Automatic code generation for supervised DES

Lenko Grigorov, grigorov@cs.queensu.ca
Course project report, ELEC843
Dec. 2002

This project introduces a method for automated code generation for supervised DES – in the sense of Java™ objects, responding to events. The code generator is implemented in Java™. It uses the object reflection mechanism to inspect a Java™ class and to generate a subclass, such that it exerts the control specifications defined by the user in a separate plain-text file (in the form of a DFA). The main advantages of this method are speed – the computer automatically constructs the correct code; simplicity – the user doesn't need to understand programming; and flexibility – the control specifications can be modified at any point without need to modify the software. The disadvantages are that some restrictions are posed on the DES Java™ classes; and that the operation of the program is not intuitive.

1. Motivation

The study of Discrete Event Systems (DES) is a relatively new field. The notion of DES is strongly related to Computer Science, since DES can be formalized with many of the constructs of Computer Science – including automata. Even so, not much supporting software exists on the market. The most notable software packages – CTCT and UMDES – are very research-oriented and lack some of the features required by non-expert users. Even professionals find the interfaces intimidating, while regular users would not be able to work with these programs at all.

As already mentioned, the correspondence between DES and Computer Science is very tight. On one hand, both areas use the same theoretic foundations: automata, petri nets, graphs. On the other hand, we can see real-world examples of how manufacturing robots are controlled by computers (i.e. software). Having these two facts in mind, it is obvious that it is theoretically possible to define a formal "mapping" from DES to software.

Being able to automate this process would be very helpful to DES users. Let us consider the following scenario. An organization purchases a bottling machine. The machine is modeled as a discrete event system, controlled by software. The machine starts when the user presses the "start" button and bottles until the user presses the "stop" button. Each bottling consists of a detector sensing the arrival of an empty bottle, the machine filling the bottle, putting the cap, and releasing the bottle. The control software comes with an API for program extensions, however it does not provide the user with any other means how to modify the functioning of the machine. If the user wishes to make the machine turn off automatically after processing 100 bottles, he or she would have to write complex extensions to the existing software (given that the user has the knowledge how to program!). If later he or she wishes to keep the machine from putting caps, he or she would have to rewrite the software extension – another long and difficult task.

Having an automatic mapping tool, the above tasks become very straightforward and fast. The user specifies the required DES behavior using much more natural (but formal) expressions like automata or pseudo-code. The specifications could be generated even more simply by using special tools. Then the user would start the automatic code generator and he or she would select the control program of the machine and the description of the specifications. The generator would produce as its output a control program, which acts in the same way as the original software, while ensuring the specifications are met. Furthermore, when the user wishes to change the specifications, he or she would have to simply replace the specifications file. Thus the user would not have to resort to programming and he or she would be able to complete the task fast and reliably.

2. Method

Since the development of a general mapping from DES specifications to software specifications (or directly to software) would be a very complex and time-consuming problem for the goals of this project, I considered a reduced version. I chose DES to be a programming object, which responds to events. The events take place when designated methods of the class are called. The events are partitioned into controllable and uncontrollable, depending on whether or not the class provides methods to control the execution of the event methods. Since many machines are controlled programmatically, and since OOP is increasingly popular, this choice might not be as restrictive as it seems at first.

The DES control (supervisor) specifications have to be formally expressed by the means of a DFA. Even though this choice reduces the generality of the solution, one should consider that most supervisor specifications are expressed in this form anyways. Furthermore, special software packages for generation and/or modification of DFA are readily available. Automata seem to be relatively easier to master, compared to other formal specifications.

The automatic code generator would inspect the object class (the DES) and would automatically discover the event methods and the control methods. It would then generate a subclass, which intercepts calls to these methods and ensures that the object behaves according to the control specifications.

The user of the software would not have to program anything. He or she would only have to ensure that the supervisor specifications are correct. Any programming objects, which fulfill the requirements for DES (i.e. they have correct method definitions), would be usable with the code generator.

3. Implementation

The implementation of the automatic code generator is done in Java™. The program is able to generate supervision code for Java™ classes – it generates a subclass of the "plant". If the original class is named `ClassName`, the subclass will be named `supervisedClassName`. Since the generated class is a subclass, it can be used in all places where the old class is used (the class identifiers can simply be replaced).

Since this solution does not aim at being completely general (as discussed above, this would not be a trivial problem), several restrictions are placed on the classes, which will be supervised.

- All events in the system must have a corresponding method named `onEventX`, where `X` stands for the name of the event (case-sensitive).
- All controllable events must have corresponding methods with the signatures `public void enableEventX()` & `public void disableEventX()`, which don't throw any exceptions and which should be used by the class to enable and disable the events (`X` stands for the case-sensitive name of the event).
- Each `onEventX` method must ensure that after the method call all controllable events are enabled or disabled according to the specification of the plant (by calling the respective `enableEventX` and `disableEventX` methods). If this is not ensured, the generated supervisor will not be able to control the plant properly.
- The class must not define a method with the signature `supervisorInit()`.
- The class must not define a field with the signature `StateMachine supervisorDFA`.
- The constructors of the class must not call any of the `onEventX`, `enableEventX`, and `disableEventX` methods.

All these restrictions are put in place, because the supervisor (the subclass) must be able to intercept the occurrence of events properly and to be able to control the enablement and disablement of controllable events. The generator uses the object reflection mechanism of Java™ to inspect the plant (the original class). This requires that the class is already compiled and that it is available to the runtime environment. Through the careful inspection, the generator determines what events can occur and which are controllable. It then generates a Java™ subclass, where each `onEventX` and each `enableEventX` method is overloaded. The class instantiates a `StateMachine` class (as discussed later). The interception of `onEventX` methods allows for keeping track of the occurrence of events and enablement and disablement of controllable events in each state. The interception of `enableEventX` methods allows for the blocking of unwanted enablements. The interception of `disableEventX` methods is not necessary, because the overall goal is to achieve an intersection of the controllable events (disablement need never be blocked).

The supervisor class instantiates a `StateMachine`, which is a very limited but lightweight implementation of a DFA. The `StateMachine` loads a DFA definition from a plain-text file, in the format generated by CTCT. The supervisor uses this instance to ensure that specifications are met. When an event occurs, the state machine is queried about the necessary configuration of the controllable events and the events are adjusted accordingly. The name of the file with the DFA definition must be `ClassName.PDS`, where `ClassName` stands for the name of the plant class. The file should be available at the start of execution of the supervisor. If modification to the behavior of the plant is required later on, it is sufficient to simply replace the file with the DFA definition. If this file was generated with CTCT and all events are numbers and if the event names are different (i.e. `send` vs. `5`), then it is necessary to rename the events in the file (i.e. replace `5` with `send`).

The work of the generator can be summarized like this: As an input it takes a compiled Java™ class, specifically designed for supervision (it is not necessary to have the source code of the class); and a DFA definition in the CTCT format, where events are labeled in the same way like in the class. As an output it produces the Java™ source code for a subclass of the original class, which ensures that the specifications are met. During execution, the DFA definition is referred, thus it is possible to replace it with a different one to alter the behavior of the class without recompilation.

4. Testing

In order to run the program, the following classes should be available: `DESAutogen`, `NoSuchTransitionException`, and `StateMachine` (the source code and the compiled versions are provided in the program directory). The program is started by running the `DESAutogen` class (e.g. by executing "`java DESAutogen`" from the prompt). "Select plant" in the "File" menu lets the users choose which Java™ class they would like to supervise. The source code for the supervisor is created immediately after the selection. Note that the Java™ class for the plant should be available to the Java™ Runtime Environment when the generator starts. If the class belongs to the default package (as do all test examples), it is sufficient to copy the compiled class to the program directory before running the generator. Otherwise the program will announce that the class is unavailable. Note also that the DFA definition is not required during the code generation.

After the supervisor class is generated, it has to be compiled: e.g. by executing "`javac supervisedClassName.java`" at the prompt (note that the original class and the `StateMachine` class should be available during compilation).

In order to use the supervised plant, the program, which uses the original class, has to be modified so that it instantiates the supervisor class instead. This is done by replacing all references to `ClassName` to `supervisedClassName`. With the test examples, this is already done. Also, the original class, the `StateMachine` class, and the DFA definition file have to present when executing the new program.

There are two test examples provided in the test directory.

4.1 Communication Protocol (`test1`)

This example is based on part 4 of assignment 3. `T12` is a simple text-based program, which scans the input for events and prints the event trace as it evolves. It can be run by executing "`java commExample`" from the prompt. It can be stopped at any point by typing quit or exit. The event trace can evolve according to the unrestricted operation of shuffle(T1,T2) as defined in the example. The DFA definition for the supervisor (in the file `T12.PDS`) is generated with CTCT and it corresponds to the NC supervisor in the assignment. In order to test the example:

1. Copy all files from the `test1` directory to the `program` directory
2. Execute "`java DESAutogen`"
3. Select the "`T12.class`" file
4. Execute "`javac supervisedT12.java`" (note that your system should have the Java™ compliler installed; if not – you can copy the "`supervisedT12.class`" file from the `compiled` directory)
5. To try the nonsupervised program, execute "`java commExample`". You can try inputing the trace 0,1,6,3,8,2 (equals $\alpha 1$, $\sigma 1$, $\alpha 2$, $\sigma 2$, $\rho 2$, $\rho 1$).
6. To try the supervised program, execute "`java commExampleS`". You can try inputting the same trace to see the difference.

Suggestions for other tests: you could model a different behavior using CTCT and replace the "`T12.PDS`" file with the new specifications. Simply restarting the program ("`java commExampleS`") will take into account the new specifications. It is necessary to model using the event numbers printed out when the program starts.

The source code for the `T12` class is provided in the `source` directory. It is put separately to show that the code availability is not required to be able to use the system.

4.2 Small Factory (`test2`)

This example is based on the Small Factory as discussed in class. `SmallFactory` is a graphical program, which lets you control how the two machines operate by pressing buttons in order to invoke the events. The program operates according to shuffle(M1,M2). The DFA definition for the supervisor was created using CTCT and it enforces (buffer over/underflow control)$\cap$(repair M2 before M1)$\cap$(do not take more than 10 parts from intput). The numbered events in the CTCT output were renamed to match the events in the Java™ class. In order to test the example:

1. Copy all files from the `test2` directory to the `program` directory
2. Execute "`java DESAutogen`"

3. Select the "`SmallFactory.class`" file

4. Execute "`javac supervisedSmallFactory.java`" (note that your system should have the Java™ compliler installed; if not – you can copy the "`supervisedSmallFactory.class`" file from the `compiled` directory)

5. To try the nonsupervised program, execute "`java SmallFactory`".

6. To try the supervised program, execute "`java factoryExample`".

Suggestions for other tests: you could rename `SmallFactory1.PDS` or `SmallFactory2.PDS` to `SmallFactory.PDS` and rerun the `factoryExample` program. The two definitions specify respectively the "buffer over/underflow control" and "take no more than 2 parts from input" requirements.

The source code for the `SmallFactory` class is provided in the `source` directory. It is put separately to show that the code availability is not required to be able to use the system.

5. Conclusion

This project proves that the automatic code generation for DES supervision is feasible. Even though the scope of the project is limited to some extend, the implemented code generator is very flexible. It can automatically generate supervisory code for any supervision-ready Java™ object – as demonstrated with the diagonally distinct test examples. It also uses plain-text DFA definitions, which can be generated either with software (CTCT) or by hand. Furthermore, once generated, the supervising code can be reused by simply replacing the control specifications file. The solution addresses the key features, expected from an automatic code generator: speed, simplicity, and flexibility. Unfortunately, time did not allow for further improvements of the implementation – like unattended program modification to incorporate the supervised class, or better user interface.

6. Reference

Without being formal:

1. ideas presented throughout the course
2. work by Lafortune, Wonham, and Rudie
3. CTCT
4. UMDES
5. Assignment 3 part 4
6. the Small Factory example
7. and of course, discussions with Karen, Ying, and Arezou