Object-Oriented Behavioral Testing Through Trace Generation

Anastasia Tircuit[*]
EECS Department
Tulane University
New Orleans, LA 70118
tircuit@eecs.tulane.edu

B. Belkhouche
EECS Department
Tulane University
New Orleans, LA 70118
bb@eecs.tulane.edu

ABSTRACT

One challenge in software design is ensuring that the design meets the requirements of the user. One way to do this is to create a prototype from the design, then devising a set of test cases to test for the behavioral requirements. Here we present our automated prototype generator and method for testing behavior.

1. INTRODUCTION

It has been shown that the longer an error persists during development, the more time and money it will cost to fix the error [1]. If, however, we find inconsistencies and errors during the design phase, we can correct them before they reach the code. To that end, much work has been done on methods to test designs. Many of these are formal, systematic, and partially automated [2, 3, 4, 5, 6, 7, 8, 9]. A typical design analysis system will begin with a definition of some rules for what makes a valid design. These rules may define syntactic and semantic requirements of the design notation. They may also identify a set of consistency requirements, that is, places where there is some redundancy of information, and that information must be coordinated. They will then provide a way to test the design against these rules, to show where the design complies and where it violates them.

Another area of design analysis is involved with ensuring that the system does what it is intended to do. In this situation, there is an intended behavior or set of behavioral requirements, and we are attempting to test the design to ensure that it meets these requirements. Several ways to do this have been presented in the literature [3, 8, 10, 11]. Here we present our method of testing design behavior via execution traces.

The contents of the paper are as follows. Section 2 discusses the issues of testing program behavior and how this problem has been tackled in the literature. Section 3 presents our behavioral notation, based on the Communicating Sequential Processes (CSP) language [12]. Section 4 presents our trace generation program. Section 5 contains the results and conclusions.

[*]Corresponding author

2. CURRENT ISSUES

The basic idea is simple. We have a set of requirements for the software system, and we have a design which is supposed to meet these requirements. How do we ensure that this is the case?

One variety of behavioral checking involves defining a set of rules for how a behavior may evolve over two or more life cycles. In this situation, the assumption is the initial life cycle has the intended behavior already. The goal is to ensure that the new life cycles evolve such that the behavior remains consistent. These methods often include a set of rules which define how the evolution may take place to insure consistency. The next step is to provide some method to check that the life cycles in fact agree with the rules. We see this form of analysis in [9, 5]. A similar method is also present in [2]. In this case the relevant design elements are not multiple life cycles, they are multiple views of the system.

In the case of life cycles and of multiple views, the issue is maintenance of behavior. The issue we are interested in is ensuring that the behavior meets its requirements. Attempts have been made at addressing this issue in [3, 8], by the method of executable designs. In this case, an array of test cases are created, then by some means, executed on the design. In the case of [3], the method involves creating an execution graph from the design. The test cases may then be run on the graph. Their method is entirely manual. As the graphs creation involves laying out loops into a series of branching structures, the graphs are likely to become unwieldy given a complex system. In [8], the authors make reference to the LOTOS SMILE simulator. They state this may be used to create executable prototypes of their specifications written in the LOTOS notation. From the information given, it is unclear exactly how powerful a testing mechanism this is.

The ideal solution for comprehensive testing is to create a fully executable prototype. In this case, the tester may run test cases on the prototype and see exactly what its behavior is. However, creating the prototype is often a time consuming task. If it has all of the functionality of the final product, it will take close to the same amount of effort to create. As we are trying to test the behavior before reaching that point, prototyping may not be an optimal solution. Several approaches to prototyping attempt to scale-down the structures created, thus eliminating part of the work involved [10, 11]. These typically either target a small set of critical functions, or create skeletons of all of the functions and focus on the interface rather than the behavior.

Here we present another view of prototyping, based on execution traces. Our method begins with two elements:

- A set of test cases to test the requirements of the system.
- A behavioral design, written in a CSP based notation.

This system automatically generates the prototype. The user may then test it with the test cases and ensure that the behavior meets the requirements. Our objectives in developing our trace generation system are:

- Provide a way to test system behavior
- Minimize the overhead for the tester
- Automate the prototype generation process

3. BEHAVIORAL DESIGN VIA CSP

In an earlier paper, we presented our text based design notation, TOODL [13]. Here we use the behavioral part of that notation as the basis for behavioral designs and trace generation. This behavioral notation will be discussed in some detail.

The formalization of our method is based on Communicating Sequential Processes (CSP) [12]. CSP is well suited to the OO execution model. For each class in the model, its behavior is described as one, or a set of, CSP processes. These models consist of an alphabet of available events, and the sequences which define the order they may be executed in. These sequences may be recursive, defining an infinite number of possible behaviors. CSP also easily allows the description of interactions between classes with its message communication constructs. This is a distinct advantage over state-based modeling approaches. State-based modeling does not capture the interactions between objects. This requires the use of a separate notation to model object interactions. When using CSP both object life history and object interactions may be modeled with a single notation. To address the issue of compositionality, we use the parallel composition construct from CSP. Two or more processes may be compositionally combined in this way to describe a larger system.

To illustrate the behavioral notation, we will use a small example. A Simple_Car_System is made of two classes: Car, and Engine. For each class, we start by defining its alphabet. The alphabet contains all of the events available to the behavior. The Car general events are:

$$\alpha Car = \{move\_forward, turn\_on, turn\_off\}$$

The alphabet is denoted by αCar. Event names are written in lower case italic letters. We then begin to define the sequences these events may be called in. Some possibilities are:

$$\begin{aligned} Car &= turn\_on \rightarrow move\_forward \\ Car &= turn\_off \end{aligned}$$

The arrow notation (→) is a CSP operation meaning "then". So we read the first sequence as: turn the car on, then move forward.

We can combine sequences to make the object life history (OLH) with the choice operator (|). When combining the two sequences above, we create:

$$Car = (turn\_on \rightarrow move\_forward \mid turn\_off)$$

There is no preference to which choice is taken. Anytime an interaction with Car takes place, either of the sequences may be chosen.

Currently both of these sequences terminate the behavior of Car. It may turn_on, then move_forward, and then it will no longer be active. We may wish to allow the Car to take another action once it has moved forward. We do this by placing a process name at the end of the sequence. If our move_forward sequences is written like:

$$\begin{aligned} Car &= (turn\_on \rightarrow move\_forward \rightarrow Car \\ &\mid \; turn\_off) \end{aligned}$$

It will return to the Car behavior, and another sequence may be chosen. This type of recursive notation allows for many options for the behavior to be described, with only a small written description.

During modeling, situations may occur where one wishes to specify that an event A must be done before some other event B. In our example, it is reasonable to require that the turn_on event comes before the Car moves. One way to accomplish this is to create another CSP process which becomes part of the Car behavior. We name our new process Car1. Processes are named with capitol letters, to distinguish them from pure events. Car1's alphabet is drawn from the alphabet of Car. It will contain the events that must follow a turn_on event. Car1's alphabet is:

$$\alpha Car1 = \{move\_forward, turn\_off\}$$

And its OLH is:

$$Car1 = (move\_forward \rightarrow Car1 \mid turn\_off)$$

To reach the Car1 process, it will need to participate in an OLH sequence of the original Car process. Our new Car OLH is:

$$Car = (turn\_on \rightarrow Car1)$$

The behavior will begin with the OLH of Car, in this situation we only have one choice, to turn the Car on. Once the Car is turned on, the OLH of Car1 is now available to the environment.

Car_System is an extended car design. It contains three additional classes: Shift, Engine, and Brake. We will use the Engine class as an example. The Engine's purpose is to sit and wait for messages from the Car. It may do two things, turn on and turn off. As it will do these only after receiving a message, these events are incoming communication events. Thus, the alphabet of Engine is:

$$\alpha Engine = \{Engine?on, Engine?off\}$$

The ? in the event name denotes that this is an incoming communication event. As the Engine's tasks are simply sitting and waiting, its OLH is:

$$\begin{aligned} Engine &= (Engine?on \rightarrow Engine \\ &\mid \; Engine?off \rightarrow Engine) \end{aligned}$$

Engine will never terminate on its own. It will only stop waiting for a message when the entire Car_System is no longer running.

We may complete the communication channels by adding matching outgoing communication events to Car and Car1. These take the form of class!event, where class is the class that the matching incoming event is housed in. Including an Engine!on call, the new Car alphabet will be:

$$\alpha Car = \{turn\_on, Engine!on\}$$

We will also include this event in our sequences. Car's new OLH is:

$$Car = (turn\_on \rightarrow Engine!on \rightarrow Car1)$$

Car1's new alphabet and OLH are:

$$\alpha Car1 = \{move\_forward, turn\_off, Engine!off\}$$

$$\begin{aligned} Car1 &= (move\_forward \rightarrow Car1 \\ &\mid \; turn\_off \rightarrow Engine!off) \end{aligned}$$

As a final step, to include both the Car and Engine in the Car_System, the composition operator may be used:

$$Car\_System = (Car || Engine)$$

4. TRACE GENERATION

4.1 Automated Creation of Trace Generator

While the CSP model provides a formal and compact description of the behavior, we may want to observe how the system behaves over time and verify specific behavioral test cases. This may be done with our trace generation program. It allows the user to see which events are available during a point in execution, to choose an event, and see the resulting behavior of the model.

We will illustrate the trace generation program with this more detailed Car OLH:

```
OLH Car
alphabet Car = {Engine!on, turn_on, Part_Car1}
Car = (turn_on -> Engine!on -> Part_Car1)

OLH Part_Car1
alphabet Part_Car1 = {move_forward,
   move_backward, Brake!press, Brake!release,
   Shift!drive, Shift!reverse, Part_Car2, stop}
Part_Car1 = (move_forward -> Brake!apply ->
   shift!drive -> Brake!release -> Part_Car1 |
   move_backward -> Brake!apply ->
   shift!reverse -> Brake!release ->
   Part_Car1 | stop -> Part_Car2)

OLH Part_Car2
alphabet Part_Car2 = {Brake!release, Shift!park,
   Engine!off, Brake!press, Part_Car1, Car,
   turn_off, park}
Part_Car2 = (turn_off -> Brake!press ->
   Shift!park -> Engine!off | park ->
   Brake!press -> Shift!park -> Part_Car1)
```

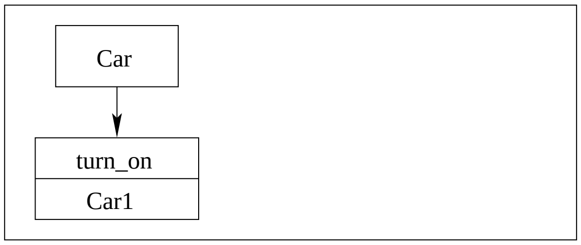

Figure 1: This is the data structure after the initial process is examined.

The Engine OLH has been described in the previous section. The Shift and Brake OLHs are not included here. They do not play a part in the trace generation.

Initially the program searches through the alphabets of each process to identify the general events. The general events are how the objects interact with the environment, so these will become our user interface during trace generation. In our example, only Car, Car1, and Car2 have general events, so these are the only processes we are concerned with here.

For each process with general events, we examine the OLH for sequences beginning with general processes. Car has one sequence, and it meets the criteria:

$$turn\_on \rightarrow Engine!on \rightarrow Car1$$

We then remove the pertinent information from this sequence and store it in a data structure.

```
Event: turn_on
Next Process: Car1
```

In our data structure, this will be stored as the only event choice under the Car process. In Figure 1 we see a figure showing the data structure after the Car process is examined. It has the one available event, turn_on. Paired with turn_on, is the next process available to the user, Car1. As we examine the rest of the process, we will see how the rest of the processes are stored in the data structure.

For Car1, there are 3 sequences which meet the criteria. We will continue to build our data structure, creating a new node for Car 1, and attaching its events. The new structure may be seen in Figure 2.

Note that the next process field may contain either a recursive process or a separate process. In move_forward and move_backward, the next process recurses on Car1. In stop, the next process is Car2.

In the event that a sequence does not end in a process name, its next event field will be blank (NULL). This is the case for the turn_off event under Car2. We see the new additions to the data structure in Figure 3.

The remaining classes will be checked in the same way for OLH sequences which begin with general events. In this example, none of those classes have any qualifying sequences, so they are not present in the structure.

4.2 Using the Trace Generator

The structure illustrated in Figure 3 is used to present the user interface in the trace generation program. The program begins by presenting the user with all of the event choices

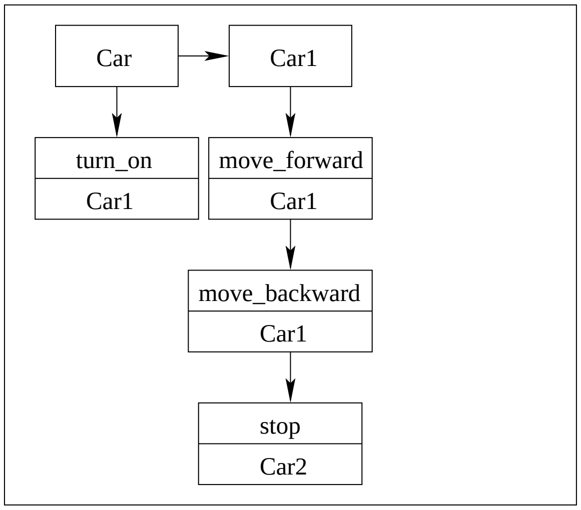

Figure 2: This is the data structure after the Car1 process is examined.

in the first column of the table. It then instructs the user to enter which event choice they would like. The user also has the option of exiting out of the trace generator. Note that the EXIT option is a function of the generator, not of the behavior itself. The initial screen for our Car_System example is:

```
turn_on
EXIT

Enter an event:
```

Upon entering "turn_on" The user will be given the options once the Car_System is in motion. These may be seen in the Car1 section:

```
move_forward
move_backward
stop
EXIT

Enter an event:
```

The move_forward and move_backward events will repeat in this section as many time as the user desires. This is consistent with the move_forward and move_backward sequences in Car1's OLH. They both end in the recursive event, Car1.

Upon entering stop, the user will be taken to the third and final menu for this example:

```
turn_off
park
EXIT

Enter an event:
```

If the user enters park, the trace generator will return to the Car1 menu, as the park sequence in our design ends with the Car1 process name. If the user enters turnoff, the trace generator program will print the trace, and offer the user the chance to create more. Here is one trace of our Car_System:

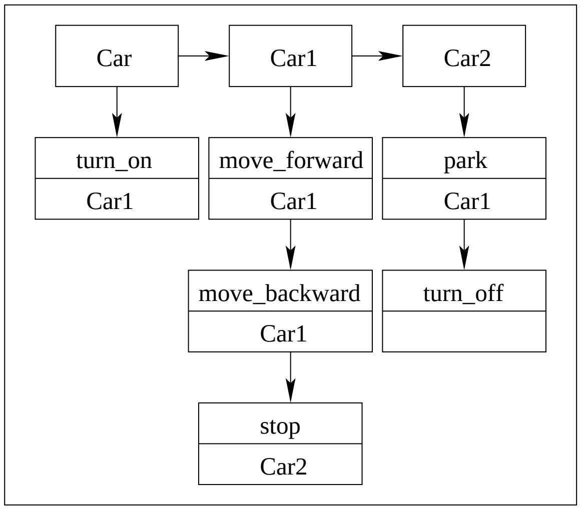

Figure 3: This is the data structure after the Car3 process is examined.

```
turn_on -> move_backward -> move_forward ->
   move_forward -> stop -> turnoff
```

The user may then continue to generate traces, verifying that the Car_System behaves in the way it is intended to.

4.3 Testing the Behavior with Trace Generation

The power of trace generation become more apparent when we add a set of test cases. The test cases may be as simple as an informal description of the intended behavior. Something like:

```
Car turns on
Car moves forward and backward
Car turns off
Car does not move until it has been turned on.
```

will outline the basic behavior of the Car_System. The tester can then attempt to generate traces which provide this behavior for the Car_System. This description will also provide a basis for a set of formal test cases later, so creating them is not additional work for the tester.

For more formal testing, we may describe the behavior as a set of traces. These traces should cover the required functionality of the Car_System. It is also helpful to test traces which, if available, would violate unwanted Car_System behavior.

The first list of test cases will ensure that the Car_System has some basic functionality.

Required Behavior

1. turn on → turn off
2. turn on → move forward → turn off
3. turn on → move backward → move forward → turn off
4. turn on → park → turn off

We run these test cases on the trace generator for Car_System. We are able to generate traces that match each of these test cases. This shows that our required behavior is actually available in the Car_System.

This second set of traces defines some behavior which we definitely do not want in the Car_System. If any of these traces are able to be generated, it shows that the Car_System has some unwanted behavior which needs to be addressed.

Unwanted Behavior

1. turn off → turn on
2. turn off → move forward
3. turn on → turn off → move forward

When we run these three test case, there are no choices available which will generate traces with this behavior. Once again we find that the behavior of the system is in accordance with our intended behavior.

It may also be the case that we want the Car_System to have some behavior not available to us. For example:

1. turn on → turn off → turn on → turn off

If we attempt to test this behavior, we will find that once the Car_System is turned off, no other actions may take place. We then know that our behavior needs some corrections.

5. CONCLUSIONS AND FUTURE WORK

Previously we stated our objectives as:

- Provide a way to test system behavior
- Minimize the overhead for the tester
- Automate the prototype generation process

We feel that our system has met these objectives. Our trace generation system does provide a way to test system behavior. It allows the tester to see what choices will be available during execution time. The tester may choose one, and will then see the outcome of that choice. This process may be repeated as many times as desired to test the behavior of the system. The tester may also try to generate traces of unwanted behaviors, to ensure these are not present in the system. This method requires little additional work from the tester. Specifying test cases is a necessary part of software development already. Here we are only requiring that they be specified by a certain time in the development process. Furthermore, they need not be formal. A casual English document outlining the desired and unwanted behavior is enough to test the behavior with our generation program. The only time spent by the tester is in generating the traces and reviewing the results. Given a design in the TOODL notation, the prototype generation is completely automated. The test need only run the utility and they will be looking at a user interface generated from their design. It also automatically states which actions are available, eliminating the possibility of choosing an impossible action during trace generation.

The future of the TOODL analysis tool includes making additions to the trace generation utility and its other design analysis mechanisms. Futher automations may be added to the trace generation program so that it is able to take a number of test cases as input, process the group, then tell the user which ones were successful. We would also like to explore having the program generate test cases automatically. Given a program with no recursion, every positive test case could be generated. When working with recursive programs, some boundries would need to be set which defined how much recursion is allowed. Otherwise, the program would generate test cases indefinitely. These plus other additions will make the tool a thorough design analysis and testing environment.

6. REFERENCES

[1] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*. International Thomson Computer Press, second ed., 1993.

[2] A. F. Egyed, "Automatically Validating Model Consistency During Refinement," tech. rep., University of Southern California, 2000.

[3] O. Pilskalns, A. Andrews, S. Ghosh, and R. France, "Rigorous Testing by Merging Structural and Behavioral Uml Representations," *Proceedings of the Sixth International Conference on the Unified Modeling Language*, 2003.

[4] G. Engels, L. Groenewegen, R. Heckel, and J. M. Küster, "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," *8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2001.

[5] G. Engels, R. Heckel, and J. M. Küster, "Rule-based Specifications of Behavioral Consistency based on the Uml Meta-Model," *Proceedings of the Fourth Iinternational Conference on the Unified Modeling Language*, 2001.

[6] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Iinconsistency Handeling in Multi-perspective Specifications," *Proceedings of the Fourth European Software Engineering Conference*, pp. 84–99, 1993.

[7] P. Fradet, D. L. Métayer, and M. Perin, "Consistency Checking for Multiple View Software Architectures," *Lecture Notes in Computer Science*, vol. 1687, pp. 410–428, 1999.

[8] A. Moreira and R. Clark, "Combining Object-Oriented Modeling and Formal Description Techniques," *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pp. 344–364, 1994.

[9] M. Schrefl and M. Stumptner, "Behavior-Consistent Specialization of Object Life Cycles," *ACM Transactions on Software Engineering and Methodology*, pp. 92–148, January 2002.

[10] R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, *Approaches to Prototyping*. Springer-Verlag, 1984.

[11] M. M. Tanik and R. T. Y. (guest editors), "Rapid Prototyping in Software Development," *IEEE Computer Soc. Computer*, vol. 22, 1990.

[12] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] B. Belkhouche and A. Nix, "Formal Analysis of Uml-based Designs," *Proceedings of the International Conference on Software Engineering Research and Practice, SERP '04*, vol. I, pp. 220–226, June 2004.