Domain-Specific Language Agents

Merik Meriste[1], Tõnis Kelder[1], Jüri Helekivi[1], Leo Motus[2]

[1] University of Tartu, Estonia
{Merik.Meriste, Tonis.Kelder, Jyri.Helekivi}@ut.ee
[2] Tallinn Technical University, Estonia
Leo.Motus@dcc.ttu.ee

Abstract. Multi-agent systems are proposing a new design concept for software – the kenetic software design – based on the concept of interactive agents. On the basis of this conceptual view, interacting agents appear as an appropriate conceptual tool for the development of domain-specific languages. Domain-specific language will be considered as a consensual collection of interrelated autonomous notions and, the language processor as a cluster of interacting agents i.e. a *multi-agent* representing language notions. This paper suggests an agent-based modeling framework as a possible methodological basis for DSL design and development.

1 Introduction

An interesting aspect in domain-specific languages (DSL) is that of formal models and frameworks for DSL design and for the development of appropriate models of language notions and language constructs. To what extent the formal methods applied support the reasonable structuring of information objects and problem solving, is a crucial aspect of modeling. The efficient implementation and accordance of the model with problem area specification language concepts are usually valued most in this process. A reasonable framework is expected to provide tools for DSL modeling and implementation on a conceptually clear basis. The variety of contextual views and application areas accentuate the complicated nature of the task. The success of system modeling depends on the extraction of surface and/or deep (regular or contextual) substructures from the information environment and their subsequent attachment to computational activities. Multi-agent systems are proposing a new design concept for software – the *kenetic program design* – based on the concepts of interactions and agents. In this paper we consider the idea of kenetic design of DSL processors. Domain-specific language is treated as a collection of interrelated autonomous notions and, its language processor as a cluster of interacting agents i.e. a *multi-agent* representing the domain-specific language by notions.

On the basis of this conceptual view the interactive attributed automata concept is considered as a potential particular framework for DSL modeling and implementation. Attributed automaton is a state transition machine introduced and applied initially for purposes of structuring and conceptual specification of knowledge on the basis of regular attributed structures [1,2,3,4,5,6,7,8,9,10]. To support the decomposi-

tion of a specification or computational task into a network of components, an attributed automaton has finite memory distributed to its states, as well as computations and communication actions at transitions. Interactive compositions of Attributed Automata (AA) serve as a tool while treating AA as complex computational agents. Basic interactive AA composition techniques are implemented in interactive AA visual development environment prototype described in this paper. Attributed automaton and interactive attributed automaton were later on identified as a kind of Wegner's sequential interaction machine and multi-stream interaction machine respectively [11,12,13].

The integrated presentation and specification of knowledge will obviously remain a research and technological development problem, for DSL too. Thus, on the one hand, the development of models for conceptual, expert and procedural knowledge, as well as of appropriate methods for knowledge presentation and management is essential for the successful use of definite DSL in practice. On the other hand, we would like to focus in this context on the domain-specific language as an environment for modeling and managing information objects and problem solving actions for the presentation of linguistic notions. The next section will consider different views of a domain-specific language – as a problem-solving environment (i.e., a kind of community of practice), a collection of intelligent agents [14], and as a cluster of interaction machines. Section 3 will present the AA concept. Section 4 will concentrate on the interactive AA approach as a DSL development and implementation tool based on the paradigm of interactive computations.

2 Agents, Interactive Computations and Communities of Practice

Design and implementation methods for software-intensive systems have undergone a remarkable evolution during the last decades. Algorithmic approach in computer science and software engineering has been substituted by interaction-centered paradigms. In artificial intelligence a similar shift, from logic-based to agent-based models, has taken place. Interaction-centered systems appear to be more powerful problem solvers than algorithm centered. It is also suggested that algorithmic models alone do not suffice to express certain expected aspects of behavior of today's systems – e.g. ability to self-organize the interaction of its components, to adapt its behavior to the changes in its environment, or in its goal function. Multi-agent approach or *kenetics* [15], is proposing a new design concept for software – *the kenetic program design* – based on the concept of interacting autonomous agents. Conceptually, each component of a system can be envisaged as *an agent* with its skills and its goals and with all the agents attempting to respond the needs of the user of the system.

Interacting autonomous agents as a paradigm for software design suggest the obvious idea to model DSL as a collection of agents, i.e. as a distributed (artificial) intelligent system. Each component of a system can be envisaged as *an agent* with its skills and its goals and with all the agents attempting to respond the needs of the user of the system. This interesting approach leads to the problem of kenetic design of a DSL processor. We will consider a (domain-specific) language as a collection of interrelated autonomous notions and, the language processor as a cluster of interacting per-

sistent agents i.e. a *multi-agent* of language notions. We consider that kind of modeling framework as a methodologically suitable basis for DSL design and development tools

On the other hand, a (domain-specific) language certainly is a consensual collection of notions which forms the basis of the problem solving environment for particular communities of practice. The concept of c*ommunity of practice* is a central concept of the theory of social learning [16]. In this approach, the primary unit of analysis is neither the individual nor social institution but the informal community of practice that is formed for some activity. The particular problem-solving environment (here DSL) will serve as a handy tool to the extent that it simulates and supports the basic notions, principles and extensibility of the language and, thereby, integrates the key components of these communities of practice.

In terms of interactive computations, the computational agents and communities of practice are a kind of interaction machines, as introduced by Peter Wegner. "Though interaction machines are a simple and obvious extension of Turing machines, this small change increases expressiveness so it becomes too rich for mathematical models. Interaction machines have single or multiple interaction streams and synchronous or asynchronous actions and can differ along many other dimensions" [11].

In general, interactive computation involves, unlike algorithmic computation, dynamic interaction streams of computational agents [11,13]. Such a computation can be considered as a kind of distributed control of information streams and agents activities in a dynamic environment [12]. A cluster of agents often shows a behavior that is rather complex and organized, despite the simplicity of each single agent. In this aspect, explicit application of interactive computations leads from systems with algorithmic behavior to systems with either sequential interactive behavior (a pair of interacting agents) or with multi-interactive behavior. This observation conforms perfectly to the software engineering principles of programming-in-the-large and programming-in-the-small. Moreover, the agent-based approach in (domain-specific) language design and implementation seems extremely intriguing in the context of these observations.

3 Attributed Automata

Attributed automata were introduced as a model of executable specifications based on regular structures with attributes attached to structure nodes. Regularity is here treated in terms of formal languages – primitive items can be composed into a structure by means of concatenation, selection and iteration operations. Attributes serve for the presentation of contextual relations, as well as of properties and meaning of underlying concepts.

Attributed automaton is a state transition machine with distributed finite memory at its states and specified at its transitions computations and communication actions. In this aspect, an attributed automaton is simply a generalization of a traditional finite automaton with attributes and computational relations attached to states and transitions of the automaton, respectively. Attributed automaton in terms of formal lan-

guages is considered as a recognizer based on regular data structures, the respective class of formal grammars is that of regular attribute grammars [17].

Consider examples of syntax-directed recognizers for binary numbers (Fig.1) and for a context-sensitive language (Fig.2). Attributed automaton (Fig.1) recognizes binary numbers and the final attribute value *a* represents the decimal value of the binary number. In another automaton (Fig.2) attributes are in a different role, they collect contextual information used in some states to select the next transition. As our examples demonstrate, there are alternative possibilities to construct traditional syntax-directed compilers on the basis of simple regular attributed structures.

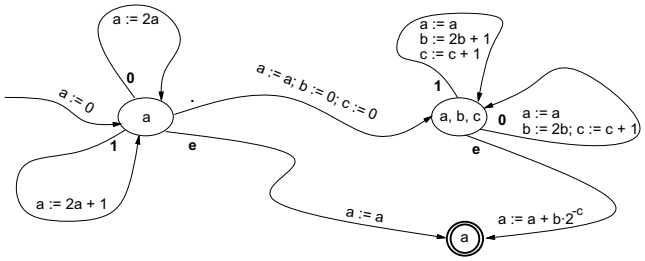


Fig. 1. Recognizer of binary numbers.

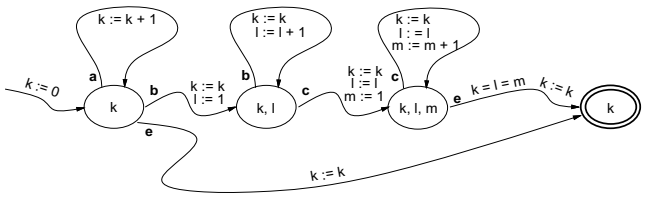


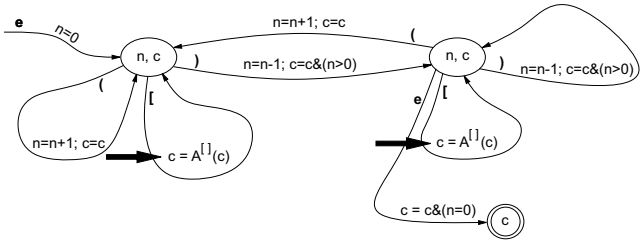
Fig. 2. Recognizer of the context-sensitive language $L = \{ a^n b^n c^n \mid n \geq 0 \}$.

There exist several extensions of the concept of finite state machine with memory. Attributed automaton can be distinguished among them by distributed memory, i.e. by allocating memory to the states of the computation. Distributed memory together with the local definition of data transformation functions helps to decompose/compose an attributed automaton. Note that hierarchical composition functions as a tool for the adequate modeling of hierarchical data structures and hierarchical computational structures. This idea is rooted in the interpreting automata concept, applied in Vienna method for defining programming languages [18].

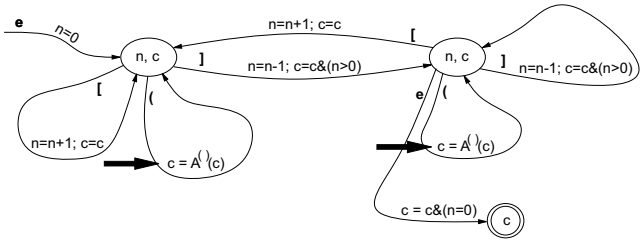
In the modeling of interactive systems it appears important for the system to react adequately to the changes in its environment. These changes cannot be predicted, such a system is inherently *interactive*, i.e. responds to changes in its environment by performing *internal* changes, which, in turn, will be registered by the environment as some internal events of the system. We call AA simulating interactive systems *interactive AA*. In these automata, the sequence of internal events (transformations of at-

tribute values) will be in some manner synchronized with external events in the environment. Interactive attributed automata represent a certain kind of *multi-stream interaction machines* introduced by Wegner [11,12,13].

For example, the parsing of Dyck languages [19] is solved by interactive AA as follows (Fig. 3). The regular structure of a string is represented by the moves of AA, counters of parentheses are represented as attributes. An interactive automaton is constructed for counting certain kinds of parentheses (‘[]’. ‘()’, ‘{}’). Automata interact with each other for recognition of a string of parentheses.



a) Accounting of parenthesis '(' and ')': c' = $A^{()}$ (c)



b) Accounting of parenthesis '[' and ']': c' = $A^{[]}$ (c)

Fig. 3. Grammar: S —> SS | ( ) | (S) | [ ] | [S]

4 Interactive Modeling of Languages

The considerations described above led us to the design and implementation of a system for the visual development of interactive AA. Such a development environment will serve as an instrumental tool for the design, as well as for the implementation of various applied software. To quote Peter Wegner [11]: “The “negative result” that interactive behavior is not expressible by Turing machines determines a “positive challenge” to develop practical models of interactive computation”. Moreover, we are encouraged by our pilot practice experience. Interactive AA compositions are expressive in solving non-algorithmic problems. Some algorithmic problems can be more efficiently solved by interactive techniques.

The basic interactive AA composition techniques are implemented in a prototype system programmed in Java. The communication technique applied is that of Java Message Service (JMS). Automata are interacting by sending messages, both point-to-point and publish-and-subscribe methods are available. Sending a message is treated as a separate process that can affect the evaluation of attributes for the next state. A message can be accepted either in the synchronous mode – acceptance included in the selection process of the next state or, in the asynchronous mode – an arriving message initiates a specific separate process of acceptance, which in its own turn may affect the values of attributes.

The collection of interactive automata designed to solve a problem forms a cluster of agents, i.e. a multi-agent. Member agents of a cluster may be activated as one complete task on a computer or, as a distributed task. In the first case, agents can exchange information by common (cluster) attributes and messages, in the second case only by messages. A cluster of agents often shows a behavior that is rather complex and organized, despite the simplicity of each single agent. In this aspect, application of interactive AA (as interaction machines) leads from systems with algorithmic behavior to systems with either sequential interactive behavior (a pair of interacting agents) or with multi-interactive behavior. This observation conforms perfectly to the software engineering principles of programming-in-the-large and programming-in-the-small. Moreover, the agent-based approach in (domain-specific) languages design and implementation seems extremely intriguing in the context of these observations.

Our prototype system of interactive AA (an agent-based problem-solving environment) is, in a sense, a system of programming where the software is constructed by means of collaborative (interacting) computational agents. Components constructed are saved as items of the common database of automata – agents. From the specification of agents of a particular task, a Java-program will be compiled. Notions and terms applied in an automaton are specified by means of the so-called *axiom-classes* (specific Java-classes, representing notions of the problem domain). Notions are implemented as particular *notion-classes* derived from axiom-classes. The application is derived from particular notion-classes and compiled in the context of constructs (terms and notions), specified by the user. Such a style of implementation supports the system's flexibility – by changes in notion-classes new properties can be introduced to the cluster as a whole. On the other hand, at some level, Java-programming is needed.

From the viewpoint of language implementation we take a 'notion view' of DSL design, in that a language is designed as a set of interrelated autonomous notions. A language processor will be constructed as a cluster of interactive AA (agents) of language notions. An agent "represents" an instance of a particular notion, i.e. the notion's representation, its structure, properties and meaning. The task of the notion agent is to secure the appropriate translation/interpretation of every notion instance in its given specific context. The contextual and structural relations of the notions included in a language are specified in terms of the properties of the notion agent and its interactions with other notion agents. If necessary, the notion agents will apply other agents for traditional subtasks of syntax-directed translation and code generation. In other words, an implementation of a language is specified as a multi-stream interaction machine [12, 13]

As an example of the multi-agent approach for DSL in the framework of interactive attributed automata, let us consider a tiny interaction language for an online ticket sales system. The example problem is borrowed from [20]. Customers buy tickets from a ticket server. The server communicates only with sales agents. Customers can ask the agent for various services. These services include reserving tickets, paying for and getting the reserved tickets, and canceling reservations. The language of communication between the customers and agents consists of a couple of notions only, as given in figure 4. Implementation of the language by automata is given in figure 5.

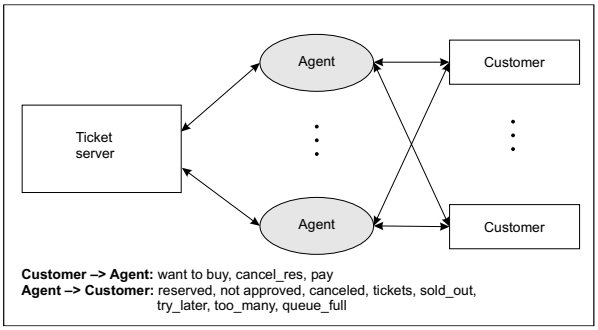
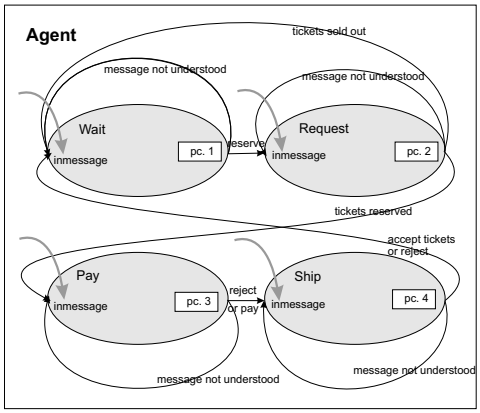


Fig. 4.



Precondition 1: Wait

server failure	==>	Wait
want to buy	==>	Request
otherwise	==>	Wait

Precondition 2: Request

tickets available	==>	Pay
tickets sold out	==>	Wait
otherwise	==>	Request

Precondition 3: Pay

request accepted	==>	Ship
request rejected	==>	Ship
otherwise	==>	Pay

Precondition 4: Ship

done	==>	Wait
otherwise	==>	Ship

Fig. 5.

5 Conclusion

During the last decade the paradigm for designing software systems has gradually shifted from the algorithm-centered to the interaction-centered approach. Idea of agent-oriented software engineering is rapidly spreading. The crucial methodological aspect of modeling software systems is to what extent the formal methods support the reasonable structuring of the problem domain. Interactive agents appear as a feasible basic framework for the design and implementation of (domain-specific) languages. Interactive translation models enable to create a semantically richer modeled world, also for domain-specific languages. As any other artificial languages, the DSL, too, are first only partially designed, evolving gradually in the process of communication and observation. As an interactive model, it leads to better structured system, better expressiveness and reduces complexity. On the other hand, the incompleteness of the model is inherent price for the freedom of multi-agent design, i.e. it remains an art of translation

References

1. Meriste, M., Penjam, J.: Attributed Finite Automata. In: *Proc. of Int. Workshop CC'92 on Compiler-Compiler*, Reports of the University of Paderborn 103, 1993, pp.48–51.
2. Meriste, M., Penjam, J.: Toward Knowledge-based Specifications of Languages. In: J.Barzdins, D.Bjorner (Eds.), *Baltic Computer Science*, LNCS, 502, Springer Verlag, 1991, pp.65–76
3. Meriste, M., Penjam, J.: *Attributed Finite Automata*. Res. Rep. CS23/91, Institute of Cybernetics, Estonian Academy of Sciences, Tallinn, 1991, 15p.
4. Meriste, M., Penjam, J.: Attributed Models of Computing. *Proceedings of the Estonian Academy of Sciences. Engineering*, 1(2), 1995, pp. 139–157.
5. Meriste, M., Penjam, J., Vene, V.: Models of Attributed Automata. *Informatica*, 9(1), 1998, pp.85–105.
6. Juhola, M., Meriste, M.: An Attributed Automaton for Recognizing of Nystagmus Eye Movements. *IAPR Papers on Structural and Syntactic Pattern Recognition*, Bern, 1993, pp.194–206.
7. Grönfors, T., Meriste, M.: Attributed Automata in Pattern Recognition of Digital Signals. *Computer Methods and Programs in Biomedicine*, 1993, pp.763–785.
8. Koski, A., Juhola, M., Meriste, M.: Syntactic Recognition of ECG Signals by Attributed Finite Automata. *Pattern Recognition*, 28(12), 1995, pp.1927–1940.
9. Grönfors, T.: *Novel Methods of Syntactic Pattern Recognition for Peak Detection of Auditory Brainstem Responses*. Doctoral Dissertation, University of Kuopio, Publications C. Natural and Environmental Sciences 28,1994.
10. Koski, A.: *On Structural Recognition and Analysis Methods Applied to ECG Signals*. Doctoral Dissertation, University of Turku, Computer Sci. Res. Reports R-97-1, 1997.
11. Wegner, P.: Why Interaction is more Powerful than Algorithms. *Communications of the ACM*, 40(5), 1997, pp.80–91.
12. Wegner, P.: Interactive Software Technology. In: *Handbook of Computer Science and Engineering*, CRC Press, 1997.
13. Wegner, P., Goldin, D.: Interaction as a Framework for Modelling. In: Chen et al (eds). *Conceptual Modelling: Current Issues and Future Directions*, 1999, LNCS vol. 1565.

14. Wooldridge, M., Jennings, N.R.: Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2), 1995, pp.115–152.
15. Ferber, J.: *Multi-Agent Systems*. Addison-Wesley, 1999.
16. Wenger, E.: *Communities of Practice*. Cambridge University Press, London, 1998.
17. Meriste, M., Vene, V.: Attributed Automata and Language Recognizers. In: *Proc. of 4th Symposium on Programming Languages and Software Tools*. Visegrád, Hungary, June 9–10, 1995, p.114–121.
18. Ollongren, A.. Definition of Programming Languages by Interpreting Automata. Academic Press, London, 1974.
19. Ginsburg, C., Greibach, S.: Deterministic context-free languages. *Inform. and Control*, 9(6), 1996, pp.620–648.
20. Wang, W., Hidvégi, Z., Bailey Jr., A.D., Whinston, A.B.: E-Process Design and Assurance Using Model Checking. *Computer*, 33(10), 2000, pp.48–53.