C# TEMPLATES FOR TIME-AWARE AGENTS

Merik Meriste, Tõnis Kelder, Jüri Helekivi
University of Tartu, 50090 Tartu, Estonia

Leo Motus
Tallinn Technical University, 19086 Tallinn, Estonia

Keywords: software agent, time-aware agent, multi-agent system, agent based programming, agent engineering environment, construction of time-aware agents

Abstract: Autonomous behaviour of components characterises today computer applications. This has introduced a new generic architecture – multi-agent systems – where the interactions of autonomous proactive components (agents) – are decisive in determining the behaviour of the system. Increasingly, agent-based applications need time-awareness of agents and/or their interactions. Therefore the application architecture is to be enhanced with sophisticated time model that enables the study of time-aware behaviour and interactions of agents. The focus of the paper is on the inner structure of a time-aware agent, the enabling infrastructure for agent's time-sensitive communication, and the class templates for the construction of time-aware multi-agents. The prototype development is carried out in C# since this platform is suitable for controlling multiple simultaneous threads, and flexible to handle sophisticated time models.

1 INTRODUCTION

Autonomous behaviour of components characterise today computer applications. This has introduced a new generic architecture – multi-agent systems – where the interactions of autonomous components i.e. agents – are decisive in determining the behaviour of the system. A typical multi-agent system has a distinctive property that is usually not present in non-agent systems – the complete list of interacting agents and the structure of their interactions cannot be fixed at the design stage. Two interesting research areas can be outlined here. First, development of appropriate techniques and tools for programming agents as well as for agent based programming. Second, for many practical applications multi-agent architecture should be enhanced with time in order to study time-sensitive behaviour and interactions of agents. For example, a real-time system can be considered as a loosely coupled collection of autonomous agents with time-critical constraints on agents' behaviour.

Multi-agent systems rely essentially on behavioural features that cannot be specified in conventional algorithmic computing, but are inevitable in real-time, autonomous, and/or proactive computing systems. Examples of such features are persistency, interaction with an environment, time-sensitivity, and emergent behaviour. Many aspects of those features cannot be completely specified in advance – their presence depends on the dynamic operational situations where the components of a multi-agent system and its environment interact. Attempts to handle and analyse the above-mentioned features within the paradigm of algorithmic computing have led to theoretical difficulties (Blass, 2003; Wegner, 1998).

In principle, *an autonomic object with full control over its state* forms a pragmatic basis for agent's implementation. An agent implementation needs additional control threads to support its time-sensitive proactive behaviour. The implementation of multi-agents with complex interactions among its members remains a serious problem in software engineering practice. The more sophisticated a community of agents and their time-sensitive interactions are, the more research needs to be involved for resolving the implementation problem.

Meriste M., Kelder T., Helekivi J. and Motus L. (2005).
C# TEMPLATES FOR TIME-AWARE AGENTS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 247-250
DOI: 10.5220/0002508402470250

A multi-agent system that operates in time-sensitive environment has a major additional feature as compared to conventional real-time systems – the participating agents and their interaction patterns may change dynamically during integration, testing, and also during normal operation. This feature has always been desirable in real-time systems, but has deliberately been avoided to increase the behavioural determinism. The component-based design and steadily increasing pro-activeness of components have raised the role of emergent behaviour in real-time systems to the level that assumes reconsidering the behavioural analysis and finding new ways of achieving behavioural determinism.

This paper describes the structure of an agent in a prototype of agent engineering environment KRATT that is being developed for engineering time-aware agents (Motus, 2002). The focus of the paper is on the inner structure of a time-aware agent, the enabling infrastructure for agent's time-sensitive communication, and the class templates for the construction of *time-aware multi-agents*.

The prototype development is carried out in C# and .NET since this platform is the most suitable for *controlling multiple simultaneous threads*, and is *flexible to handle sophisticated time models* (Selic, 2003). C# classes for the composition of agents are described in section 2.

2 C# CLASSES FOR AGENTS

Time-aware agents exist and interact in a computer system that is distributed across a set of, not necessarily homogeneous, networks. The agents can exist completely in a virtual world – interacting only with the other agents, or also interacting with non-agent components of the system.

An application consists of administrative agents, and application agents. All the agents are generated from classes pre-specified in C# language. Classes form as a namespace *AgentComponents*.

2.1 Namespace AgentComponents

An agent (see also figure 1) is an instance of a base class Agent in the namespace AgentComponents. The class determines functionalities and lists related components that are to be applied to generate an instance of a time-aware agent. As a rule, a multi-agent system comprises several agents; each agent is implemented as Windows application (WinA).

Basic components of class Agent are:

- Communicator, exchanging time-stamped messages with the other agents;
- Manager, managing strategic and operative time-sensitive control of agent's functioning;
- Actor, performing the functional tasks;
- Monitor, monitoring specified aspects and time-stamped events in the agent's behaviour.

Each component is described by a class in the namespace AgentComponents. In the most cases the Communicator is standardised, whereas the other components are different in different applications.

Definition of a class of agents starts from defining, at least, four *primitive classes*.

1) Property. Property is a specification of an agent or its component and is used for introduction of an agent to its partners. Public class Property belongs to the Namespace *AgentComponents*.

2) Service. Service defines public services that an agent can offer to its partners, the list of services is stored by the Agent Management System (AMS) and is available to agents registered with this particular AMS. Each service has a name, service type, and a list of service properties. An agent may provide several services, described as particular instances of public class Service.

3) AgentMessage is an object that Communicator sends to Manager (or to other components) for further processing. A message usually has a form that is imposed by communication protocols combined with actions of encryption and/or decryption systems. A public class AgentMessage specifies the object AgentMessage. An AgentMessage comprises two parts – header of the message and body of the message. The header is

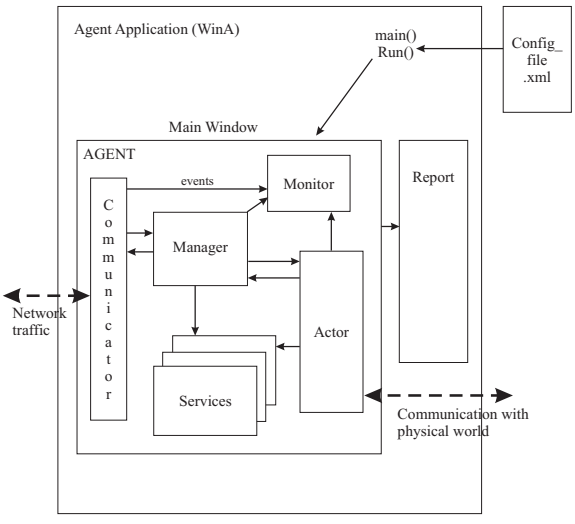

Figure 1: Generic description of an agent

processed by methods of the class AgentMessage, within the class Communicator. The body of the message is not processed by Communicator but is forwarded to Manager (or to the other addressees respectively).

4) AgentDescription contains the agent's name, lists of properties and services (meant for public use) of the agent. AgentDescription is an instantiation of a public class AgentDescription. *AgentDescription* is applied to configure agents and to register agents with an AMS.

2.2 Application Agents

Majority of agents, instantiated from the classes in the namespace AgentComponents, carry out their autonomous and proactive tasks. Application agents belong to the public class Agent: AgentDescription. Class Agent, the basic class of the namespace, is derived from the class AgentDescription and may contain all components of an agent, as variables:

```
Communicator communicator;
Manager manager;
Actor actor;
Monitor monitor.
```

When specialising a class Agent it is necessary, as a rule, to specialise its components Communicator, Manager, Actor, and Monitor. In the same WinA are constructed all components of the Agent. One has also to rewrite the respective constructor in order to substitute the construction of components with the construction of specialised (inherited) objects. Instantiation of a representative of the base class Agent is carried out by method *init* that has to be rewritten if one wants to specialise the particular class of agents.

2.3 Agent's components

An application agent is a complex that comprises several components described as follows.

1) Communicator implements the inter-agent communication and interaction with the other agents in the system. *Communicator's* functionality is standardised, e.g. it may implement the layers of Agent Communication Language, FIPA. In principle it could be an agent on its own. In its present implementation it is an object instantiated from public class Communicator.

Method Communicator.init invokes a thread that waits for information from network, addressed to port Port. This thread is simultaneous with the other threads invoked within this agent. When a message from the network is received, a method Communicator.client is invoked – the thread waiting for new network messages remains active. Thus, the Communicator can process several network messages concurrently. Specific synchronisation services or appropriate time services support the synchronisation of concurrent processing of messages, whenever necessary. Those services and primitives can be implemented as specific application agents providing those services for other agents of a particular multi-agent system.

Method Communicator.SendReply sends a message to the partner agent and waits for response. The response (possibly empty) is transformed to the AgentMessage form and is the value of this method.

A method Communicator.shutdown terminates the network message waiting thread of the Communicator. Terminating of the Agent automatically terminates the Communicator.

2) Manager is a public class whose instance guides interaction between Communicator and Actor, maintains the goal function(s) that define and rule the proactive behaviour of the agent, and controls learning and/or adaptability of the Actor. The only method defined in the class is Manager.Handle that is typically invoked by Communicator. The method Manager.Handle executes as an independent thread and must be re-enterable. The other methods are to be defined during the derivation from the class Manager.

3) The public class *Actor* implements direct functional and non-functional requirements of an agent. Hence its methods and attributes depend heavily on a specific application, and very little can be predefined. The modified constructor guarantees communication with the other components of the agent, and communication with the real world objects (outside of the computer system) if necessary.

4) Monitor is a component that monitors the operation of the agent. Based on the monitoring results it prepares periodic reports and forwards those (upon request) to MonitorAgent that is responsible for behaviour monitoring and self-diagnosing of this particular multi-agent system.

Monitor records events and processes voluntarily reported by the other components of an Agent. A public class MonitoredEvent defines an abstract event as an instantaneous phenomenon that has time instant of occurrence, type of the phenomenon, optional description of the phenomenon, and some quantitative characteristics of the event. A public class MonitoredProcess defines an abstract process as an activity that has a starting time, termination time, type of the activity, optional description of the activity, and some quantitative characteristics of the process.

Request for recording is done by a Monitor method Record (MonitoredEvent event) or Record (MonitoredProcess process). Monitor provides a report in XML format; the report is based on monitored events and processes, and includes self-diagnosis results.

2.4 Administrative Agents

Administrative agents are specific in a sense that they store and execute the rules, and provide common services, required for expected normal operation of application agents. Administrative agents are also instances of classes of the namespace AgentComponents.

1) Agent AMS for management of application agents
Instances of this class provide specific services, such as authentication and registration of agents who plan to operate within the domain of this particular AMS; keeping a directory that lists the agents, their location, their public resources and services, etc. Agents of this class also keep track of the active agents, agents who have temporarily suspended the operation, and agents who have left the directory for various reasons. Agents of this class also assign each registered agent a port for messages from the other agents provide search engine services for registered agents who look for potential cooperating partners can suspend, or terminate operation of any agent, if necessary.

2) Agent ProxyAgent
Representative of this class may serve as a switchboard for transferring messages to and from the addressee whose address is not publicly available – e.g. agents that reside behind a firewall. The second area for using a *ProxyAgent* is debugging and monitoring the traffic between application agents. ProxyAgents of have to register themselves with the corresponding agents from class *AMS.*

3) Agent MonitorAgent
Representatives of this class are agents that collect and process activity reports from Monitor classes within application agents. This information can be used for analysing the properties of the application system run-time, or during the post-mortem analysis. Again, the designer may instantiate several agents of this class with slightly different functions, if necessary.

3 CONCLUSIONS

A clear rise of interest can be observed to applying agents, and multi-agent systems in situations that assume time-awareness, or even are essentially time-critical. This could be caused by the successful practice of building component-based systems, combined with the fact that increasingly the autonomous, and proactive components are being used. The evolution of computer science is gradually reaching the understanding that interactive systems represent a new paradigm in computation (Wegner, 1998; Blass, 2003). The basis for this *empirical* computer science research relies on two contradictive concepts – inside-view to prescribe the behaviour of an agent (i.e. programming), and outside-view to design and analyse (i.e. modelling).

In this paper the agent specific structure implemented for agent engineering environment KRATT was considered. KRATT enables to develop time-aware software agents. The focus of this paper was on the inner structure of an agent that explicitly enables to elaborate its time-awareness features, on communication primitives of agents, and on the classes of agents.KRATT is still under development.

ACKNOWLEDGMENTS

This research has been partially financed by Estonian Science Foundation (ETF) grant no. 4860, and by grants no. 014 2509s03 and no. 018 2565s03 from the Estonian Ministry of Education. This support is gratefully acknowledged.

REFERENCES

Blass, A. and Gurevich, Y., 2003. Algorithms: A quest for absolute definitions. *Bulletin of European Association for Theoretical Computer Science*, no. 81, 30pp.

Motus, L., Meriste, M., Kelder, T., Helekivi, J., 2002. An Architecture for a Multi-agent System Test-bed. *Proc. of the 15th IFAC World Congress*, vol. L, Elsevier Science Publ., 6p.

Selic, B. and Motus, L., 2003. Modeling of Real-time Software with UML. *IEEE Control Systems Magazine*, vol.23, no.3, 31-42.

Wegner, P., 1998. Interactive foundations of computing. *Theor.Computer Science*, vol. 192, 315-351.