
Available from: http://dx.doi.org/10.1109/AAMAS.2004.39

Copyright © ACM 2004.
The definitive version was published in Proceedings of AAMAS, 2004.

This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. If your library has a subscription to these conference proceedings, you may also be able to access the published version via the library catalogue.
Achieving Dynamic Interfaces with Agent Concepts

Thomas Juan
Department of Computer Science and Software Engineering,
University of Melbourne, 3010, Victoria, Australia
{tlj, leon}@cs.murdoch.edu.au

Leon Sterling

Abstract

Traditionally, interfaces of software entities, modules and components are immutable at runtime and carry no information on the meanings of the underlying implementation. We believe this definition of interface imposes a rigid view or context on the interaction of software entities, which impedes software re-use and the development of open / adaptive systems.

We propose a novel analysis and design construct called a dynamic interface. Dynamic interfaces capture the social ability of agents and can be evolved consistently at runtime. Semantic information about the underlying implementation is also built into the dynamic interface, while preserving information hiding. We envisage the new dynamic interface construct to be complementary to traditional immutable interfaces. The two constructs can be used consistently in applications to address different requirements.

We describe a prototype implementation of the dynamic interface construct. The implementation leverages the agent concepts of goals, roles, protocols, agents and services from the ROADMAP meta-model. The initial evaluation on its flexibility and performance indicates that dynamic interfaces have potential as an industry strength design and implementation construct.

1. Introduction

Over the last few decades the main progress in software engineering has been the introduction of progressively higher-level abstractions. The high-level abstractions allow developers to analyze and design the systems abstractly, focusing on high-level engineering issues, without being bogged down in low-level implementation details.

In addition, a software engineering paradigm offers concrete mechanisms as basic infrastructure to support the abstractions. By assuming the existence of such mechanisms and reusing the mechanisms, developers avoid re-inventing and re-implementing their own equivalent mechanisms for each new application. For example, the object oriented paradigm provides abstractions such as classes and objects, with basic mechanisms like encapsulation, inheritance and polymorphism.

Intelligent agents have emerged as a promising candidate for future software development [6]. They have attractive theoretical properties like autonomy, pro-activeness, reasoning of knowledge and social ability to negotiate interactions dynamically. These properties render agents suitable for future distributed, knowledge-intensive, open and adaptive applications.

However, before agent-oriented software engineering can be widely used by industry developers, the attractive theoretical properties of agents must first be translated into simple, light-weight, concrete constructs and mechanisms that are of similar granularity as objects, methods, inheritance and polymorphism.

In this paper, we focus on the social ability of agents to negotiate (new) interaction dynamically and design the dynamic interface as the corresponding concrete construct for use in developing real applications. Indeed, the autonomous and adaptive nature of agents suggests the need for dynamic interfaces.

In the rest of this paper, Section 2 examines potential problems with the current definition of a software interface. Section 3 gives a high level overview of the dynamic interface. Section 4 describes agent concepts from the ROADMAP meta-model. These concepts are used to describe our implementation of dynamic interfaces. Section 5 describes the actual implementation of the dynamic interfaces, using ROADMAP agent concepts. Section 6 presents illustrating examples using the dynamic interfaces. Section 7 provides early evaluation of the construct and the implementation in terms of quality attributes such as performance. Section 8 concludes.

2. Problem with abstract, immutable and semantic-free interfaces

Traditionally, an interface, whether to a software module, a component, or an object, will contain a set of function signatures which allows the underlying functionalities to be invoked. The set of APIs provided by an operating system is a typical example. The interface is abstract and declarative in the sense that it will not be instantiated at runtime. The interface is considered an
immutable contract between software entities. Changes to interfaces during the development are important design decisions that could cause incompatibilities between software entities. Direct change to the interfaces at runtime is normally not possible.

The semantics of the underlying implementation is normally not provided in the interfaces. In fact, such information is normally not explicitly expressed in the system at all, but in documents created during the development.

The immutable and semantic-free nature of abstract interfaces impedes software reuse and the development of open / adaptive systems.

The immutable interface imposes a particular view on a software entity, normally of large granularity such as a software sub-system. It forces developers of other interacting peer software entities to accept that view which is often very specific to a particular application and context. As a result the entire underlying implementation is sealed from being reused unless another application happens to share exactly the same view and the same context.

The interface, being semantic-free, does not provide support to represent the meaning of the underlying implementation. Existing tools such as compilers can at most assure correct invocation with properly typed parameters and return values. Human effort is always required to ensure that the calling entity and the called entity have the same understanding and agree on what exact function is to be performed. The significant human effort often prohibits reuse as the effort to understand a component and verify the correctness of its behaviour is often more than the effort of re-implementing the component. As a well known fact, unplanned opportunistic reuse is usually not fruitful [1,2]. We consider the current definition of immutable interfaces a major contributing factor to this fact.

Similarly, the rigid partitioning of the system prevents openness and limits the scope of possible adaptation. Entities in adaptive systems may change their behaviours according to environment stimuli. If the semantic meaning of their interaction is not represented, it would not be possible to determine whether the interaction is still correct after (mutual) adaptation. Similarly, in open systems, software entities may be self-interested or even malicious. The lack of semantics in interfaces prohibits runtime verification of entity behaviours and consequently impedes the development of open / adaptive systems.

3. Dynamic interfaces

To address the above problems, we propose a novel construct of a concrete, dynamic and semantic-rich interface. The dynamic interface is a complementary conceptual tool to the abstract immutable interface. We expect the dynamic interface to have a runtime instance and therefore can be reasoned about and manipulated. The dynamic interface will also support the representation of semantics of the underlying implementation.

The vision is that given the same underlying implementation, the concrete dynamic interface can be more flexible, as its runtime instances can be modified easily when necessary to present multiple views to access the underlying implementation. This allows peer software entities of different views and contexts to interact with the interface consistently.

At a higher-level, by providing concrete dynamic interfaces between sub-systems within the system, the partitioning of the overall system is captured in a concrete and dynamic architecture design. The architecture, being represented in concrete dynamic interfaces, can be reasoned about and changed at runtime, to better serve the users.

The dynamic interface should also support the representation of semantics of the underlying functionalities. To preserve information hiding, only the abstract objectives (goals) and the externally observable behaviours of the underlying functionalities are to be represented, without revealing the low-level implementation details. The representation of objectives and externally observable behaviour allows other developers to quickly understand the function of a component, and easily verify its correctness and other quality attributes. The same reasoning and verification can also be done by software agents at runtime, representing the possibility of runtime composition of components or agent services.

Although the concept of the dynamic interface arises from our attempt to create a concrete design and programming construct to capture the social ability of agents, it deals with common problems in software engineering. It is therefore possible to see it as a general construct, able to be implemented in different paradigms or in different forms, such as being part of a new programming language. However, our implementation is agent-oriented, based on the ROADMAP meta-model.

4. The ROADMAP meta-modal

In this paper we leverage agent concepts from the ROADMAP approach to implement the dynamic interface construct [3, 4].

ROADMAP promotes an organizational view of computing similar to [7,8]. The ROADMAP meta-model is shown in figure 1. All the entities shown in the meta-model are instantiated at runtime as changeable first-class entities from their respective development time classes. For simplicity, the development time classes are omitted.
Roles and agents are at the centre of the meta-model. Roles represent the higher level system specification at runtime and constrain agent behaviour. Agents represent the concrete system implementation at runtime and implement their respective roles. The relationship between roles and agents is similar to that between traditional interfaces and components. However, as agents are more autonomous than components, roles are more expressive and changeable at runtime to better constrain the behaviour of agents. Roles contain protocols, like traditional interfaces containing method signatures. Agents contain services, like objects containing methods. Figure 2 shows interaction of implementation (agents) through concrete dynamic interfaces (roles).

For scalability, roles can aggregate sub-roles recursively while agents can aggregate sub-agents. We expect a multi-agent system to contain a hierarchy of roles and a hierarchy of agents, separating the system specification (roles) from implementation (agents) (see figure 3).

Roles are defined by a name, sub-roles, knowledge, goals, responsibilities, permissions and protocols. Figure 4 shows an example of a role performing low-level network transport of data. It aggregates two sub-roles, namely Controller and Connection. The role also contains a knowledge component called Application Traffic Profile, recording the typical traffic pattern of known applications.

Knowledge components are modular units of domain knowledge. They can be used to parameterize roles, protocols, agents and services for more flexibility. The exact parameterization is specified as part of the role's permissions. In figure 4, the Connect protocol is parameterized with the Application Traffic Profile knowledge component.

**Role Name:**
NetworkTransport

**Sub-Roles:**
Controller, Connection

**Knowledge:**
Application Traffic Profile

**Goals:**
TransportData
Max. Reliability () during Work
Max. Throughput () during Transmit
Min. ActiveConnection () during Work

**According to** Prioritize ()

**Responsibility:**
R1: Reliability () > 8 during Work involves Connection . Safe2
R2: Throughput () > 3 during Transmit involves Connection . Safe1
accessible external functions in agents or objects.

The responsibilities are properties that the agent acting on the role must preserve. Responsibilities are also expressed as logic statements using evaluation functions.

Permissions specify what resources the agent fulfilling this role can or cannot access or modify. Some roles may have permissions to access and modify other roles or protocols. This is the mechanism we use to model the dynamic nature of the new dynamic interface construct.

We use keywords before, during and after to specify when goals, responsibilities and permissions will take effect. We use these keywords to model the pre-conditions, invariants and post-conditions of protocols.

The involves keyword links role attributes such as responsibilities and protocols to sub-role attributes. When children properties are violated, involved parent role properties will be considered violated too and parent role can perform higher-level responses while the children role performs lower-level responses.

Protocols can address and implement hard goals and are specified using the fulfills keyword. In figure 4, the Work protocol is specified in regular expression and implements the main goal of the role, TransportData.

Figure 5 shows an example agent definition. The implements keyword binds services of agents to protocols of roles. Like the binding using all the other keywords, this binding is also dynamic and can be changed at runtime.

Agents and roles communicate by message passing. The use of messages to model interaction reduces the coupling between agents, as messages can be rejected, forwarded to other agents, logged or played back. Official interaction in the agent organization (the system) must go through roles. When agents interact directly without going through roles, the interaction is considered private and does not have the same official status within the organization. For certain organizations private interaction can be undesirable and therefore prohibited.

Agent Name:
NetworkAgent

Roles Implemented:
NetworkTransport

Sub-Agents:
ControlAgent, ConnectionAgent

Services:
ConnectSve implements Connect
TransmitSve implements Transmit
DisconnectSve implements Disconnect
WaitSve implements Wait

Figure 2 shows an example of official interaction in an organization between Agent A and Agent B. Here we see the concrete dynamic interfaces (roles) are used to separate their implementation (agents). The message from Agent A is first sent to and validated by its role. If all constraints are satisfied, the message propagates to Agent B’s role. After the message is validated, Agent B receives the message and can now respond to it. As part of the organizational arrangement, the message is also forwarded to Agent C’s role and to Agent C after validation for monitoring purposes.

If the message fails to satisfy constraints from any roles concerned, the message will be rejected and actions will be taken to handle the error. This mechanism ensures the interaction respects perspectives of all roles involved. In addition, quantitative results may be produced by evaluation functions within the roles. Such results provide
indications to agents on how well their interaction satisfies functional requirements and quality attributes of the system. The evaluation functions are somewhat similar to the fitness function in a genetic algorithm.

In summary, the roles and protocols represent the dynamic interface construct at runtime. The dynamic nature of the construct comes from the ability to have permission to modify instantiated runtime roles and protocols.

The rich semantics of the underlying implementation is represented in two ways. The objectives of the implementation are specified in terms of hard and soft goals, linked to both roles and protocols. The externally observable behaviour of the underlying implementation is specified as various attributes of the roles such as responsibilities, evaluation functions and permissions. All these attributes are externally observable and none of them reveals the internal implementation details. Therefore information hiding is preserved while rich semantic information is given. The hierarchical nature of the roles and protocols allows us to take this approach consistently from a high level architectural view to a low-level class/method view and to use dynamic interfaces at all levels.

5. Realizing dynamic interfaces with roles, protocols, agents and services

Consider an example protocol class called Negotiate. Figure 6 shows the complete Negotiate (contract net) protocol class using a notation based on AUML [5].

The development-time protocol class in figure 6 includes only the message pattern. At runtime, protocols are instantiated by their parent roles from the protocol class, with some additional information.

Figure 6. Negotiate Protocol

Once instantiated, the protocol will possess the message pattern from the protocol class, access to all the relevant attributes from the parent role, such as responsibilities, goals, and permissions, and a status showing if it is the initiator or the responder of the protocol. The status allows the protocol to enter and execute the correct half of the message pattern.

In figure 7 we show two agents interacting through their respective roles. It is worth noting in this particular scenario, each agent, role, service and protocol have a dedicated execution thread. The scenario illustrates the case where agents cannot commit their execution threads to run services, as they still need to keep track of their environment and be ready to handle other messages. Roles have execution threads so they can start or stop protocols whenever necessary. Protocols have execution threads so they can perform independent monitoring of agent behaviour without relying on the agents' execution threads. The object lifelines in figure 7 are threads while the activation bars on lifelines represent active processing running on the threads. Interaction is done through asynchronous message passing. The scenario shown in figure 7 is a special case where no error or rejection occurred.

Figure 7. A scenario

At the top-left corner of figure 7, Agent 1 performs some reasoning and decides to start a buying service. The service attempts to send a Query message to the Agent 2 via the roles. The Buyer Role, fulfilled by Agent 1 performs some reasoning on receiving Query and instantiates the appropriate Negotiate protocol. This
The actual pattern of interaction into roles and protocols now contain the message pattern from figure 7, all relevant attributes and constraints from the Buyer role, and the status as the initiator of the message sequence.

The protocol checks the pre-condition constraints before passing the message on and continues to monitor invariants and post-conditions of the interaction. The pre/post conditions and invariants are specified in the definition of the Buyer role as responsibilities before, after, and during the Negotiate protocol. The constraints may include quality of service requirements such as a deadline for a Quote message. As Query reaches the Sales Role, played by Agent 2, Sales Role recognizes the message and starts the appropriate Negotiate protocol to validate the interaction. This protocol now contain the message pattern from figure 7, all relevant attributes and constraints from the Sales role, and the status as the responder of the message sequence.

The protocol checks for pre/post conditions, invariants and quality constraints specified in the Sales role and forward the Query to Agent 2. Agent 2 starts the Sell service to handle the Query. The same process follows when Agent 2 sends the Quote message back to Agent 1, through the two protocols, and Agent 1 sends the Order message to Agent 2 to finish the Negotiate protocol.

The two runtime protocols include the same message pattern from the protocol class, and differ in the constraints and permissions they obtain from their parent roles, in that one is instantiated as the initiator of the interaction and the other as responder of the interaction. Both protocols step through the message pattern as prescribed in the same protocol class with opposite operations. When one sends the message the other one receives and vice versa.

Once instantiated, the two protocols also have associations to their parent role. Through the association, they can invoke various functions referred to in the parent roles for constraint checking. For example, the Negotiate protocol instantiated by the Buyer role has access to all Buyer role responsibilities, goals, and permissions marked relevant before, during and after the Negotiate protocol. For example, the Buyer role might have an upper bound on purchase price that cannot be exceeded and a goal to minimize the price. Similarly, the Negotiate protocol instantiated by the Sales role has access to Sales role safety responsibilities, goals, and permissions marked relevant before, during and after the Negotiate protocol. Therefore different checking is performed by protocols instantiated from the same protocol class but by different parent roles.

In this sense, we separate the context of interaction and the actual pattern of interaction into roles and protocols respectively. For example, the same Negotiate protocol can be used between other roles, while the correct context of the interaction, in terms of constraints and resource permissions, will be supplied by different roles.

**Figure 8. Steps in protocol**

Figure 8 shows a service and the protocol it implements, similar to the left-half of our scenario in figure 8. The right-half responding pair of protocol and service is omitted here. According to our design, the protocol can include any sequence of four possible operations: sending a message, waiting to receive a message, initiating a sub-protocol, and responding to a sub-protocol initiated by other roles. In figure 9, the protocol has exactly four steps to show all four possible operations. For each step in the protocol, there is a list of fine-grained sub-services that will run inside the service, and a separate and independent list of tasks that will run inside the protocol. For example, the first step in the protocol is the sending of Msg 1 to A. The service runs sub-service 1 to invoke the sending of the Msg 1 in the protocol. In this case, no additional tasks are present in the task-lists of both the protocol and the service. In the next step, the service runs sub-services 2 and 3 as preparation and waits to receive Msg 2. At the same time the protocol independently checks for pre-conditions, sends Msg 1 to B, and then waits to receive Msg 2. When the protocol receives Msg 2, it passes Msg 2 to the service, which runs sub-service 4 to handle Msg 2. This is the point where parallel processing in the service and the protocol is synchronized, as indicated by the line connecting the receiving of Msg 2 in the protocol to the running of sub-service 4 in the service. After this synchronization point, the service continues to run sub-service 5 at the same time the protocol waits to receive Msg 3 and invoke a method on an object.

When all actions in both lists in the protocol and the service finish running, the protocol moves into the next step. The service runs sub-service 6 to initiate the sub-protocol 1 through the protocol. In this example both task-lists are not loaded with additional tasks. When all
processing related to sub-protocol 1 finishes for both the service and the protocol, the protocol moves into the next steps and waits to respond to sub-protocol 2 initiated by another protocol. Once sub-protocol 2 is initiated, the protocol notifies the service and starts sub-service 7 to handle sub-protocol 2.

In summary, a protocol can have any number of steps from the four possible operations. In each step the service and the protocol runs two separate task-lists of actions, with one synchronization point. When both lists finish running, the protocol and the service move into the next step.

To change a dynamic interface at runtime, one can either change the role, or the protocol, or both. All attributes of a role can be added, deleted or modified. The binding between attributes, specified with keywords in Figure 4, can also be added, deleted and modified at runtime.

There are two ways to change a protocol. Firstly, the protocol class can be changed (see figure 7) and steps in the protocol can be added or deleted. Secondly, each step in the protocol can also be modified. For each step in the protocol, there are two attributes that may be changed given proper authorization, namely the address and the task-list. For a step that sends a message or initiates a sub-protocol, the address is the destination role. For a step that receives a message or responds to a sub-protocol, the address is the source role. By changing the corresponding addresses, the message or sub-protocols can be re-routed.

![Figure 9. Re-routing interaction](image)

Figure 9 shows the dynamic change of the protocol to re-route a message. Instead of sending message from Protocol 1 to Protocol 2 directly, the address in the protocols can be changed and the message is now sent to Protocol 3 first before sending to Protocol 2. The services implementing Protocol 3 (in gray) will transform the message for additional functionalities before sending it to protocol 2. For example, the message was originally sent from a data source to the file system (A in figure 10). In light of new security threats, the system can change Protocol 1 to send the message to be encrypted first before forwarding to be stored on the file system (B in figure 10). This change of control and data flow in the system can happen dynamically at runtime. As long as constraints on all roles are preserved, there is assurance that correctness and other quality requirements are still being met.

The task-list of a protocol step can be changed as well. The tasks in the task-list of the protocol can modify content of the cached message from the sending service before it is sent, or after it is received but before delivered to the handling service. By adding or deleting tasks from the task-lists, we can add or remove transformations on the messages.

For every step of the protocol, both lists and the synchronization point in the protocol and the service are dynamic and are subject to modification given proper authorization. All three types of changes on roles and protocols can be done by developers during development, or by agents playing authorized roles at runtime. Hence the dynamic interface construct is implemented.

### 6. Application examples

With the above realization of dynamic interfaces, application development can be done at a higher level of abstraction. When immutable interfaces are used, control flow and data flow in the system via function/method calls are hard-wired into the body of other functions or methods. The control/data flow is hard to understand, hard to search and undocumented. With our design of dynamic interfaces, the allowed control flows and data flows are captured in the protocol diagrams, allowing the developer to work at a higher level of abstraction.

Extending the functionalities of the system is also made easy. Developer can install fine-grain tasks/sub-services into existing protocols and services by applying aspect tags to the protocol, or certain parts of the protocol. For example, applying an aspect tag like: [encrypt, type = RSA, strength = 1024 bits, binary = local_path/filename] to a protocol may load encryption code into the task lists of every protocol step and encrypt/decrypt all messages exchanged appropriately. The developer may still need to check the order when multiple tags are applied to ensure correctness. However, she could rely on the dynamic interface infrastructure to take care of much low level work.

Interesting and commercially valuable application examples can be modeled by dynamic interfaces.

Consider flexible enterprise applications and infrastructures that change according to the business process, strategy and organizational structure of the company. Assuming the internal of the system is bond together by dynamic interfaces, when the business processes, the company strategy or the organizational structure of the company changes, the interface can change accordingly. The implementation behind the interface can be re-structured accordingly down to the granularity of a method-level service. With explicit
constraint checking at all points of the system, there is more assurance on the achievement of correctness and other quality requirements. Tremendous re-integration effort within the company can be saved in this scenario.

7. Evaluation and future work

The dynamic interface construct formalize and represents at runtime the following aspects:
- control and data flow in the system, in terms of allowed patterns to invoke functionalities through messages.
- semantic constraints to represent correctness and other quality requirements.

The level of impact of the dynamic interface construct on application development also depends on the exact implementation of the construct. At the University of Melbourne, we are developing a prototypical programming framework, based on the ROADMAP meta-model, built on top of Microsoft's .NET platform. It consists of a set of library classes to be extended by application developers. The framework implements the dynamic interface according to the design described in this paper. Initial performance evaluation indicates invoking an empty service via the dynamic interface with empty task lists is more efficient than invoking an empty Java method.

As future work we plan to formally evaluate the dynamic interface construct and compare it to other approaches such as semantic web [9], aspect oriented programming [10] that address potentially similar problems.

8. Conclusion

Traditionally interfaces of software entities and components serve the purpose of invoking the functionalities encapsulated in entities or components. The interfaces are immutable at runtime and carry no information on the semantics of the underlying implementation. We believe this definition of interfaces can be extended to make them more flexible.

We proposed a novel analysis and design construct called a dynamic interface. Dynamic interfaces capture the social ability of agents and can be evolved at runtime in a consistent manner. Semantic information about the underlying implementation is also included, while preserving information hiding. We envisage the new dynamic interface construct to be complementary to traditional immutable interfaces. The two constructs can be used consistently in applications to address different requirements.

We describe one implementation of the dynamic interface construct using agent concepts from the ROADMAP meta-model. We also describe some illustrating uses of the dynamic interface and provide early evaluation of the construct in terms of support for quality attributes such as performance.

References
[1] Bosch, J. Design and Use of Software Architectures Addison-Wesley, May, 2000