REOS Home Page

This is the home page for the Workshop on Requirements Engineering and Open Systems (REOS).

September 8th, 2003, Monterey, CA

In conjunction with 11th IEEE International Requirements Engineering Conference (RE03).

Workshop on Requirements Engineering in Open Systems

Integration and interoperation have become the critical issues in engineering multi-stakeholder distributed systems (MSDS) like the Internet electronic mail system, networks of web services, modern telephone networks, and the Internet itself. MSDS require rethinking requirements engineering and validation, because they do not admit globally consistent high level requirements, requiring instead a personalized and time-dependent view of requirements, and because single stakeholders are largely ignorant of the detailed operation of nodes outside their sphere of control, making validation problematic.

Target audience. This workshop is intended to bring together researchers and practitioners in the fields of

- requirements engineering
- component-based design (including Enterprise Application Integration (EAI))
- verification and validation
- ontology engineering and knowledge representation
- web services
- XML-based standards for representation (DAML+OIL, DAML-S, RDF and OWL)

and related fields to discuss the challenges of designing and using open systems.

Goals and structure. Our goals for the workshop are both to improve awareness of how open systems create novel problems for requirements engineering and to begin to explore potential solutions. To help focus the discussion, we have selected some open system scenarios (see full call for participation) and encourage each presentation to discuss how its ideas address or relate to the problems illustrated in the scenarios. The format of the presentations will include extra time for audience discussion of each presentation, hopefully allowing the group both to better understand each

set of ideas and to relate them to other presentations and to the workshop themes.

A minimum two-page abstract should be submitted via e-mail to either of the workshop co-chairs in pdf (preferred) or ascii text format. Please see The Call For Proposals (CFP) for more information.

```
Workshop papers due: June 27, 2003
Notification to authors: July 18, 2003
Camera-ready: August 4, 2003
Workshop date: September 8, 2003
```

Note that for workshop presentation choices, the committee will give preference to papers that address one or more of the topics/examples below (or those closely related).

The workshop will be part of the larger 2003 International Requirements Engineering Conference - see the conference web page at http://conferences.computer.org/RE/.

Detailed Description

Integration and interoperation have become the critical issues in engineering multi-stakeholder distributed systems (MSDS) like the Internet electronic mail system, networks of web services, modern telephone networks, and the Internet itself. Consistent, well defined protocols and other low level requirements enable these systems to function, but higher level requirements placed by diverse users are often ephemeral and typically inconsistent when viewed together. Thus, for the field of requirements engineering to deal with open MSDSs at all, we need to shift our thinking from systems having consistent, global requirements to those in which requirements can be user-relative and ephemeral.

Beyond that issue, however, lurks a second major challenge dubbed the "ignorance problem": since the nodes of an MSDS are controlled by stakeholders with different goals, priorities, and capabilities, just knowing what they all do is a challenge. For example, email features and functionality have grown so complex that merely knowing a host serves TCP port 25 (SMTP) does not give enough information to know whether one's email message will be handled correctly. Current web services provide the means to discover method signatures. However, formal service standards have yet to be defined.

This workshop is intended to bring together researchers and practitioners in requirements engineering, component-based design (including Enterprise Application Integration (EAI)), verification and validation, and related fields to discuss the challenges of designing and using open systems in which requirements are ephemeral and user-relative, and in which it is difficult or impossible to know the behaviors of all the parts of the system.

Topics of interest

1. In an open system, up front system analysis is at best of limited, heuristic usefulness. The system is amorphous and dynamic: it comprises whatever components are available at the moment. The best one might hope for is just-in-time analysis: can requirement R be met at this moment with system S?

2. Three new types of requirements are prevalent in open systems:

    1. How do we capture and reason about ephemeral requirements? Ephemeral requirements have a lifetime that can be measured in days, hours, minutes or even less. They may be recurring, but may also be one-off.
    2. How do we capture and reason about user-relative requirements, or what we might call "personal requirements"? Personal requirements are not global to the system, but are specific to a single user or stakeholder. They are highly contextual.
    3. When a system will be used by (or affect) multiple users with different personal requirements (personal utility functions), how do we reconcile their differing preferences in setting requirements for a system. For instance, in an email setting, spam can be explained as a clash between the sender's personal requirement (get a user to respond to the message) and the receiver's personal requirement (don't be bothered by material that's irrelevant, often offensive, and possibly fraudulent). How do we reason about utility-based requirements?

3. Requirements in open systems are often obstructed by the environment, i.e., by components not under the user's control. It is less interesting to know that a requirement R cannot be met than knowing how it might fail. Both runtime monitoring and failure-recovery mechanisms come to the fore.

4. Given that many components of the system will not be under a user's control, how can we reason about the system's behaviors that may be relevant to the user? Can components provide an external description that will allow us to reason about requirement achievement? Can components be made transparent (e.g., reflective) so we can directly reason about their behavior?

5. Can open modeling standards be established that would allow personal requirements validation tools to help a user discover and reason about personal requirements satisfaction? If so, how can communities of interest establish shared ontologies/theories that can support capabilities like semantically meaningful model composition, scenario simulation, animation, theorem proving, model checking, and other formal reasoning tools? What information (beyond the standard OO entity-relationship information) must be included in such shared theories? Are current web-based standards such as RDF, RDFS, or OWL, enough for these tasks in terms of expressiveness, tractability, ease-of-use, etc.?

6. Deceptive practice is turning out to be a problem with some web sites in todays competitive market. For instance, it is difficult to ascertain what the information-privacy policy is of some sites. Even when sites state their policy, it is difficult to verify it. How do privacy and security fit into the larger picture of MSDS? Should game theory be part of requirements analysis in an MSDS?

Examples

We list some examples to help with the focus of the workshop. Ideally, a submission would explicitly explain how their approach, tool, idea, etc, would handle similar examples.

EXAMPLE 1: an email request and reply.

This example is drawn from a real project (contact fickas@cs.uoregon.edu for more details). User A wishes to get a ride to the doctor from user B. User A decides to send an email request to B asking for the ride. In slightly more formal terms, A's requirement is a yes or no answer from B to the ride request, in time to do A some good. The

requirement is ephemeral: it endures for this ride request only, and may never come up again. The requirement is personal: it is A's requirement and contextualized to the components in place, at the request moment, to get a question to and reply from B.

It is an open system problem in that neither the Internet connectivity of A and B, the communication channel (the SMTP servers and post-office server) between A and B, nor the email client of A and B are under the control of all parties. In particular, A and/or B can be disconnected, email servers on either A's or B's side may filter email under administrative policy, the actual email clients themselves can filter email. Further, from A's point of view, B (the human) is part of the environment: B can ignore the request. In summary, the ride-request requirement can fail in many ways and it is impractical to attempt to guarantee success.

The questions this example raise in terms of the workshop are as follows:

1. Can we reason at all about the potential success or failure of the ride request? Do we know or can we discover what system components must be involved in the request? If we know the components, can we also know their behavior? If we know their individual behavior, can we reason about success and failure from a system-wide level?
2. Once we know how the request can fail, can we do more than hope for the best? Can we monitor its progress? Can we be warned of impending failure and potentially circumvent it? If failure does occur, can we recover in a way that keeps the request alive?
3. Is it feasible, in a limited domain like email, to develop a shared ontology, one that all email server and client vendors might adopt?
4. Privacy issues arise in this problem. It may be the case that B's email client is willing to make explicit the general rules it follows, (e.g., email from certain domains on a watch-list are deleted) without providing details (e.g., whether A's domain is on the watch list).

EXAMPLE 2: A Web Services Example

This example comprises four stakeholders:

1. User C has a web browser. C knows what Internet web services he or she wishes to access. One of these is UpToTheMinuteNews.com (U2M).
2. Corporate IT (IT) requires that all web requests traverse a web proxy on the corporate firewall.
3. A company called Acme Web Speedup Services (AWS) provides a caching proxy web service that is billed as "speeding up the average web access by N%!!!"
4. UpToTheMinuteNews.com provides (for a fee) the very latest news stories on their web site.

IT decides to improve everyone's life (on average) by connecting the corporate web proxy to AWS. Soon after that decision, C notices he or she is no longer receiving the latest news from U2M. This example has several relevant characteristics for the workshop:

- No single stakeholder has the information necessary to anticipate or even diagnose the problem. Can we come up with ways of decreasing the ignorance in this MSDS?
- The problem arises dynamically. One day U is happily receiving the service for which he or she is paying, and the next day an invisible (to C) change causes it to stop working. Thus, any analytic tool must be sensitive to change. How can any proposed solution deal with the constant change that is prevalent in MSDSs?
- IT is making design decisions based upon imperfect understanding of the requirements or other stakeholders

and/or the implementations of other nodes.
- AWS and U2M may not be implemented optimally or even correctly. Thus, can one stakeholder's tools compensate for ignorance and imperfect design?

Note that it is difficult to know without detailed debugging knowledge what causes the problem. Here are some alternative hypotheses. (1) AWS's contract has fine print explaining that it does not guarantee freshness of its pages, (2) AWS's contract is purposely inaccurate or unclear on this point. This would be an example of deception in MSDSs (see Example 3 below). (3) AWS's implementation is buggy, old, or incorrectly configured so that it does not honor "no-cache" header information. Note that AWS's service would still be acceptable for all users who don't access time critical sites. (4) U2M's implementation is buggy or incorrectly configured, so that it does not produce "no-cache" headers with its pages. Note that in this case, U2M would still work perfectly well for all clients who do not use caching. (5) IT failed to read or correctly interpret the AWS contract. (6) IT failed to realize that some of its users required freshness. (7) C failed to communicate to IT that it needed access to a time critical web site, even though IT surveyed its users on this point at some time previously. (Or maybe C's needs changed since the survey.)

Requirements and validation tools must deal with all these issues, since it is not reasonable to assume every node is implemented well or even in compliance with published standards; it is not reasonable to assume all requirements are accurately known, or that all stakeholders are honest; and it is not reasonable to assume that contracts (e.g. published interfaces) are correct, complete, and unambiguous. Clearly, a stakeholder needs ways to discover actual behavioral details, and to monitor plans as they execute for failures. How can we support these needs?

EXAMPLE 3: Deception

The previous two examples have shown the problems when well-intentioned components/stakeholders must be brought together to meet a personal requirement. But components in either example could have hidden goals. For instance, an SMTP server could be monitoring the email that it processes for marketing opportunities, harvesting email addresses for further bulk mailing. The AWS component could be monitoring visited pages and making the information available, for a price, to those interested in the surfing habits of the IT corporation. Ideally, we would like to be able to have a clear-box view into the behavior of the components, thus making hidden goals transparent. Barring that, are there other means of reasoning about deception in open systems? Are there ways to discourage it? Can we borrow reasoning methods from other fields that deal with potentially dishonest agents, e.g., game theory?

Submissions

A minimum two-page abstract should be submitted via e-mail to either of the workshop co-chairs in pdf (preferred) or ascii text format. Deadlines:

```
Workshop papers due: June 27, 2003
Notification to authors: July 18, 2003
Camera-ready: 8/4/03
Workshop date: September 8, 2003
```

Note that for workshop presentation choices, the committee will give preference to papers that address one or more of the topics/examples below (or those closely related).

Workshop committee

Workshop co-chairs:

Stephen Fickas, University of Oregon, fickas@cs.uoregon.edu
Robert J. Hall, AT&T Labs Research, Florham Park, NJ, hall@research.att.com

Workshop Program Committee:

Annie Anton, North Carolina State University
Carlo Ghezzi, Politecnico di Milano
Klaus Havelund, NASA AMES
William N. Robinson, Georgia State University
Axel van Lamsweerde, Universite Catholique de Louvain

Workshop on Requirements Engineering for Open Systems

Co-Chairs
Stephen Fickas, University of Oregon
Robert J. Hall, AT&T Labs Research

September 8, 2003

830 Introduction

R.J. Hall
RE in Open Systems: Problems and Some Approaches

1000 Break

1030-1200 Formal Methods

L. Baresi, E. Di Nitto, C. Ghezzi
Inconsistency and ephemerality in a world of e-services

W. Robinson
Monitoring web service interactions

X. Fu, T. Bultan, J. Su
A top-down approach to modeling global behaviors of web services

1200 Lunch

1330-1500 Negotiation and Ontological Alignment

P. Gruenbacher, F. Stallinger, N. Maiden, X. Franch
A negotiation-based framework for requirements engineering in multi-stakeholder distributed systems

K. Breitman
Semantic interoperability by aligning ontologies

F. Marschall, M. Schoenmakers
Classifying requirement conflicts for multi-stakeholder distributed systems

1500 Break

1530 Discussion and Summary of Themes

Inconsistency and Ephemerality in a World of e-Services

Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi

Politecnico di Milano – Dipartimento di Elettronica e Informazione
piazza L. da Vinci, 32 I-20133 Milano (Italy)

baresi|dinitto|ghezzi@elet.polimi.it

ABSTRACT

Service-based systems comprise components owned by different organizations and providing so called *e*-Services. *e*-Services operations are published through interfaces that focus almost exclusively on defining syntactical aspects. The distributed ownership of components and the partial visibility of their behavior complicate system management and preclude the capability of reasoning at design time on the system as a whole. These new problems require the definition of new techniques to support the composition of services and validate the result. In addition, this implies capabilities to deal with run-time misbehaviors and dynamic reconfigurations that go far beyond the currently available approaches.

The many standards proposed in these years for composing *e*-Services do not address the consistency of designed cooperations. In this position paper, we briefly describe a possible solution to enforce consistency and deal with ephemeral requirements in a world of *e*-Services. In such a world, services can be selected dynamically, stake-holders may change their services, users may change the context in which they use the system, or simply change their needs and taste. Static analysis techniques are not enough to ensure consistency in this case. Our proposal emphasizes pre and post conditions to both describe the functionality provided by the *e*-Services and how they should be used in deployed compositions. Based on pre and post conditions, run-time monitors may trigger possible inconsistencies between supplied services and how they are supposed to behave.

1. INTRODUCTION

Service-oriented architectures are becoming the means to integrate and deploy complex distributed systems. Typically, they comprise components supplied by different organizations and offering *e*-Services published through some *interfaces*. Such interfaces are usually focused on the syntactical aspects related to the definition and invocation of operations, but hide all other details (e.g., the semantics of invocation, the meaning of results, etc.). Such a framework poses two interesting problems: the distributed ownership of components and the partial visibility of their behavior. The former makes system management more complex and does not allow the user to trust a single provider: We need a mean to "negotiate" the services supplied by each provider on the basis of particular quality of services. The latter precludes the capability of reasoning on the behavior of the system as a whole. We must require that the *e*-Services expose as much information as possible to let designers "reason" on the systems they deploy. Indeed, we must provide proper mechanisms to support composition of *e*-Services and validation of the resulting system.

In these years, we are assisting to a proliferation of different XML-based proposals that should pave the ground to the solution of the problems described so far. WSDL (Web Service Description Language, [2]) is already the de-facto standard to describe *e*-Service interfaces, that is, sets of operations supplied by a single provider, WSOL (Web Service Offerings Language, [12]) is just one of the many proposals to negotiate the quality of required services, and BPEL4WS (Business Process Execution Language for Web Services, [3]) is the most recent proposal for composing *e*-Services to build more complex services. Orthogonally, we can also mention DAML-S (DARPA Agent Markup Language - Services, [1]), which with its three layers – profile, model, and grounding – covers both the description of services and their composition.

Even if all these proposals aim at a seamless and easy deployment of service-based systems, they all assume that designers be in charge of the *consistency* of the system. Almost all proposals concentrate on modeling, but offer little or no support to the validation phase. Thus, how can we compose *e*-Services in a consistent way? Are the simple syntactical interfaces of operations enough to this end? The feeling is that composition of *e*-Services requires more knowledge on the operations they offer than what is actually published. Important information ranges from aspects related to quality of service to a specification of how the operations behave, what sequence of operations can be correctly executed, etc. Only DAML-S goes in the direction of disclosing such information by allowing designers to specify operations in terms of pre and post conditions. However, it leaves designers free to use the language they want, thus resulting in a difficulty of automatically interpret and evaluate such conditions. Indeed, DAML-S does not provide any suggestion on how to exploit the defined pre and post conditions through the life cycle of *e*-Services.

In a closed environment, where we can trust and control providers, we could think of exploiting pre and post conditions to perform *static analysis* to reason on the consistency of service-oriented systems. But, if component services are outside our control, they can change freely, and we cannot assume the consistency between their published interfaces and the corresponding running implementations. So, even if the former could be compatible with our expectations, the latter could be faulty. In this case, the only mean to *reason on* consistency is through *run-time monitoring*.

Similar problems also arise when we do not have fixed bindings between *e*-Service definitions and providers, but we require that actual *e*-Services be dynamically discovered and negotiated through some kind of repository. Used services are discovered only during the execution and thus the comparison between published interfaces and running implementations can be done only at run-time. Services can change with respect to the context in which the application is deployed, but also user requirements (i.e., user profiles) can vary and thus impose new or different requirements on the system. For instance, users can change the devices they use to interact with the system, they can move, or they can simply change their preferences (and taste). Again, ephemerality emphasizes on-line monitoring.

The idea is not completely new: Robinson [10] propose run-time monitors to match requirements and system executions. His work concentrates on high-level requirements. From them, he discovers obstacles, and derives monitors. In contrast, we are closer to the implementation, concentrate on simpler properties, and leave to the designer more freedom to tailor the degree of accuracy he wants to adopt to define pre and post conditions.

In the rest of the position paper we exemplify our ideas on one of the examples proposed in the call for papers [5]. Section 2 describes the example in terms of *e*-Services and lists the issues it raises. Section 3 sketches our proposal that is based on the idea of asserting pre and post conditions to use them as run-time monitors. Finally, Section 4 briefly discusses our proposal with respect to available technologies and Section 5 concludes the paper.

2. THE PROBLEM

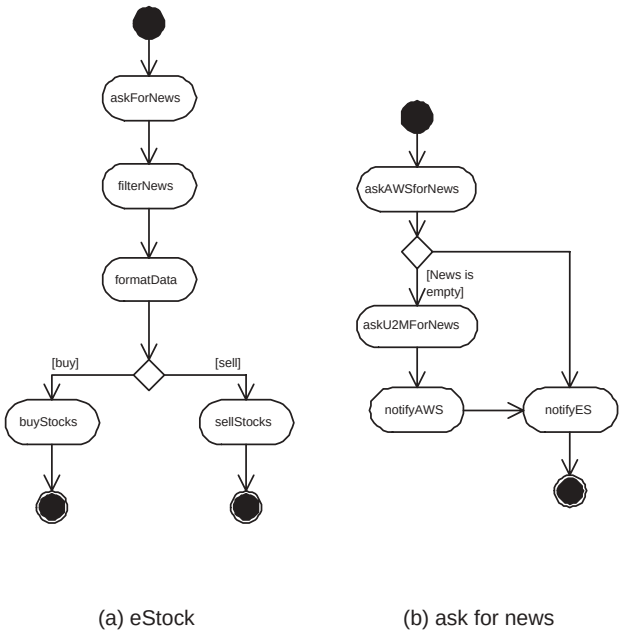
To exemplify the problems that we have pinpointed in the previous section, but also to pave the ground to a possible solution, we recall and adapt one of the exercises suggested in the workshop call for papers [5]. The example includes four stake-holders:

- *UpToTheMinuteNews.com* (U2M) provides the very last news stories. News belong to various topics: for instance, sport, finance, and politics.
- *eStock* (ES) offers to users the capability of trading stocks on-line. It exploits U2M to provide the very last news on the stock market.
- Users (U) exploit *eStock* to know the very last news on the stock market and manage their stock portfolio. U can access the system using different devices (PCs, PDAs, smart phones).
- *Acme Web Speedup Services* (AWS) provides a caching proxy Web service that is billed as "speeding up the average Web access by N%".

We can imagine ES, AWS, and U2M as *e*-Services that offer a predefined set of operations. Each stake-holder controls the *e*-Service it provides, but has no control on the others.

Figure 1.(a) shows a process description that defines the behavior of ES. The `askForNews` activity causes the execution of a request to the U2M service. The `filterNews` activity is in charge of extracting from all received news the ones concerning stocks, `formatData` formats and display data for the user who, in turn, can then decide to buy or sell stocks.

At a certain point in time, ES decides to improve its availability to customers' requests by binding the `askForNews` activity to a new service that, based on a few experimental data, appears to be more stable than U2M and is compliant with the same interface as



(a) eStock (b) ask for news

Figure 1: Example processes

U2M. Being outside the control of ES, it does not know that such service exploits the AWS caching service as shown in the process description of Figure 1.(b). Therefore, as a result, it happens that ES users start experiencing that stocks news are not updated anymore and stop buying or selling stocks through the system.

The set of cooperating services raises the following problems:

- No stake-holder has enough knowledge to anticipate problems arising from *e*-Service composition nor to locate them.
- Problems arise dynamically. The process in ES has been designed correctly and all components (i.e., *e*-Services) seem to work together properly. The availability of partial information is not enough to assure the correct execution of the system.
- The selection of *e*-Services (operation) has been based on published information, but they could be incorrect, incomplete, and ambiguous.
- Errors can arise from several different reasons. It could be because of the addition of a new service, and thus a change in the process, but also because one of the service providers changes (the internals of) its operations, or one of the users starts using a device that is not well-supported by the system.
- The match making process can be major source of problems. The discovery and selection process in a given context can be considered a typical example computation that must deal with ephemeral requirements: The constraint that must be matched and the services available in a given context are usually different from those in other contexts.
- Last, but not least, problems may be due to the unmatched quality of non-functional (e.g., availability of service, time efficiency, security issues) rather than functional parameters.

Even if important, the last bullet is left out of the scope of this position paper, but we feel that our proposal can be properly extended to cope with it.

3. OUR PROPOSAL

To tackle the aforementioned problems we are currently exploring the use of assertions. Our proposal is inspired by the many assertion systems available for programming languages. For example, we can mention ANNA (ANNotated ADA, [7]) as the first proposal, Eiffel [8] and the latest version of Java as other examples of programming languages that embed assertions, and ADL (Assertion Description Language, [11]) as a cross-language approach. All these languages allow designers to associate *assertions* (constraints) with particular parts of the code and check them at runtime. Assertions are added to the code as special-purpose annotations; ad-hoc preprocessors transform them into code and thus allow them to be checked at run-time.

To the best of our knowledge, none of the languages for *e*-Service composition allow designers to add these assertions. For example, neither BFEL4WS nor others such as WSCI [4] leave room to them. Instead, we argue that the usage of assertions offers an effective programming tool to define pre and post conditions to predicate on the invocation of operations offered by externally controlled *e*-Services. Such a programming mechanism is nicely separated from the language constructs that are used to define the logic of *e*-Services composition.

In our approach, *e*-Service owners can associate pre and post conditions with the operations they publish on their *e*-Service interfaces. Such conditions define the contract they declare to be able to fulfill when some user invokes the corresponding operations. The semantics is the usual one: whenever the precondition is satisfied, they promise that the results of the operation will fulfill the postcondition. At the same time, users of *e*-Services exploit assertions to associate the process fragments with pre and post conditions that predicate on their obligations and expectations with respect to such *e*-Services. Assuming that the process definition is correct, preconditions are guaranteed to be true before calling some *e*-Service operations. Also, postconditions are expected to be true after completion of the same operations. A service provision matches a service request if a) the precondition defined as a part of the invocation is equivalent to or stronger than the precondition defined as a part of the operation description, and b) the postcondition defined as a part of the invocation is equivalent to or weaker than the postcondition defined as a part of the operation description. Pre and post conditions can then be used to:

1. Statically analyze the consistency of the designed process: such analysis can be performed at development time and can be used to prune bugs that are introduced within the code that is under the control of the process designer.

2. Select the actual services to be exploited within a service-oriented system by checking the compatibility of the *e*-Service operation invocations with the corresponding operation definitions.

3. Act as *run-time monitors* to raise possible problems and scope them. In fact, by asserting pre and post conditions at the side of the invoking system it is possible to check that the *e*-Services being exploited behave in a way that is compatible to the expectations of the invoking system itself (assuming that this last one is behaving correctly, i.e., it allows the preconditions of the invoked services to hold). Such a monitoring allows the invoking system to gain control over the execution of the external *e*-Services and to discover the cases where, for some reasons, their actual behaviors do not respect the expected contract.

A faulty behavior signaled by the run-time monitoring mechanism can be caused by one of the following factors:

- The process describing the composition is internally inconsistent thus leading to a situation where a precondition expected to be true it is not. This case should not happen if the process is properly validated.

- The asserted pre and post conditions are still compatible with the pre and post conditions declared in the interface of the *e*-Service being used, but the corresponding implementation does not respect them: The stated interface of the invoked service is inconsistent with the corresponding implementation. This is the case of services that lie about their behavior.

- The asserted pre and post conditions are not anymore compatible with the pre and post conditions declared in the interface of the *e*-Service being used, for instance because it has been changed independently.

In the second and third case, the identification of the problem at run-time allows the composed system to take some corrective steps. The most obvious one could be the activation of a matchmaking mechanism that tries to find a new *e*-Service compatible with what is expected, and the binding to and activation of such service so that the composite system can continue execution. Corrective steps can be expressed by exploiting the exception handling and/or compensation mechanisms made available by the existing orchestration languages.

Let us consider again the example of Figure 1.(a), and in particular the `askForNews` activity. It could be associated with the following pre and post conditions[1]:

```
news = askForNews.getNews():
 pre: true
 post: forall n in news:
  cTime - someConstantValue < n.time < cTime
```

where `news` is the set containing all results produced by the invocation of the *e*-Service in charge of producing the `news`. Intuitively, the postcondition expression states that the time at which all news have been produced has to be close enough but not greater than `cTime`. Notice that we implicitly assume that all quantifiers – `forall` and `exists` – predicate on finite domains.

If service U2M or any other news service defines in its public interface the pre and post conditions below, it is compatible with the `askForNews` activity defined in the composed *e*-Service provided that `anotherConstantValue` $\leq$ `someConstantValue`:

```
Set getNews():
 pre: true
 post: forall s in Set and all n in s:
  cTime-anotherConstantValue < n.time < cTime
```

Let us assume that the new *e*-Service that replaces U2M exposes the above pre and post conditions, but its actual implementation does not comply with them. In this case, if during the execution of ES the assertion mechanism is activated, the problem related to the freshness of the news is dynamically discovered and can be solved by rebinding to the old U2M service.

By adding pre and post conditions to the process defining the composed *e*-Service we provide a mechanism to define how an invoked service is supposed to be used and to check *e*-Service behavior at runtime.

[1]For the sake of readability, pre and post conditions are rendered using a "generic" language. The same concepts could have been rendered using a language like OCL.

Pre and post conditions in the process definition can also be complemented by *invariants*. An invariant represents a property that must hold in any valid state of the process. At runtime it can be enforced after a failure in order to guarantee that the composed system remains in a valid state.

4. APPLYING OUR APPROACH TO CURRENT TECHNOLOGIES

After describing our proposal, we want to briefly discuss how it applies to available technology. The idea is to work in the same way as the assertion systems for programming languages do. We can exploit comments – both in BPEL4WS or in DAML-S – to state the pre- and post-conditions associated with each service invocation. We could also imagine invariants that must hold true on the complete process or on some fragments. These comments do not change current standards and can simply be discarded by the engines that execute the specifications: for instance the BPEL4WS Java Run Time by IBM [6] or the DAML-S virtual machine by Carnegie Mellon University [9].

Special-purpose pre-processors can read these comments and transform them in a *validation service*, which is held by the stakeholder that executes the process and provides the operations to check the consistency of defined assertions.

A more complex solution would consider the extension of both specification languages and execution engines. This solution is similar to the recent extension to Java, which now embeds assertions as language feature. Since we are thinking of *run-time monitors*, and not just oracles during the validation phase, this solution would be much more robust, but also much more complex to implement.

The right compromise is to aim at the second solution, but start doing experiments following the first approach, which is lighter and can provide interesting feedbacks on the soundness of the proposal.

5. CONCLUSIONS

This position paper has discussed the importance of inconsistency and ephemerality in service-based systems. Besides identifying the problems that we must address and try to solve, the paper proposes the use of pre, post conditions, and invariants defined both in the interface of *e*-Services and in the process describing a composition of *e*-Services. Such conditions can be exploited to perform static analysis by using traditional approaches. In addition, they can be asserted at runtime to *detect* inconsistent steps while executing the composed system. This is only one side of the problem; the other side is the capability of dynamically reconfiguring the process – executed by system – to deal with both wrong services and ephemeral requirements. This is part of our future work.

6. REFERENCES

[1] DAML-S Coalition: A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web service description for the Semantic Web. In *Proceedings of 1st Int'l Semantic Web Conf. (ISWC 02)*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363, 2002.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.2. `www.w3.org/TR/wsdl12/`, June 2003.

[3] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.1. `www6.software.ibm.com/software/developer/library/ws-bpel11.pdf`, May 2003.

[4] A. Arkin et al. Web Service Choreography Interface 1.0. Technical report, BEA Systems, Intalio, SAP, Sun Microsystem, 2002. `http://wwws.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf`.

[5] S. Fickas and R. J. Hall. Call for papers of the Workshop on Requirements Engineering and Open Systems (REOS), September 2003. `http://www.cs.uoregon.edu/~fickas/REOS/`.

[6] IBM. The ibm bpel4ws java run time. `www.alphaworks.ibm.com/tech/bpws4j`.

[7] D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe. *Anna - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[8] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[9] M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proccedings of WWW 2003*, pages 111–118, 2003.

[10] W. Robinson. Monitoring web service requirements. In *Proccedings of the International Conference on Requirements Engineering*, 2003. To appear.

[11] S. Sankar and R. Hayes. ADL: An Interface Definition Language for Specifying and Testing Software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994.

[12] V. Tosic, K. Patel, and B. Pagurek. WSOL - Web Service Offerings Language. In *Proceedings of the Workshop on Web Services, e-Business, and the Semantic Web - WES (at CAiSE02)*, volume 2512 of *Lecture Notes in Computer Science*, pages 57–67. Springer-Verlag, 2002.

Monitoring Web Service Interactions

William N. Robinson
Georgia State University
wrobinson@gsu.edu

Businesses increasingly rely on web services. Advantages, such as supply chain efficiencies and agility, are gained universally. Disadvantages, such as supply chain failures, occur universally, but are understood less. While electronic commerce has increased the speed of on-line services, the technology of monitoring on-line services has lagged behind. Consequently, businesses are becoming increasingly vulnerable to the problems of their on-line partners.

Monitoring provides an initial solution. Ideally, impending failures in electronic supply chains can be detected and repaired without user intervention and without a perceptible decrease in performance. Researchers must provide answers to reach the ideal of dynamically evolving supply chains. Now, practitioners use monitors to provide service failure alerts. More advanced monitoring systems include alerts of impending individual and aggregate service failure, analysis of historical failure data, and extensive reporting.

1 Monitoring Systems

A simple conceptual model underlies approaches to monitoring, as illustrated in Figure 1. The design-time model represents systems requirements. The run-time model represents a view of the internal workings of the system; it is a refinement of the design-time model, typically. Monitoring is the observation of system actions—including internal actions—interpreted through the run-time and design-time models. Monitoring systems vary in the complexity of their models and the efficiency of their run-time systems.

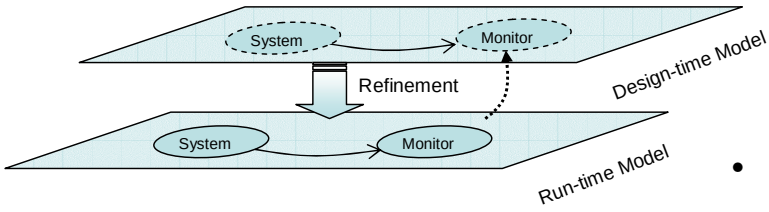


Figure 1 Monitoring models.

The implementation of run-time monitors can be understood in terms of the network architecture illustrated in Figure 2. The architectural components can reside on a single computer, or they can be distributed over complex networks. Event adaptors translate world events into monitored events. For example, a web service adaptor captures web service requests and replies as monitored web service events. Broadcasters forward the monitored events to other broadcasters and listeners. A requirements monitor is a specific type of listener that interprets an event stream in terms of requirements satisfaction.

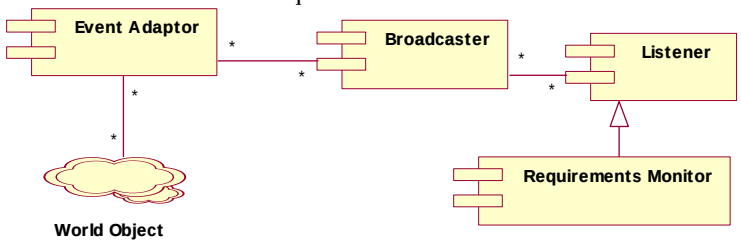


Figure 2. Service monitoring network architecture.

Choices on the service monitoring models and network architecture affect the expressibility, adaptability, efficiency of the requirements monitoring system. Consider each of the network architecture components:

- The event adaptor can be integrated into the network protocol stack, which includes SOAP message for web services. As requests and replies are deserialized and serialized, the event adaptor can extract the SOAP message and forward it to the broadcaster. Although efficient, the protocol stack event adaptor must be installed on each server. Some web service implementations provide web service filters, which greatly simplify event adaptor installations. Intermediaries, such as web service gateways or routers, also simply web service monitoring. These web service specification extensions allow an intermediate server to monitor web service messages.
- The broadcaster receives the monitored messages from the event adaptor—an efficient adaptor filters messages, and sends only relevant messages to the broadcaster. The broadcaster then sends the messages to registered listeners. Broadcasters and listeners can

be arranged into network hierarchies—for example, to aggregate information. However, most monitoring systems do not have a broadcaster: messages are sent directly to a single listener. In such an architecture, however, the listener becomes a bottleneck.

- The listeners receive the messages as a series of events, such as Request and Reply. Listeners may also receive events from other listeners, via the broadcaster. For example, a listener may broadcast a Requirement Failure notification when it observes a requirement failing. In most monitoring systems, listeners simply show a trace of messages. Some types of listeners can provide warning lights when web service requirements fail.

1.1 Monitoring Requirements

Expressibility of the design-time language and efficiency of the run-time monitoring system distinguish the different approaches to service monitoring. For example, many web service monitors only determine the satisfaction of the following simple requirement.

```
After a request, Rq, service S shall provide a
corresponding response, Rs, within time t.
```

Notice that this requirement concerns a request-response pair of single service. Many web service monitors send a dummy request, a sort of "ping", to see if the service is responding. If the service responds, then it is assumed that all similar requests are being satisfied. Such monitoring systems use a listener only, as presented in Figure 2.

Fewer monitors analyze expressive requirements, such as the one below:

```
After each request, Rq, service S shall provide
a corresponding response, Rs, within time t.
```

Here, the monitor does not send dummy requests. Instead, it monitors each service request and response using an event adaptor.

Service requirements can reference message correspondences, temporal relationships, data integrity, and historical information. For example, consider the following requirement that limits the average response time.

```
The average response time of service S shall be
within time t.
```

Here, the monitor must record each response time and raise an alert when the average response time drops below the threshold.

There is overhead in tracking actual requests and their responses, recording service information, and analyzing complex relationships. Consequently, many service monitors gain run-time efficiencies by relying on the simpler method of "pinging" services to determine their availability.

1.2 Monitoring with ReqMon

Ideally, monitoring systems support the expressive requirements and high-level feedback demanded by end users, while ensuring usage through practical efficiency. ReqMon does so[2]. It is a distributed, scalable requirements monitoring application framework. Lightweight extensions of web service technologies provide run-time practicality and efficiency, while design-time activities make use of object-oriented requirements analysis.

The ReqMon approach is most similar to that of FLEA[1]. Both approaches use an object-oriented requirements language (KAOS) that includes real-time temporal logic operators. ReqMon builds on FLEA by: (1) providing tactics for deriving monitors from requirements, (2) addressing distributed concurrent transactions, (3) distributing monitoring and analysis, (4) providing temporal logic primitives, and their aggregates, for monitoring, (5) relying on standard tools (e.g., SQL, web services), and (6) providing automated analysis of web services.

In short, ReqMon compiles high-level monitor definitions into a network of web service monitors. Local site databases record local web service actions. When its database queries determine that high-level system requirements have failed, ReqMon notifies users.

2 Discussion

The presentation of red "danger lights" on service failures is the goal of many monitoring systems. However, much more is possible. For example, users can be warned when a service is about to fail. A monitoring system can model the requirements hierarchy, monitor low-level requirements failure, and present a high-level warning when a lower-level failure occurs. The combination of high-level requirements and an events database provides for a wide range of run-time analysis. Requirements that reference aggregate historical information can be monitored, such as, "A retailer shall average no more than 10 simultaneous credit confirmation requests." Data violations, workflow anomalies, performance degradation, and other non-functional requirements can also be monitored.

Using ReqMon, we have demonstrated requirements monitoring of web services and their relationships. The expressive requirements language and networked implementation architecture provide a wide range of analyses. However, the approach leaves many opportunities for future improvements. These include: (1) assisting obstacle identification, (2) design-time reasoning about monitors, including performance analysis of the distributed system, (3) assisting design-run-time mapping, (4) improving visualization of monitored results, and (5) assisting the run-time responses to warnings and failures. For

web services, failure response planning and execution is particularly interesting, as web services can be dynamically reconfigured. Thus, systems may be able to reconfigure themselves in response to partial failures. Many benefits may derive from requirements monitoring of web services and their relationships.

References

[1] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," presented at Proceedings of the International Workshop on Software Specification and Design (IWSSD'98), Isobe, 1998.

[2] W. N. Robinson, "Monitoring Web Service Requirements," presented at 12 IEEE International Conference on Requirements Engineering, Monterey Bay, CA, 2003.

A Top-Down Approach to Modeling Global Behaviors of Web Services

Xiang Fu Tevfik Bultan Jianwen Su

Department of Computer Science
University of California at Santa Barbara
{fuxiang, bultan, su}@cs.ucsb.edu

Abstract

Due to the distributed nature of modern composite web services, designers are facing new challenges in both requirement specification as well as logic validation. This paper proposes a top-down design/verification strategy that helps construct composite web services to meet preset system goals. The key to this approach is to specify desired global behaviors with a "conversation protocol" and verify preset system goals on the global protocol. Then peer implementations are synthesized from the conversation protocol. Three realizability conditions are provided to guarantee that the composition of synthesized peers will satisfy the previously verified system goals.

1 Introduction

Web services are revolutionizing the way that many e-commerce, consumer software, and telecommunication applications are provided, as indicated by the rapid growing development in the industry standards (e.g., SOAP, UDDI, WSDL, BPEL4WS) and technology (e.g., IBM's Web services Toolkit, Sun's Open Net Environment and JiniTM Network technology, Microsoft's .Net and Novell's One Net initiatives, HP's e-speak). Research communities are providing complimentary technologies from different perspectives. Modeling at a more fundamental level both e-services themselves, and frameworks for combining them have been studied in [5, 8, 14, 15, 29, 16, 2, 1, 17]. New languages for defining services were proposed in [3, 19]. Specialized type systems were considered in [22]. Finally, tools were developed for annotating e-services and for planning, aiming at combining web services automatically to achieve a specified functionality [25, 4, 27, 13]. In this paper, we discuss the issues and techniques in the design, specification, and verification of composite web services.

Since each component of a composite web service is autonomous, no single peer has the control over the global interaction process. Such a distributed nature makes it extremely hard to ensure the correctness of the composite web service merely through the design of each peer individually. In this paper, we argue for a top-down approach from a global perspective in specification and design of web services. On one hand, we show that the bottom-up approach of designing composite web services may result in more complex global behaviors. On the other hand, we illustrate that the top-down approach may further enable existing tools for verification of web services.

In this paper we extend a web service model introduced in [10] and further studied in [20]. A composite web service in this model consists of a set of peers that communicate via asynchronous message passing. In particular, each peer is modeled using a guarded finite state automaton, which abstracts emerging web service choreography standards (e.g. BPEL4WS [6], WSCI [30], BPML [7], ebXml [18]) to characterize behaviors of complex long running services. The asynchronous message passing is achieved by associating each peer with a queue for storing its input messages. This FIFO queue based model resembles many industry efforts like Java Message Service (JMS) [24] and Microsoft Message Queuing Service (MSMQ) [26]. Unlike JMS and MSMQ, there is a virtual "global watcher" in our model that "records" the sequence of messages as they are sent by the peers. A central focus on the global behavior of a web service is to study the set of message sequences generated by the web service, where temporal logics such as LTL [28] can be extended to this framework to specify "good" behaviors. Our previous work in [10] and [20] concentrates on a contentless message model and on how to design a "realizable" global specification, from which (FSA) peers can be synthesized to ensure specified global behaviors without a global coordinator.

Specifically, we define a *conversation protocol* as a set of permissible sequences of messages observed by the global watcher. In [10, 20], we show that it is possible to realize a

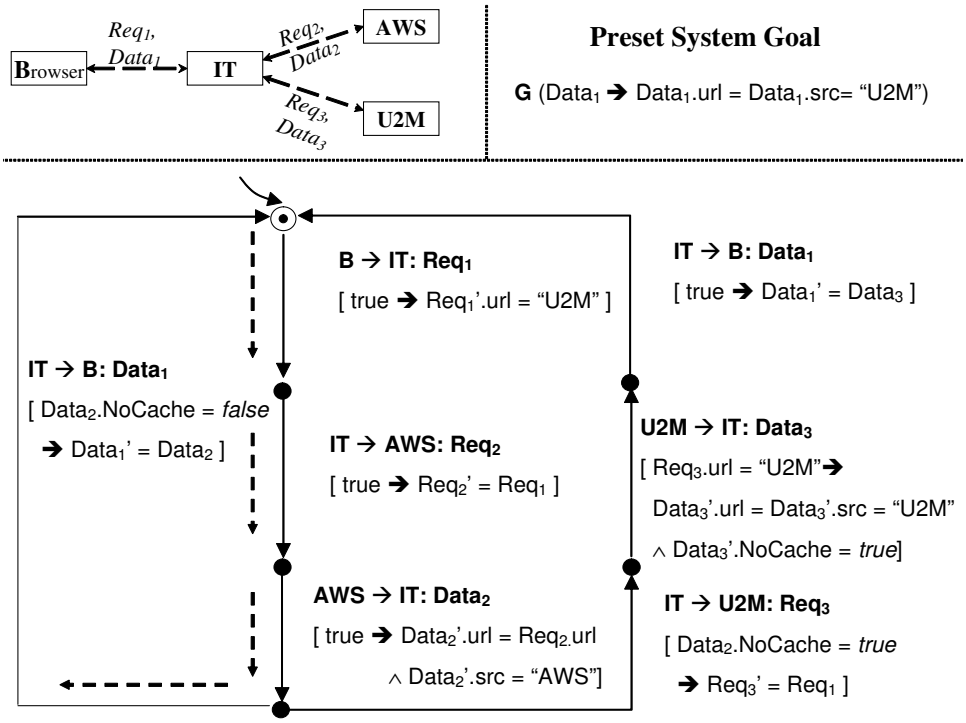


Figure 1. Global conversation specification

conversation protocol using a set of finite state peers, if the protocol satisfies three conditions. Our framework enables a top-down verification strategy where

1. A conversation protocol is specified by a realizable Büchi automaton [9],
2. The properties of the protocol are verified on the Büchi automata specification, and
3. The peer implementations for the conversation protocol are synthesized from the Büchi automaton via projection.

In contrast, we also present a negative fact about the alternative bottom-up approach of specifying the peers of a composite web service in isolation. We show that the composition of finite state peers may result in a non-ω-regular behavior set globally, which makes it difficult to use model checking techniques. In this paper, we generalize the framework of [10, 20], which considers only message classes (names), by allowing messages to have contents. We show that this technique can be used to tackle the U2M problem described in the workshop announcement.

This paper is organized as follows. Section 2 illustrates the conversation protocols with the U2M example in the workshop announcement. Section 3 defines a variation of LTL logic to express system goals such as the "freshness" requirement in U2M scenario. We apply formal model checking techniques in Section 4. In Section 5, we synthesize each peer based on the global conversation protocol in Section 4. As a comparison, Section 6 shows a negative result concerning the bottom up approach. Section 7 concludes the paper.

2 Conversation Protocols

Consider the UpToTheMinuteNews.com (U2M) example: A user accessing the pages of U2M using a web Browser has to go through a Corporate IT (IT) web proxy on the corporate firewall. A company called Acme Web Speedup Services (AWS) provides a caching proxy web service which is used by IT for accelerating web access. This has the undesirable effect of displaying stale web pages from U2M at the user's Browser.

We argue that the system goal that Browser always receives fresh web pages from U2M is fundamentally a global constraint. Although one could derive ad hoc solutions that are local, it is more desirable to obtain a more general global solution, depending on the properties of IT, AWS, and U2M. In Figure 1, we present a conversation protocol specifying the global web service, which consists of four peers: Browser, IT, AWS, and U2M.

A *conversation protocol* is a guarded Büchi automaton enhanced with message contents, and each transition of the automaton consists of two parts:

1. a *message* transmitted from one peer to another, and

2. a *transition guard* that specifies the condition to take the transition as well as assigns the contents of the message being sent.

We use a conversation protocol to characterize the set of *conversations*, i.e., all possible sequences of messages communicated between peers. Then we check whether the conversations meet some preset goals.

As shown in Figure 1, there are two types of messages in the U2M scenario: `Req` and `Data`. Note that we use subscripts to distinguish the same message class transmitted on different channels, e.g. $\texttt{Req}_1$ and $\texttt{Req}_3$. The two message classes are declared in the following.

```
class Req{
   string url;
   ...
}
```

```
class Data{
   string url;
   string src;
   bool   NoCache;
   string htmlPage;
   ...
}
```

Message class `Req` represents an http request, and its attribute `url` contains the address (original source) of the requested web page. Message class `Data` is the response, where `htmlPage` is the web page content, `src` is the actual address it is retrieved from (e.g., a cache server), and attribute `NoCache` is set to `true` if the header of `htmlPage` contains a "no-cache" tag.

In Figure 1, each transition guard is written in the form of "*condition* $\rightarrow$ *assignment*". Take as an example the transition labeled with "$\texttt{IT} \rightarrow \texttt{U2M}$: $\texttt{Req}_3$". The condition "$\texttt{Data}_2\texttt{.NoCache} = \textit{true}$" means that only if the web page returned from AWS contains a "no-cache" tag can the transition take place. The assignment "$\texttt{Req}_3\texttt{'} = \texttt{Req}_1$" means that IT simply relays the request $\texttt{Req}_1$. Note that here primed variables refer to the contents of the current message being sent, and non-primed variables denote the corresponding fields of the latest transmitted message of that message class.

Intuitively, the desired conversations specified by the protocol in Figure 1 are as follows. In each round of a conversation, the first message is a request ($\texttt{Req}_1$) from Browser to IT. IT relays this request to cache service AWS, and waits for its response $\texttt{Data}_2$. AWS guarantees that $\texttt{Data}_2$ is a matching response for $\texttt{Req}_2$, by ensuring that their `url` are equal; and AWS also sets the actual source `src` of the response to the value "AWS". IT then examines the contents of $\texttt{Data}_2$ from AWS, if the page does not contain a "no-cache" tag, IT just sends this cached page to Browser; otherwise, it will fetch the page directly from U2M. Note that U2M guarantees that each page it sends contains the "no-cache" tag, and their `url` and `src` are properly set.

3 Using LTL to State the System Goal

Now the immediate question is how to express the preset system goal that Browser should always get non-cached U2M news pages from IT. We extend the linear temporal logic (LTL) [28] to fit into our message passing framework. To facilitate the discussion, we clarify some of the technical notions first. Given a conversation $w = w_0, w_1, w_2, \ldots$, a sequence of *messages* with contents, let w_i denote the i-th message in w, and $w^i = w_i, w_{i+1}, w_{i+2}, \ldots$ denote the i-th suffix of w. An *atomic proposition* is either in the form of c where c is a message class, or $c.pred$ where $pred$ is a predicate over the attributes of c.

Let AP be the set of atomic propositions. A message m is said to *satisfy* an atomic proposition $\psi \in AP$, written as $m \models \psi$, if

1. when ψ is a message class, the type of m is ψ, and

2. when ψ is in the form of $c.pred$, then the type of m is c and $pred(m) = true$.

LTL properties are constructed from such atomic propositions, logical operators $\wedge$, $\vee$, $\neg$, and LTL operators **X**, **G**, **U**, **F**. Given LTL formulas ϕ, and φ, and an atomic proposition $\psi \in AP$,

$w \models \psi$	iff	$w_0 \models \psi$ if $\psi \in AP$
$w \models \neg\phi$	iff	$w \not\models \phi$
$w \models \phi \wedge \varphi$	iff	$w \models \phi$ and $w \models \varphi$
$w \models \phi \vee \varphi$	iff	$w \models \phi$ or $w \models \varphi$
$w \models \mathbf{X}\phi$	iff	$w^1 \models \phi$
$w \models \mathbf{G}\phi$	iff	for all $i \geq 0, w^i \models \phi$
$w \models \mathbf{F}\phi$	iff	there exists $i \geq 0, w^i \models \phi$
$w \models \phi\mathbf{U}\varphi$	iff	there exists $j \geq 0$, s.t. $w^j \models \varphi$ and, for all $0 \leq i < j, w^i \models \phi$

Intuitively temporal operator **X** means "next", **G** means "globally", **F** means "eventually", and **U** means "until". We give some examples of LTL properties and their semantics in the following.

1. **G** `Data` : every message appeared in the conversation is of type `Data`.

2. **G**(`Req.url="U2M"` $\Rightarrow$ **F** `Data.url="U2M"`) : for each message `Req` with `url` equal to "`U2M`" eventually there is a matching response `Data` with `url` equal to "`U2M`".

Similarly, the "freshness" system goal of U2M scenario can be expressed as

$$\mathbf{G}(\texttt{Data}_1 \Rightarrow \texttt{Data}_1\texttt{.url=Data}_1\texttt{.src="U2M"}) \quad (1)$$

That is, every U2M news page retrieved by IT should be a non-cached fresh page.

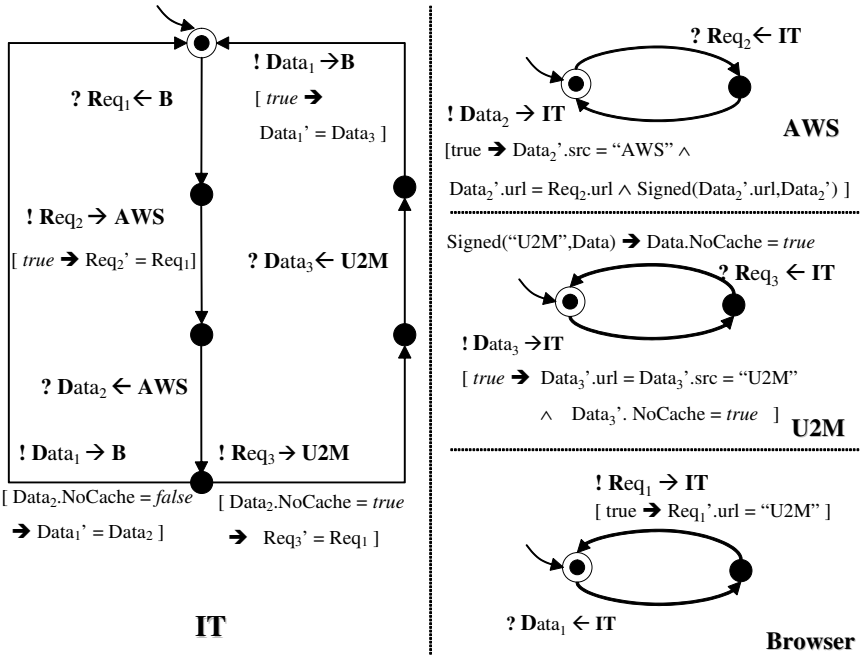


Figure 2. Synthesized implementation of each peer

4 Model Checking the U2M Design

Given a Büchi specification of conversation protocol, it is possible to transform it into the language of a model checker, such as Spin [23], SMV [12], and Action Language Verifier [11]. Note that, depending on the restrictions on data types and domains imposed by model checkers, the translation may require abstractions. After the translation, we can verify whether the proposed system goal is satisfied by the conversation protocol using model checking. For the example presented in Figure 1, model checking can reveal that the LTL property (1) is not guaranteed by the initial design, and an error trace is marked using dashed arrows in Figure 1. The problem with the initial design is that AWS may forge a page whose attribute `NoCache` is false, which is later relayed by IT to the Browser. Thus we need to require that AWS is always "honest". To express this concept, we introduce a predicate `Signed(url: site, Data: page)`, which means intuitively that `page` is digitally signed by the web service at url `site`. Then the following formula can be inserted into the guard of the transition $\texttt{AWS} \rightarrow \texttt{IT} : \texttt{Data}_2$ in Figure 1.

$$\texttt{Signed(Data}_2\texttt{.url',Data}_2\texttt{')} = \texttt{true} \quad (2)$$

Interestingly, even if AWS makes the "no-deception" promise, it still cannot guarantee the freshness requirement. For example, if at some point, U2M forgets to insert "no-cache" tag into its web page, and somehow this page happens to be stored in AWS. When IT requests the page, AWS can send this digitally signed "bad" page to IT which causes the failure of freshness. Therefore if we strengthen the design of U2M with the following system assumption:

$$\texttt{Signed("U2M", data)} \Rightarrow \texttt{data.NoCache} = \texttt{true} \quad (3)$$

we can safely reach the conclusion that the LTL property (1) is satisfied. Model checking of the new composed system with guard (2) and system assumption (3) requires the ability to handle first order formulas.

5 Synthesis of Peers

Synthesis of peers is obtained by projecting the conversation protocol to each peer by removing non-relevant transitions for each peer. Then we detach guards for those transitions that are labeled with incoming messages, since a peer cannot control the contents of its incoming messages. The projection results in a guarded Büchi automaton for each peer. As an example, in Figure 2, we present the synthesized peers for the refined version (enhanced with Equations 2 and 3) of the U2M example in Figure 1.

It can be verified that, in an asynchronous message passing environment (where a FIFO queue is used to store incoming messages), the composition of finite state peers in Figure 2 produces exactly the same conversation set as described by the refined protocol of Figure 1. However, not every conversation protocol has this "realizable" property.

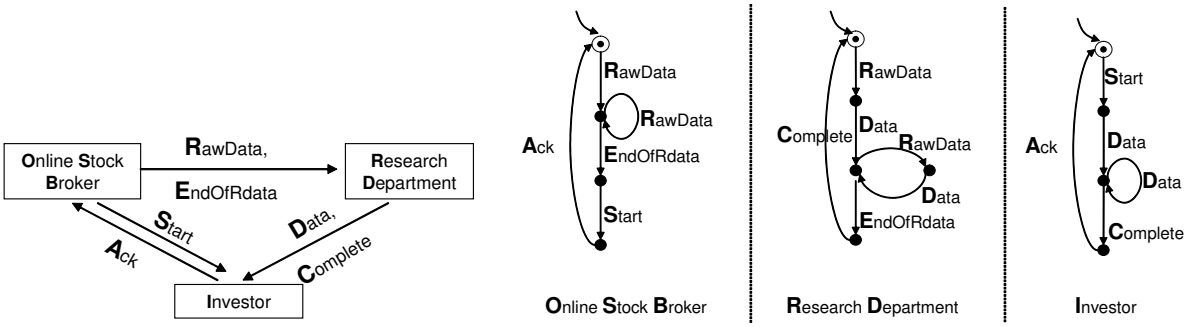


Figure 3. Fresh Market Update Service

In [20], we presented three conditions that can ensure the realizability of a conversation protocol. We briefly introduce them below:

Lossless join property requires that a conversation protocol should be equivalent to the Cartesian product of its projections to each peer.

Autonomous property requires that at any moment according to the protocol, each peer can make a deterministic decision on whether to wait, or to send, or to terminate.

Synchronous compatible property requires that there is no "illegal" state in a conversation protocol where some peer is ready to send a message that is not expected by its receiver.

We argue in [20] that conversation protocols satisfying these three realizability conditions can still capture a large category of web service patterns. However results in [20] cannot be directly applied to conversation protocols with message contents. In [21] we show that by employing the state space exploration technique, for a conversation protocol with finite domains, we can always construct a standard guardless protocol which bisimulates the original protocol. Running realizability check on its guardless bisimulation usually suffices to justify a realizable guarded conversation protocol with message contents.

6 Problems of Bottom-up Approach

One natural question concerning the bottom-up specification of composite web services, i.e., to specify each single peer first and then compose them, is whether we can always construct such a global conversation specification recognized by a finite state automaton? A positive answer would imply that many verification techniques become immediately available. Unfortunately, we show that the answer is negative, even when message contents and guards are not considered. There are composite web services whose conversation set cannot be recognized by finite state automata.

Consider the scenario shown in Figure 3. There are three participants, namely OSB (Online Stock Broker), RD (Research Department), and Investor, involved in a "Fresh Market Update" (FMU) service. We describe each service using a Büchi automaton, and note that each service is equipped with a FIFO queue to store incoming messages under the asynchronous message passing environment like the Internet.

The interaction pattern between the three peers is described as follows. In each round of message exchange, OSB first collects "Rawdata" (e.g. the market price and volume of each stock) from the market, and then sends them to RD for further analysis. After all "Rawdata" are collected and sent, OSB sends the message "EndofRdata" to mark the end of "RawData", and it sends the message "Start" to inform Investor about the planned arrival of a sequence of "Data". RD processes each "Rawdata" and generates a corresponding polished report named "Data". After all "RawData" have been processed, RD sends the message "Complete" to Investor. Once informed by the "Complete" message, Investor sends the message "Ack" to OSB so that OSB can start another round of market information collection and analysis.

The seemingly simple FMU scenario produces a non ω-regular language. To see why this is the case, consider its intersection with an ω-regular language[1] $(\mathbf{R}^*\mathbf{ESD}^*\mathbf{CA})^\omega$. One can infer that the result is $(\mathbf{R}^i\mathbf{ESD}^i\mathbf{CA})^\omega$. By an argument similar to pumping lemma, we can show that this intersection cannot be recognized by any Büchi automaton, and hence the set of conversations is not ω-regular. In fact, given a set of finite state peers, the problem of checking if all conversations generated by them satisfy an LTL property is undecidable due to the unbounded input queues associated with peers. This negative result is one of the motivations for our top-down approach to specification of web services.

[1] We denote each message by its first letter. For example, **R** is the "Rawdata".

7 Discussions

While using the top-down approach enables us to take advantage of model checking techniques, there are other challenges. One possible drawback of the top-down approach may be that for the same design, the global specification can be much larger than its bottom-up counterpart. Another drawback can be that the top-down approach does not work well when we try to compose a service from existing services which do not allow alteration of their internal implementations. In addition, the current version of conversation protocol requires that the participants are fixed, i.e., we cannot dynamically determine the destination of a message, e.g., "check the `url` of the `Req` from Browser, and then send a second request to `Req.url`". We are investigating the trade-off between the top-down and bottom-up approaches to address these challenges.

Automatic verification and validation of composite web services is a new area with interesting challenges — the difficulties arise from both the open system aspect and the hardness of verification problem itself. As we mentioned earlier, to verify the design of U2M example in Figure 1, a model checker with abilities to handle first order constraints is required. We are also looking into the issue of enhancing model checkers with theorem provers to validate a non-trivial composite web service design.

Acknowledgments

Bultan was supported in part by NSF grant CCR-9970976 and NSF Career award CCR-9984822; Fu was partially supported by NSF grant IIS-0101134 and NSF Career award CCR-9984822; Su was also supported in part by NSF grants IIS-0101134 and IIS-9817432.

References

[1] S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, M. Hascoet, C-C. Kanne, B. Koechlin, D. LeNiniven, A. Marian, L. Mignet, G. Moerkotte, B. Nguyen, M. Preda, M-C. Rousset, M. Sebag, J-P. Sirot, P. Veltri, D. Vodislav, F. Watezand, and T. Westmann. A dynamic warehouse for XML data of the Web. *IEEE Data Engineering Bulletin*, 2001.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. ACM Symp. on Principles of Database Systems*, 1998.

[3] Philippe Althern. The scala home page. `http://lamp.epfl.ch/scala/`.

[4] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web service description for the semantic web. In *Proc. Intl. Semantic Web Conf. (ISWC)*, July 2002.

[5] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Self-coordinated and self-traced composite services with dynamic provider selection. Technical report, University of New South Wales, March 2001. (Available at `http://sky.fit.qut.edu.au/~dumas/selfserv.ps.gz`).

[6] Business Process Execution Language for Web Services (Version 1.0). `http://www.ibm.com/developerworks/library/ws-bpel`, 2002.

[7] Business Process Modeling Language (BPML). `http://www.bpmi.org`.

[8] R. Breite, P. Walden, and H. Vanharanta. C-commerce virtuality - will it work in the Internet? In *Proc. of International Conf on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000)*, 2000. (`http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`).

[9] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[10] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[11] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, 2001.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, January 1990.

[13] C. Bussler, R. Hull, S. McIlraith, M.E. Orlowska, B. Pernici, and J. Yang, editors. *Proceedings of Workshop on Web Services, E-Business, and the Semantic Web (WES)*. Springer-Verlag Lecture Notes in Computer Science, number 2512, Toronto, 2002.

[14] F. Casati, S. Sayal, and M. Shan. Developing e-services for composing e-services. In *Proceedings of CAISE 2001*, Interlaken, Switzerland, June 2001.

[15] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

[16] V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong, and M. Xiong. Beyond discrete e-services: Composing session-oriented services in telecommunications. In *Proc. of Workshop on Technologies for E-Services (TES)*, Rome, Italy, September 2001.

[17] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Proc. Int. Conf. on Data Engineering*, 2002.

[18] Electronic Business using eXtensible Markup Language. `http://www.ebxml.org`.

[19] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for web service specification and composition. In *Proc. Int. World Wide Web Conf. (WWW)*, 2002.

[20] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. Int. Conf. on Implementation and Application of Automata (CIAA)*, 2003.

[21] X. Fu, T. Bultan, and J. Su. Model checking conversation protocols: A top-down approach to specification and verification of web services. *manuscript*, 2003.

[22] S. Gay and M. Hole. Types for correct communication in client-server systems. Technical Report CSD-TR-00-07, Department of Computer Science, Royal Holloway, University of London, December 18 2000.

[23] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[24] Java Message Service. `http://java.sun.com/products/jms/`.

[25] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. In *IEEE Intelligent Systems*, March/April 2001.

[26] MicroSoft Message Queuing Service. `http://www.microsoft.com/msmq/`.

[27] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. Int. World Wide Web Conf. (WWW)*, 2002.

[28] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[29] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May 2000. (`http://www.w3.org/TR/SOAP/`).

[30] Web Service Choreography Interface (WSCI) Version 1.0. `http://www.w3.org/2003/01/wscwg-charter`.

A Negotiation-based Framework for Requirements Engineering in Multi-stakeholder Distributed Systems

Paul Grünbacher *Fritz Stallinger*
Johannes Kepler University
Systems Engineering & Automation
Linz
Austria
{pg,fs}@sea.uni-linz.ac.at

Neil Maiden
City University London
Center for HCI Design
London
UK
n.a.m.maiden@city.ac.uk

Xavier Franch
Universitat Politècnica
de Catalunya
Barcelona, Catalonia
Spain
franch@lsi.upc.es

1 Introduction

Negotiation methods, techniques, and tools for identifying and reconciling expectations of success-critical stakeholders in requirements elicitation are available for a while now [2][3][16]. The win-win negotiation model for example provides a simple yet effective approach to identify and capture stakeholder interests and reconcile conflicts. In win-win the personal goals of stakeholders are expressed as *win conditions;* constraints, risks, and uncertainties are modeled as *issues*; alternatives overcoming these issues are captured as *options*. Win conditions not raising any issues or options are turned into *agreements* describing mutually satisfactory conditions [1].

While at the first glance one might expect that these negotiation artifacts are centered solely on goals and constraints about the problem to be solved, our experience in numerous real-world negotiations shows that for a negotiation to be successful the solution space has to be considered equally. Consequently the win conditions, issues, options, and agreements also deal with architectural concerns. For example, stakeholders often state win conditions about adopting a certain component or service. Issues are often related to constraints arising from existing components or IT infrastructure. Addressing both the problem space and the solution space in requirements negotiation is further required as partial solutions are often available (e.g., legacy assets, COTS).

Similar challenges resulting from the need to address both the problem and the solution space are faced in the context of engineering multi-stakeholder distributed systems (MSDS), in particular when these are deployed as dynamic configurations of instantiated components and connectors. According to [9] a multi-stakeholder distributed system (MSDS) "is a distributed system in which subsets of the nodes are designed, owned, or operated by distinct stakeholders." These nodes are often designed or operated in ignorance of one another or with different, possibly conflicting goals. Popular examples for MSDS are electronic mail, or networks of web services. In an MSDS the requirements placed by diverse stakeholders are often ephemeral and conflicting since details about the elements of such a dynamic system are largely unknown to single stakeholders and outside their sphere of control.

In this paper we explore the idea of how a negotiation-based approach can help in tackling the challenges posed to requirements engineering by open MSDS. We present a framework to explain how different aspects of MSDS are interrelated. In this context we will briefly discuss our existing research on reconciling requirements and architectures [8] and on applying the *i** actor-based modeling approach for modeling architectures [6]. We will also explore the framework in the context of the problems exemplified in the workshop call.

2 Framework Overview

Figure 1 depicts the overview of a layered framework for requirements engineering for MSDS. The purpose of the framework is to bridge the stakeholder view and the Open System view in requirements engineering for MSDS. The framework supports the separation of concerns in MSDS by proposing clearly defined layers reflecting the 'translation' of personal, time-dependent, often tacit stakeholder interests and needs into the runtime elements of an Open System. It facilitates the clarification of the roles and responsibilities involved at different layers of the translation process and the classification of problems and mismatches in MSDS. It further helps to examine the role of negotiation within the different layers and emphasizes the need to integrate requirements and architectures [15].

The framework distinguishes the following layers:

Stakeholder Needs: The personal, possibly ephemeral needs of a dynamic set of stakeholders are assumed to be the starting point for requirements engineering in MSDS. Some of the stakeholders might be representing other stakeholders. They might pursue a mutual goal but can also act as individuals. Their needs and interests are driven and influenced by their values and beliefs [11].

Requirements shall usually be reconciled with existing software assets. On the one hand, when the ongoing MSDS is replacing an existing system, some legacy assets may be kept, either permanently or just in the early stages of development, to assure that a particular service will be offered without interruption. On the other hand, early exploration of the COTS marketplace reveals the strengths and limitations of the available components that can be used in the MSDS, making new requirements to appear and also identifying which ones will be more important to discriminate among components [12]. Flexibility of requirements becomes crucial to avoid COTS selection failure or to require component modifications.

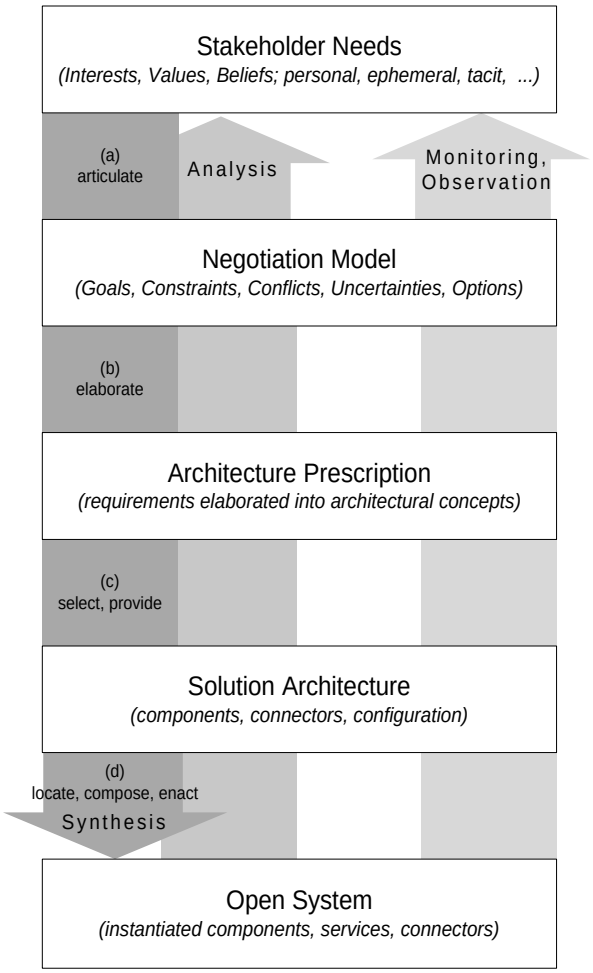


Figure 1: Framework for MSDS RE

Negotiation Model: Stakeholders articulate their individual goals in some negotiation language, for instance the win-win model [1]. A goal such as "ride to doctor by B" mentioned in the example of the workshop call could be expressed as a win condition. Win conditions represent personal desires and possibly conflict with win conditions of other stakeholders. In the case of a conflict an issue is created. Issues make the risks, conflicts, and uncertainties explicit thus providing an incentive for stakeholders to negotiate to achieve a solution. Issues they are resolved by proposing options that overcome the identified problems. Stakeholders can turn win conditions raising no issues or options into agreements. The goal is to achieve the win-win equilibrium requiring that all win conditions are covered by agreements and there are no unresolved issues.

Negotiation is particularly important when selecting COTS components to be integrated in the MSDS [1]. In addition to dilution of control, changes in the COTS marketplace makes requirements vulnerable; new releases may include features that conflict with current requirements. Negotiation becomes then a continuous process.

Architecture Prescription: The elements of the negotiation model (e.g., win conditions, issues, options) are generally stated informally and can hardly be directly mapped onto the architectural elements of the open system. The Architecture Prescription [4] layer is therefore introduced as an intermediate layer to represent the negotiated requirements in terms of architectural concepts [17]. In the context of component-based software engineering this layer is expected to hold the specification of the components and their relationships.

Deriving the elements of this intermediate layer can be done intuitively but could also benefit from existing methods for identifying architecturally relevant information from stakeholder goals, like the CBSP (Component, Bus, System, Property) approach that helps to extract architecturally relevant information from requirements [8]. A win condition "ride to doctor by B" could for example be refined into several desired components ('mail client a', 'mail client b', 'mail server a', 'mail server b') and properties ('reply within two days').

Another innovative approach is our application of the *i** [5] actor-based modeling approach for modeling software architectures, not in terms of connectors and pipes, but in terms of actor dependencies to achieve goals, satisfy soft goals, use and consume resources, and undertake tasks [6]. *i** dependency types were used to model dependencies between actors that fulfill roles in the architecture and can be instantiated by different software components with different characteristics. We identified some architectural properties aimed at suggesting how well an instantiation complies with different non-functional requirements such as dependability and usability. These properties were defined in terms of the dependencies that appear in the model, both before and after the instantiation, taking into account that a component may play more than one role in the architecture and thus may hide some dependencies among them. As a result, we have defined a framework for comparing different types of architectures for a system in terms of how they fulfill the dependencies

among them. Such modeling, which integrates requirements- and architecture-modeling based on the notion of actor, has real potential for negotiation-based requirements processes. The key concept, that different components can instantiate different actors to provide the consequences of different architectural permutations in the requirements domain, offers us a novel insight into architecture prescription.

Solution Architecture: The elements defined in the Architecture Prescription are generally just candidate elements (or rather specifications of those) that have to be mapped to the real world elements of the open system [17][13]. For example, a mapping of candidate element 'mail client a' to the real world component 'Eudora' is required. In the context of open MSDS it must be noted that the implementation and evolution of these real world components are often not under control of the stakeholders associated with the top layer of the framework.

Open System: The Open System layer finally represents the dynamic configuration of components, services, and connectors with their properties at runtime which is supposed to actually satisfy the needs of the stakeholders represented in the top layer of the framework. In the context of open MSDS in general and of component-based software engineering or web Services in particular it must again be highlighted that the deployment and operation of these real world elements are typically not under control of these stakeholders.

The presented framework is intended to support both the synthesis (i.e., selection, composition) of a system based on stakeholder needs as well as its analysis where these activities are assumed to take place iteratively at each framework level. These two generic activities traditionally take place during system design and implementation. Remarkably, analysis may be used to support return of investment for future MSDS developments; for instance, the development process itself may be tailored to specific scenarios and improved through experiences, and best practices may be identified, e.g. construction of quality models for representing the information about software quality [7].

In the context of open MSDS we argue that a monitoring and observation activity (i.e., feedback from the system during operation back to the stakeholder) focusing on the runtime behavior of the system and potential mismatches with stakeholder needs is necessary to cope with the problems emerging from open MSDS. Effective monitoring mechanisms to address actual performance of services and observing the open system to ensure satisfaction of stakeholder goals and alert the stakeholder about the violation of goals (e.g., by registering issues in the negotiation model) is crucial to requirements engineering in an open MSDS context and expected to change the role of the before mentioned synthesis and analysis activities to ongoing activities during the lifetime of the system.

It must further be noted that Figure 1 only captures the view of one stakeholder or one group of stakeholders onto the open system, while generally there are often multiple such stakeholder groups simultaneously placing requirements on parts of the system. The number of these groups and their requirements are generally unknown to the group of stakeholders under consideration but represent an inherent property of the open system particularly causing evolution and change in the two bottom layers of the framework. It is this evolution and change together with the time-dependency of user needs that pose the need for above mentioned monitoring and observation mechanisms.

3 Conclusions

In this paper we have discussed that ephemeral and user-relative requirements can be captured in a negotiation model and have shown that such a model can be further translated into an open system based on best practice in component-based software engineering [18][10] and the integration of existing methods [6][8]. We have further argued that the inherent properties of open MSDS and the way stakeholders use them to fulfill their needs require the integration of monitoring and observation activities into the requirements engineering process for MSDS taking over the role of traditional validation activities and leading to a changed, more dynamic role of 'traditional' synthesis and analysis activities. A dynamic view on requirements and their translation into a system is thus essential in MSDS. Traditional negotiation models need extension and integration with monitoring and observation mechanisms to deal with the challenges of open MSDS. The presented framework is intended to provide the skeleton for further discussion and organization of this challenging task.

4 References

[1] Alves C., Finkelstein A. Negotiating Requirements for COTS-based Systems. REFSQ' 2002, Essen, Germany, September 9-10 2002.

[2] Boehm, B., Bose, P., Horowitz, E., Lee, M.J. Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach, *Proc. ICSE'95*, IEEE CS Press, Los Alamitos, Calif., 1995.

[3] Boehm, B., Grünbacher, P., Briggs, B. Developing Groupware for Requirements Negotiation: Lessons Learned, *IEEE Software*, pp. 46-55, May/June 2001.

[4] Brandozzi, M., Perry, D. E.. Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. *ICSE 2001 Workshop From Software Requirements to Architectures*, Toronto, May 2001.

[5] Chung L., Nixon B.A., Yu E. & Mylopoulos J., 2000, Non-Functional Requirements in Software Engineering, Kluwer.
[6] Franch X., Maiden N.A.M., 2003, Modeling Component Dependencies to Inform their Selection, to appear in Proceedings 2nd International Conference on COTS-Based Software Systems, Lecture Notes on Computer Science 2580, Springer.
[7] Franch X., Carvallo J.P., Using Quality Models in Software Package Selection. IEEE Software, 20(1), 2003.
[8] Grünbacher P., Egyed A.F., Medvidovic N., Reconciling Software Requirements and Architectures: The CBSP Approach, Proceedings 5th IEEE International Symposium on Requirements Engineering (RE '01), August 27-31, 2001, Toronto, Canada.
[9] Hall, R.J. "Open modeling in multi-stakeholder distributed systems: requirements engineering for the 21st Century," in Proc. First Workshop on the State of the Art in Automated Software Engineering, U.C. Irvine Institute for Software Research, June 2002. URL ftp://ftp.research.att.com/dist/hall/papers/openmodel/openmodel-asewshp02.pdf
[10] Henderson-Sellers, B., Stallinger, F., Lefever, B., The OOSPICE Methodology Component: Creating a CBD Process Standard. F. Barbier (ed.): Business Component-Based Software Engineering, Kluwer Academic Publishers, 2002, pp. 135-149.
[11] Krumbholz M. & Maiden N.A.M., 2001, 'The Implementation of ERP Packages in Different Organisational and National Cultures', *Information Systems Journal*, 26(3), 185-204.
[12] Maiden N., Ncube C. Acquiring Requirements for COTS Selection. *IEEE Software* 15(2), 1998.
[13] Medvidovic, N. Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93 (January 2000).
[14] Nuseibeh, B., Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115-117, March 2001.
[15] Paech, B., Detroit, A.H., Kerkow, D., von Knethen, A., Functional requirements, non-functional requirements, and architecture should not be separated, REFSQ' 2002, Essen, Germany, September 9-10 2002, http://panoramix.univ-paris1.fr/CRINFO/REFSQ/02/.
[16] Robinson, W.N., Volkov, V. Supporting the Negotiation Life Cycle. 95-102, *Communications of the ACM*, Vol. 41, 1998.
[17] Shaw, M., Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.
[18] Stallinger, F., Henderson-Sellers, B., Torgersson, J., The OOSPICE Assessment Component: Customizing Software Process Assessment to CBD. F. Barbier (ed.): Business Component-Based Software Engineering, Kluwer Academic Publishers, 2002, pp. 119-134.

Semantic Interoperability by Aligning Ontologies*

Karin Koogan Breitman
Pontifícia Universidade Católica do Rio de Janeiro
karin@inf.puc-rio.br

Researchers from industry and academia are now exploring the possibility of creating a "Semantic Web," in which meaning is made explicit, allowing machines to process and integrate Web resources intelligently. This technology will allow interoperability among development of intelligent internet agents in large scale, facilitating communication between a multitude of heterogeneous web-accessible devices. Unfortunately the majority of the information available is in a format understandable to humans alone, thus creating the need to provide an adequate amount of semantics to allow for some of the information filtering to be done by machines. The emergent technology to address this problem is the codification of the information using ontologies, i.e., conceptual models that embody shared conceptualizations of a given domain [Gruber93].

We believe the task of ontology building belongs to requirements engineers - after all we are trained in conceptual modeling techniques, and that is what building ontologies is all about. We have developed a process in which we use a special lexicon as a starting point to building ontologies [Leite93, Breitman03]. The idea of using a glossary of terms in the early development of ontologies is not new, and it is supported by some ontology development methodologies [Ushold96, Fernandez-Lopez97, Gruninger95]. The basic idea is to start with an informal definition and use a stepwise refinement process until the desirable level of formality is achieved. In our case the output of the process is a machine processable ontology written in DAML + OIL.

We are currently developing automated support to our process using the open source C&L tool, available at http://sl.les.inf.puc-rio.br/cel/aplicacao/. The tool was initially proposed as a lexicon and scenario edition and management environment. The ontology generation plug-in is currently under way and is scheduled to be made public in August, 2003.

Central to our research is Tim Berner's Lee [Berners-Lee01] belief that in the near future every site and web application will have to make available its ontology and Jim Hendler's notion that instead of having a few carefully crafted (by AI experts, such as the WordNet [Fellbaum98] and CYC efforts [Guha90]) ontologies, there will be a "*great number of small ontological components consisting largely of pointers to each other*" [Hendler01]. The result will be a great variety of lightweight ontologies both built and maintained by independent parties (not necessarily with expertise in ontology development).

Our recent experiences with ontologies have demonstrated that ontology development is not particularly challenging compared to building any other conceptual model used in our RE practice such as KAOS [Bertrand98] and i* [Yu97]. The constructs are not extremely complex, neither is the level of formality required. Any person familiar with basic concepts of first order logic and the notions of *subsumption* and *aggregation*, should not experience major difficulties in the process. In terms of implementation languages, although today's scenario may seem confused,

* This research was supported in part by CNPq under contract ESSMA- 552068/2002-0, and by CAPES.

there seems be a convergence to DAML+Oil (or its equivalent OWL DL sublanguage) and a consensus that the language while being expressive enough, still allows for adequate automatic support, as put by the W3 consortium " *maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computed) and decidability (all computations will finish in finite time).* " In addition there are editors, such as OilEd, that provide automated support to the edition and maintainance of ontologies. Structural consisitency can be automatically verified with the FaCT tool. Both tools are available at http://oiled.man.ac.uk/.

The real bottleneck is, in our opinion, to secure what is commonly referred to as *"semantic interoperability"*. That means that open system applications with different ontologies will have to undergo a negotiation process. This operation is named ontology alignment and it aims at an intermediate representation that can be shared by both applications. While aligning ontologies one can merge the two ontologies into one, integrate (negotiate and decide on a representation that uses concepts from both ontologies) or simply translate. For this purpose no tools, to the best of our knowledge, are available other than translation mechanisms between specific ontologies, e.g., the iCal to DAML agenda ontologies [translator].

The initial requirement to align different ontologies is being able to list differences and inconsistencies between both ontologies. We are currently specifying a mechanism to detect such differences between ontologies. In particular we are aiming at identifying :

Concepts using different names (labels) for the same meaning

- Differences in the number of restrictions (differentiate among cases where there is intersection of restrictions)
- Differences in the properties used in the restriction - related concepts are similar
- Differences in the related concepts used in the restriction - properties are similar

Concepts with the same name (label) with different meaning

- Identify differences in restrictions
- Identify differences in the properties used

Properties with different name (label) and same meaning

- Verify if all concepts the properties relate are equivalent in both ontologies

Properties with the same name (label) and different meaning

- Verify if the concepts that use the property are consistent in both ontologies

Early attempts were made comparing the ontologies in their native representation language, DAML+OIL. Some problems arose from the expressiveness (or lack) of DAML+Oil itself. Neither concepts nor properties accept synonyms, therefore the concept dog from Ontology1 and dogs from Ontology2 would be considered different. The use of synonyms could alone avoid mismatches caused by plural/singular (dog/dogs), male/female (salesman, saleswoman) and verbal time (pays, pay).

We are currently experimenting with the intermediate version we produce while applying our process to the lexicon. In this representation we have a database that contains information present in the lexicon and the ontological structure that is built as a result of the application of the process. This repository contains more information that is currently provided by DAML+OIL, synonyms and structured descriptions of the terms of the lexicon (denotation and connotation). At

this point, we have not experimented enough to make any suggestions to a possible need of additional information to the DAML+OIL notation. We have noticed, however, that the use of synonyms decreased the number of items in the list of discrepancies. Of course this attempt can only be applied to ontologies to which we have a lexicon available, more so, a lexicon modeled using the LEL notation [Leite93]. In parallel, we are investigating the possibility of a mechanism that translates the ontologies to a lexical representation. One of the reasons is to facilitate validation with users. We have noticed that the visualization of ontologies is somewhat difficult to users[1]. No tool, to the best of our knowledge, is able to display a broad overview of an ontology but, instead, most tools provide a fish eye view, concept per concept (as it is the case with OilEd or Protégé). Of course that in the conversion process, some information will be lost, for lexicons are flat, as opposed to the hierarchical structure of ontologies. There is also no clear way to represent an axiom in the lexicon.

Despite our efforts our perception is, even at this early moment, that the future is not in trying to obtain total alignment in ontologies. The effort involved is too great and may not be justifiable in the context in which the ontologies will operate. Among the problems are the duration of interaction between two applications - do we have enough time to align the ontologies? Are the requirements so ephemeral in nature that it is cost effective to allow the interaction even in the presence of inconsistency? How much mismatch/inconsistency is allowable? Are levels of similarity an acceptable measure? Is it possible to analyze the impact and the risks involved in tolerating inconsistency between ontological representations? Can we apply classical inconsistency handling approaches to ontologies, such as the one proposed in [Nuseibeh00]?

We are convinced that the solutions to the ontology integration problem are intertwined with our abilities to tolerate and live with inconsistencies, that as put by Easterbrook and Chechnik are "a fact of live" [Easterbrook01]. |It is not an easy shift however, for we have been trained to strive for completeness, consistency and to avoid conflict.

References

[Berners-Lee01] – Berners-Lee, T.; Lassila, O. Hendler, J. – The Semantic Web – Scientific American – May 2001 - http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html
[Bertrand98] - P. Bertrand, R. Darimont, E. Delor, P. Massonet, A. van Lamsweerde GRAIL/KAOS: an environment for goal drivent requirements engineering - Proceedings ICSE'98 - 20th International Conference on Software Engineering, IEEE-ACM, Kyoto, April 1998.
[Easterbrook00] - Easterbrook, S.; Chechik, M. - 2nd International Workshop on Living with Inconsistency - Summary, - IEEE - 2001.
[Fellbaum98] - Fellbaum, C.; ed - WordNet: An electronic Lexical Database - Cambridge, MA - MIT Press - 1998.
[Fernandez-Lopez97]- M. Fernandez, A. Gomez-Perez, and N. Juristo. METHONTOLOGY: From Ontological Arts Towards Ontological Engineering. In Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering, Stanford, USA, pages 33--40, March 1997.
[Gruber93] – Gruber, T.R. – A translation approach to portable ontology specifications – Knowledge Acquisition – 5: 199-220

[1] Which, perhaps, is intentional, as ontologies for the semantic web and, in particular DAML+OIL, were created to provide MACHINE interoperability as opposed to facilitate human understanding.

[Gruninger95] – Gruninger, M.; Fox, M. – Methodology for the Design and Evaluation of Ontologies: Proceedings of the Workshop on basic Ontological Issues in Knowledge Sharing – IJCAI-95, Montreal, Canada, 1995.
[Guha90] - Guha, R. V., D. B. Lenat, K. Pittman, D. Pratt, and M. Shepherd. "Cyc: A Midterm Report." *Communications of the ACM* Vol.33 , No. 8 - August, 1990.
[Hendler01] - Hendler, J. – Agents and the Semantic Web – IEEE Intelligent Systems – March/April - 2001. pp.30-37
[Lamsweerde]
[Nuseibeh01] - Nuseibeh, B.; Easterbrook, S.; Russo, A. - Leverage Inconsistency in Software Development - Computer - Vol 33 No. 4 - April 2000 - pp. 24-29
[translator] - http://www.daml.ri.cmu.edu/site/projects/DMA2ICal/index.html
[Ushold96] - Ushold, M.; Gruninger, M. – Ontologies: Principles, Methods and Applications. Knowledge Engineering Review, Vol. 11 No. 2 – 1996. pp. 93-136
[Yu97] - Yu, E. - Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering - Proceedings of the Third International Symposium on Requirements Engineering - RE97 - IEEE Computer Society Press - p1997 - pp.226-235

Classifying Requirement Conflicts for Multi-Stakeholder Distributed Systems

Frank Marschall, Maurice Schoenmakers
Technische Universität München
Lehrstuhl Prof. Dr. Manfred Broy
Software & Systems Engineering
(marschal|schoenma)@in.tum.de

Abstract

Multi stakeholder distributed systems become more and more widespread and raise a lot of integration problems. One problem is that conflicts arise often only at runtime if a single system component is changed. The whole composition of systems then doesn't behave like at least one of the stakeholders expects. The following paper provides a classification of the potential conflicts and gives some guidelines how to handle and overcome these conflicts using this classification. The classification is based on the fact that some parts of a system implementation can be linked to explicit stated stakeholder requirements, while others are just implementation specific parts that are not related to any explicitly stated requirement. Therefore a prerequisite for a successful conflict resolution is the traceability between requirements of a requirements model and the affected parts of an implementation model.

1. Introduction

Resolving requirement conflicts and combining reusable software components are often tasks performed once during the system development. The system is deployed in a controlled environment, and after deployment, each requirement change enforces new requirement engineering and system integration cycles.

Today systems are more and more composed of distributed services which are under control of loosely coupled stakeholders with possibly conflicting interests. The services are deployed independently and combined at runtime. Examples for such scenarios are web service architectures for B2B e-commerce systems with a large set of business partners or enterprise application integration systems which compromise a large set of single applications of different departments.

The result is that the requirements are likely not to be propagated throughout all participating parties, thus conflicts may not appear although they exist. There is no central explicitly coordinated consistent conceptual model of the system at all times. Instead each participant has its own conceptual model. System changes are performed without notice of other stakeholders, which results in unexpected misbehavior.

In this paper we propose a model for the classification of requirement conflicts and show how to reason about conflicts in terms of a conceptual model by using a little example.

2. Example

The following example stems from [7] and describes a web service scenario. There are four stakeholders: the *user* who uses the *UpToTheMinuteNews* news service, *Corporate IT* that provides internet access for the user through a proxy, and the *AWS company* that provides a caching proxy that is used by Corporate IT.

From the moment when Corporate IT starts using the AWS proxy, the user experiences that the news form UpToTheMinuteNews isn't up to date any more. Obviously this does not meet the user's requirements while Corporate IT and AWS might not even have reasoned about this topic.

3. Requirements

In our model we suppose that stakeholders have requirements which are basically statements about the systems in their scope. These requirements form a requirements model of the desired system. This model is refined into an implementation model that is enriched by design decisions which are not considered in the conceptual system model. When the models are formal an approach like considered by the Model Driven Architecture (MDA) approach [5] may be chosen to derive platform specific models from more abstract platform independent models and computational independent models. Ideally the requirements can be

traced so that one knows from which parts of the requirements model a certain part of the implementation model stems. Since the implementation model is a refinement of the requirements model it has in general properties that were never explicitly required and thus fulfil never stated "requirements".

Thus the implementation model can be partitioned into two parts: one contains the model elements that can be seen as a direct refinement of the conceptual model. The other part contains the elements that were not explicitly specified in the conceptual model. This is typical since requirement models are usually underspecified, i.e. they often leave (intentionally or not) room for design decisions.

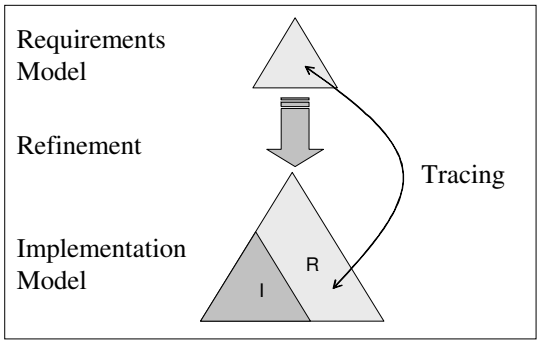


Figure 1. A single system. Assumption: the implementation satisfies the requirements

Figure 1 depicts these facts. The implementation model comprises a part R that was derived from the requirements model and a part I that cannot be mapped to any elements of the requirements model but is necessary for the system to work.

When combining two systems, there exist overlapping parts that both systems have to deal with in the requirements and in the implementation model. For example the requirements models of both systems have to consider the common goals of their collaboration, exchanged data types, messaging mechanisms etc. Often the latter arise only in the implementation model because they didn't seem to be relevant to the stakeholders and thus the decision was left to the developers.

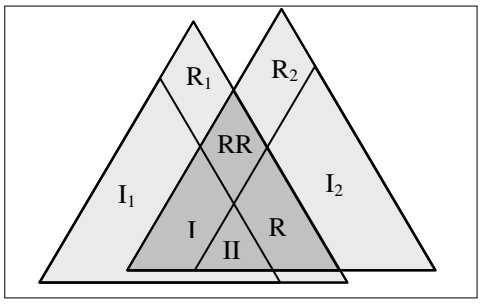


Figure 2: Conflicts between two systems can occur in implementation areas RR, II, IR/RI.

Figure 2 depicts the conflicts that may arise if two participants integrate their systems: the following conflicts can occur simultaneously. There may occur a conflict between parts of the systems that

- reflect the requirements of both participants: **RR conflicts**
- where not considered by the requirements of any participant**: II conflicts**
- where only considered by the requirements of just one of the participants: **IR/RI conflicts**

Requirement conflicts (RR) are fundamental conflicts which must be resolved before the two parties can form a system satisfying to both parties. Normally one tries to prevent these conflicts by exchanging some kind of requirements model description concerning the overlapping parts. For example IT and AWS sign a contract in advance, which contains the rules of operation between these parties. Identifying such a conflict at runtime must result in an adaptation of the requirements at one or both sides or more likely results in dissolving the contract between the two parties.

Conflicts concerning parts of the implementations that are not considered in the requirements model of at least one stake holder, the II and RI/IR conflicts, can be overcome by adapting the implementation without changing the original requirements. However such a conflict indicates that the requirements specification is incomplete and should be completed. Therefore mechanisms are needed that identify the context of the conflicting parts of the implementation model and identify the appropriate area of the requirements part where information is missing. For example if the requirements model doesn't state anything about the message exchange between two components and during operation it turns out that this message exchange fails such a mechanism could lead a stakeholder to the involved components that failed to communicate. Thus the stakeholder would be confronted with the problem in terms of the requirements model, not with a technical error. He can make a decision at the high level requirements model and refer it to the developers. These refine the new requirement into a consistent solution, presumed that the requirements model is not still underspecified for the collaboration of the components.

In case of a legal contract one party may enforce the other party to adapt to an implementation. However, this is not always possible or feasible because each partner controls which implementation is deployed, and the implementation can be chosen at the other party because of other internal requirements. Conflicts in these classes occur often by not having explicitly modelled the requirements on one or both sides.

Some RI/IR conflicts between requirements of one party and implementations of the other party can be very problematic and should generally be avoided. One party has a requirement accidentally fulfilled by the implementation of the other party while it was not explicitly guaranteed by the conceptual model or contract. The other party can change the implementation at any time which may cause the requirement to become unfulfilled. Therefore such requirements should be explicitly stated and exchanged in advance to lift the fulfilled feature up to the RR class.

In the previously described example, where the user experiences that the news form UpToTheMinuteNews is not up to date any more, because Corporate IT started to use the AWS proxy, there are seven possible reasons given in [7] for the conflict. We now look at these reasons from the classification perspective:

1. AWS's contract has fine print explaining that it does not guarantee freshness of its pages. In this case the requirements model of the user / Corporate IT is underspecified, thus this is an IR conflict. AWS has specified a requirement the user / Corporate IT didn't consider. However adding the new requirement that the information provided by the AWS proxy must be fresh enough would yield to a RR conflict in the requirements specifications that must be eliminated by the involved stakeholders.

2. AWS's contract is purposely inaccurate or unclear at this point. Here two cases can occur: The AWS implementation is caching and delaying requests by purpose then there is an AWS internal requirement hidden for the IT department, which is not stated in the external contract. In this case there is an IR conflict as the IT department did not specify its requirements sufficiently. If the AWS implementation causes the conflict just because the developers choose the implementation accidental, then both parties did not specify the requirements thus an II conflict occurred.

3. AWS's implementation is buggy, old, or incorrectly configured so that it does not honour "no-cache" header information. In this case the refinement of the AWS requirements model to its implementation model failed. Such conflicts must be avoided by efficient testing of the implementation model against its requirements model. The other party can in this case insist on the adaptation of the implementation if possible.

4. U2M's implementation is buggy or incorrectly configured, so that it does not produce "no-cache" headers with its pages. This is the same case as (3).

5. IT failed to read or correctly interpret the AWS contract. In this case the conflict detection between requirements models failed. The RR conflict was not detected. With more formal requirements specifications (e.g. B2B Specifications like ebXML [3]) some of these conflicts can be avoided. However there may still be cases when these specification languages are not expressive enough to formalize all desired requirements.

6. IT failed to realize that some of its users required freshness. In this case the requirements model of IT or that of the users is underspecified (if it doesn't state anything about the freshness of information) or it is simply wrong and needs to be reworked.

7. The user failed to communicate to IT that it needed access to a time critical web site, even though IT surveyed its users on this point at some time previously. Again the requirements model of IT is wrong and needs to be reworked. Both IT and the user didn't specify parts of their overlapping requirement model in an explicit contract.

4. Position

In our opinion the following criteria must be fulfilled to correctly handle occurring conflicts:

- *A) Differentiate between required (R) and not required (I) implementation parts at both parties.* One must be able to differentiate between the system properties which originate from real requirements and those properties stemming from implementation refinement steps chosen by developers for technical reasons unrelated to any stated requirements.

- *B) Tracing back from implementation components to the requirement model.* Because conflicts arise at the implementation level tracing a requirement from the implementation level up to the original requirement level becomes an essential feature. The affected implementation parts of the conflict at implementation level must be related to the requirements in the original requirement model. This allows an expression of the conflict at the language level used by the stakeholders. This in turn allows an effective requirement conflict resolving discussion.

We propose that each partner formulates requirements explicitly and forms a conceptual model to relate elements of the conceptual model to system components as well as to ensure traceability.

Traceability is very important for open distributed multi stakeholder systems. Exchanging model information may be used to prevent RR conflicts but cannot prevent II, RI/RI conflicts. Furthermore as requirements are often not made explicit and requirements and implementations change over time *requirement conflicts may arise only at runtime in form of implementation conflicts*. Only traceability between conceptual model elements and system components of the implementation can then be used to identify to which classes RR, RI, IR, or II the conflict really belongs. To which class a conflict belongs in turn shows how to resolve it and this may reveal the legal and financial consequences.

A conflict is classified as follows: If the conflicting system component is related to a requirement at each party, then the conflict may belong in the RR class, if these requirements are conflicting. In this case there is a fundamental problem. If the conflicting system component is related to a requirement at just one party, the conflict belongs to the RI or IR class and without any related requirements at either side it belongs to the II class. In any case a conflict belonging to the RI/IR or II may also point to an incomplete conceptual model. The parties then can at least pinpoint the problematic areas and discuss resolution possibilities.

In the research project KOGITO [6] the authors address some of the topics mentioned here. The scope of KOGITO is requirement engineering for multi stakeholder B2B internet systems. Typical problems are the integration of different existing open distributed systems. To ensure the above mentioned tracing capabilities KOGITO has defined a multilevel conceptual model. Requirements in documents individually reference subsets of the conceptual model elements. The levels of the conceptual model range from coarse grain business process areas to fine grained message exchange. During system development the refinement steps result in linking the different levels of the conceptual model. The steps become traceable and fine grained system components become related to single requirements at the conceptual level. This ensures the traceability and the capability to differentiate between those implementation parts directly linked to requirements and those not linked to requirements. Furthermore the integration of formalized ebXML [3] business process descriptions at a middle level allow to check and ensure consistent overlapping requirements and helps avoiding RR conflicts by reusing predefined business processes as some kind of contracts.

Until now we discussed cases where both parties were able to get in contact to each other directly, can sign contracts and resolve conflicts in a direct discussion with stakeholders.

This is only possible for open distributed systems with a limited number of participants and a rather stable requirement model, where stakeholders may have a contractual relationship. For fine grained fast changing systems with a high number of participants tracing conflicts back to requirements and resolving conflicts in discussions is not feasible. Especially as the stakeholders may not have any direct contractual relationship. Examples for such systems are large distributed web service [2] or Jini [9] based systems. To trace back conflicts or to generate conflict resolutions automatically would require a high degree of formalization [8], [4]. For such systems it is probably more feasible to avoid conflicts and to check for RR conflicts automatically up to a certain level instead of resolving them. Thus stakeholders will have to agree on standards or models for which they claim to provide a correct implementation.

While the proposed principles and conflict classes do not change, the way conflicts and requirements and are identified and are resolved would change: As RR conflict checks must be performed frequently and automatically. We propose that implementations are accompanied by an explicit formulated abstract conceptual model which expresses the requirements and defines the collaboration between the participating systems (for example as an ebXML business process description [1] with an appropriate role profile). Each implementation would thus be accompanied by a simple formalized version of a requirements model of Figure 1. Before components would be combined the models then could be checked automatically for RR conflicts, this roughly would reflect the case of signing a contract. If no conflicts arise the implementations could be combined. Conflicts of any of the classes II, RI, RI and RR could still occur. However the change of conflicts in the RI and IR classes would be lowered as the requirements would be quite explicitly defined and modeled and RR conflicts could only stem from requirements which cannot be formulated in the formalized requirement description model.

The problem of inconsistencies in the conceptual model respectively between the contract and the implementation is not addressed by this proposal. Such problems occur because the implementation doesn't fulfill the requirements either by purpose (deception by a stakeholder) or by failure would still arise. While it could be guaranteed that such problems would not occur by proving each refinement step in a formal way, it would probably more feasible to combine the requirement models and their implementations with certificates or ratings. A trust relation could be established in the following way: A third party checks if the requirements model of a stakeholder is fulfilled by the stakeholders implementation. If both, the stakeholder using the system and the stakeholder providing the system, trust this third

party, then they can be confident to a certain degree that conflicts will not arise because of deception or failure.

5. Conclusion

The proposed conflict classification allows to reason about occurring conflicts, for example if a conflict is a simple technical implementation issue or a fundamental requirement conflict. It therefore provides valuable hints how to solve these conflicts and to consider the conflict implications.

To classify conflicts the basic capability identified was traceability of refinement during development, how requirements are related to system components of the implementation. This ensures the capability to trace a conflict back from the implementation to the requirement level. This in turn is crucial to identify if a fundamental requirement conflict occurred or merely a technical failure.

The best way to handle conflicts is to avoid them upfront by formulating contracts on a conceptual requirement level. We gave an outlook how more or less formalized conceptual models like ebXML process descriptions could help avoiding requirement conflicts

References

[1] ebXML Business Process Specification Schema Version 1.01 6, http://www.ebxml.org/specs/ebBPSS.pdf, 11 May 2001

[2] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing, 6(2):86{93, March 2002

[3] Electronic Business using eXtensible Markup Language, UN/CEFACT and OASIS, http://www.ebxml.org

[4] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-perspective Specifications", IEEE TSE, 20(8): 569-578, 1994

[5] MDA Guide Version 1.0, Joaquin Miller and Jishnu Mukerji, http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf, Mai 2003 OMG

[6] Towards Model-Based Requirements Engineering for Web-Enabled B2B Applications, Frank Marschall, Maurice Schoenmakers, Proceedings 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), p. 312, April 07 - 10, 2003

[7] Workshop on Requirements Engineering and Open Systems (REOS), Home Page, http://www.cs.uoregon.edu/~fickas/REOS/, 2003

[8] Robinson, W.N., Volkov, S., Conflict-Oriented Requirements Restructuring, GSU CIS Working Paper 99-5, Georgia State University, Atlanta, GA, April 9, 1999

[9] Jim Waldo. The Jini Architecture for Network-centric Computing. Communications of the ACM, pages 76--82, July 1999