E-CARES Research Project: Extraction of State Machines from PLEX Code

André Marburger
Aachen University of Technology
Department of Computer Science III
Ahornstr. 55, 52074 Aachen, Germany
marand@cs.rwth-aachen.de

Dominikus Herzberg
Ericsson Eurolab Deutschland GmbH
CNM – Node Product Unit MSC
Ericsson Allee 1, 52134 Herzogenrath, Germany
Dominikus.Herzberg@eed.ericsson.se

Abstract

Re-engineering of legacy systems means among others to find out as much as possible about a specific system (reverse engineering). The information gathered in this part of re-engineering, sometimes also called system understanding*, is a prerequisite to any change to the system. In almost any approach described in the literature, the reverse engineering starts with analyzing the source code using parser techniques. The reason is that often the source code is the only reliable source of information. But, additional knowledge like domain knowledge, information about the specific development cycle, etc. plays a special role. During our research activities in the E-CARES project we found out that domain knowledge is very important when the re-engineering process is designed to be interactive. The information extracted from the source code and other sources of information, its representation, and its visualization has to meet both the demands of the re-engineer and the demands of the system experts involved. Telecommunication systems and the like are for example often planned and modeled using state machines. That is, system experts think in terms of state machines and protocols. This paper describes how state machine extraction is realized in case of the E-CARES project and how it helps to understand telecommunication systems.*

1. Introduction

The E-CARES[1] research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and the Department of Computer Science III, RWTH Aachen, deals with understanding complex legacy telecommunication systems. The subject of study is Ericsson's Mobile-service Switching Center (MSC) called AXE10. The cooperation aims to develop methods, concepts, and tools to support the processes of understanding and restructuring this kind of embedded systems.

In contrast to business applications embedded systems in general, and telecommunication systems in particular, have additional requirements regarding fault tolerance, reliability, availability, and response time. Even the development cycle differs. While modeling business applications normally involves typical modeling languages like the Unified Modeling Language (UML), there is modeling in terms of protocols represented by interacting state machines when it comes to telecommunication systems. This difference has significant impact on the kind of information needed from the code analysis, on the interpretation of the extracted information, on the internal representation of the extracted information in the re-engineering tools used, and on the visualization of relevant information to the re-engineer and to domain experts.

During our research activities in the E-CARES project we found out, that domain knowledge becomes more important the more the re-engineering process is designed to be interactive. In this case, the term *interaction* means a complex sequence of information processing and reaction to this information. Therefore, just pressing the OK-button to confirm a splash-screen doesn't meet the meaning of interactive in this paper. Successful interaction, with respect to accomplishing a certain goal, prerequisites that all parties of an interaction do speak a kind of common language. That is, the information available from a re-engineering tool, its visualization to an expert or re-engineer, and the interpretation by the expert or re-engineer have to be compatible.

In the context of the E-CARES project there are different flavors of interaction: There is interaction between the re-engineer and the re-engineering tools he is using and developing. There is also interaction between the domain experts and the re-engineering tools to validate results, give hints,

[1] The acronym E-CARES stands for **E**ricsson **C**ommunication **AR**chitecture for **E**mbedded **S**ystems.

or just get new insights for themselves. But, there is as well interaction between the re-engineer and the experts in the expert group to discuss findings, to evolve new ideas, and to develop new requirements from the results of the discussions. This can lead into problems because both parties have different interests concerning the re-engineering tools and judge the re-engineering results from different perspectives. As both the re-engineer and the domain experts are using the same re-engineering tools, these tools have to provide the appropriate information, visualization, and interaction possibilities for both of them.

In this paper we focus on the extraction of state machines as this domain concept is used to build and implement the telecommunication system studied in the E-CARES project. Furthermore, we will briefly describe how the additional information on state machines can be utilized to improve system understanding.

2. Extracting State Machines from PLEX Code

The extraction of state machines from PLEX code is not as easy as it seems. Though the majority of PLEX programs in the analyzed system was originally modeled and implemented as a state machine, there is no strict design rule of how to implement such state machines. Neither, there is a dedicated support for state machine implementation from the underlying programming language PLEX. Therefore, we first had to find out how state machines were implemented in different cases. Then we had to define a heuristic that allows to determine whether a program contains a state machine or not. In the third, step we developed an algorithm that allows to extract and re-assemble the original state machine on "architecture level" stepwise. These three steps are discussed in more detail in the remainder of this section.

Manual analysis of a couple of PLEX programs showed that there is a common sense at Ericsson of how to implement state machines in PLEX. Therefore, we were able to formulate the following list of facts indicating that a PLEX program implements a state machine:

- The program implements file size alteration to dynamically increase and decrease the number of instances of this program at runtime[2].
- The program contains a record that is stored persistently. This record contains a symbol variable of a string-valued enumeration type. The possible values of this state variable represent the possible states of the program or its instances respectively. Therefore, the current value of the state variable represents the current state of a program or program instance.
- The name of this state variable contains the substring `STATE`.
- The set of possible values of this state variable encloses the values `IDLE` and `SEIZED`.
- Each signal entry is immediately followed by a case statement whose condition queries the current value of the state variable[3]. Each branch of this case statement triggers different executions in the program.
- A state change is represented by an assignment to the state variable.

If all these facts apply to a certain program then this program contains the implementation of a state machine. Therefore, the heuristic in question is just to check for all the items in the list above. But, this heuristic just tells that there is a state machine implemented. It doesn't tell us how it looks like.

Some more analysis of the interdependence of signal reception, signal sending, and state changes showed that in each path of execution following a signal reception the program's/instance's state is changed at most a single time. Normally, the state is changed just before the execution control is handed back to the runtime system. Therefore, the algorithm for stepwise state machine extraction in PLEX program could be formulated as shown in figure 1.

The algorithm is divided into three functions written down in pseudo code. The main function is `extract-Statemachine`. It first checks whether there is any state variable present in the program specified by the parameter `block`. For this purpose the function `identifyStat-eVariables` is used. If the set `stateVariables` is empty, the program to analyze does not contain a state machine (according to our criterion). In this case, the state machine extraction is terminated immediately. If there is at least one state variable present, the state extraction is proceeded. Now the function starts with the first signal reception statement – a so-called signal entry – to trace subsequent statement sequences according to the control flow. For each statement sequence that is reachable starting from the signal entry, the algorithm detects state changes within these statement sequences. That is, it searches for assignments to one of the state variables. For every state change detected a corresponding state transition is added to the internal representation of the state machine in the extraction tool. In figure 1, this part of the algorithm is represented by the function `checkStatementSequence`.

[2] Though it is not a feature of the PLEX programming language, the system analyzed contains a mechanism that allows to create multiple instances (processes) of a single program. These instances are managed by the control part of the program's code. The computation state is kept persistent by means of a huge data record per instance. These records are stored in a file.

[3] PLEX programs and their instances communicate by means of signals.

```
extractStatemachine(block) {
  Variable stateVariables[] = empty;
  State states[] = empty;
  StateChange completeSetOfStateChanges[] = empty;

  identifyStateVariables(block, out stateVariables);
  if not(empty(stateVariables)) then
    for each (signal entry in block) do
      if (CASE-statement follows) then
        if (CASE condition queries element in stateVariable) then
          for each (CASE-branch) do
            states = states in current branch guard;
            checkStatementSequence(self.-goto->, states, null,
                                   out completeSetOfStateChanges);
          end for
        end if
      end if
    end for
    for each (element in completeSetOfStateChanges) do
      addToStatemachine(element);
    end
  end if
}

identifyStateVariable(block, out stateVariables) {
  Variable symbolVariable[] = empty;

  for each (record in block) do
    if (record contains symbol variable) then
      if (    (variable name contains substring 'STATE')
          and (variable is string valued)
          and (variable is dynamically stored)
          and (possible values contain 'IDLE' and 'SEIZED'))
      then
        symbolVariables = (symbolVariables + symbolVariable);
      end if
    end if
  end for
  stateVariables = symbolVariables;
}

checkStatementSequence(statementSequence, states, setOfStateChanges,
                       out completeSetOfStateChanges){
   StateChange mySetOfStateChanges[];
   StateChange tmpSetOfStateChanges[];
   StatementSequence setOfSubsequentStatementSequences[];
   Condition condition = empty;

   mySetOfStateChanges = setOfStateChanges;
   for each (statement in statementSequence) do
     if (isStateChange(statement)) then
       for each (state in states) do
         addToSet(mySetOfStateChanges,
                  new Tuple(state,signal,condition,newState));
       end for
     else if (isGotoStatement(statement)) then
       addToSet(setOfSubsequentStatementSequences,statement.gotoTarget);
     else if (isIfStatement(statement)) then
       condition = (condition AND statement.condition);
     end if
   end for
   for each (statementSequence in setOfSubsequentStatementSequences) do
     checkStatementSequence(statementSequence,
                            mySetOfStateChanges,
                            out tmpSetOfStateChanges);
     mySetOfStateChanges = tmpSetOfStateChanges;
   end for
   completeSetOfStatechanges = mySetOfStateChanges;
}
```

Figure 1. Extraction Algorithm

Summarized, the algorithm presented here re-creates a state machine on the "modeling level" by extracting it transition by transition and state by state from the code that implements it. Note, if there are several state variables detected in the code of a single program the corresponding state machines are extracted "simultaneously".

3. Benefits from State Machine Extraction

Once all the state machines are extracted the question arises how to use this information and how to combine it with other information available to gain additional benefit. First of all, the possibility to extract and visualize the state machines implemented by a PLEX program is a step towards a domain compliant abstraction from the plain source code. In combination with the multi-level abstraction/visualization of the analyzed systems structure, control flow, and data flow provided by the E-CARES re-engineering prototype [2] the understanding of telecommunication systems is improved.

Additional improvement can be reached if information on the runtime behavior of the analyzed system is available. The E-CARES prototype contains a trace simulation tool that extracts and processes information on a systems dynamics [1]. This information is visualized as a kind of collaboration diagram or sequence diagram with respect to UML [3]. The state machines extracted by our algorithm are now used to annotate the current state of a program instance at runtime to its representation during trace simulation. Furthermore, we can use the dynamic information to verify the transitions of the extracted state machines. That is, we can do consistency checks.

4. Future Work

Currently, we are implementing the state machine extraction algorithm for our re-engineering toolkit. In parallel, we are preparing the integration of the resulting state machine graphs with the structure graph and the trace simulation facility. First attempts to apply the algorithm manually to some selected programs have shown that the technique is feasible. But, we are still lacking a complex case study with (semi) automated state machine extraction. Therefore, such a case study is planned for the next stage of our work. This case study will include an evaluation of the resulting state machines by domain experts.

References

[1] A. Marburger and D. Herzberg. E-CARES Research Project: Utilizing Dynamic Information. In *Proceedings of the 3rd Workshop Software Reengineering*.

[2] A. Marburger and D. Herzberg. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 139–147. IEEE Computer Society Press, 2001.

[3] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison Wesley, 1999.