A Language and Environment for Rigorous Modeling and Analysis of e-commerce Architectures*

V. S. Alagar, V. Bhaskar, D. Muthiayen, O. Popistas, V. Srinivasan, Z. Xi

Department of Computer Science
Concordia University
Montreal, Quebec H3G 1M8, Canada
{alagar,bhaskar,d_muthi,popistas,sriniva,xi}@cs.concordia.ca

Abstract

The main issue in the development of agent-based architectures for e-commerce applications is to produce agents that collaborate with correct functional and temporal properties. They should be able to cope with large, complex, multi-lingual, and multi-platform systems. In order to achieve this potential, specific techniques for modeling, specification, analysis, and evolution of agent-based architectures must be provided. This paper advocates and presents one such approach.

1 Introduction

Internet offers a new way of marketing products and services that require distributed transactions to be carried out at prescribed times. Dependability and security are two major concerns for the developers and users of e-commerce systems. To deal with these concerns, it is crucial that e-commerce software be designed with a sound architecture. A good architecture simplifies the construction of large complex systems, and accomodates the design changes forced by a steady stream of new requirements. This paper describes a method, a language, a set of constructs, and tools that facilitate the design of architectures in e-commerce domain. More importantly, the presented approach enables rigorous validation and analysis of the design.

Software architectures provide a high-level view of the system, enabling developers to abstract on the relevant issues and focus on the "big picture". Most architectural descriptions for e-commerce applications are typically expressed only informally, in terms such as "client-server organization", "layered systems", "blackboard architecture" and in terms of box-and-line drawings showing the global organization of computational components and their interactions. More detailed descriptions of architectures are traditionally provided by Module Interconnection Languages (MILs) and Interface Definition Languages (IDLs). Informal descriptions are usually imprecise and incomplete. MIL and IDL descriptions focus on *implementation* relationships among parts of the system. Although such descriptions are essential, neither of them is suitable for analyzing the internal consistency, communication patterns, and the presence of the safety and security properties in an architecture. The distinction between abstract architectural descriptions and implementation-dependent descriptions, and the need to describe architectures at a level above their implementation details are clearly brought out in [4].

In this article we combine visual descriptions with rigorous notations. We use UML notational extensions as the visual architectural description language (VADL) to model e-commerce architectures. The developer uses only Rose [12] tools and a graphical user interface (GUI) [13] to model and analyze agent-based e-commerce applications. The graphical descriptions are mechanically translated into a formal architectural description language (FADL). The GUI provides the interface to compile the formal specification generated mechanically from UML diagrams, to validate, debug and reason about the architectural designs, and to verify safety and security properties.

Basic Concepts

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations. For us, software components are *agents*, where an agent is an autonomous and encapsulated system that *interacts* with the environment. continuous basis. The following properties characterize agents: (1) an agent always reacts to a stimulus from its environment, which is a collection of agents; (2) an agent has sufficient computational power to process and respond to a stimulus so that the time elapsed between a stimulus and its response is acceptale to the relative dynamics of the environment; (3) an agent is encapsulated - the internal state of an agent is not observable; and (4) an agent communicates with its environment through messages at its *ports*, where a port is an abstraction of an access point for a bidirectional communication between an agent and its environment.

An *agent type* is a generic reactive agent (GRA). Instances of agents having different environmental interactions can be created from an agent type and used in different designs. A typical e-commerce application involves four agent types: client, merchant, transaction system, and payment gateway. The agents created from these agent types interact by exchanging events. The system of interacting agents is secure if every message is responded within the specified time bound. These properties should be verified at the architecture level, prior to implementation and deployment of the system on the internet.

2 A Theory of Agents

An agent has a well-defined structure, interface, and behavior. Agents have no knowledge of other agents in their environment. Ports provide the only interface for an agent to communicate with its environment. An agent receives or delivers its messages at the ports attached to it. That is, agents do not communicate directly with other agents. Every message that occurs at a port is communicated along the connector that links that port to a *compatible* port, which is attached to another agent. It is the desiner's responsibility to ensure that the configuration of agents are sound and complete.

An agent type is parameterized with port types and specifies the time-dependent functionalities of agents of the type. A port type defines the set of events that can be received at a port of that type. An agent may have several ports of a particular type. Port reference identifiers and abstract data types are the only data members for an agent. The behavior of an agent is captured by a hierarchical state machine with logical assertions on the transitions and time constraints. By preserving the interface (for external communication) and behavior (for internal consistency) of an agent, the agent becomes reusable in a black-box fashion in different designs.

Figure 1 shows the template (type) of a generic reactive agent (GRA). The filled arrows in the figure indicate flow of events. An input event is the result of an incoming interaction defined by the external stimulus, the current state of the agent, and the port constrained by the port-condition. Every event occurrence causes a state transition and may also involve a computation. A computation updates the agent's state and attributes, shown by the arrow labeled with 'Att.Func.' The dotted arrow connecting the block of computation to that of time-constrained reaction signifies the enabling of a reaction due to a computation. Based on the clock, an outstanding reaction is fired by the agent, thereby generating either an internal event or an external event. All generated output events will result as a response at the port specified by the port-condition. A state update may also result in the disabling of an outstanding reaction.

The significant features of the abstraction are the following: (1) A GRA can be specified individually. (2) The GRA specification is not biased towards any language or implementation. (3) Timing constraints are encapsulated, thereby precluding an input event from being a time constrained event. That is, an agent has no control over the times of occurrences of input events since they are under the control of the environment. (4) An agent has complete choice over selecting the port (and hence the external agent) to communicate.

3 The Language

We use UML as the VADL for describing an architecture with reactive agents. The grammar for the FADL is based on a formal description of GRAs and subsystems.

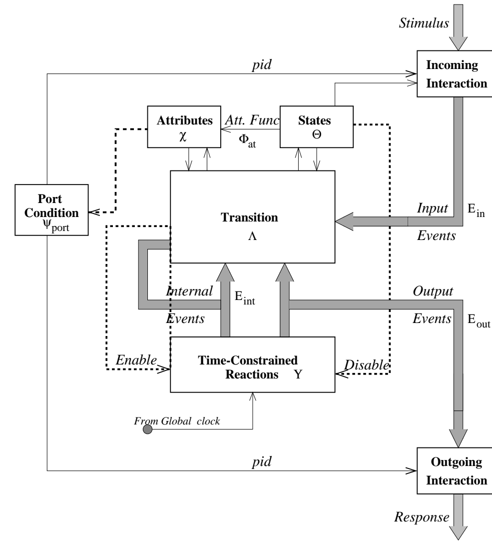

Figure 1: Anatomy of a Generic Reactive Agent

The Formal Language (FADL)

A formal definition of the different components of a reactive agent as described in Section 2 is given below. A full description of the grammar, the language semantics, and its expressive power in specifying real-ttime reactive systems have been discussed in [1, 2].

A reactive agent is an 8-tuple ($\mathcal{P}$, $\mathcal{E}$, Θ, $\mathcal{X}$, $\mathcal{L}$, Φ, Λ, Υ) such that:

- $\mathcal{P}$ is a finite set of port-types with a finite set of ports associated with each port-type.
- $\mathcal{E}$ is a finite set of events and includes the silent-event tick.
- Θ is a finite set of states. $\theta_0 \in \Theta$, is the *initial* state.
- $\mathcal{X}$ is a finite set of typed attributes. The attributes can be of one of the following two types: i) an abstract data type ; ii) a port reference type.
- $\mathcal{L}$ is a finite set of LSL traits introducing the abstract data types used in $\mathcal{X}$.
- Φ is a function-vector (Φ_s, Φ_{at}) where,
  - $\Phi_s : \Theta \to 2^{\Theta}$ associates with each state θ a set of states, possibly empty, called *substates*. A state θ is called *atomic*, if $\Phi_s(\theta) = \emptyset$.
  - $\Phi_{at} : \Theta \to 2^{\mathcal{X}}$ associates with each state θ a set of attributes, possibly empty, called the *active* attribute set.
- Λ is a finite set of *transition specifications* including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple : $< \langle\theta, \theta'\rangle; e(\varphi_{port}); \varphi_{en} \implies \varphi_{post} >$; where:

- $\theta, \theta' \in \Theta$ are the source and destination states of the transition;
- event $e \in \mathcal{E}$ labels the transition; φ_{port} is an assertion on the attributes in $\mathcal{X}$ and a reserved variable pid, which signifies the identifier of the port at which an interaction associated with the transition can occur.
- φ_{en} is the enabling condition and φ_{post} is the postcondition of the transition. φ_{en} is an assertion on the attributes in $\mathcal{X}$ specifying the condition under which the transition is enabled. φ_{post} is an assertion on the attributes in $\mathcal{X}$, primed attributes in $\Phi_{at}(\theta')$ and the variable pid and it implicitly specifies the data computation associated with the transition.

- Υ is a finite set of *time-constraints*. A timing constraint $\upsilon_i \in \Upsilon$ is a tuple $(\lambda_i, e'_i, [l, u], \Theta_i)$ where,
  - $\lambda_i \neq \lambda_s$ is a transition specification.
  - $e'_i \in (\mathcal{E}_{out} \cup \mathcal{E}_{int})$ is the *constrained event*.
  - $[l, u]$ defines the minimum and maximum response times.
  - $\Theta_i \subseteq \Theta$ is the set of states wherein the timing constraint υ_i will be ignored.

Figure 2 shows the template for agent specification in FADL. A subsystem architecture specification (SAS) is defined by composing reactive agents or by composing smaller subsystems. Figure 3 shows the template for a subsystem architecture specification: the include section lists subsystem architectures to be used in constructing the one under specification; the instantiate section lists agents with their attributes initialized and port-types instantiated with their cardinalities; the configure section specifies the connectors between agents.

GenericAgent $< name > [port - typelist]$
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end

Figure 2: Template for Generic Agent Specification.

UML Notation (VADL)

In this section we describe briefly how to model generic reactive agents and subsystem architectures in Rational Rose. A detailed discussion with examples appears in [11, 2]. It is assumed that the readers are familiar with UML and Rational Rose. Each subsection describes a component of the formal model introduced earlier.

Subsystem $< name >$
    Include:
    Instantiate:
    Configure:
end

Figure 3: Template for System Architecture Specification.

Generic Reactive Agent - A generic reactive agent is the basic abstract unit of a e-commerce system. UML provides an extension mechanism using stereotypes. We introduce generic reactive agents in Rose as classes with stereotype $\langle\langle GRA \rangle\rangle$. The class icon has three compartments in Rose: the name compartment, the attribute compartment and the operations compartment. The name compartment is mandatory, but the other two are optional.

Relationship between a GRA and its Port Types - Following the same extension mechanism with stereotypes as for GRA, we introduce port types in Rose as classes with stereotype$\langle\langle PortType \rangle\rangle$. The name of the port type class must start with the symbol "@". It has only one attribute, named events, of type Set, which is considered to be constant. The initial value of this attribute denotes the list of input (?) and output (!) events that can occur at a port of this type. We represent the relationship between a GRA and its port types as a binary association between the GRA symbol and the port type symbol in a class diagram (Figure 4). The association name is optional. The association end corresponding to the GRA must have the composition aggregation indicator, meaning that the GRA is a composite of port types.

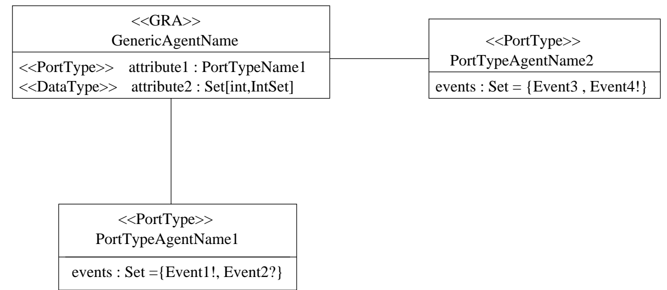

Figure 4: UML Class Diagram with one GRA and its two port types

Events and States - All the events that may occur for a generic reactive agent (input, output and internal) and its states are modeled in Rose using the statechart diagram belonging to the GRA. In Rose, the triggering event is defined as the event that triggers a state transition. Graphically, the event name is represented as the label on the correspond-

ing state transition. Input and ouput events are also listed in the attribute section of the corresponding port type classes as described above. Only one start state can exist in each statechart diagram. When applying nested state, one start state should be defined in each context. A state may be nested to any depth level. We may define complex states for a GRA using Rose's nested states concept.

Typed Attributes - All typed attributes of a generic reactive agent are represented in Rose as attributes of the GRA. For generic reactive agents, attributes can be of two types:port types and data types. Port type attributes are represented in Rose as attributes with the $\langle\langle PortType \rangle\rangle$ stereotype. The type expression must be the name of a port type class already defined in an aggregate association to the GRA. Data types such as set, queue, and relation are specified as Larch Shared Language traits [6]. Data type attributes are represented in Rose as attributes with the $\langle\langle DataType \rangle\rangle$ stereotype.

Transition Specifications and Time Constraints - Transition specifications in the generic model have the following components: initial and final state, triggering event, port-condition, enabling-condition and post-condition. These components are modeled in Rose using the guard and action features of the statechart diagram. The guard condition of a transition has the form:

> port-condition && enabling-condition && time-constaint condition.

The action of a transition has the following format:

> Post-condition && Time-constraint-initialization.

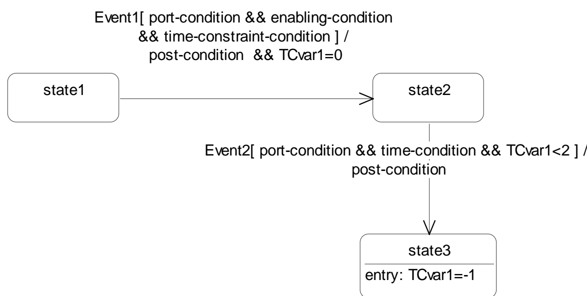

Figure 5: Constrained Events in Statechart Diagram

A time constraint in the generic agent model has the following components: constraining event, constrained event, lower and upper bounds and a set of disabling states. Time constraints will be introduced in Rose in the statechart diagram, as follows. For each time constraint, a reserved variable of type interger will be defined. This variable should be named *TCVarN*, where **N** is a numeral (for example TCvar1, TCvar2, etc.). This variable has to be initialized to 0 on the transition of the constraining event, as the second Action. On the transition corresponding to the constrained event, the *guard condition* has to include the time-constraint condition, as a third predicate. Figure 5 shows part of a statechart diagram illustrating the transition specifications and time constraints.

System Architecture Specifications - In Rose, we model a subsystem as a collarboration diagram. This collaboration diagram contains generic reactive agents instantiated with ports and port links for communication (Figure 6). The purpose of this collaboration diagram is to statically specify the architecture composed from agents, ports and port links. Therefore no messages should be shown on the links.

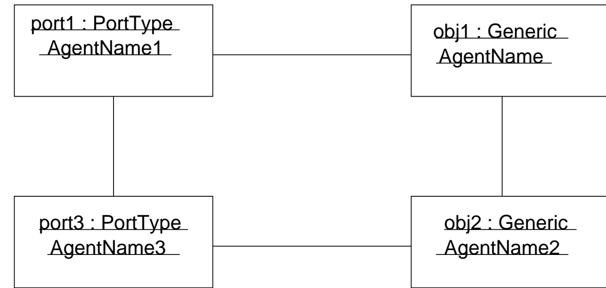

Figure 6: A Collaboration Diagram of an Architecture with two GRAs

A reactive agent is defined in Rose as an instance of a generic reactive agent. A port is defined in Rose as an object of the class named with the port type name. We define a reserved variable **pid** as the identifier of the port at which an interaction associated with a transition can occur. This reserved variable can be used in the logical assertions in the port-conditoins, enabling-conditions and post-conditions. An agent may have several ports of each port type defined for the class. This relationship is shown as a link between the agent and the port instances in the collaboration diagram of the subsystem. A link between two port objects belonging to two reactive agents defines a port link between the two objects.

4 The Environment

This section describes the environment which supports the modeling, analysis, evolution, and implementation of e-commerce architectures based on reactive agents. The environment includes support tools for UML modeling and translation to the formal specification, an interpreter for syntactic and semantic checking, a validation assistant, a verification assistant, a browser, and a graphical user interface.

Visual architectural models are created in Rational Rose [12]. The environment provides a link to Rational Rose, allowing the visual composition of reactive agents and subsystems in terms of UML classes, statecharts and collaboration diagrams. It also includes a translator [11] for generating formal specifications from the graphical descriptions.

The browser [9] provides a navigational link to a library of components including data models, generic reactive agents,

and subsystem configuration specifications. These components, stored in a repository, are checked for syntactic correctness and internal consistency by the interpreter. Hence, they may be reused or adapted for use in different architectures.

The interpreter [14, 13] performs lexical and semantic analysis on formal descriptions of agents and subsystems that are generated from the visual models. An internal representation of the architecture specification is used to instantiate reactive agents and perform validation and verification. The graphical user interface (GUI) [13] provides a comprehensive interfacing facility to the different tools in the environment. The interpreter, simulator, and verification assistants can be invoked through GUI.

The simulator [8] is the central piece of the validation assistant and it simulates computational steps of agents in the architecture based on the operational semantics of reactive agents. The simulation process is useful for validating agent performance, system properties, functional requirements and timing constraints, debugging system designs, incremental development of systems, and for providing a level of assurance that security and safety properties are not violated. During the simultation process, the clock can be frozen before handling the next event from the event-list. The state of the system at this point and the histories accumulated during the simulation of computational steps up to this point can be inspected, and analyzed to determine the causes of the current system status. The verification assistant [10] generates a set of axioms from the abstract syntax tree description of the modeled architecture created by the interpreter. The goal is to use them in formally verifying security and safety properties in the system. For some applications, this set of axioms may have to be supplemented with additional axioms.

5 Developing an e-Commerce Application - An Example

We consider a generic e-commerce system consisting of five components, Client, Internet, Merchant, Bank, and Data WareHouse. Many specific applications can be conceived based on the high-level architecture shown in Figure 7. We assume the following pattern of event-driven transactions in this model:

> The client provides the address for the webpage to the internet, which in turn contacts the respective merchant, receives a confirmation from the merchant and gives the message that the webpage is available for the client to view. The merchant informs the data warehouse to make available the catalogue to the client. The client can then select items and the required quantity from the webpage. After the selection of items, when the client is ready to make the purchase the client can request for the order form through the internet. The internet in turn requests the merchant for the order form. Once the client is satisfied with the order form, the client provides the credit card information and requests for the invoice. The internet in turn requests the merchant to give the invoice. The merchant gives the client information to the bank in order to check the validity of the client information.

The data warehouse is the repository of data from which the merchant, bank and client retrieve information in a secure manner. That is, at any instant the data revealed by the warehouse is completely dependent on the port at which a message is received. Since messages determine the port-types and these are determined at the architectural level, from the formal specifications we can verify the security of message communication and hence ensure the privacy of transactions. Moreover, at the expense of increased message traffic we reduce the amount of data transfer between the agents. A complete architectural design with Rose models and formal specifications are given in [3]. In this section we present only partial views and specifications of the architecture.

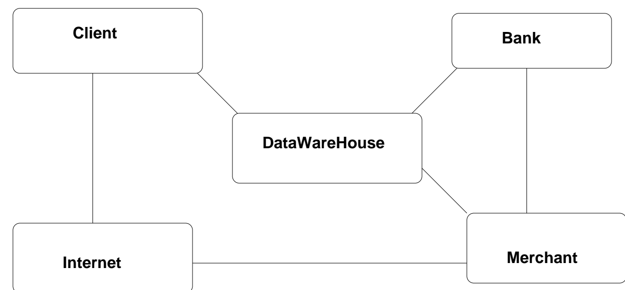

Figure 7: Generic e-commerce Architecture

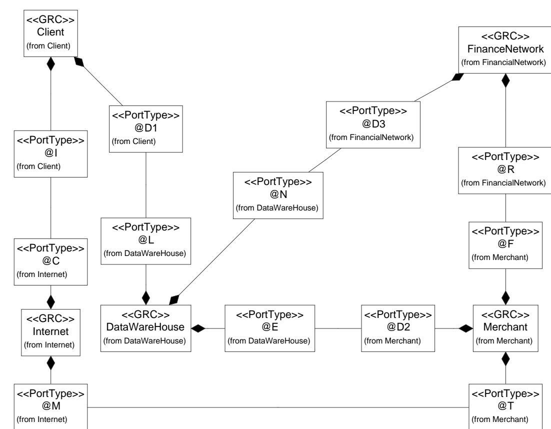

Figure 8: Main Class Diagram

Visual Models - VADL Descriptions

From the architecture shown in Figure 7 we determine the port types for each agent. From the description of the model we extract the messages for communication, and partition the messages according to the port-types. For instance, the

bank requires two port typess: one for communication with the merchant agent and the other for communication with the data warehouse. In our model the financial network consists of a gateway which regulates a secure communication between the bank and merchant and between the bank and data warehouse. The event names and the port types associated with them are shown in Table 1.

Following the method outlined in Section 3.2, we construct the UML class diagrams for the agents. The main class diagram for the full architecture is shown in Figure 8. The class diagram for the bank shown in Figure 9 describes the events that can occur at its port types, and the abstract data type *Queue* included in it. The queue data structure is necessary to enqueue the merchant requests received at the bank and service them on a first-in-first-out basis.

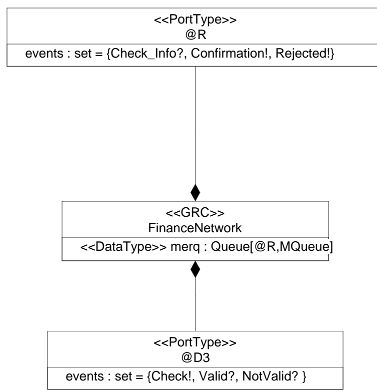

Figure 9: Class diagram for Bank

The dynamic behavior of the bank agent is captured in the statechart diagram shown in Figure 10. The agent is in *idle* state when there is no request to process. In the state *contacting* the agent has received a request to process a transaction. In the sate *waiting*, a request is being processed. In the state *validating*, the transaction has a valid credit card number. In the state *invalid*, the transaction has an invalid card number. In every state except *idle*, the agent enqueues requests received. In the state *waiting*, the request at the front of the queue is deleted. The events *Confirmation* and *rejected* are time constrained events in the sense that within the time interval [3, 10] from the instant the event *Check* occurred, either the transaction must be confirmed or rejected.

Figure 11 shows the architecture of an e-commerce subsystem with six clients, one internet, two merchants, two data warehouses, and three banks. The agents are linked through connectors at their respective compatible ports for communication. For instance, the port $@I1$ of client $C1$ is linked to the port $@C1$ of the internet $I1$. That link is not shared by any other agent. Consequently, the architecture specification ensures secure communication.

Formal Specifications - FADL Descriptions

The Rose-GRA translator [11] translates the visual models

Port @R - Merchant Agent	
Events	Meaning
Check_Info? (idle → contacting)	to validate card
Check_Info? (invalid_form → invalid_form)	same as above
Check_Info? waiting → waiting	same as above
Check_Info? (validating → validating)	same as above
Check_Info? (contacting → contacting)	same as above
Confirmation! (validating → idle)	confirm card
Confirmation! (validating → contacting)	same as above
Rejected! (invalid_form → idle)	reject card
Rejected! (invalid_form → contacting)	same as above
Port @D3 - DataWareHouse	
Events	Meaning
Check! (contacting → waiting)	request to validate
Valid? (waiting → validating	receive confirmation
NotValid? (waiting → invalid_form)	receive rejection

Table 1: Financial Network Events

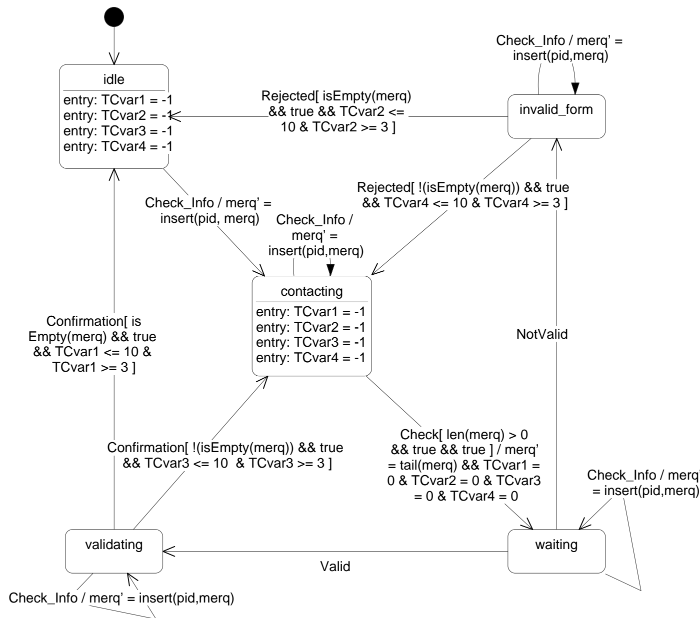

Figure 10: State Chart diagram for the Bank

into formal specifications in the syntax described in Section 2. The architecture of the translator is designed in such a way as to emphasize the information flow between the corresponding visual and formal components.

The user invokes the translator from the Rose Graphical User Interface, which is linked to the GUI in the environment. The interpreter in the environment can be invoked from the GUI to type check the formal specifications. The architectural design is both incremental and iterative. If errors are found in a specification, Rose tool can be reentered to modify visual models, translate and the interpret them. GUI privides the necessary facilities for such design evolution.

Figure 12 shows the formal specification generated by the translator for the e-commerce subsystem shown in Figure 11. The subsystem specification can be type checked and analyzed for semantic consistency. GUI provides facilities to invoke the simulator and view the simulated scenarios of the interpreted e-commerce system. Simulator results give insight into the temporal and functional behavior of the modeled system.

6 Conclusion

Architecture-based software development offers great potentials for software reuse and software evolution. This paper has adapted a technique developed in [2] for real-time reactive systems to the context of e-commerce applications. Since agents in an e-commerce application are reactive and time-dependent the adaption looks appropriate.

By improving the UML model and integrating it with formal notation, we have provided a two-tiered development technique that combines the advantages of visual modeling and

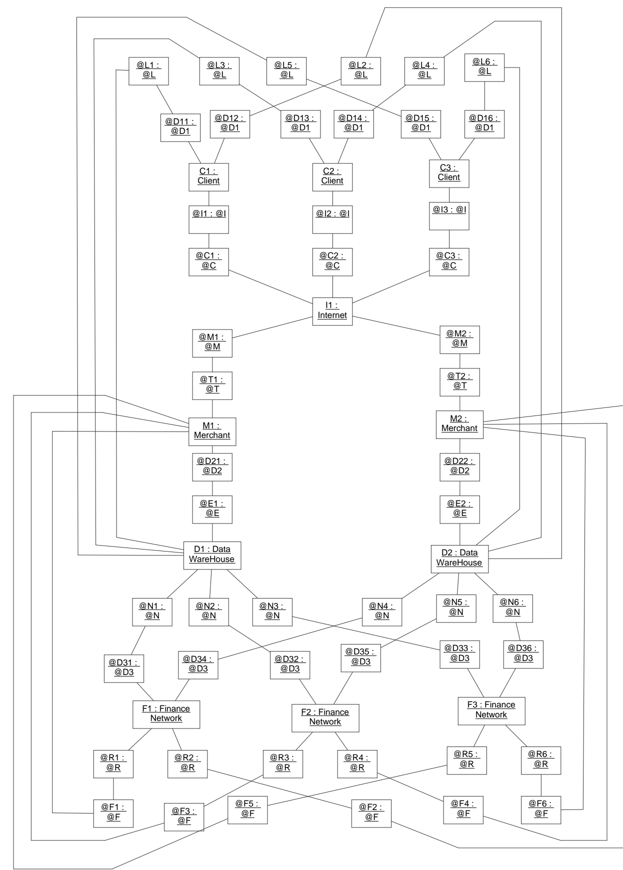

Figure 11: Subsystem Architecture

formal modeling. Application specialists use the widely accepted industrial standard Rose tool to create UML models. The Rose-GRA translator [11] translates them into formal specifications, that are rigorously analyzed by the tools in our environment. Thus, the architecture-based approach discussed in this paper has the great potetial for improving the quality of the resulting software through early analysis, validation, and verification.

Agent specifications can be refined through the addition of states and events, strengthening of timing constraints, and adding new port types. Ensuring specific properties at the architecture level is of little value unless they are also ensured in the resulting implementation. We are currently working on these issues. In addition, we are working on code generation and test-case generation methods from the architectural specifications.

REFERENCES

[1] R. Achuthan. *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems*. Ph.D. thesis, Concordia University, Montreal, Canada, October 1995.

[2] V.S. Alagar and D. Muthiayen, A Formal Approach to UML Modeling of Complex Real-Time Reactive Systems, Submitted for publication, September 2000.

[3] V.S. Alagar, V. Bhaskar and Z. Xi, Visual Modeling of e-Commerce Architectures, Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, August 1999.

[4] R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, July 1997.

[5] V. S. Alagar, D. Muthiayen, and R. Achuthan. Animating Real-Time Reactive Systems. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'96, Montreal, Canada, October 1996.

[6] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specifications.* Springer-Verlag, 1993.

[7] G. Haidar, *Reasoning System for TROMLAB Environment*, M.S. Thesis,(to be submitted in September 1999), Department of Computer Science, Concordia University.

[8] D. Muthiayen. *Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment.* Master's thesis, Concordia University, Montreal, Canada, October 1996.

[9] R. Nagarajan, *VISTA: A Visual Interface for Software Reuse in TROMLAB Environment*, 1999. M.S. Thesis, Department of Computer Science, Concordia University

[10] F. Pompeo, *A Verification Assistant for TROMLAB Environment*, M.S. Thesis, (to be submitted in September), Department of Computer Science, Concordia University.

[11] O. Popistas, *Rose-GRC translator: Mapping UML visual models onto formal specifications*, M.S. Thesis, Department of Computer Science, Concordia University, 1999.

[12] Rational Software Corporation, *UML Notation Guide, Version 1.1*, September 1997.

[13] V. VangalurSrinivasan *An Integrated Visual Interface for the Evolutionary Development of Reactive Systems in TROMLAB Environment*, M.S. Thesis (to be submitted in September 1999), Department of Computer Science, Concordia University.

[14] H. Tao. Static Analyzer: A Design Tool for TROM. Master's thesis, Concordia University, Montreal, Canada, August 1996.

```
SCS ECommerce
    Includes:
    Instantiate:
        F1::FinanceNetwork[@R:2, @D3:2];
        F2::FinanceNetwork[@R:2, @D3:2];
        F3::FinanceNetwork[@R:2, @D3:2];
        D1::DataWareHouse[@E:1, @N:3, @L:3];
        D2::DataWareHouse[@E:1, @N:3, @L:3];
        M1::Merchant[@T:1, @F:3, @D2:1];
        M2::Merchant[@T:1, @F:3, @D2:1];
        I1::Internet[@C:3, @M:2];
        C1::Client[@I:1, @D1:2];
        C2::Client[@I:1, @D1:2];
        C3::Client[@I:1, @D1:2];
    Configure:
        F1.@D31:@D3 ↔ D1.@N1:@N;
        F2.@D32:@D3 ↔ D1.@N2:@N;
        F3.@D33:@D3 ↔ D1.@N3:@N;
        D2.@N4:@N ↔ F1.@D34:@D3;
        D2.@N5:@N ↔ F2.@D35:@D3;
        D2.@N6:@N ↔ F3.@D36:@D3;
        D1.@E1:@E ↔ M1.@D21:@D2;
        D2.@E2:@E ↔ M2.@D22:@D2;
        M1.@T1:@T ↔ I1.@M1:@M;
        M2.@T2:@T ↔ I1.@M2:@M;
        I1.@C1:@C ↔ C1.@I1:@I;
        I1.@C2:@C ↔ C2.@I2:@I;
        I1.@C3:@C ↔ C3.@I3:@I;
        C2.@D13:@D1 ↔ D1.@L3:@L;
        C3.@D15:@D1 ↔ D1.@L5:@L;
        D1.@L1:@L ↔ C1.@D11:@D1;
        D2.@L2:@L ↔ C1.@D12:@D1;
        D2.@L4:@L ↔ C2.@D14:@D1;
        D2.@L6:@L ↔ C3.@D16:@D1;
        M1.@F1:@F ↔ F1.@R1:@R;
        M2.@F2:@F ↔ F1.@R2:@R;
        M1.@F3:@F ↔ F2.@R3:@R;
        M2.@F4:@F ↔ F2.@R4:@R;
        M1.@F5:@F ↔ F3.@R5:@R;
        M2.@F6:@F ↔ F3.@R6:@R;
end
```

Figure 12: Formal specification of Subsystem Architecture