of LOTOS execution traces. Comput. Netw.: Int. J. Comput. Telecommun. Netw. **40**(5), 665–681 (2002)
10. Liu, S., Chen, Y., Nagoya, F., McDermid, J.A.: Formal specification-based inspection for verification of programs. IEEE Trans. Softw. Eng. **21**(2), 259–288 (2011). IEEE Computer Society Digital Library, IEEE Computer Society
11. Liu, S.: Integrating specification-based review and testing for detecting errors in programs. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 136–150. Springer, Heidelberg (2007)
12. Miller, T., Strooper, P.: Model-based specification animation using testgraphs. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 192–203. Springer, Heidelberg (2002)