Chapter 4

SYSTEMS AND TOOLS

Living Agents: An Industry-proven Agent Server and Development Toolset

Christian Dannegger, Klaus Dorer
Living Systems GmbH, Germany

Abstract: Software agents are a fascinating research area since more than 15 years now. In this time agent systems also matured in terms of business application as well as technological support. After an introduction to the agent paradigm this section describes the Living Agents agent server and development toolset as an example of how tool support helps setting up a business application using an agent system.

Key words: agent system, agent development tool, Living Agents, agent server

1. INTRODUCTION

The term agent is used in numerous ways in software research and development. Besides a definition of the term agent we find it also important to have a short insight into the roots of agent research and the domains that originated the use of agents.

1.1 Agent Definition

You can find hundreds of agent definitions. It seems to be impossible to define the power behind this paradigm in one or two sentences only. So let us first define what agents are not: Real software agents or not only renamed search buttons or a meta search engine. They are not only price finders and comparers. They are not only the 3D representation of help functionality. They are definitely not spies.

Instead agents are electronic assistants and act in the sense of a representative, broker or secretary. The following definition of Living Systems is the shortest to summarize all important characteristics of software agents how we understand it:

> „Agents are software objects that proactively operate on behalf of their human masters in pursuing delegated goals."

1.2 The Paradigm Shift

Since the Neumann-Computer and the hereby associated machine level programming the software-industry has gone through several levels of solution analysis, design and implementation.

- First there was the introduction of commands (Mnemonics) for the assembler code and the programming style was sequential.
- The idea of reusing code blocks (modules, subroutines) led to the paradigm of function orientation. The result were 3rd generation languages (3GL) like C and Pascal.
- The next step in software development was the shift from strictly functional thinking to object orientation, where functions combined with its data was hidden as black box within an object. The result were OO-languages like Java or C++.
- With the idea of software agents, researches are talking about the next paradigm shift (e.g. Jennings, Wooldridge, 2001). Software assistants (agents) playing a role within an application space take over goals. Agents as a new, higher abstraction level encapsulates not only traditional objects but also the strategy how to use them to pursue delegated goals.

In each case, first there was the paradigm shift followed by technology support, meaning the introduction of new programming languages.

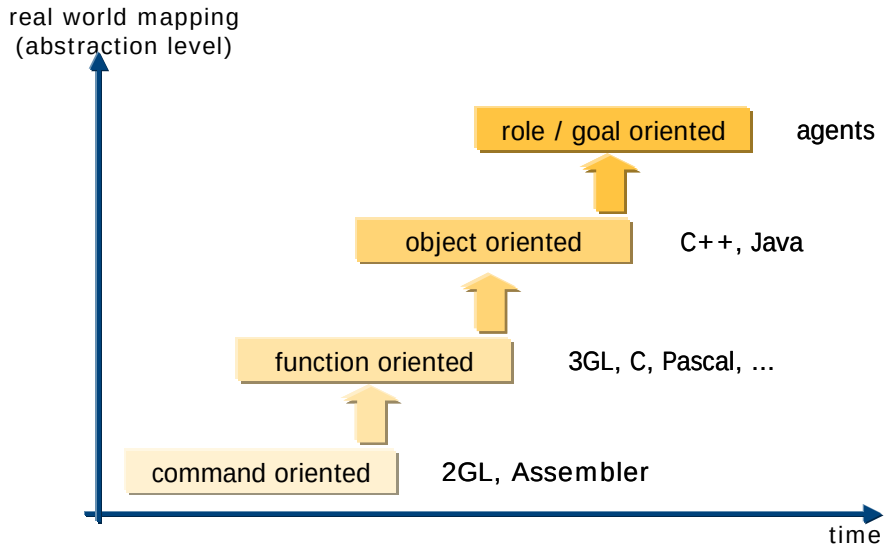

Figure 1-1. Paradigm shift in software development

Agent technology is not more mysterious than Cobol, Pascal, C++ or any other 3GL or 4GL. It is the consequent result of analyzing the real world's complexity and dynamics and how we deal with it in our daily life. Each company, each organization, each social structure is a complex network in which each participant plays certain roles. Associated with each role are a number of responsibilities and rights. All of these networks do not work because of a centralized optimization and centralized decisions. It is the opposite: decentralized decisions based on decentralized optimization. Though all participants of the organization may share a common overall goal, all of them also have their own (sub)goals. These goals build the decision base of a role/person. The system does work, if the right roles and the right goals are defined.

1.3 Agent Solution Space

As discussed above the agent paradigm has its origin in complex and dynamic environments. As more complex and more dynamic a problem is as more appropriate it is to be solved with agent technology. Or even more: extreme complex and dynamic challenges may only be solved with an agent approach (Russel, Norvig, 2002).

Less dynamic areas like strategic planning or less complex scenarios like single node in-house processes are already covered by traditional systems like ERP- and DSS-systems. The real world consists of endless dynamic

challenges. Most software solutions are planning systems based on historical data for an ideal and static future. But the reality is a sequence of permanent exceptions, which makes it complex and very dynamic.

Examples are production execution, where plans are made for weeks or even months and within a planning period exceptions like supply shortages, machine breakdowns or additional orders happen abundantly. The same applies for a whole supply network, where more than ever “time is money” to react on deviations from plan. Even faster decisions are needed in the financial area, where brokers and the systems they are using have to react in seconds.

Software agents are the result of long and intensive research to build solutions for complex and dynamic environments and – at the same time – keeping these solutions manageable.

1.4 Planning vs. Execution ?

A considerable number of existing software solutions are planning systems. E.g. “ERP” stands for Enterprise Resource Planning. Planning here as in other solutions means collecting historical data, aggregating the results into general tendencies (e.g. average), combining this with guesses for the future (forecasting) and calculating out of this a plan for the future. If you agree with that and read again through this process you will recognize that at no point in time a planning system takes the actual situation into account.

All the deviations from plans, all the real time exceptions are still managed (in most cases) by humans following manual processes. When a supplier fails to deliver in time the buyer has to find alternatives very quickly. When a machine breaks down, the production manager has to deal with that. In a positive exception when a customer increases his order the whole production process has to be adjusted very quickly with lowest impact to other orders in the most effective (cheapest) way. All of these exceptions do not only happen now and then – unfortunately exceptions are the rule in real life. This process layer of exception handling and real time optimization sits on top of three other already existing layers (see Figure 1-2 from bottom to top):

1. The layer of strategic planning in timeframes of 3 years and more helps to decide network strategies.
2. The layer of tactical planning looking at a period of 1 to 18 months is the basis for capacity planning and asset management.
3. The operational layer optimizes and plans with existing resources and orders 1 day, 1 week or even 1 month in advance.
4. The new layer of real time optimization within a planning period leads to the agent paradigm.

With this layer concept you understand the question mark in the heading of this paragraph. It is not about planning versus execution. It is real time execution on top of the existing planning layers. Planning systems were and still are needed for the purposes mentioned in the overview above. Exception handling in the 4th layer contains a huge unused optimization potential, which can ideally be unleashed by agent technology. Agent-based automation and bottom-up optimization extends existing planning solutions by getting a calculated plan at the beginning of a planning period, dealing with the exceptions and constraints at real time and reporting back results to the planning systems at the end of the period.

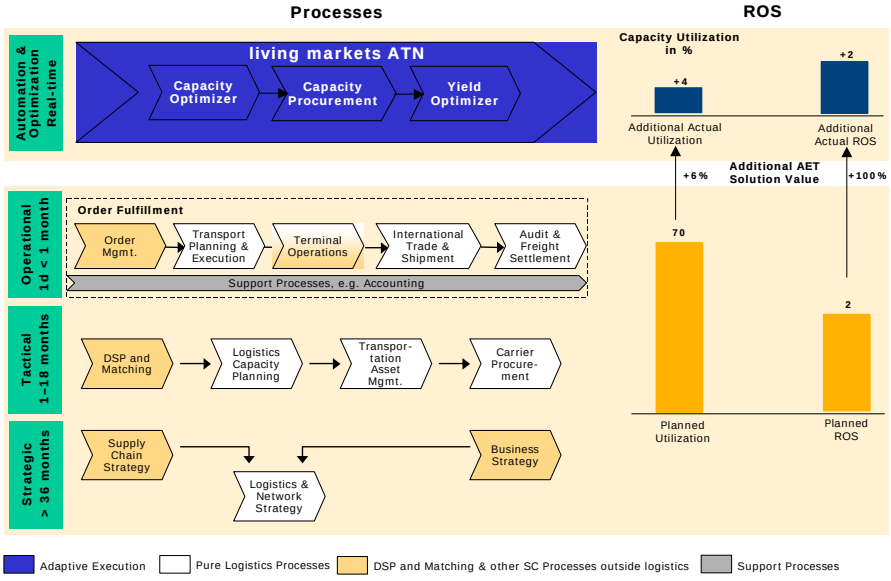

Figure 1-2. Enterprise planning layers

The rest of this article is organized as follows: section 2 describes how to design an agent application and the tool supporting the creation of the design. Section 3 explains in detail the Living Agents runtime environment needed to run agent applications. It also covers the tools that help to manage and debug running agent applications. Both sections are illustrated by an example agent application in the logistics domain. Section 3 also contains some results achieved using an agent application in the logistics domain. Section 4 draws some final conclusions.

2. DESIGNING AGENT APPLICATIONS

Agent solution architects have discussed several approaches and styles of agent analysis and design methodologies (e.g. Wooldridge, Jennings, Kinny, 2000; Odell, Parunak, Bauer, 2000; Kinny, Georgeff, Rao, 1996). All approaches are very comprehensive. Most of them, however, lack the support of well-integrated tools. Either tools are not existing at all or, as is the case for Agent UML, of the shelf tools are existing, but do not provide easy deployment integration for agent servers.

In this section we describe the design approach used at Living Systems to easily design and deploy agent applications. The tool used for this is the Living Markets Development Suite explained in section 2.2. After that section 2.3 gives an example for a design for a logistics application.

2.1 Design Approach

To be able to provide easy to use tools and based on our practical experience we developed a very simple agent design approach having 3 steps.

1. First of all the roles have to be defined. Role stands for: agent, electronic assistant or any active object with a certain task. The major decision at this stage is to define the granularity of the solution – the level of detail of each agent. A higher level in a logistics scenario would be a truck or even a distribution center. A low, detailed level would be a package or a single product.
2. The second step is to assign the domain capabilities to each of the agents. Some need financial capabilities (expertise). Others need trading or logistics expertise.
3. The third step – the most important one – is to define the strategy of each of the agents in terms of goals and business knowledge. The goals and their importance are the basis to calculate the action with the maximum utility at runtime. Business knowledge is created by combining actions and perception out of the capability libraries (the domain expertise) to behavior rules. The capabilities are functional primitives created to be reusable and are the building blocks for the agent's business logic.

These three steps cover all levels of an agent implementation from the role concept to the business strategy of each agent down to the capabilities implemented in Java.

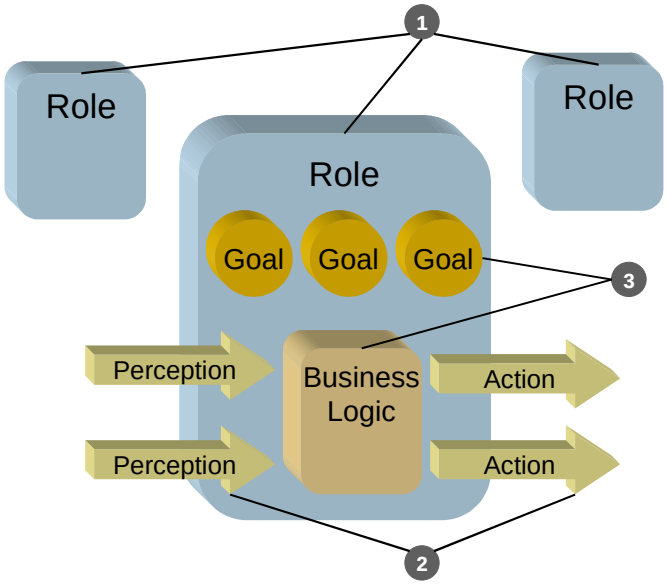

Figure 2-1. Three steps to design an agent system: 1) define roles 2) assign domain capabilities 3) define goals and business knowledge using domain capabilities

2.2 Living Markets Development Suite

Designing Living Agents applications is supported by the Living Markets Development Suite (LMDS). It directly supports the steps described above.

In a first step the roles can be defined by adding new roles to a scenario. Multiple scenarios are used in large projects to allow different views on the roles. Associated with a role is a name, a description and an icon that later identifies the agents having this role. The responsibilities of a role are specified in step 3 when the goals and business logic is defined. In this first step the interactions between roles may also be defined. Usually agents interact by asynchronous communication using messages. LMDS allows specifying messaging by linking the according roles with an arrow labeled with the name of the message. This only defines that two roles exchange a message. The time at which this is done is defined in step 3.

In the second step the user can assign the domain capabilities. This will depend on the domain the agent will be responsible for. A broker will need financial capabilities, a dispatcher will need logistics capabilities. Assigning domain capabilities either means to select from an existing library of capabilities. Especially for projects in new domains, it will be rarely the case that capabilities already exist. Therefore LMDS also allows creating ad hoc

capabilities that later in the implementation phase of the project have to be implemented.

Step 3 contributes to two important properties of agents that distinguish them from conventional software objects: autonomy and proactivity. Autonomy means that an agent should be able to decide on its own what the best action is in the current situation. Proactive means that an agent not only reacts to triggers from outside but also acts on its own initiative. Agents are therefore also called active objects. The decision engine of an agent determines the degree of autonomy and proactivity. Living Agents and LMDS currently supports two decision engines: a workflow engine and an extended behavior network (EBN) engine (Dorer, 1999). The first allows defining a deterministic workflow that contains the business logic. The goals are only implicitly specified by such a workflow and the degree of autonomy and proactivity is low. Workflows are used in deterministic domains, where the flow of actions of an agent is fix in a certain situation. EBNs on the other hand allow specifying explicitly the goals of an agent as well as the importance and the situation dependent relevance of the goals. Behavior rules contain the effects of the agent's actions and their probabilities in order to plan proactively to satisfy the agent's goals. Multiple rules may be appropriate in a situation. The agent has the autonomy to select whichever rule has the highest expected utility in a given situation. EBNs are used in continuous, dynamic and non-deterministic domains.

The result of this design process is an overview of all roles involved in a solution and their communication as well as their high-level business logic. If all of the capabilities used in the business logic are already implemented, roles may already be instantiated by agents and deployed to the agent server. Connecting LMDS to an agent server supports design time deployment of agents. A simple click starts an agent on the connected platform.

2.3 Example Application

Living Systems used the design process described above in several agent-based applications (e.g. Fritschi, Dorer, 2002). Here it shall be illustrated by an example in the logistics domain. The example is taken from a business application (Living Systems Adaptive Transportation Network - ATN) for a large logistics company. To keep the example concise we show here a highly simplified design of the real application. The general idea, however, should become clear.

The core of the application is that a dispatcher receives orders from customers to be transported from one place to another. The dispatcher's task is to find allocations of orders to trucks in a way to have optimal load of the trucks on the one hand and to stick to all constraints on the other. Constraints

are for example pickup times and delivery times of orders and drive times of drivers. Apart from new orders unexpected events occur and are reported to the dispatcher. Global (re)planning is prohibited by the complexity of the domain. A local, agent-based approach is necessary.

Figure 2-2 basically shows the result of step 1 of the design process. The two most important roles involved are the dispatcher and the truck. The dispatcher is responsible for allocating orders to trucks. The trucks are responsible to proactively optimize the allocation by exchanging orders between different instances (agents) of the truck role.

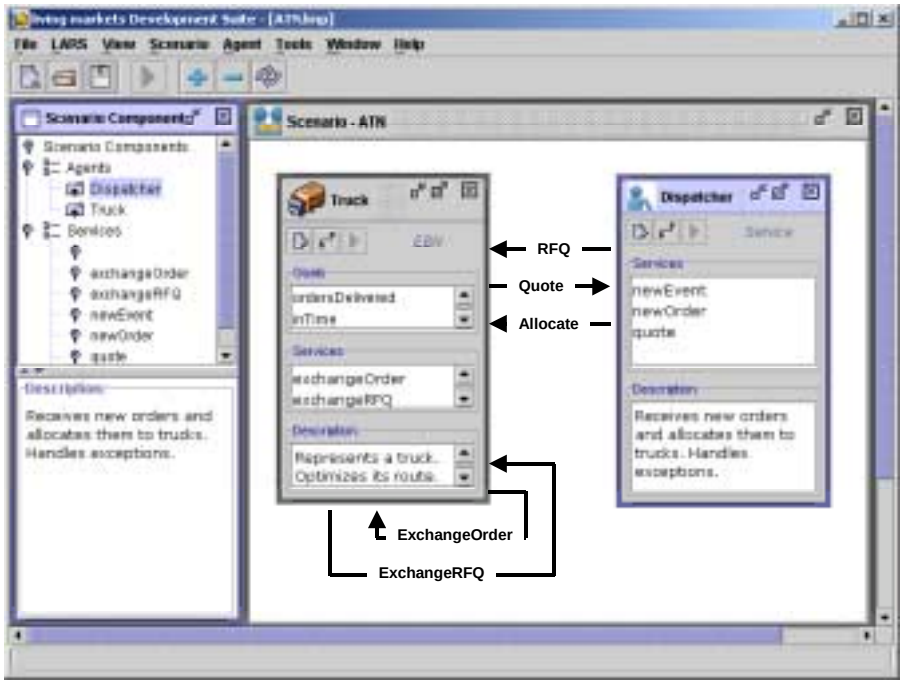

Figure 2-2: Scenario view of a logistics application in the LMDS

In step 2 both roles need capabilities of the logistics area, the dispatcher also needs market capabilities. The specific logistics capabilities needed differ of course from truck to dispatcher. The truck role needs the capability to get the distance between two locations or cost estimations for a route. The dispatcher needs the capability to run a request for quote (RFQ) among all the trucks he is responsible for. The capabilities, if not available, are the pieces that have to be developed later in the implementation phase. If properly implemented, the RFQ capability can be reused in other market scenarios.

The logic engine, goals and business logic of the roles are defined in step 3. The dispatcher in this example has straightforward deterministic behavior. Whenever a new order arrives it starts an RFQ and assigns the order to the truck with the lowest quote. Therefore the dispatcher's business knowledge is represented using a workflow engine. It uses a standard workflow for RFQs. Goals are not explicitly represented. The truck role uses an EBN engine. Its goals are that orders are delivered, that it is in time and with low costs. Behavior rules contain rules for exchanging orders with other trucks or rules for quoting on a new order. By using the EBN engine these rules are evaluated continuously and may fire proactively without triggers from the outside.

Part of the design phase for business applications is also a database design to store application data. Design also contains object-oriented design for complex capabilities. As mentioned above, the design phase is followed by an implementation phase implementing the not yet existing capabilities.

3. RUNNING AGENT APPLICATIONS

It is a long way from reading about agent technology and being fascinated by the new concept to be able to implement reliable solutions for the industry. To run an agent application you need at least one component: an agent server or agent runtime system that provides the necessary infrastructure and services for the agents. Other helpful tools allow remotely observing and managing the runtime system by visualizing the state of the agents (see also Dannegger, Kluge, Katzenberger, 2002).

3.1 Living Agents Runtime System - LARS

An agent server is needed as the basic platform and runtime environment for agents. Agents can only run on an agent server. This means that you have a well-defined environment for the agents. The agent server runs as a black box and does not need to be changed by agent developers. The Living Agents Runtime System (LARS) is an agent server that's development was started in 1997. More than 30 customer projects have been developed since then based on the LARS agent server. An agent server has to exhibit several properties that are described next.

3.1.1 Standards

Two different groups of standards can be distinguished, which have to be recognized by agent solution architects:

Firstly there are web application and Internet standards like HTTP, XML, J2EE, JMS, Corba, SOAP (Web-Services), JSP, JDBC, JNDI. All of these and other standards are used, implemented or supported by Living Agents.

Secondly there are agent standards defined by FIPA (Foundation for Intelligent Physical Agents, http://www.fipa.org), promoted by e.g. AgentLink (European Union funded organization, http://www.agentlink.org) and practically used by the agent community. Agent standards are also promoted by projects like agentcities (http://www.agentcities.org) and agent related activities of the W3C (World Wide Web consortium, http://www.w3c.org) and the OMG (Object Management Group, http://www.objs.com/agent). Most important standard here is the ACL (Agent Communication Language) of FIPA, which standardizes the message exchange between agents.

For Living Agents XML is one of the most important standards. XML is used for configuring agents and their business logic and strategy and XML is used as communication syntax between agents and to the outer world. Being aware of the performance issues with XML, LARS optimizes internal XML conversions by suppressing them whenever possible. It still provides transparent XML communication to external systems.

It is beyond the scope of this paper to discuss all the standards, their status and their special advantages in combination with software agents. Please refer to the mentioned resources for more information.

3.1.2 Agent Layers

The layered architecture of Living Agents allows optimizing, enhancing or exchanging each of the layers separately (see Figure 3-1). The layers start from bottom with the communication layer. It implements and allows configuring one or more of the standard communication protocols like Socket, JSocket, JSecureSocket, RMI and JMS. The next layer decrypts incoming and encrypts outgoing messages. It supports configurable key length of at least 4096 bit, which means increasable security level. The XML layer parses XML messages into internal XML objects for further processing. Outgoing XML messages in object format are converted back to XML documents.

The standard logic layer buffers incoming messages in the agent's inbox, processes them if they are system messages and forwards them to the business logic if they are regular messages. The business logic layer is the one containing the customized part of an agent. Different decision engines can be plugged in as described above in section 2.2. The persistence layer is the database and file system interface for an agent. Some agents may need

database persistence. Others are only needed at runtime without database access.

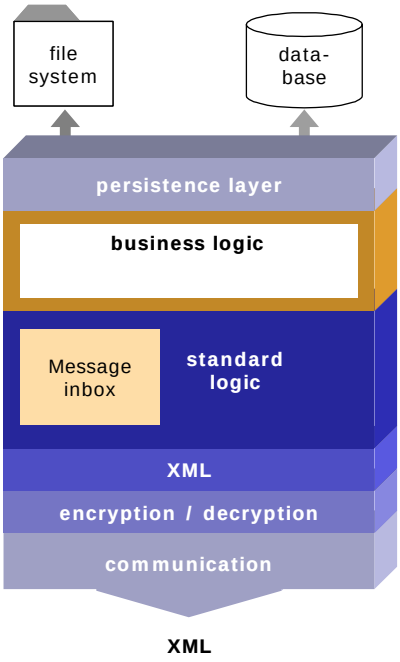

Figure 3-1. Layers of an agent

3.1.3 Logic Engines

As mentioned in section 2.2 agents are active objects. The core principle of agents is sense the environment and then act accordingly. Between the sense and the act there is the decision what to do – choose the next action that maximizes the utility for the agent and by that for the whole solution.

Choosing the next action based on the current knowledge can be very easy in certain environments and can be very complex for others. Depending on different tasks or problem types there are different approaches to model the solution strategies as there are:

- Deterministic service-oriented, reactive workflows allowing traditional sequences, branches and loops.
- Goal-oriented proactive extended behavior networks for continuous domains.
- Decision trees which can learn from sample data.
- …

The logic engine framework of Living Agents allows extending this set of action selection strategies very easily. All logic engines make use of the capability libraries with perceptions and actions written in Java. The business logic of the respective logic engine always is stored in an XML-structure. This gives the flexibility to either use a tool like LMDS to design and maintain an agent's business logic or to edit the XML-representation directly. It also opens the ability for other tools to be used or developed in the future to create this XML-based business logic.

3.1.4 Footprint / Micro-Kernel-Architecture

The footprint of LARS for a system can be very small. It depends on the workload we put on the agents. A single agent can even run on a PDA, whereas the same technology is scalable to support high performance, heavy loaded internet platforms like eBay.

This is possible through the micro-kernel architecture of LARS, which means that the kernel consists only of the absolute minimum to start and run the agent server. Almost all system functionality is implemented by so called system agents. If you do not need certain system functionality you simply do not start the respective agents.

3.1.5 Scalability

Scalability is a core characteristic of agents. Each agent always holds the data it needs for decisions and communicates with other agents through Internet protocols. This even allows an agent to work on one single server exclusively if needed for its task. On the other side an agent can share a server with hundreds or thousands of other agents. An agent architecture allows you to make use and leverage each piece of resource.

Mobile agents can move to another server if the current one is overloaded or is scheduled for maintenance and by that balance the load across all resources of the IT infrastructure. This is managed by load balancing agents on each agent server following an individual strategy to balance the load.

3.1.6 Flexibility

An installation of an agent-based solution can be distributed across several computer systems. Reasons for that are strategies for load balancing, fault tolerance, firewall security, and solution integration.

3.1.7 Integration

By hiding protocols within a single agent, agent technology is ideal for heterogeneous integration challenges. Data transformation ideally is done by XML-based XSLT technology. The interface itself sits within an agent. Web-Services and the underlying SOAP-protocol based on HTTP-requests are the perfect fit for agents. Web-Services are ideally implemented by agents.

Traditional integration solutions like bulk processing by exchanging data via structured files or using API-libraries can be added to agents by adding these capabilities to the respective integration agent.

Last but not least Living Agents of course fits into an existing EAI-infrastructure like e.g. TIBCO, MQ-Series, webMethods, Entire-X or other message-based platforms.

3.1.8 Reusability

The permanent goal of software developers is to increase the reusability of once developed functionality. The clear separation of business strategy (goals and behavior rules) and functional primitives (capabilities) is a major step into this direction. All capabilities are organized in libraries. They are used by graphical agent development tools like LMDS to build specific business agents. All details of a Java-based capability are hidden while implementing or changing the agent's strategy.

3.1.9 Security

Each agent can be configured to accept only messages of a certain type from certain agents from certain agent servers. This security shell is outside the agent's logic and does neither affect logic nor the logic has to take care about this "agent private firewall".

3.2 Composer and Visualizer

An important step between the design an running an agent application is the instantiation of roles with agents. This can be done by using the Living Markets Composer and Visualizer (LMCV). It loads the role definitions of LMDS and allows defining a setup of any number of agents per role. This is done by dragging and dropping the agent icon to a composer window.

The LMCV can also be used to visualize agent states during runtime. This is especially useful if the agent represents a role with position

information. In the logistics example from above this can be used to display the truck agents and their position on a map (see Figure 3-2).

Figure 3-2. The Living Markets Composer and Visualizer

3.3 Control Center

The Living Agents Control Center (LACC) is a tool for managing the agent servers and the agents running on them (see Figure 3-3). It connects to an agent server as if it was an agent. All communication is done using agent messages. LACC allows monitoring the state of the platform and the agent running on it. It also allows changing the state of the agents and platforms by sending messages to the agents. Finally it integrates debugging functionality by observing the messages on the platform.

The LACC displays a list of all agent servers to which the current agent server is connected. It can show a list of all agents running on a platform. Any agent can be asked to send description information of it containing version information, some messaging statistics and a list of services the agent provides.

Any administration task can be performed by either manually sending the appropriate message or by pre-configured messages stored in an XML configuration file. Changes may be starting or stopping agents, migrating agents to other platforms or changing the state of agents in any other way by

sending appropriate messages. For example, one can change the log level of the agent or manually trigger a service of an agent.

The LACC also integrates debugging functionality with respect to the message traffic on the server. It therefore allows tracing specified messages on the server done by a debug agent on the server platform. The trace can be used to create charts of the message traffic to detect bottlenecks. It can also be used to create message sequence diagrams to verify protocols and the order of messages exchanged between agents.

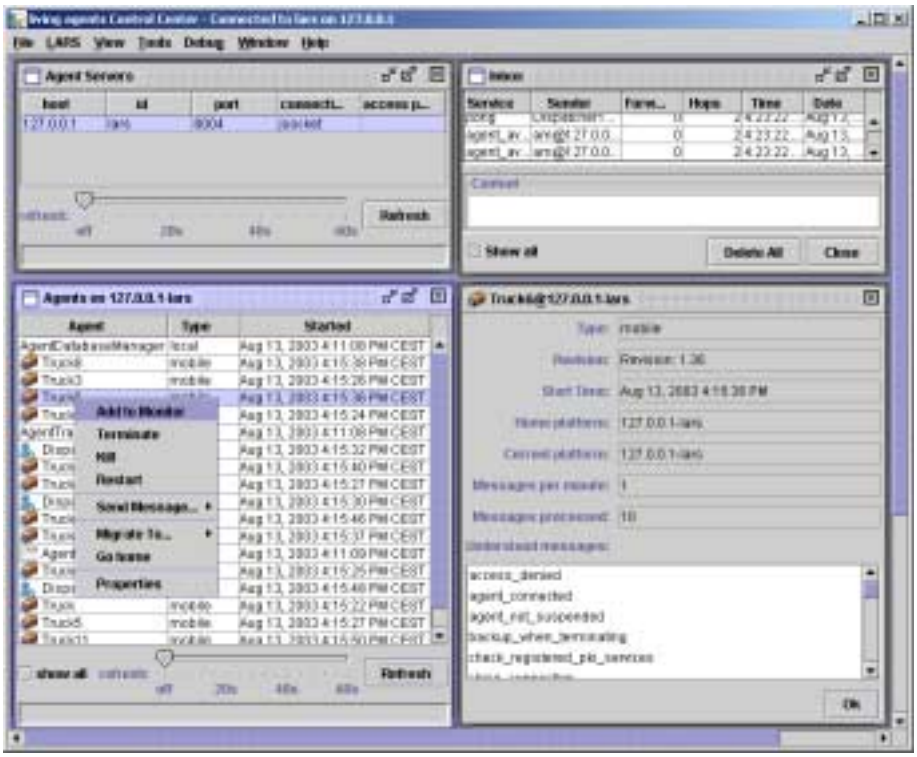

Figure 3-3. The Living Agents Control Center

3.4 Running the Example Application

As described in section 1 and 2 one advantage of agent-oriented software engineering is its very natural way to design an agent application. The important question is if such a design results in a system that produces the desired results in the dynamic domains it is designed for. In this section we show some results achieved in the example logistics domain. The results described here are not results of the simple design used as illustration in

section 2. They are the result of the more complex agent design of the Living Systems ATN software used in an industry application.

Figure 3-4 shows the results gained by running ATN on real data of a big logistics company[1]. The goal was to reduce the overall driven kilometers to transport the specified orders and keeping the number of constraint violations (time delays) as low as possible.

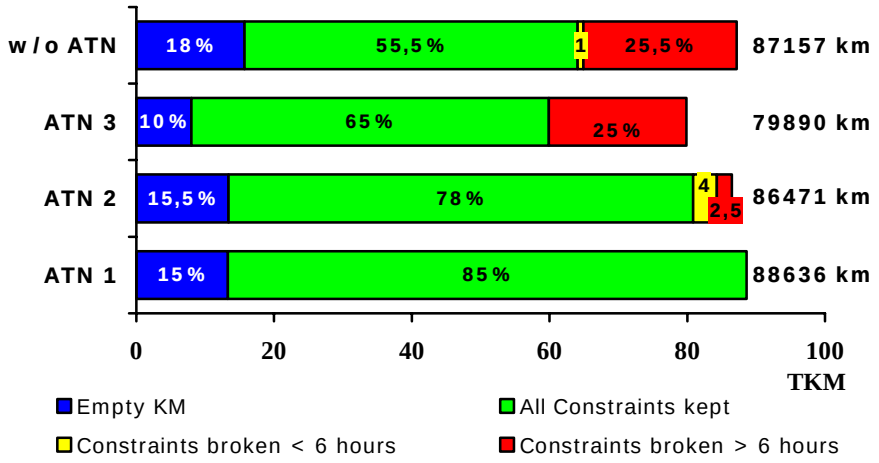

Figure 3-4. Results on real logistics data of human dispatchers (w/o ATN), and running agent-based ATN on same service level (ATN 3), higher service level (ATN 2) and total fulfillment i.e. no constraint violations (ATN 1).

The top bar (w/o ATN) shows the result that was achieved by the human dispatchers of the company. The first part of the bar represents the empty kilometers of all trucks, the second the kilometers driven with all constraints kept, the third kilometers driven with orders violating pickup or delivery times with less than 6 hours and the fourth with more than 6 hours.

The second bar (ATN 3) shows the result of ATN running on the same service level as the human result. This means that violations of pickup or delivery times for orders are about the same amount as with the human dispatchers. The overall driven kilometers have been reduced by about 9%. Especially the empty kilometers are reduced considerably.

The third bar (ATN 2) shows the result when running ATN on a higher service level. It shows that when producing about the same amount of driven kilometers this can be achieved by a considerable fewer amount of constraint violations meaning delivering more orders in time.

[1] The data was a subset of the complete transport orders in the LTL (less than truckload) business of the company.

Finally the bottom bar (ATN 1) shows running ATN with complete fulfillment, i.e. no constraint violations allowed picking up and delivering all orders in the specified timeframe. This only produces a roughly 2% higher amount of kilometers having a much higher quality of service.

A big difference to conventional dispatching systems is that the agent-based system is able to deal with real time changes like delays of trucks or new or changed orders. Due to its local and decentralized optimization there is no need to re-plan all orders and trucks. Changes are optimized locally by the affected agents and then spread across the network to get closer to a global optimum. With this, the reaction time of the system to such unforeseen events is between fractions of a second to few seconds.

4. CONCLUSIONS

Software is continuously getting more complex over time. One reason is that it deals with increasingly complex and dynamic domains. A new paradigm evolved that deals with this complexity by distributing tasks and acting proactively in a network of responsibilities: agent-based systems. The agent paradigm has a number of appealing properties. In this paper we have exemplified two of them. Firstly, the design of such agent systems is straightforward, because the agent design maps very closely to the organizations in the real world. Secondly, the properties of agents like collaboration, proactivity, autonomy and goal-directedness exhibit a runtime behavior usable in complex and dynamic domains.

As with other paradigms the success is highly dependent on the availability of tools and infrastructure supporting the user in applying the paradigm. The Living Agents toolset supports an agent developer in the phases of design, deployment, debugging and maintenance of an agent system. This is a necessary condition to run applications in an industrial environment.

The results obtained so far from industrial application of agent technology are encouraging. Saving 9% of the overall kilometers of a logistics company, as described above, is quite a success. Equally valuable is the ability to optimize the logistics network in real time. This allows taking the daily exceptions and opportunities into account without huge manual intervention as it is done today.

REFERENCES

1. Dannegger Ch., Kluge Ch., Katzenberger R. Agents in Real-World Applications: living markets. AgentLink Newsletter 9, 2002.
2. Dorer K. Behavior Networks for Continuous Domains using Situation-Dependent Motivations. Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99), 1233-1238, Morgan Kaufmann, Stockholm, 1999.
3. Fritschi C., Dorer K. Agent-oriented software engineering for successful TAC participation. Proceedings of the First International Joint Conference on Autonomous Agents & Multi-Agent Systems, AAMAS 2002, July 15-19, Bologna, Italy, ACM, 2002.
4. Jennings N.R., Wooldridge M. "Agent-Oriented Software Engineering." In *Handbook of Agent Technology*, J. Bradshaw, ed., AAAI/MIT Press, 2001.
5. Kinny D., Georgeff M., Rao A. A Methodology and Modelling Technique for Systems of BDI Agents. Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. Springer: Berlin, 1996.
6. Odell J, Parunak H.V.D., Bauer B. Extending UML for agents. Proceedings of the 2nd International Workshop on Agent-Oriented Information Systems, AOIS 2000, June 5-6, Stockholm, Sweden, iCue Publishing, 2000.
7. Russel S., Norvig P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd Edition, 2002.
8. Sycara K., Paolucci M., Van Velsen M., Giampapa J. The RETSINA MAS infrastructure. Autonomous Agents and Multi-Agent Systems 7(1/2), July/September 2003, K. Sycara, M. Wooldridge, eds., Kluwer Academic Publishers, 2003.
9. Wooldridge M., Jennings N.R., Kinny D. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems 3 (3): 285-312, 2000.