SBES 2007
XXI Simpósio Brasileiro de Engenharia de Software

Using Interaction Laws to Implement Dependability Explicit Computing in Open Multi-Agent Systems

Rodrigo Paes, Carlos Lucena, Gustavo Carvalho

Laboratório de Engenharia de Software
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua M de S Vicente 225, Gávea – Rio de Janeiro – RJ – Brazil

{rbp, lucena, guga}@inf.puc-rio.br

Resumo. *Neste artigo, ilustra-se a aplicação das idéias de Dependability Explicit Computing (DepEx) em uma abordagem de leis de interação para a construção de sistemas multi-agentes abertos fidedignos (dependable). Mostra-se que as especificações das leis podem tratar explicitamente conceitos de fidedignidade, e auxiliar na coleta e publicação de dados sobre fidedignidade. Estes dados podem ser utilizados, por exemplo, para auxiliar na construção de aplicações guiando decisões tanto em tempo de projeto quanto em tempo de execução. As principais vantagens da utilização de uma abordagem de leis para a especificação de preocupações de fidedignidade são: (i) definição explícita das preocupações; (ii) coleta automática de metadados usando a infra-estrutura de mediadores presente na maioria das abordagens de leis; e (iii) habilidade de especificar estratégias para reagir a situações não-desejadas, auxiliando na prevenção de falhas de serviço.*

Abstract. *In this paper we propose to incorporate the Dependability Explicit Computing (DepEx) ideas into a law-governed approach in order to build dependable open multi-agent systems. We show that the law specification can explicitly incorporate dependability concerns, collect data and publish them in a metadata registry. This data can be used to realize DepEx and, for example, it can help to guide design and runtime decisions. The advantages of using a law-approach are (i) the explicit specification of the dependability concerns; (ii) the automatic collection of the dependability metadata reusing the mediators' infrastructure presenting in law-governed approaches; and (iii) the ability to specify reactions to undesirable situations, thus preventing service failures.*

1. Introduction

Many current systems are open and dynamic. A key characteristic is that they demand dynamic binding, i.e. the selection and use of components, or agents, at run-time. Therefore, they do not consist simply of agents selected during an on-line design activity. Instead, they are open to agents arriving, departing or being modified. They are dynamic in order to provide services on a continuous basis, and do so even when agents or the environment change [Serugendo et al. 2006]. The greater the dependence of our society on such open distributed multi-agent systems, the greater will be the demand for dependable applications.

The dependability of a system can be defined as the ability to avoid service failures that are more frequent and more severe than is acceptable [Avizienis et al. 2004]. Dependability is an integrating concept that encompasses the following attributes [Avizienis et al. 2004]: (i) availability: readiness for correct service; (ii) reliability: continuity of correct service; (iii) safety: absence of catastrophic consequences on the user(s) and the environment; (iv)

integrity: absence of improper system alterations; and (v) maintainability: ability to undergo modifications and repairs.

One way to promote dependability is by implementing a Dependability Explicit Computing (DepEx) approach [Kaâniche et al. 2000]. DepEx treats dependability metadata as first-class data. The means for dependability (i.e., fault prevention, fault tolerance, fault forecasting and fault removal) should be explicitly incorporated in a development model focused on the production of dependable systems [Kaâniche et al. 2000].

While developing the system, the data is specified by explicitly incorporating dependability-related information into system development from the earliest possible phases; by annotating design and implementation files with dependability-related metadata, which are updated when the files are processed (e.g. when development moves from one phase to another); and by maintaining this information to reflect the system and environment states afterwards.

Then, at the runtime or at the design time, the dependability metadata can be exploited to aid decision-making. Some examples of metadata are safety integrity level, failure rates, failure modes, pre and post conditions, MTBF, reliability, response time, resources consumed, component specification, fault assumptions, types of encryption, etc.

Achieving dependability in open multi-agent systems is particularly challenging. Such systems are characterized by having little or no control over the actions that agents can perform. Besides, the internal aspects of the agents (such as implementation language and architecture) are inaccessible. The research in interaction laws deals with this problem by explicitly specifying behavioral rules, and by providing mechanisms that check if the actual interactions conform to the specification at runtime. The mechanisms are usually implemented by either a central mediator [Paes et al. 2006] or by a decentralized community of mediators [Minsky and Ungureanu 2000]. These mediators perform the active role of monitoring the interaction among the agents and interpreting the laws to verify if the actual system behavior is in conformance with the specifications.

In this paper we propose to incorporate the Dependability Explicit Computing ideas into a law-governed approach in order to build dependable open multi-agent systems. We show that the law specification can explicitly incorporate dependability concerns, collect data and publish them in a metadata registry. This data can be used to realize DepEx and, for example, it can help to guide design and runtime decisions. The advantages of using a law-approach are (i) the explicit specification of the dependability concerns; (ii) the automatic collection of the dependability metadata reusing the mediators' infrastructure presenting in law-governed approaches; and (iii) the ability to specify reactions to undesirable situations, therefore, preventing service failures.

This paper is organized as follows. In Section 2, we present a flexible law-governed approach called XMLaw. We use this approach throughout the examples given in this paper. In Section 3 we present in details of a case study where we discuss how the laws can be used to promote Dependability Explicit Computing. In Section 4, we specifically relate our research to previous work, explaining how the problem of promoting dependability in open multi-agent systems has been addressed so far. Finally, in Section 5, we present some discussions about this and future work.

2. XMLaw

Law-governed architectures are designed to guarantee that the specifications of open systems will be obeyed. The core of a law-governed approach is the mechanism used by the mediators to monitor the conversations between components. M-Law [Paes et al. 2007]

[Paes et al. 2006] is a middleware that provides a communication component, or mediator, for enforcing interaction laws. M-Law was designed to allow extensibility in order to fulfill open system requirements or interoperability concerns.

The middleware was built to support law specification using XMLaw [Paes et al. 2005], [Paes at al. 2007b]. XMLaw is used to represent the interaction rules of an open system specification. For readability purposes the codes written in XMLaw presented in this paper use a simplified syntax that is more compact than the one used in early XMLaw publications. These rules are interpreted by the M-Law mediator that, at runtime, analyzes the compliance of agents with interaction law specifications. A law specification is a description of law elements that are interrelated in a way that makes it possible to specify interaction protocols using time restrictions, norms, or even time sensitive norms. XMLaw follows an event-driven approach, i.e., law elements communicate by the exchange of events.

The conceptual model of XMLaw is composed of the following main elements: {Event, Protocol, Transition, State, Scene, Clock, Norm, Constraint, Action}. The elements are described as follows.

Event - an event models an occurrence related to the elements of the law. It can represent a change of state of a protocol, the arrival of a message sent by an agent, the moment in which an agent acquires an obligation, an announcement that a certain amount of time has elapsed and much more. The semantic of each event is determined by its type. There are many types of events, which are summarized as follows. Type of events = {*message_arrival, compliant_message, transition_activation, failure_state_reached, successful_state_reached, failure_scene_completion, sucessful_scene_completion, scene_creation, time_to_live_elapsed, clock_activation, clock_tick, clock_timeout, clock_deactivation, norm_activation, norm_deactivation, norm_not_fulfilled, constraint_not_satisfied, action_activation*}.

Protocol - A protocol defines the possible states that an agent interaction can evolve. Transitions between states can be fired by any XMLaw event, instead of only message arrivals. Therefore, protocols specify the expected sequence of events in the path of interaction among the agents.

Transition - a transition is a directed arc between a source state and an end state. It represents the change between two different situations in the course of the interactions caused by a response to the occurrence of an XMLaw event. Transitions may depend on norms and constraints to fire. If there is an obligation associated with the transition, then the obligation must be inactive in order to activate the transition. Instead, if there is a permission associated with the transition, the permission must be active in order to fire the transition. Constraints may act as fine-grained filters for transitions. A constraint could access a database, do some math, calculate date periods or perform any other complex domain-dependent operation in order to allow the transition to fire.

State - A state models a possible step in the evolution of the agents' interaction. States can represent static or dynamic situations, such as "*waiting for buyer's answer*", "*deciding about the deal*", and so on. There are three types of states: successful, failure or execution. A successful state means that the protocol stops upon reaching success. A failure state means that the protocol stops with failure when the state is reached. For its part, when reached, an execution state does not stop the protocol.

Scene - Scenes help organize interactions. The concept of scenes here is similar to those in theater plays, where actors play a role according to well defined scripts, and the entire play is composed of many connected scenes. A scene models an interaction context where protocols, actions, clocks and norms can be composed to represent complex normative

situations. Furthermore, from the problem modeling point of view, a scene permits decomposing the problem into smaller and more manageable pieces of information. They can be viewed as building blocks of normative interactions. Normative interactions are situations in which agents interact through a set of behavioral rules, or social conventions.

Clock - Clocks represent time restrictions or controls and can be used to activate other law elements. Clocks indicate that a certain period has elapsed producing clock_tick events. Once activated, a clock can generate clock_tick events. Clocks are activated and deactivated by law elements.

Norm - A Norm [Paes et al. 2005] [Paes at al. 2007b] is an element used to enable or disable agents' conversation paths. For instance, a norm can forbid an agent to interact in a negotiation scene. There are three types of norms with different semantics in XMLaw: obligations, permissions and prohibitions. The obligation norm defines a commitment that software agents acquire while interacting with other entities. For instance, the winner of an auction is obligated to pay the committed value and this commitment might contain some penalties to avoid breaking this rule. The permission norm defines the rights of a software agent at a given moment, e.g. the winner of an auction has permission to interact with a bank provider through a payment protocol. Finally, the prohibition norm defines forbidden actions of a software agent at a given moment; for instance, if an agent does not pay its debts, it will not be allowed future participation in a scene. The structure of the Permission (Table 1), Obligation and Prohibition elements are equal. Each type of norm contains activation and deactivation conditions. In Table 1, an assembler will receive the permission upon logging into the scene (scene activation event called negotiation) and will lose the permission after issuing an order (event orderTransition). Furthermore, norms define the agent role that owns it through the second parameter. In Table 1, the assembler agent ($assembler) will receive the permission. Norms can also have constraints and actions associated with them. Norms also generate activation and deactivation events. For instance, as a consequence of the relationship between norms and transitions, it is possible to specify which norms must be made active or deactivated for firing a transition. In this sense, a transition only could fire if the sender agent has a specific norm.

Table 1 - XMLaw specification of the permission structure

```
// norm definition
01: assemblerPermissionRFQ{permission, $assembler, (negotiation), (orderTransition)
// constraint declared in the context of the norm
02:    checkCounter{br.pucrio.CounterLimit}
// actions declared in the context of the norm
03:    permissionRenew{(nextDay), br.pucrio.ZeroCounter}
04:    rfqTransition{(rfqTransition), br.pucrio.RFQCounter}
05: } //end norm definition
```

Constraint - A constraint [Paes et al. 2005] [Paes at al. 2007b] is a restriction to norms or transitions and, generally, it specifies filters for events, constraining the allowed values for a specific attribute of an event. For instance, messages carry information that is enforced in various ways. A message pattern enforces the message structure fields. A message pattern does not describe what are the allowed values for specific attributes, but constraints can be used for this purpose. In this way, developers are free to build as complex constraints as needed for their applications. Constraints are defined inside Scene (Table 2) or Norm (Table 1) elements. Constraints are implemented using Java code. The Constraint element defines the class attribute that indicates the Java class that implements the filter. This class is called when a transition or a norm is supposed to fire, and basically the constraint analyzes if the message values or any other events' attributes are valid. Table 2 shows a

constraint that verifies if the date expressed in a message is valid; if it is not, the message will be blocked. In Table 1, a constraint is used to verify the number of messages that the agent has sent until now; if it has been exceeded, the permission is no longer valid.

Table 2 - Constraint checkDueDate used by a transition

```
01: negotiation{
...
09:    t1{s1->s2, rfqMsg, [checkDueDate]}
...
14:    checkDueDate{br.pucrio.ValidDate}
...
20:} // end scene
```

Action - An action supports the definition of the moment when the mediator should call a domain-specific service. Actions are domain-specific Java code that runs in an integrated manner with XMLaw specifications. Actions can be used to plug services into a governance mechanism. For instance, a mechanism can call a debit service from a bank agent to charge the purchase of an item automatically during a negotiation. In this case, we specify in the XMLaw that there is a class that is able to perform the debit. Of course, this notion could also be extended to support other technologies instead of Java, such as direct invocation of web-services. In XMLaw, an action can be defined in three different scopes: Law, Scene and Norms. Since actions are also XMLaw elements, they can be activated by any event such as a transition activation, a norm activation and even an action activation. The action structure is shown in the example of Table 1 at lines 03 and 04 (in this example: a norm action). The class attribute of an Action specifies the Java class in charge of the functionality implementation. The first parameter references the events that activate this action, and as many events as needed can be defined to trigger an action.

2.1. XMLaw for dependability

The flexibility achieved by using the event-driven approach at a high-level of abstraction is not present in the other high level law approaches [Esteva 2003] [Dignum et al. 2004]. The advantages claimed in favor of the use of events as a modeling element are also present in LGI [Minsky and Ungureanu 2000], however at a low level of abstraction. A flexible underlying event-based model as presented in XMLaw can allow conceptual models for governance to be more prepared to accommodate changes. This is specially needed when we consider using the law-approach to deal with new concerns not considered in its original specification, such as dependability. For this reason, we have used XMLaw to specify and implement our case study.

2.2. Grammar

Table 3 shows a simplified version of the XMLaw’s grammar. The laws in XMLaw were originally written in an XML-based language [Paes et al. 2006] [Paes et al. 2007] [Paes et al. 2005] [Paes at al. 2007b]. That is the reason for the name XMLaw. However, the simplified notation presented here allows for a much clearer and compact specification of laws.

Table 3 – XMLaw Simplified Grammar

```
General Syntax:
| OR
[] optional
'' reserved symbol

Message = message-id'{'sender','addressee','content'}'
Transition =transition-id'{'sourceState'->'destinationState',' message-id '}'
        transition-id'{'sourceState'->'destinationState',' message-id ',' Lists '}'
Lists = '[' list of constraints ids ']' |
        '[' list of norms ids ']' |
        '[' list of constraints ids ']' ',' '[' list of norms ids ']'
Clock = clock-id'{' Time ',' Clock_Type ',' ActivationEvents ',' DeactivationEvents '}'
Time: IntegerLiteral[Unit]
Unit: 's' | 'm' | 'h' | 'd'
Clock_Type = 'periodic' | 'regular'
ActivationEvents = Events
DeactivationEvents =  Events
Events = '('')' |
        '('ElementRef')' |
        '('ListsOfElementsRef')' |
        '('element-id'..'element-id')'
ElementRef = element-id |
        '('element-id ',' event-type ')'
Contraint = constraint-id'{'java-class'}'
Action = action-id'{'ActivationEvents ',' java-class'}'
Norm = norm-id'{' NormType ',' owner ',' ActivationEvents ',' DeactivationEvents '}'
NormType = 'obligation' | 'permission' | 'prohibition'
```

3. Implementing DepExp Using XMLaw

In this section, we present a motivating case study to illustrate how to specify the laws in such a way that the dependability metadata is treated as first-class data. The problem description was already reported in [Yi and Kochut 2004], and it was slightly modified to this case study.

3.1. Problem Description

Consider the task of creating a system composed of three types of agents: a travel agent, a user agent and an airline agent. The airline provides several related operations, which must be invoked according to a complex conversation protocol. Assume that the airline agent provides five different operations: *checkSeatAvailability, reserveSeats, cancelReservation, bookSeats,* and *notifyExpiration.* Each operation performs a single airline travel related task. The operations must be invoked by a client according to the following conversation rules:

- *checkSeatsAvailability* must be the first operation to be invoked;
- *reserveSeats* may only be invoked if a client has already invoked *checkSeatsAvailability* and the requested seats are indeed available; the reservation is held only for a certain amount of time;

- *bookSeats* or *cancelReservation* may be invoked, but only if the seats have been reserved (by a successful invocation of *reserveSeats*) and the reservation has not expired;
- if neither *bookSeats* nor *cancelReservation* has been invoked by the client within a specified amount of time, the airline agent will itself invoke *notifyExpiration* to inform the client that the reservation has expired.

A traveler, represented by the user agent, planning on taking a trip submits a *TripOrder* (through *getItinerary* message*)* to her travel agent, hoping to get an *Itinerary* proposal. The *TripOrder* contains information such as departure and destination, departure date and time, and return date and time (for a round trip), the number of maximum connections and the number of travelers.

The travel agents finds the best itinerary to reach the destination based on the traveler's criteria such as the cheapest fare, availability of flights, or frequent flyer miles accumulated by the traveler. Before the *Itinerary* can be proposed to the traveler, the travel agent invokes the airline agent to verify the availability of seats (*checkSeatsAvailability*). In the event the seats are available, the travel agent notifies the traveler and waits for the traveler to submit a modified *TripOrder*.

If seats are available, the proposed *Itinerary* is sent to the traveler for confirmation. She then decides to reserve the seats for the *Itinerary* and gives the travel agent her contact information so that the airline agent will be able to send her an e-Ticket.

Next, the travel agent interacts with the airline agent to electronically finalize the reservation (*reserveSeats*). Let us assume that the airline holds such reservation for one day, and that if a *BookRequest* is not received within one day, the seats are released and the travel agent is notified. The travel agent sends a *ReserveResult* message to the traveler as an acknowledgment.

At this point, the traveler can either book or cancel the reservation. If she decides to book the trip, she sends a *BookRequest* to the travel agent containing her credit card information. The travel agent then invokes the *bookSeats* operation of the airline agent to finally book the seats. As a result, the airline agent books the seats for the proposed *Itinerary*, and issues an e-ticket to the traveler.

3.2. Architecture

The architecture of the system is shown in Figure 1. The architecture is based on the metadata architectural model presented in [Serugendo et al. 2006]. The architecture was conceived for the achievement of predictable levels of dynamic resilience in distributed systems. We have chosen this architecture because it already contains the components to enable DepExp. It has a metadata registry, a runtime reasoning/adaptation service and a metadata acquisition component.

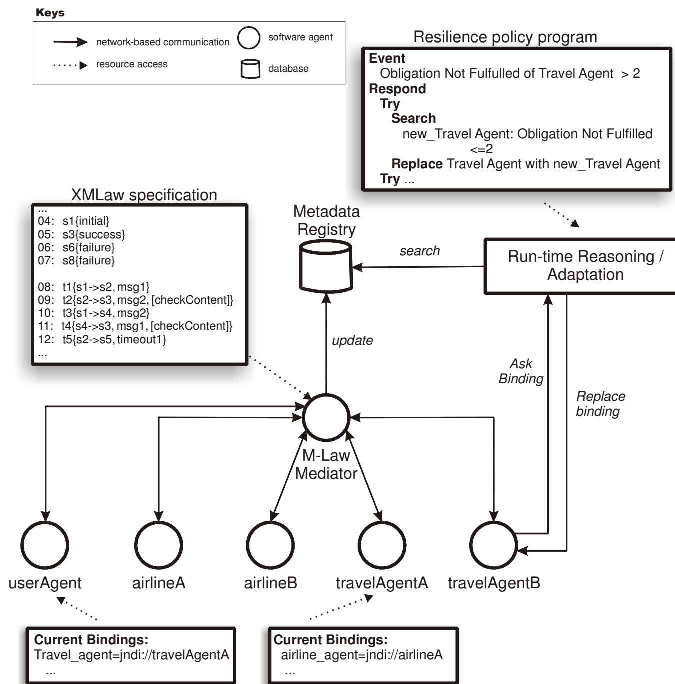

Figure 1- System architecture

In this case study, the role of the metadata acquisition component is performed by the M-Law middleware. M-Law works by intercepting messages exchanged between agents, verifying the compliance of the messages with the laws and subsequently redirecting the message to the real addressee, if the laws allow it. If the message is not compliant, then the mediator blocks the message and applies the consequences specified in the law (Figure 2). This architecture is based on a pool of mediators that intercept messages and interpret the previously described laws. A more detailed explanation about how this architecture was in fact implemented can be found in [Paes et al. 2006]. As more clients are added to the system, additional mediators' instances can be added to improve throughput.

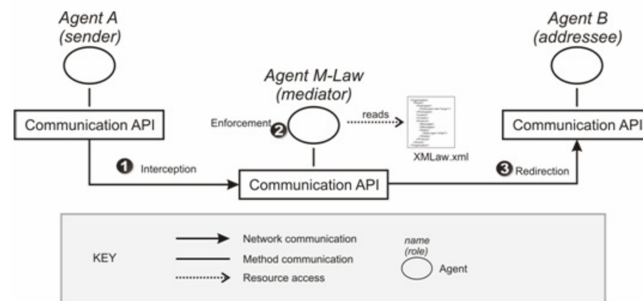

Figure 2 – M-Law architecture

The M-Law mediates the communication between the agents. The behavior of the M-Law is specified in the XMLaw file that it reads. In the XMLaw specification there are instructions that tell the mediator how to update the metadata registry. At runtime, the metadata registry can be used in two ways: directly by the agents or through the runtime reasoning. The agents can proactively search for metadata and self-adapt to reflect their dependability requirements. As an example, a user agent can search for a travel agent that has not broken any obligation during the last month.

In this case study, there are two available travel agents (*travelAgentA* and *travelAgentB*). At runtime, the user agent is able to choose which travel agent she will interact with based on the dependability metadata. The travel agents have two available airline agents with which they can interact. The choice of which of them is used can be also based on the dependability metadata available at the metadata registry

The runtime reasoning provides different tasks related to the processing of metadata stored in the metadata registry, such as comparison/matching of metadata, determination of equivalent metadata information and composition of metadata [Serugendo et al. 2006]. This service manages the list of agents, seamlessly activating or connecting the ones that will be used according to a specified resilience policy program. As an example, in the Figure 1, the *travelAgentB* is interacting with the runtime reasoning. In this paper, we focus on how we can specify the laws to automatically update the metadata registry.

3.3. The meta-data

We are using the laws to specify metadata concerning availability, service failure and enforcement of pre and post conditions.

- Availability – every time an agent sends a request to other agent, the receiver should answer within a pre-specified amount of time. The absence of an answer implicates that at that time the receiver is not available with the required quality level.
- Service failure – during the interaction, agents acquire obligations that they must fulfill. The fulfillment of these obligations represents the expected correct behavior for the agents. Therefore, each obligation that is not fulfilled can be interpreted as a service failure, i.e., the actual system execution deviates from the correct behavior.
- Pre and post conditions - Agent specifications may likely change as agents evolve, but resilience may be maintained if it is possible to reason dynamically about the functional properties of agents, including abnormal behaviors. Dynamic resilience mechanisms require agent specifications as metadata, including both specifications of services offered and services required. These can be given by pre-/post-conditions, potential failure behaviors and responses when the components are used outside their pre-conditions. As an example of pre-condition, let us suppose that we want to enforce that the values of the departure and destination attributes specified in the *TripOrder* (*getItinerary* message) must belong to the set of possible attributes *S={"Toronto", "New York", "London", "Tokyo", "Rio de Janeiro"}*. Enforcing this constraint guarantees that the travel agent is receiving a parameter that is within its specification scope.

Therefore, considering the case study description (Section 3.1) and the metadata concerns described above, it is possible to specify the following law requirements:

Requirement #1 - The whole process must occur within two days. After two days, the process is cancelled and all the rules are no longer valid. All the interactions must restart.

Requirement #2 – All interactions must occur as the pre-defined order specified in the problem description (Section 3.1).

Requirement #3 - If the airline agent says that there is a seat available, this seat must be saved to the travel agent for at least five minutes. This way, the user has some time for deciding about the confirmation of the reservation. If the time has elapsed, and the airline has not received any confirmation, then the airline is allowed to answer with a *not-available* message and book the seat for another client.

Requirement #4 - When the airline agent sends a *result-ok* message in response to a seat reservation, then the reservation must be held for at least one day

Requirement #5 - The *TripOrder* (*getItinerary* message) must belong to the set of possible attributes *S={"Toronto", "New York", "London", "Tokyo", "Rio de Janeiro"}*.

Requirement #6 - Every request that does not require user interaction must be answered within 15 seconds by any agent.

3.4. Metadata acquisition through XMLaw specification

The interaction protocol is shown in Figure 3 and the complete XMLaw specification can be found in Table 5. The scene is declared in lines 01 and 02. Lines 03 to 16 contain the pattern of messages that agents are expected to exchange. Lines 17 to 20 specify the initial and final states of the interaction protocol. The transitions are specified in lines 21 to 37. The transitions refer to the states, messages, constraints and norms present in the law. Clocks are specified in lines 38 to 40, constraint in line 41, actions in lines 42 to 44, and norms in lines 45 and 46. Next, we show how the six law requirements were specified in the laws.

Requirement #1: This requirement is implemented as the *time-to-live* scene attribute in line 02.

Requirement #2: The interaction protocol in Figure 3 reflects exactly the possible paths of interactions described for the case study. This protocol is specified in lines 03 to 37. These lines declare all messages, states and transitions present in the protocol.

Requirement #3: This requirement demands a combined use of various XMLaw elements. First, it is necessary to identify when the airline agent "says there is a seat available". Then, we have to start to count five minutes. The airline is not allowed to answer *not-available* within these five minutes. Table 4 shows the sequence of observed events that makes it possible to specify this requirement. This table is mapped to the XMLaw specifications in lines 35, 39, 43 and 45 (highlighted in Table 5).

Table 4 – Rationale for the XMLaw specification of the requirement #3 in lines 35, 39, 43 and 45.

Airline agent sends *itinerary-1* message to the travel agent. It means that airline agent is saying: "there is a seat available". Then, we activate the clock to start counting the time. Moreover, we also activate the obligation *hold-seat*, and give it to the airline agent.
WHEN (t3, transition_activation) ACTIVATE hold-seat-clock, hold-seat
If the time that the airline must hold the seat has elapsed (*clock_tick* event), then the obligation does not need to be fulfilled, i.e., the airline agent can answer with a *not-available* message.
WHEN (hold-seat-clock, clock_tick) DEACTIVATE hold-seat
If the airline agent answers with a *result-ok* to a *reserveSeats* requisition, it means that the airline agent has fulfilled its obligation of holding the seat. Then, the obligation should be deactivated.
WHEN (t7, transition_activation) DEACTIVATE hold-seat
The transition *t15* only fires if the obligation *hold-seat* is deactivated. If the airline sends the *not-available* message while the obligation is still active, then a *norm_not_fulfilled* event will be generated and the transition will not fire. As we are concerned with acquiring metadata about agents that do not follow the rules, the non-fulfillment of an obligation should be reported to the metadata registry. The action *updateHoldSeatMetadata* is in charge of obtaining the contextual information such as the agent, the obligation id (in this case *hold-seat*) and updating the registry.
WHEN (hold-seat, norm_not_fulfilled) ACTIVATE updateHoldSeatMetadata

Requirement #4 – This requirement is specified in XMLaw using an idea similar to requirement #3. The transition *t16* (line 36) only fires if the obligation *hold-reservation* (line 46) is deactivated. The clock *hold-reservation-clock* (line 40) counts the time until one day. And the action *updateHoldReservationMetadata* updates the metadata registry with information about agents that do not fulfill the obligation.

Requirement #5 – The constraint *checkContent* specified in line 41 is invoked by the transition *t1* (line 21). The constraint verifies if the values of the variables dep and dest (line 03) belong to the set of the pre-defined cities. The constraint implementation is shown in Table 6.

Requirement #6 – This requirement states that agents that do not wait for input from the user must not take too long to provide an answer. The *availability-clock* specified in line 38 counts 15 seconds every time an agent receives a request. The clock is reset when the agent answers the request. The transitions *t1,t2,t3,t5,t6,t7.t9,t10,t11,* and *t13* specified in the clock, represent requests to agents. Note that transitions such as *t4* are not present in this list. This is because *t4* represents a message that is sent to the user (through the user agent). Every time an agent does not answer within the 15s, the clock generates a *clock_tick* event. This event is listened to by the action *updateClockMetadata* (line 42). The action updates the metadata registry indicating that the agent was not available at that time. The code for this action is shown in Table 7.

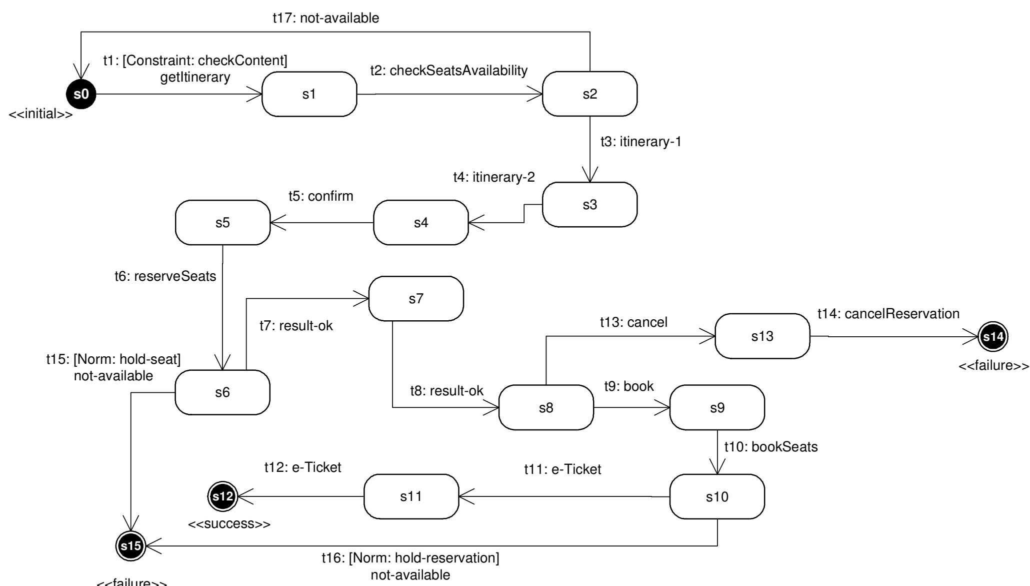

Figure 3 – Protocol Specification

Table 5 – XMLaw specification

```
// scene specification
01:planningATrip{
02:   time-to-live=2d
// pattern of messages
03:       getItinerary{userAgent,travelAgent,trip_order($dep,  $dest,  $depDate,  $depTime,
$retDate, $retTime, $maxCon, $travellers)}
04:   checkSeatsAvailability{travelAgent,airlineAgent, $trip_order}
05:   itinerary-1{airlineAgent, travelAgent, itinerary($id,$details)}
06:   itinerary-2{travelAgent, userAgent, itinerary($id,$details)}
07:   confirm{userAgent, travelAgent, confirm($id)}
08:   reserveSeats{travelAgent, airlineAgent, reserveSeats($id)}
```

```
09:   result-ok-1{airlineAgent, travelAgent, ok($id)}
10:   result-ok-2{travelAgent, userAgent, ok($id)}
11:   book{userAgent, travelAgent, book($id)}
12:   bookSeats{travelAgent, airlineAgent, bookSeats($id)}
13:   e-Ticket{$sender, $receiver, e-ticket($ticketId)}
14:   cancel{userAgent, travelAgent, cancel($id)}
15:   cancelReservation{travelAgent, airlineAgent, cancelReservation($id)}
16:   not-available{airlineAgent, travelAgent, not-available($id)}

// initial and final states
17:   s0{initial}
18:   s12{success}
19:   s14{failure}
20:   s15{failure}

// transitions
21:   t1{s0->s1, getItinerary, [checkContent]}
22:   t2{s1->s2, checkSeatsAvailability}
23:   t3{s2->s3, itinerary-1}
24:   t4{s3->s4, itinerary-2}
25:   t5{s4->s5, confirm}
26:   t6{s5->s6, reserveSeats}
27:   t7{s6->s7, result-ok-1}
28:   t8{s7->s8, result-ok-2}
29:   t9{s8->s9, book}
30:   t10{s9->s10, bookSeats}
31:   t11{s10->s11, e-Ticket}
32:   t12{s11->s12, e-Ticket}
33:   t13{s8->s13, cancel}
34:   t14{s13->s14, cancelReservation}
35:   t15{s6->s15, not-available, [hold-seat]}
36:   t16{s10->s15, not-available, [hold-reservation]}
37:   t17{s2->s0, not-available}

// Clocks
38:           availability-clock{15s,    regular,    (t1,t2,t3,t5,t6,t7.t9,t10,t11,t13),
(t2,t3,t4,t6,t7,t8,t10,t11,t12,t16,t17)}
39:   hold-seat-clock{5m, regular, (t3), (t6)}
40:   hold-reservation-clock{1d, regular, (t7), (t10)}

// Constraints
41:   checkContent{br.pucrio.CheckContent}

// Actions
42:   updateClockMetadata{(availability-clock), br.pucrio.DecAvailability}
43:   updateHoldSeatMetadata{((hold-seat, norm_not_fulfilled)), br.pucrio.HoldSeat}
44:           updateHoldReservationMetadata{((hold-reservation,    norm_not_fulfilled)),
br.pucrio.HoldReservation}
// Norms
45:   hold-seat{obligation, airlineAgent, (t3), (hold-seat-clock, t7)}
46:   hold-reservation{obligation, airlineAgent, (t7), (hold-reservation-clock , t11)}
47:}
```

Table 6 – Java Implementation of the CheckContent Constraint

```
class CheckContent implements IConstraint{
    private static List<String> allowed = new ArrayList<String>();
    private void init(){
        allowed.add("Toronto");
        allowed.add("New York");
        allowed.add("London");
        allowed.add("Tokyo");
        allowed.add("Rio de Janeiro");
    }
    public boolean constrain(ReadonlyContext ctx){
        String dep = ctx.get("dep");
        String dest = ctx.get("dest");
        if ( !allowed.contains(dep) || !allowed.contains(dest) ){
            return true; // constrains, transition should not fire
        }
    }
}
```

Table 7 – Action updateClockMetadata implemented as the java class DecAvailability

```
class DecAvailability implements IAction{
    private Datasource metadataRegistry;
    ...
    public void execute(Context ctx){
        String addressee = ctx.get("lastAddressee");
        Event event       = ctx.get("activationEvent");
        metadataRegistry.insert(event, addressee);
    }
}
```

3.5. The Metadata registry

In this case study, the metadata registry is composed of two entities: *agent* and *dependability_data*. The entity-relationship model is presented in Figure 4, and it is described in Table 8.

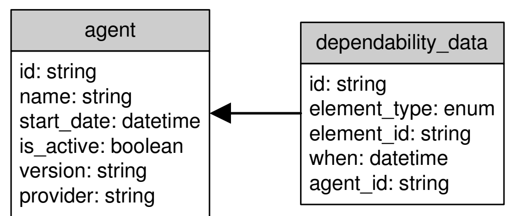

Figure 4 – Entity-relationship model of the metadata registry

Table 8 – Description of the Attributes

agent	dependability_data
id – unique database identifier of the agent.	id – unique database identifier of the data.
name – the unique name of the agent	element_type – type of the XMLaw element. (eg: obligation, clock, …)
start_date – date when the agent was added to the registry	element_id – id if the XMLaw element
is_active – true if the agent is still running	when – date and time of the insertion of the metadata
version – version of the agent	agent_id – agent identification associated with this metadata
provider – organization that is in charge of the agent	

The actions *updateClockMetadata*, *updateHoldSeatMetadata*, *updateHoldReservationMetadata* (lines 42, 43 and 44) are responsible for updating the metadata registry. In fact, these actions update the dependability metadata of the agents at runtime. Figure 6 and Figure 5 show screenshots of the registry database. For example, Figure 6 shows that the *airlineA* (agent_id=1) has not fulfilled the obligations *hold-seat* at February 1st and *hold-reservation* at February 3rd.

It is important to notice that these actions are very simple elements that obtain the contextual information at runtime and update the metadata registry. However, it is the law specification that tells when the actions should execute. In other words, the acquisition of the dependability metadata is done through the combined use of various XMLaw elements. It can clearly be seen in Table 4, where a transition, a clock, a norm and an action were connected to update the metadata.

Resultset 1

id	name	start_time	is_active	version	provider
1	airlineA	2007-01-23 10:27:32	1	1.0	Airsafe Corp.
2	airlineB	2006-01-23 08:00:12	1	1.2	Linucsi Corp.
3	travelAgentA	2006-08-26 15:03:45	1	0.56	Nostress Travel Corp.
4	travelAgentB	2005-07-19 20:12:22	1	2.1	The Traveller Corp.

Figure 5 – Examples of the agent

Resultset 1

id	element_type	element_id	when	agent_id
1	obligation	hold-seat	2006-02-01 16:00:13	1
2	obligation	hold-reservation	2006-02-03 10:23:06	1
3	clock	availability-clock	2006-02-01 14:45:22	2

Figure 6 – Examples of the dependability_data.

Surely, complex queries can be built using this simple model. For example, to query the number of not fulfilled obligations of the *airlineA*, one can write an SQL command such as follows.

```
SELECT count(id) as "Obligation Not Fulfilled" FROM dependability_data WHERE
element_type='obligation' and agent_id='1'
```

4. Related Work

In [Dobson 2006], a position paper is presented that illustrates the use of OWL for dependability specifications. An advantage claimed by the author is that the ontology includes the definition of a controlled vocabulary. With the potential of confusion between parties, and domains about the meaning of dependability metrics, an ontology is therefore valuable for disambiguation.

In [Chen et al. 2005], a tool was presented for monitoring the dependability and performance of Web Services. The metadata acquisition and maintenance occurs from a specific location of a client (reflects problems with routers and service WSs). The results are collected and updated in an openly accessible DB. The tool measures the dependability of Web Services by acting as a client to the Web Service under investigation. The tool monitors a given Web Service by tracking the following reliability characteristics: (i) availability: the tool periodically makes dummy calls to the Web Service to check whether it is running; (ii) functionality: the tool makes calls to the Web Service and checks the returned results to ensure the Web Service is functioning properly; (iii) performance: the tool monitors the round-trip time of a call to the Web Services producing and displaying real time statistics on service performance; (iv) and faults and exceptions: the tool logs faults and exceptions during the test period of the Web Service for further analysis. Although the tool can be useful for many existent applications, when compared to the solution presented in this paper, the laws allow for a much more expressive and flexible way to collect domain-specific situations. For example, in the tool it is not possible to express any of the obligations stated in our case study.

Furthermore, to the best of our knowledge there is no solution that encompasses the various features presented in this paper: (i) enforcement of the interaction behavior; (ii) flexible and declarative behavior of the interactions; (iii) explicit incorporation of dependability concerns into the specification; (iv) and openly accessible database about dependability metadata (metadata registry).

5. Discussions

Our society is becoming increasingly dependent on complex software systems. This dependence in turn makes the task of building dependable systems a critical part of software development. Dependability explicit computing states that there is a need for integrating the dependability concerns at the very early stages of the development process. In this approach the dependability metadata should be specified as first-class entities that are available to guide decisions both at the design time and at run-time.

On the other hand, the law-governed approaches have already proposed various high level elements that allow for a flexible specification of the overall system behavior. Furthermore, they provide mediators that ensure that the behavior is being followed as expected. Law-governed approaches also promote dependability in the sense that the system becomes more predictable and some system failures can be prevented by the intervention of the mediator.

In this paper, we have shown that DepEx and Law approaches are complementary. The laws can provide a powerful way to monitor and specify complex dependability metadata. To be more specific, we have incorporated the dependability explicit computing into the XMLaw approach. A detailed case study was presented with the goal of illustrating the acquisition of the metadata. The case study presented has three main contributions: (i) first it shows the integration of a law mechanism (M-Law) into the architecture that allows for dynamically resilient systems based on a metadata approach; (ii) second it shows that the laws can be a very powerful way not only to acquire dependability metadata, but also to interfere in the system execution when necessary (iii) Finally, with the laws, the dependability concerns are explicitly considered and precisely specified mostly in a declarative way. We have also shown the model of a metadata registry and how this model can be queried to return dependability metadata.

The approach presented in this paper has the advantage of flexibility and reuse. Flexibility, because in contrast to the related work, the high level abstractions presented in XMLaw allow for a very expressive and domain dependent way to acquire the metadata, while still

preserving the declarative nature of the laws. And reuse, because we do not have to rebuild a new language nor a new mediator to perform the metadata acquisition. Therefore, XMLaw was shown to be flexible enough to incorporate various dependability concerns.

One promising research direction to this work is to improve the current support given to Requirement Engineering activities. Some works such as the ones presented in [Webster et al. 2005] and [Chung et al. 1999] have been trying to connect the non-functional requirements and the functional requirements. In this sense, it is possible to represent in a requirements document the association among the functional requirements, the non-functional requirements and the law specification to deal with them. As an example, the Figure 7 shows how one can represent the case study presented in Section 3 using the i* notation proposed in [Yu 1994]. This diagram shows the association of the (six) defined case study requirements to the specific attributes availability, service failure and pre and post-conditions.

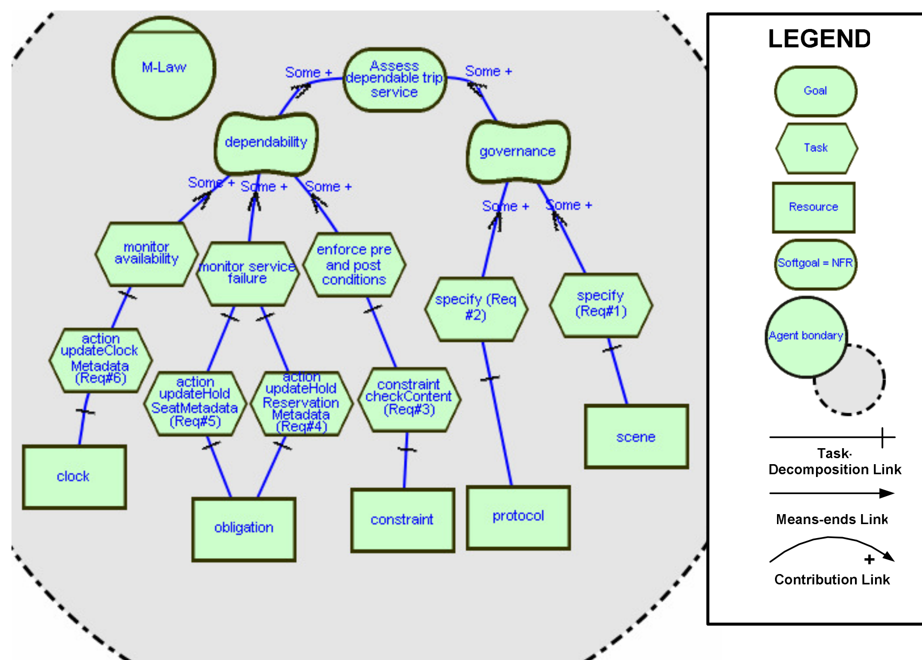

Figure 7 – I* Diagram of the case study.

Acknowledgements

The authors would like to thank Dr. Alexander Romanovsky for indicating the possibility of using our law-governed approach in the dependability domain. The authors would like to thank Antonio de Padua Albuquerque Oliveira for helping with the i* diagram. This work is partially supported by CNPq/Brazil under the project "ESSMA", number 5520681/2002-0 and by individual grants from CNPq/Brazil.

References

Serugendo, G., Fitzgerald, J., Romanovsky, A. and Guelfi, N., (2006) "Dependable Self-Organising Software Architectures - An Approach for Self-Managing Systems", Technical Report No: BBKCS-06-05, School of Computer Science and Information Systems, Birkbeck College, London, May

Avizienis, A., Laprie, J-C., Randell, B., and Landwehr, C., (2004) “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transactions on Dependable and Secure Computing, vol. 1, n. 1, pp. 11-33, January-March.

Kaâniche, M., Laprie, J., and Blanquart, J. (2000). “A Dependability-Explicit Model for the Development of Computing Systems”. In Proceedings of the 19th International Conference on Computer Safety, Reliability and Security. F. Koornneef and M. v. Meulen, Eds. Lecture Notes In Computer Science, vol. 1943. Springer-Verlag, London, 107-116.

Paes, R., Gatti, M., Carvalho, G., Rodrigues, L., Lucena, C., (2006), “A middleware for governance in open multi-agent systems”, Tech. Rep. MCC 33/06, PUC-Rio, http://wiki.les.inf.puc-rio.br/uploads/8/87/Mlaw-mcc-agosto-06.pdf.

Minsky, N. and Ungureanu, V., (2000) “Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems”, ACM Trans. Softw. Eng. Methodol. 9 (3) (2000) 273--305.

Paes, R., Carvalho, G., Gatti, M., Lucena, C., Briot, J.-P., Choren, R., (2007) “Enhancing the Environment with a Law-Governed Service for Monitoring and Enforcing Behavior in Open Multi-Agent Systems”, In: Weyns, D.; Parunak, H.V.D.; Michel, F. (eds.): Environments for Multi-Agent Systems, Lecture Notes in Artificial Intelligence, vol. 4389. Berlim: Springer-Verlag, p. 221–238.

Paes, R., Carvalho, G., Lucena, C., Alencar, P., Almeida, H., Silva, V. (2005) “Specifying laws in open multi-agent systems”, in: Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM, Utrecht, The Netherlands.

Paes, R., Carvalho, G., Lucena, C., (2007b) “XMLaw specification: version 1.0”, Tech. Rep. to appear, PUC-Rio, Rio de Janeiro, Brasil.

Esteva, M., (2003) “Electronic institutions: from specification to development”, Ph.D. thesis, Institut d'Investigaci en Intel.ligncia Artificial, Catalonia – Spain - October.

Dignum, V., Vazquez-Salceda, J., Dignum, F., (2004) “A model of almost everything: Norms, structure and ontologies in agent organizations”, in: Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), Vol. 3.

Yi, X. and Kochut, K., (2004) “Process Composition of Web Services with Complex Conversation Protocols: a Colored Petri Nets Based Approach”, Proc. of Design, Analysis, and Simulation of Dist. Sys. Symposium.

Dobson, G., (2006) "OWL and OWL-S for Dependability-Explicit Service-Centric Computing", Service-Oriented Computing: Consequences for Engineering Requirements (SOCCER'06 - RE'06 Workshop), p. 4, September.

Chen, Y., Li, P., Romanovsky, A. (2005) Web Services Dependability and Performance Monitoring. Proc. of 21st UK Performance Engineering Workshop. Newcastle upon Tyne. UK. July.

Webster, I., Amaral, J., Cysneiros, L. M. (2005) "A Survey of Good Practices and Misuses for Modelling with i* Framework", in Proc. of VIII Workshop in Requirements Engineering, Porto, Portugal,pp:148:160, ISBN 972-752-079-0

Chung, L., Nixon, B., Yu, E., Mylopoulos, J. (1999) "Non-Functional Requirements in Software Engineering" Kluwer Publishing.

Yu, Eric. (1994) “Modelling Strategic Relationships for Processing Engineering”, Ph.D. Thesis, University of Toronto.