TEMPURA REENGINEERING

Vladimir Valkanov, Damyan Mitev

Abstract: *Interval Temporal Logic provides time-dependant formal description of hardware and software. Such formalism is needed for description of behaviors of the middleware of AOmLE project, depending on different scenarios of operation. In order to use ITL, we need an interpreter. Tempura provides executable ITL framework, written in C language. We cannot use Tempura as is, because AOmLE is developed entirely in Java. For this reason we need Java version of Tempura. This paper describes our plan for reengineering of C-Tempura and creating Java version if the ITL interpreter.*

Keywords: framework, reengineering
2010 Mathematics Subject Classification: 97R40

1. Introduction

The main goal of the Distributed eLearning Center (DeLC) project [1] of Faculty of Mathematics and Informatics (FMI) University of Plovdiv "Paisii Hilendarski" is creation of infrastructure for distributed eLearning [2]. The project goal is creating of a flexible, adaptive, collaborative, context-aware service and agent-oriented eLearning environment, that uses InfoStation [3] network architecture and delivers personalized, mobile, any-time and any-where assess to educational content and services.

The InfoStation architecture consists of three tiers: InfoStation Center, InfoStations and mobile devices [4]. InfoStations serve as wireless connection points and service providers for mobile devices. The operation of the InfoStation is governed by agent-oriented middleware – a multi-agent system, in which different agents will perform specific activities [5]. This middleware has two primary objectives. The first one is to manage the connection with mobile devices and the Personal Assistant agents that reside on these devices [6]. The second is to provide a platform for various (existing and new) services and to facilitate the access to these services by the mobile devices [7].

The middleware functionality is governed by scenarios [8], depending on the movement of mobile device while a service is being executed. We foresee four different base scenarios:

- The user does not change his mobile device and does not move out of range of the current InfoStation (is constantly connected);
- The user does not change his device, but moves out of range of (disconnects from) the current InfoStation and connects to another InfoStation;

- The user stays in range of the current InfoStation, but changes his mobile device (disconnects with one device and connects with another to the Same InfoStation);
- The user changes his mobile device and the InfoStation.

These scenarios mandate different behavior of the middleware, due to various factors, i.e. different device profiles (capabilities), cached information in the InfoStations, availability of services in different InfoStations and others.

We need formal description of the scenarios in order to create flexible mechanisms for scenario detection and control, and a mechanism to interpret that description. Existence of such formalism and accompanying interpreter would allow us to create new (sub) scenarios and define their corresponding middleware behavior without the need of major rewrite of the system.

In this paper we discuss why we have chosen Interval Temporal Logic for formal description of the scenarios, why we have chosen to perform reengineering of ITL's interpreter Tempura, show the main cycle of the interpreter, analyze the source code structure of Tempura, present our plan for reengineering, enlist the tools we plan to use and finally summarize the paper's contents.

2. Interval temporal logic and tempura

The most important characteristic of the scenarios is that they are time dependant; therefore it is difficult to describe them formally with first order logic. We found Interval Temporal Logic to be a perfect solution to that problem [9]. Interval Temporal Logic (ITL) is a temporal logic for representing both propositional and first-order logical reasoning about periods of time that is capable of handling both sequential and parallel composition. Instead of dealing with infinite sequences of state, interval temporal logic deals with finite sequences - intervals. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL [10].

Interval Temporal Logics find application in computer science, artificial intelligence and linguistics. Interval Temporal Logic is a specific form of temporal logic, originally developed by Ben Moszkowski for his thesis at Stanford University. It is useful in the formal description of hardware and software for computer-based systems. Tempura provides an executable ITL framework.

In order to use ITL formulae in Agent-Oriented mLearning Environment project, we need ITL interpreter. Tempura is an interpreter for executable ITL formulae, developed originally by Roger Hale and now maintained by Antonio Cau and Ben Moszkowski. Tempura was originally programmed in Prolog, but later rewritten in C. The AOmLE middleware, however, is developed in Java, so current Tempura sources can not be used directly. For that reason a Java version of tempura needs to be developed.

We were considering three different approaches in using Tempura from the Java middleware – wrapping existing Tempura executable with Java IO redirection; development of entirely new project, based on execution rules of ITL; and reengineering of Tempura and rewriting it in Java.

3. Why reengineering

We decided that reengineering of Tempura and rewriting it in Java is the best approach. There are several reasons for this. First of all the middleware of our system is developed in Java and the most natural way is to write Tempura interpreter in Java. In this way we can fully use the advantages of object-oriented architecture and Java Platform. Also a Java version of Tempura interpreter will be much more flexible and easy to reuse, which can be beneficial to future projects. On other side we have the advantage not to face some of the typical problems of reengineering like missing parts of source code or large systems which are not fully documented and etc. Also we have a permission, from original developers of the interpreter, to use C-Tempura source code.

We choose to reengineer the existing C-Tempura, as opposed to write new Java-Tempura interpreter from scratch, because we prefer to use mature system, which has proven its performance and stability through the years. The first C-Tempura interpreter was written in 1985 by Roger Hale at Cambridge University. In next years the interpreter evolves, by adding new functions and operators, thanks to Roger Hale, Ben Moszkowski and Anonio Cau. The last version of the Tempura interpreter on which we focus our attention is Tempura version 2.16.

4. Main interpreter cycle

As a first step of reengineering process we need to acquire basic understanding of Interval Temporal Logic as Tempura is made to interpret it. We focused our attention on Ben Moszkowski’s book “Executing Temporal Logic Programs”. There he describes the basics of Interval Temporal logic as well as operators of Tempura and execution algorithm of the system.

The main program cycle of Tempura is based on the ITL theory model, where formulae are executed over intervals. The way in which Tempura execute formulas is a loop, where in each step the interpreter transforms the formulas to logically equivalent conjunction of two formulas - *present_state* and *what_reamains*.

The formula *present_state* consists of assignment to the program variables and also indicates whether or not the present state is the last one. The formula *what_reamains* is a reduced form of original formula and it is executed in subsequent states if the interval does indeed continue on. In the next step of the loop, if the interval has length more than 1, the formula which interpreter execute is formula *what_reamains* from the first step.

There are four variables which are used by interpreter when execute program:

- *Program*: This variable contains the Tempura program itself. After execution of each state, the variable is transformed to form which describes what should be done in the next state.
- *Memory*: This variable is in fact an indexed list of cells, where each cell can be empty or contain values as an integer or a list descriptor. Every cell form this list is assigned the empty cell at the beginning of each state.
- *Current_Env*: This variable is also a list which describes the environment. It has separate entry for each variable in Tempura program. Every entry is a pair; first part is the name of a variable and second is an index to the memory cell where his value is placed.
- *Current_Done_Cell:* The variable is equal to an index of a memory cell called the *done_flag*. During the execution Tempura program places either *true or false* in the done flag during at every state. This indicates whether or not the current state is last one.

In terms of these four variables the executing algorithm can be presented in the following pseudo code:

```
begin
   local Program, Memory, Current_Env, Current_Done_Cell;
   prepare_execution_of_program;
   loop
          execute_single_state
   exit when Memory[Current_Done_Cell]=<true>
   otherwise
          advance_to_next_state
end.
```

In *prepare_execution_of_program* the interpreter’s variable *Program* is assigned the program’s syntax tree and the *Current_Env* is initialized to indicate references to the memory. The memory itself is allocated to have cell for each variable plus one where to be placed the value of the done flag from *Current_Done_Cell variable.*

The transformation of the variable *Program* and check to ensure that the done flag has been set to *true or false* are made in *execute_single_state.* Also the assignments in the current state are reflected in the values of the memory’s cells.

In *advance_to_next_state* if the current state is not the last, preparation are made for going to the next one. This is done by clear the content of the memory’s cells and prepares the reduced form of *Program* variable.

The transformations of the variable *Program*, which are made in procedure *execute_single_state*, are done in a loop, which executes procedure *transform_stmt(Program)*. It repeatedly transforms the program, applying special rules based on ITL theory, until the specific test shows that the program is reduced to appropriate form. Each iteration of this loop corresponds to one pass over the program.

The procedure *transform_stmt* has the form *transofrm_stmt(Statement)*. An interesting point is that the syntax of Tempra permits one statement to be perceived in different ways. For example the formula *(I=4)∩(J=1+I)* can be viewed as a statement or as a boolean test and the variable *I* can be considered as an expression or as a location. For this reason the Tempura interpreter needs different functions for reducing of statements.

5. Tempura reengineering plan

The reengineering of Tempura will be conducted in several iterations. Each iteration will consist of three main phases: research, implementation and testing. The research phase will include series of experiments of reverse engineering methodologies over a subset of the code. These experiments will show us whether the selected methodology is usable and should be applied over the entire source. The research phase will also include usage of various reengineering tools. The result of the implementation phase will be fully functional interpreter. With each iteration we will achieve source code with higher degree of compliance with the object-oriented paradigm. In the testing phase we will conduct series of tests by running the same tempura scripts on the original C-Tempura and the reengineered Java version. Needless to say, we aim at identical behavior of both systems. These tests will ensure the correctness of the reengineered code. It is important to note that our main focus is at creating correctly functioning ITL interpreter, paying less attention at optimization, memory and performance issues.

Our goal in the first iteration is to merely rewrite the original C source into Java, avoiding object-oriented features (such as inheritance and polymorphism) as much as possible and using Java OOP principles at their minimum. Unfortunately, this task will not be trivial, because in Tempura sources are used constructs of C, which are not available in Java. These constructs include macro definitions, memory pointers, function pointers, structure and union types. On the other hand, our reengineering effort will be made easier due to the fact, that the C-Tempura authors used uniform set of conventions through the source.

5.1. First Iteration

Our reengineering plan for the first iteration can be outlined with the following:

I. Research

1. Acquire understanding of Interval Temporal Logic;
2. Learn code structure, main program flow, structures and variables of Tempura interpreter.

II. Implementation

1. Create a Java class for every C source file and its associate Header file;

2. Create a static method in the Java class for every function in the C file;
3. Create static variable in the Java class for every global variable in the Header file;
4. Create a static method in the Java class for every macro function in the Header file (where appropriate);
5. Expand the contents of every macro, not covered by the previous step;
6. Replace function pointers with instances of specially created Interface, which will call the desired Java method (delegates).

After examining the foundations of temporal logic, source code of the C version of tempura and basic life-cycle of the interpreter we began the actual translation of source code from C to Java.

Following the implementation plan of the first iteration phase we completed the translation of the source files, replacing typical C structures by their respective equivalents in Java. The most common differences and difficulties in interpretation were related to the replacement of pointers used by the original developers, with structures in Java. The nature of problems in translation is caused by relatively different types of languages that we are using. C programming language has a lower level of abstraction; it is intended for programming systems with high speed and precise control of memory. It has mechanisms, such as pointers to interact and directly manipulate the memory cells. Java, which is our primary means of developing the system, is more modern and fully object-oriented programming language. This gives a lot of advantages such as components and capabilities for handling containers, which in turn enable us to create and manage our Middleware agents. Languages with high level of abstraction, such as Java, have no means of directly handling memory. Because of this a relatively small portion of code consumed a lot of time to convert C memory handling to semantically equivalent Java constructs.

Most of the code was translated easily; in order to ease this translation in the first version, we neglected some of the conventions for writing Java code. This was done in order to minimize differences in source code, which in turn enables us to easily test the Java version.

III. Testing

1. Run the examples, which come with the original sources, on the Java interpreter and compare results;
2. Contact Antonio Cau for further test cases;
3. Strategy – embed the same trace information in original tempura sources and in JTempura, then compare outcomes of execution of the same tempura expression in both interpreters.

After finishing the initial translation, we began testing Java-tempura. To be sure of the results of the test cases we used tests and scripts that we have been provided with the source code of the original developers. Tests themselves are carried by parallel execution of test scripts by the two versions of the interpreter and comparison of outcome. About 90% of test cases completed with a result identical to that of the original interpreter. It should be noted that at this stage of development we are not concerned with the speed of the Java version. Perhaps in the future we will have optimize Java-tempura, because the nature of DeLC, part of which will be the interpreter, is to provide real-time electronic services and performance is of great importance for the system and the users themselves.

After the initial tests we contacted one of the original developers of C-Tempura, Antonio Cau. He was kind enough to provide us with additional test cases, which were divided in three topics – I/O, Lists and sub lists and Miscellaneous. Using his directions we were able to pinpoint and eradicate some implementation problems.

5.2. Plans for Second Iteration

Upon completion of first Java version of the interpreter, next step is to make two versions of Java Tempura – object-oriented and agent-oriented. The initial translation is direct and it will keep almost entirely the structure and philosophy of the original project. The creation of object-oriented version will enable us to take advantage of the full benefits of object-oriented languages such as Java. Moreover, this version of the interpreter will be much more understandable, adaptable and easier to use in future projects. Agent-oriented version will be specially designed to serve the needs of the project DeLC and his agent-oriented middleware. Optimizing the performance of Java-tempura will also be important in the practical realization of the system as it currently is ignored in order to quickly achieve a working prototype.

6. Conclusion

In this paper we presented our motivation for creating Java version of the C-Tempura interpreter. We discussed why we need ITL interpreter and why we choose reengineering existing C code over creating a new project from scratch. We also described the main flow of interpreting Tempura formulae and main functions, structures and variables used in the interpreter. We proposed step-by-step plan for reengineering the ITL interpreter and rewriting it in Java. We plan several iterations of reengineering; each iteration consists of phases of research, implementation and testing. The first iteration is one-to-one translation from C to Java (preserving the procedure-oriented paradigm of the program), while the second iteration will produce object-oriented and agent-oriented version.

Acknowledgment

This publication is partly financed by project ДО02-149/08 of the Fund "Science researches" of the Ministry of Education.

References

[1] S. Stojanov, I. Ganchev, I. Popchev, M. O'Droma, R. Venkov. 2003. "DeLC – Distributed eLearning Center". In Proc. of the 1st Balkan Conference in Informatics BCI'2003, Pp. 327-336, 21-23 November. Thessaloniki, Greece. ISBN 960-287-045-1

[2] S.Stoyanov, I.Ganchev, I.Popchev, M.O'Droma, From CBT to e-Learning, Journal "Information Technologies and Control", No. 4/2005, Year III, Pp. 2-10, ISSN 1312-2622

[3] Ganchev I., S. Stojanov, M. O'Droma, D. Meere, "An InfoStation-Based University Campus System Supporting Intelligent Mobile Services". Journal of Computers (JCP, ISSN1796-203X), Vol. 2, No. 3, Academy Publisher, May, Pp. 21-33.

[4] Iacono A. L. and C. Rose. "InfoStations: A new perspective on wireless data networks. In Next Generation Wireless Networks, Defining Applications and Services for the Next Generation." Kluwer Academic Publishers, 2000

[5] Stoyanov, S., I. Ganchev, M. O'Droma, D. Mitev, I. Minov, Multi-Agent Architecture for Context-Aware mLearning Provision via InfoStations, In: Proc. of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology, October 28-31, 2008, Paris, pp.549-552, 2008, ACM 978-1-60558-046-3/08/0003.

[6] Ganchev I., D. Meere, S. Stojanov, M. Ó hAodha, M. O'Droma. 2008. "On InfoStation-Based Mobile Services Support for Library Information Systems". Proc. of the 8th IEEE International Conference on Advanced Learning Technologies (IEEE ICALT'08), Pp. 679-681, 1-5 July, Santander, Spain. ISBN 978-0-7695-3167-0.

[7] Ganchev, I., M. O'Droma, S. Stojanov, I. Popchev, "Provision of mobile services in a distributed eLearning center". Proc. of the International Conference on Automatics and Informatics, Pp. 79-82, Sofia, 6-7 Octomber, 2003

[8] Ganchev I., S. Stojanov, M. O'Droma, D. Meere. "Communications Scenarios for InfoStation-based Adaptable Provision of mLearning Services". In Proc. of the 2nd International Conference on Modern (e-)Learning (MeL 2007), Pp. 98-104, Varna, Bulgaria. ISSN 1313-0095 (paperback), ISSN 1313-1168 (CD), ISSN 1313-1214 (online). 1-7 July 2007.

[9] B. Moszkowski. Executing Temporal Logic Programs. Cambridge University Press, Cambridge, England, 1986.

[10] A. Cau , Interval Temporal Logic A not so short introduction, http://www.tech.dmu.ac.uk/STRL/ITL/itl-course/index.html (to date)

Vladimir Valkanov,
Faculty of mathematics and Informatics
236, Bulgaria Blvd.
2003 Plovdiv, Bulgaria
e-mail: assarel@abv.bg

Damyan Mitev,
"Trakia" bl. 201A, aprt. 9
Plovdiv, Bulgaria
e-mail: damyan_mitev@mail.bg