Computer Science Journal of Moldova, vol.23, no.2(68), 2015

Communicative automata based programming. Society Framework*

Andrei Micu, Adrian Iftene

Abstract

One of the aims of this paper is to present a new programming paradigm based on the new paradigms intensively used in IT industry. Implementation of these techniques can improve the quality of code through modularization, not only in terms of entities used by a program, but also in terms of states in which they pass. Another aspect followed in this paper takes into account that in the development of software applications, the transition from the design to the source code is a very expensive step in terms of effort and time spent. Diagrams can hide very important details for simplicity of understanding, which can lead to incorrect or incomplete implementations. To improve this process communicative automaton based programming comes with an intermediate step. We will see how it goes after creating modeling diagrams to communicative automata and then to writing code for each of them. We show how the transition from one step to another is much easier and intuitive.

Keywords: Communicative Automata, XML Automata, Automata transfer, Distributed systems, Traveling code.

1 Introduction

The last decades of evolution for computing machines brought a significant increase in computing power and their diversity. The rise of parallel computing, the important foundations of modern computers,

©2008 by A. Micu, A. Iftene

*This work was supported by MUCKE (Multimedia and User Credibility Knowledge Extraction) project Reference No. 2 CHIST-ERA, three years from 01.10.2012.

* This paper represents an extended version of the paper presented at FOI2015.

has revolutionized the world of software and hardware making it possible to create artificial intelligent systems. Whether it is a desktop, mobile phone, mainframe, or any other computing system, it is able to simultaneously perform a number of tasks that sometimes depend on each other. Their synchronization is essential in most cases and it depends on the states in which the processes or threads are at a time. Synchronization is not easy to achieve if the source code is not structured in terms of states. Sometimes it happens that a seemingly stable code in terms of errors to work as expected for successive runs with the same input data, but at a certain running (with the same input data) to give a wrong result.

Automata have been used since before the beginning of modern computers to solve mathematical problems. Nowadays they have applications in many of the components of a software product such as lexical analyzers, parsers which use regular expressions or network communication protocols [1]. In 2003 Russian scientist Anatoly Shalyto published an article about automata based programming [2]. This paper presents a new way of programming mechanisms for simulation of states, transitions and input/output operations. In designing of large applications state charts and activity diagrams can be used, and they are very similar to automata. The problem arises when you have to translate these diagrams into source code. Shalyto senses this transposition and connects his theory with the association between diagram elements and automata elements.

Communicative automata based programming has elements from the object-oriented version of the Russian researcher and, additionally, it solves the problems mentioned above. It proposes an improved model of the application, dividing the tasks of the control automata in the object-oriented model to several independent machines that communicate with each other. Every communicative automata is self-contained and do not share information, the only way to exchange data is the transmission of messages. So the code of applications benefits from high cohesion without sacrificing coupling and it can be reused easily.

The main strength in the technology based on communicative automata is that the application can be easily distributed across multiple

computing machines. Each system has its suite of communicative automata, which communicates with the rest of the automata by the same type of messages; in this case the communication channel is the network. Moreover, these systems can switch automata between them, which do not depend on a particular machine, providing task balancing. This is useful particularly in the client-server model, where the client is a computing machine with low capacities, such as a mobile phone. In this paper we will see how we created the premises for the communicative automata based programming paradigm, which is based on object-oriented programming concepts. This paradigm intensively uses the concept of automaton; code structure is given by the states and transitions. Novelty to classic automata based programming is to treat automata as atomic elements at application-level and the introduction of using transmission of messages between automata. Society Framework implements the basic elements and concepts described by communicative automata based programming. The framework allows creation of native and XML automata, the XML ones having an important advantage because they can be serialized locally and deserialized on another machine at runtime.

2 Communicative automata based programming

2.1 Automata based programming (classical version)

For the first time in software engineering, Shalyto describes an application model composed only from automata. Its technology uses intensive enumerations and switch-case instructions, so that is also called "Switch Technology" [2]. In this solution, although the code is easy to understand, there are big problems when the number of states increases. This is because the program code increases with each added state and transition. The solution appears later in the paradigm "object-oriented programming based on states" [2]. It combines the advantages of automata for easier understanding of the program behavior and advantages of object-oriented programming for easier understand-

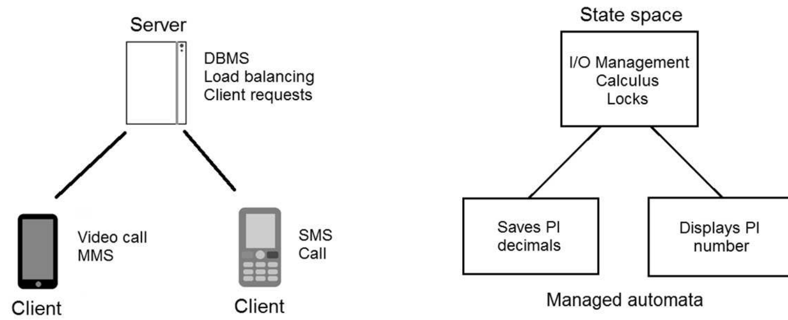

Figure 1. Comparison between client-server model and Shalyto model

ing of the structure. The new technique shifts from "switch-case" instructions to classes and objects to describe automata, thus avoiding nested switches. Shalyto takes in discussion the existence of several automata in one application. Their management is difficult when you have to synchronize certain transitions or when performing operations of reading/writing from the same memory location. Therefore the notion of "space of states" [2] arises, a set of conditions designed to control objects. Knowing which states control objects, we can program automata to have synchronous access to memory. Moreover, if we consider one object for each automaton, the space of states may act as a supervisor for the other automata. Thus, the application model begins to look like client-server model, as in Figure 1.

Another novelty in this technique is the capacity of system to respond to events, a necessary feature for communication between state space and the other automata. So, not only I/O operations can change the state of automata, but also other automata by generating its own events. Object-oriented programming based on states solves many of the initial problems presented by Shalyto, but even this option cannot be used in large projects. The main reason would be that as the code grows in size, it becomes more difficult to extend. State space must be rewritten for each new added automaton and becomes more and more complex, and harder to understand.

2.2 Communicative automata based programming

Programming based on interconnected automata expands object-oriented programming, inheriting all its elements and rules. On top of them there is the added notion of communicative automaton, with new rules related to its functionality. The major differences between the automata from object-oriented version and communicative automata are their atomic characteristic and the communication based on messages. In what follows, an automaton is a finite automaton with epsilon-transitions, without isolated states, with one or more final states and a single initial state [3].

Communicative automata bring new elements compared to the classical automata, leading to changing application architecture. The main elements are: (1) **State** – a series of instructions viewed as an atomic part. It contains code that performs the actual work of the automaton; (2) **Transition** – a series of instructions viewed as an atomic part. In contrast with states, transitions contain only the code needed to determine the next state where it will pass; (3) **Message** – an entity that contains data transmitted by an automaton to another automaton or by a code that is not part of an automaton; (4) **List of messages** – a comprehensive list of received messages by automaton.

A communicative automaton may contain, in addition to the base elements, other resources such as variables, operating system resources or references/pointers to other automata. In most cases, when a system runs more communicative automata, it is desirable to execute their code in parallel. The general solution in modern systems is to run the code for each automaton on a separate thread. Some programming languages such as JavaScript before HTML5 [4, 5], do not support working with multiple threads. If the parallel execution is not possible, the programmer will have to use an own method of allocation and arbitration of automata to the processor. This approach has an important advantage in that the programmer can choose the convenient moments when to deallocate an automaton to ensure a consistent state at each step of the execution. This approach has a disadvantage, though. At any point in execution it cannot run more than one automaton, so others

have to wait.

Each automaton must provide ways to add messages to its message list. The problem occurs when an automaton should reference another automaton, to which it must send a message. A naive way to solve this problem is to keep a reference/pointer to every possible destination, which is set by the function that creates it. Such practices, however, are extremely hard to maintain since it requires changing the code of the function that instantiates automata each time when we add a new type of automaton. A better approach is to mediate communication with an object that keeps track of automata. This object, called "router", has an implementation similar with the Observer pattern. Automata must register using a unique identifier to this mediator and must be able to handle messages from any source, automaton or not. The router methods can send messages based on recipient identifier, thus obtaining a total decoupling of automata in the system. When the router handles messages sent through the network, security problems may appear which must be taken into account. A security system must provide a separation between user automata to avoid situations in which an automaton sends a compromising message to another automaton.

3 Society framework

3.1 Architecture

Society Framework is a project developed to demonstrate the advantages of communicative automata based programming. Its source code and examples are publicly available on https://code.google.com/p/society-framework/. Its target is to facilitate the development of communicative automata based applications and to minimize the errors that can happen in such an application. There are two framework implementations, one written in Java language and one written in C# language, the reason being the demonstration of its interoperability.

The three major modules of this framework are the following: (1) **Base module for communicative automata** – contains interfaces,

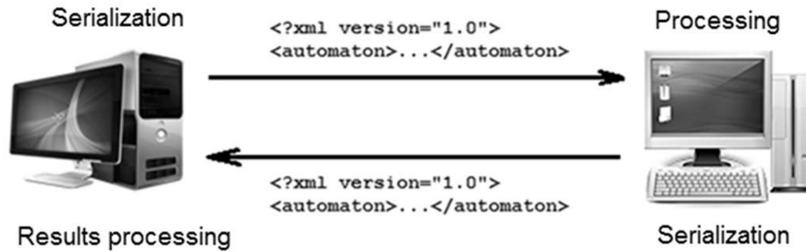

Figure 2. An efficient use of a communicative automaton

abstract classes and completely implemented classes to create a native automaton; (2) **XML communicative automata module** – contains interfaces, abstract classes and completely implemented classes to create an XML automaton; (3) **Communication module** – contains classes with role in automata communication, both locally and through the network, on different applications.

In Society the communicative automata are divided in two large categories: native automata, with Java or C# code based on the framework implementation and XML automata, for which the code is written using an extension of XML. The main reason is the fact that, unlike the native automata, the XML ones can be serialized, sent through the network to another application and re-instantiated on that machine, the process being intuitively described in Figure 2. The XML code inside the serialization is transformed in one more object trees representing the instruction that must be executed in the states and the transitions. For this reason we cannot say that the framework compiles the code and neither that it interprets it. It constructs its own code using objects corresponding to the instructions described by the XML.

XML automata can use only a restricted set of instructions, on which the framework can construct XML specifications. The reason is the fact that XML automata are not intended for complex or intense processing due to the great overhead compared to the native ones. Their utility is the fact that they can communicate with the na-

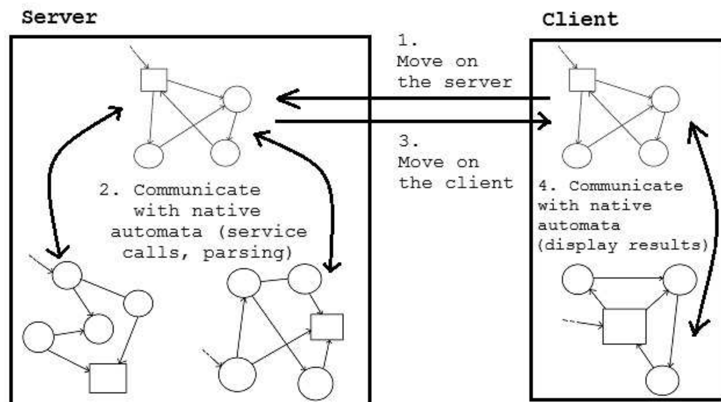

Figure 3. Example of optimization using a native automaton and an XML automaton

tive ones through messages, the latter fulfilling the tasks described in the messages much more efficiently. The alternative usage of the two types enables optimizing the application processing, an example being described in Figure 3.

The efficiency relies on the fact that the only data sent between the server and the client is the automaton serializations. Thus, the transfer of data between server and client is minimized and it is replaced by service calls, the result being a constant number of connections to the server for executing a task. Fortunately an XML automaton plays the role of manager for the native automata in the system and they don't require much code, so traveling to another machine through network doesn't imply a lot of data transfer. Another major advantage in this approach is the reduced overhead of the native automata because, as we previously mentioned, the native automata are written directly in the language/platform of the framework implementation. The fact that a native automaton can use any instruction of this type raises security problems regarding the actions permitted to XML automata. XML automata can only execute instructions that don't compromise security, the only way of communicating with the machine it runs on being the

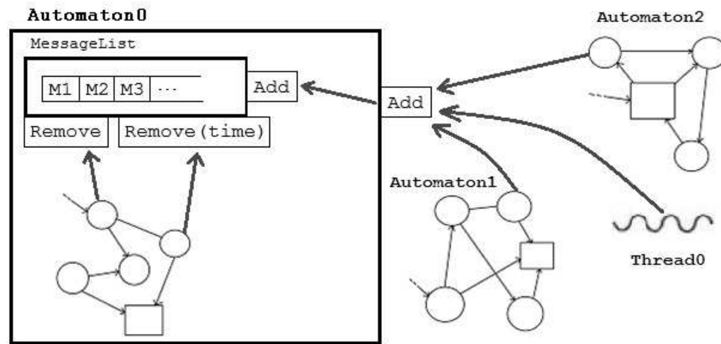

Figure 4. Interaction with the message list (queue)

message sending to other automata.

3.2 Base module for communicative automata

The base module contains the base class for all the automata created in a system (BaseAutomaton) and classes for automaton components. In both framework implementations these are: (1) **State** – the base class for the states in an automaton; (2) **Transition** – the base class for the transitions in an automaton; (3) **TransitionGroup** – class that contains transitions and acts like a normal transition; (4) **MessageList** – the class that implements the queue of messages received by an automaton.

The State, Transition and TransitionGroup classes are classes inside the BaseAutomaton class. The reason is the fact that these classes must have access to the fields and the methods in BaseAutomaton.

The MessageList class implements a special type of queue with synchronized add and remove methods (see Figure 4). The add operation is synchronized to ensure consistency to the list while more than one execution threads add messages simultaneously. The remove operation is synchronized to block the thread that calls it when there are no messages in the queue until another thread places a message in it.

The message queue is encapsulated in the automaton, the only permitted operation being the message addition by using the Add method at automaton. Inside the State, Transition and TransitionGroup classes there is direct access to the MessageList object, meaning that they can both add and remove messages.

Each state contains a transition or a transition group which indicates the next state and is kept in an indexed list (HashMap in Java and Dictionary in C#) with string identifiers. To create a new state the State class must be extended and the stateCode (or StateCode in C#) method must be overridden. Then the Transition class must be extended (or TransitionGroup if the state has more than one transitions) and the transitionCode (or TransitionCode in C#) method must be overridden, such that the new implementation returns a valid transition name. The next executed state is decided by the name returned by the transition. Societys communicative automata can run both on the current thread, by calling the run (or Run in C#) method, and on a newly created thread by calling start (or Start in C#). To stop the thread that runs the automaton without risking data corruption the stopSafely (or StopSafely in C#) method can be used. The run method contains a while loop that ends at the stopSafely call. This loop executes states one at a time and the order of execution is driven by each states transition or transition group.

Another important class in the base module is the SocietyManager which manages the automata inventory from the current application and the XML automata transfers. This can be extended to implement the saving and loading methods for the automata. For automata transfer SocietyManager runs a special native automaton called AutomataTransferAutomaton responsible with managing the connections and sending/receiving automata.

3.3 XML communicative automata module

XML communicative automata are an extension of the native ones, their base class being BaseAutomaton. The XML automaton name is given by the serialization and deserialization of this type, which is

achieved by using the Society XML, an extension of XML.

The serialization of XML automata contains 6 major elements: (1) **Automaton name** – also named identifier, it is the character sequence attribute of the < *automaton* > element with role in a possible subscribe of the automaton at the message router; (2) **Current state name** – current state identifier, also stored as a character sequence; (3) **Current message** – the serialization of the last message pulled from the message queue; (4) **Message list** – the serialization of the automaton's message queue; (5) **Variables** – the list of variables and their values; (6) **States** – the list of states in the automaton and their code.

The current state name must match the name of a state in the state list. If this rule is violated, then the automaton cannot start. Also, if the name which was provided for the automaton serialization differs from the name it had in the system before, when the automaton is re-instantiated, all entities which send messages to that automaton must be aware of the change and send message for the new name.

Variables are key-value pairs. In the Society framework automata work with 5 data types: Boolean, Integer, Double, String and Map. The Boolean type is represented using characters delimited by dots: .T. for true and .F. for false. Integer and Double types have the same representation as in Java or C#. String type is delimited by apostrophes and it can contain anything exception apostrophe characters. Map type can represent vectors with any number of dimensions; the only limit is the machine memory. To ensure this property, the framework uses a series of indexed lists (the type of list depends by framework implementation). The indexed values can be of any type, including Map type. Thanks to this flexibility we can create vectors that contain other vectors, any number of times and in any combination, the result being as much dimensions as the memory can hold.

The state serialization contains the state identifier (name attribute), the executed code (< *code* > element) and the transition (< *transition* > element) or the transition group (< *transition_group* > element). A transition group contains any number of transitions and a < *code* > element which contains the instructions that manage the

returned values for each transition. From the moment when the automaton is started, the initial state code is executed indicated by the value in the *current_state* attribute. Then the transition code from that state is executed (or the transition group code if it's a group of transitions) and the next state name is obtained. The process continues until the automaton is stopped, either from a state code, or from an execution thread outside it.

It should be noted the fact that inside a transition group each transition must have a name (specified in the name attribute), so that their code can be called from the $< code >$ element of the group. If a state has just a transition, and not a transition group, then that transition doesn't have to specify a name. The instructions represent the imperative part of Society XML and there are two types of them: (1) **Simple instructions** – instructions that don't contain other instructions inside them (empty elements); (2) **Compound instructions** – instructions that contain other instructions inside them (non-empty elements).

Simple instructions are the base for the code executed in the states and transitions. These are represented by XML elements without content, the only parameters being their attributes. The following simple instructions can be used in Society XML: *get_next_message*, *send_message*, *execute*, *continue*, *break* and *return*. Compound instructions are usually loops (*while*, $do - while$, for), but they can be other types, like the $if - else$ instruction or the $switch - case$. Their behavior is identical with the one in the framework's implementation language, with small differences to enable much more flexibility by using the expressions. The following compound instructions can be used in Society XML: *while*, *do_while*, for, if, *else*, *switch*, *case* and $default$.

Unlike the other elements, where the declaring order does not drive the code behavior, instructions must be written in the exact order in which they must be executed. Regarding the calculability of Society XML, it contains enough instructions to be Turing-complete: at least one assignation operation, one conditional operation and one jump instruction. This means that XML communicative automata can solve any problem which can be transformed in an algorithm. The input and

the output are ensured by the router and the message queue inside the automaton. Native automata must provide communication methods with the user for the XML ones because the language of the latter does not allow native calls for reading, writing or displaying data. The advantage is the fact that XML automata don't have any security issues so long as the native ones verify and control the requests. The expressions have an important role in Society XML because they provide values for the attributes in the instructions presented earlier. An expression will always return a value, even if this value is null, and some expressions may even change the state of variables during the execution, like the assign operation or the function call. The expressions may also be used to access the values of the received messages.

XML communicative automata deserialization

The BaseAutomaton class incorporates methods to serialize and deserialize automata. For parsing the XML data the Society framework uses SAXParser in Java and XMLReader in C#. For constructing the objects that compose the XML automaton functionality the framework uses a special automaton called DeserializationAutomaton. For each beginning and ending tag the parser sends a message to the deserialization automaton containing the corresponding data (tag name, attributes, and tag type). According to the state and message data the automaton will create the objects and perform transitions.

The major components in the XML automaton serialization are the following: the variables, the messages, the states, the transitions and the transition groups. When the deserialization automaton encounters an expression inside an attribute it must analyse it and construct an instruction tree equivalent. The constructExpression method inside DeserializationAutomaton has a string representation of the expression as input and an instruction tree as output. It uses an algorithm inspired from the infix to prefix expression transformation algorithm [6]. Instead of constructing the list of prefix ordered symbols the algorithm was modified to construct the instruction tree. To extract the symbols (tokens) from the expression the deserialization automaton uses the LexerAutomaton. The lexer provides these symbols as instructions. The paranthesis (OpenedBracketInstruction and Closed-

BracketInstruction) are also considered instructions, though they only have functionality inside the deserialization process. Because both the expressions and the instructions implement the same interface, Instruction, they are treated in the same way: instructions are executed by calling their code method and, for their part; the instructions call the same method for the expressions they contain. The call of a state or transition is done by only calling their code method, as the other calls are done through call chaining.

The lexer automaton is similar to the automata constructed for regular expressions. Its implementation has a string as input and for each of its runs it provides an Instruction object in the final state. This object corresponds to the next symbol found in the expression string, therefore situations when the lexer automaton must run multiple times on the same expression are often. If the automaton reaches the final state and provides a null value it means no more symbols can be found in the current expression and the constructExpression method returns the created tree. Lexer Automaton holds data about the current parse at each run. These pieces of information are named accumulators and stored in a list. In the final states of the automaton these are used to construct the returned instruction. The first accumulator is always the string of the expression to parse. There are cases when the lexer will also call the constructExpression method to create a subtree of a substring in the expression. An example would be the function parsing: for each substring delimited by parenthesis and commas constructExpression is called to obtain the subtrees corresponding to the parameters.

XML communicative automata serialization

XML automata serialization is a much simpler process thanks to the tree structure of the instructions inside them. The serialization process implies the construction of the XML based on the objects inside the automaton. To achieve this it is necessary to inspect the variables, the message list, the special members (automaton name, current message, etc.) and a single BFS traversal of object trees in the states, transitions and transition groups.

The expression serialization is an exception from the BFS: an expression tree is traversed in-order. The reason for this is the fact that

the expressions linear structure requires the left subtree of a node to be written before the operator and the right subtree to be written after the operator. To send serialization through the network, Society framework uses TCP connections which it tries to keep alive during the execution. The non-ASCII characters are encoded using UTF-8. This means that the values and names in an XML automaton can use any character from Unicode.

3.4 The communication module

The communication module includes the classes responsible with sending and routing the messages: (1) **Message** – the class which represents a message; (2) **MessageRouter** – the class responsible with message transfers.

The Message class contains two fields (or properties in C#): from and data. The from field holds the identifier with which the sender automaton has subscribed to the router or any other name if the message was not sent by an automaton. The data field is a reference to the sent object and it can be of any type. The MessageRouter class manages the message transfers for both automata in the same application and automata on different machines. It implements the Observer, Singleton and Lazy Initialization patterns and its unique instance can be accessed anywhere in the code. Automata can subscribe to receive messages using the subscribe(String name, BaseAutomaton automaton) method and they can unsubscribe through the unsubscribe(String name) method. The name parameter will have the value of a unique identifier for that automaton, usually its name. The sendLocal(String to, Message message) method sends the message provided as parameter to an automaton in the current application. It returns whether the automaton was found in the application and, in case it was found, the message is added to its message queue.

To send messages to an automaton outside the application the router uses an automaton which is responsible with the message transfer through the network called NetworkMessagingAutomaton. This looks similar to the DeserializationAutomaton inside the SocietyMan-

ager class, the only difference being the type of sent information. The network messaging automaton contains a message queue from which messages are sent starting from the moment when the connection is established with the application in the network. To place a message in this queue the sendRemote(String to, Message message) method is used. The send(String to, Message message) method first tries to send the message locally, then, if the recipient automaton is not found in the current application, it places the message in the NetworkMessagingAutomaton's queue.

4 Comparisons

4.1 Loose code vs. native communicative automata

Loose code means any object-oriented code written without the constraints of the communicative automaton based programming paradigm. By applying these constraints to loose code the native communicative automatons can be obtained, the performance difference being minimal.

To create a native automaton the following steps must be followed: (1) **Extending the BaseAutomaton or Automaton classes** – if the automaton state doesn't have to be persisted, then the Automaton class is used, otherwise the BaseAutomaton class is extended and the serialization/deserialization methods are implemented; (2) **Adding the member variables** – necessary for the automaton functionality; (3) **Creating the nested classes** – corresponding to the states, transitions and transition groups; (4) **Instantiating and adding the previously created classes at the current automaton** – these steps can be made in the automaton constructor.

The code executed in the stateCode and transitionCode methods by the automaton is the imperative (procedural) code corresponding to the language of the framework implementation. The automaton code, as a whole, has a structure enforced by the programming paradigm: it is grouped in states, transitions and transition groups.

The run method (or Run in C#), which was previously mentioned

in this article, is responsible with the correct execution of the automaton regarding the order in which the states, the transitions and the transition groups are executed. The management instructions in this method have O(1) complexity, except the operation that searches a state in the state set. After a transition returns an identifier, in the run method, a search for the state corresponding to that identifier is attempted. This implies a get operation in a HashMap (Java) or Dictionary (C#) with a complexity in the worst case scenario of O(n) [7, 8], where n is the length of the identifier hash. Though the complexity in the general case is greater than when using normal vectors, the programmer can minimize the execution time by offering identifiers with a minimum number of characters. The execution time also depends on the hash algorithm on which the search is based (specific for the platform).

While the automata are designed, the programmers and architects must decide how to split the automaton tasks and what are their states and transitions. Fewer automata and states/transitions means less allocated memory and less time consumed by the context switches between automaton threads. This approach implies more code written per state or transition, therefore the imperative part of the automata is favored. On the other hand, more automata and states/transitions mean a better modularization of the code and the possibility to allocate the tasks to a greater number of processors or machines, favoring the declarative part of the automaton.

4.2 Native communicative automata vs. XML communicative automata

The XML communicative automata, as mentioned in the previous chapters, are an extension of the native communicative automata. Their states, transitions and transition groups are constructed in a certain manner to ensure they can be serialized and deserialized using the Society XML language. The code inside the XML automaton is composed of an object tree and the objects are implementing the Instruction interface. Executing its code means calling the code method of the root

object which will trigger directly or indirectly the call of code in the other objects from the tree.

The overhead compared to the native ones is visibly greater because each instruction implies a method call at the level of implementation. Starting from the moment the automaton tries to assign a value to a Map object at an inexistent level, the algorithm creates the necessary levels based on the indexes from the left operand. If on one of the levels that must be created there is an object of a type different from Map, then it is replaced by a new Map object.

In the native automata the variable access has O(1) complexity thanks to the fact that they are direct members of the automaton class. XML automata keep all the variables in an indexed list, the same way the states are stored, therefore the access algorithm complexity is O(n), where n is the length of the variable name hash [7, 8].

When comparing XML automata and native automata it can be inferred that the native ones must be used for intense processing and system calls and XML ones must be used for the business logic, control and code that must be transferred between applications. This way we can take advantage of both without sacrificing execution time or code modularity. The connection between the two types of automata is the messaging which ensures a uniform communication. Because Society framework was written using just the base platform for each language it was implemented in, there are no additional dependencies for it to run. An advantage of this decision is the ease of extending the framework for the Android platform. In Android the Java code runs on a special virtual machine called Dalvik [9]. To run the intermediary Java code (Java byte-code) it is transformed into intermediary Dalvik code (Dalvik byte-code) for optimizations [10]. This way the Android extension for the Society framework was created which contains specialized classes for that environment in the society.framework.android package. The ActivityAutomaton class is an extension of the Automaton class which contains a reference to the current Activity in the application instance. The reference to the Activity is necessary for some actions on the application resources: user interface changes, system resources usage, service starting, etc. The AndroidSocietyManager class extends

the SocietyManager class and implements the serialization and deserialization methods for saving and loading the automata state from a persistent environment.

5 Conclusions

Communicative automata based programming makes the process of moving from the design to the implementation easier and the state or the activity diagrams to be found directly in the source code, without the need of a detailed documentation. The paradigm combines both imperative programming elements and declarative elements for obtaining a higher quality code with less effort. The new features it brings on top of the classic automata based programming, automaton atomicity and messaging communication, enforce practical rules with an aim for minimizing the errors: code modularization, data encapsulation at automaton level and request verification. These advantages have a great impact in production, especially in large projects where the work is assigned to a great number of programmers and architects.

Society framework has reached its purpose: the demonstration of the advantages brought by the communicative automata based programming. The differentiation between native automata and XML automata resulted in the development of two categories of automata, each with its advantages and disadvantages. The native ones are intended to provide methods to access the machine resources in a controlled and efficient manner and the XML ones have the role to use these functionalities, to execute the business logic and to travel in the network at the most suitable place for a certain task. The alternative and efficient use of those has results superior to the current approaches for some problems: minimizing the number of connections and the amount of data sent through the network, efficient execution of a distributed application or providing universal processing services on a powerful machine.

The Society framework, though it is not the most efficient implementation of the mechanisms in the communicative automata based programming, it draws closer to the industry needs. Large distributed

applications or the ones that intensively use network transfers can be boosted by Society framework. Nonetheless, based on the application necessities, different mechanisms can be implemented for the automata. The target would be code optimization, security (message encryption, authentication, error tolerance, etc.) or the sending of messages through other environments (embedded systems, Bluetooth).

References

[1] E. Gribko. *Applications of Deterministic Finite Automata.* (2013).

[2] A. Shalyto. it Technology of Automata-Based Programming. (2004).

[3] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation, Second Edition.* Addison-Wesley, (2000).

[4] J. Edwards. *Multi-threading in JavaScript.* (2012).

[5] R. Gravelle. *Introducing HTML 5 Web Workers: Bringing Multi-threading to JavaScript.* (2012).

[6] S. Singhal. *Infix to Prefix Conversion. Sharing ideas, Sharing experiences.* (2012).

[7] Arno. *HashMap vs. TreeMap.* (2010).

[8] K. Normark. *Generic Dictionaries in C#.* (2010).

[9] J. Hildenbrand. *Android A to Z: What is Dalvik.* (2012).

[10] Security Engineering Research Group, Institute of Management SciencesPeshawar, Pakistan, *Analysis of Dalvik Virtual Machine and Class Path Library.* November (2009).

Andrei Micu, Adrian Iftene,
Received September 20, 2015

Andrei Micu, Adrian Iftene
Institution: "Alexandru Ioan Cuza" University
Address: General Berthelot, No. 16
Phone: 004 - 0232 - 2011549
E–mail: andrei.micu@info.uaic.ro, adiftene@info.uaic.ro