Requirement Traceability: A Model-Based Approach

Omar Badreddin
Northern Arizona University
Flagstaff, U.S.A
Omar.Badreddin@nau.edu

Arnon Sturm
Ben-Gurion University of the Negev
Beer Sheva, Israel
sturm@bgu.ac.il

Timothy C. Lethbridge
University of Ottawa
Ottawa, Canada
tcl@eecs.uottawa.ca

***Abstract*—Requirements tractability remains challenging, particularly in the prevalence of code centric approaches. Similarly, within the emerging model centric paradigm, requirements traceability is addressed only to a limited extent.**

To facilitate such traceability, we call for representing requirements as first class entities in the emerging paradigm of model-oriented programming. This has the objective of enabling software developers, modelers, and business analysts to manipulate requirements entities as textual model and code elements. To illustrate the feasibility of such an approach, we propose a Requirement-Oriented Modeling and Programming Language (ROMPL) that demonstrates how modeling abstractions can be utilized to manage the behavior and relationships of key requirements entities.

***Index Terms*—Requirements, Modeling, Action languages, Domain Specific Language, MDA.**

I. Introduction

Software traceability is "the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process" [10]. The importance of software traceability is further stressed in [11]. Nevertheless, there are multiple challenges in order to achieve the desired traceability that include the specification, analysis, and maintenance of such traces. In our view, the initiation of software systems relies on the requirement engineering stage and thus we put emphasis on tracing requirements.

The majority of development methodologies treat requirements activities as separate from the development activities. Agile methodologies, which focus on coding/programming, attempt to bridge the gap between requirement elicitation and the specification and technical development activities. However, requirements engineering activities remain to a large extent distinct from the development activities. This results in many challenges, including managing the links between requirements artifacts and their corresponding development artifacts. Even when the effort and technical infrastructure are in place for managing the links, it is often the case that the development and requirement artifacts go out of sync as the system evolves [8].

Model-oriented methodologies, which adopt the Model Driven Architecture (MDA) or similar approaches, place the focus on models, rather than code. Such models are typically more concerned with system entities (e.g., classes) and behavior (e.g., state machine), than the system's goals, use cases and non-functional properties.

In this paper, we call for treating requirements engineering concepts in a manner integrated with other development activities. We demonstrate how this can be achieved by weaving key requirements elements into a modeling and programming language. Our view is that business analysts, software modelers, and developers should and can model/program requirements that are tightly integrated into development artifacts, effectively managing traceability. To demonstrate this vision, we provide a textual syntax for requirements concepts and integrate them within model-oriented code.

The paper is organized as follows. The next section gives a quick overview of current approaches to trace requirements back and forth and discusses major requirements concepts as presented in requirement engineering research and practice. We then introduce the language abstract and concrete syntax followed by an example from the healthcare domain. Next, we present a discussion on the potential benefits of using the approach and finally, we conclude and refer to future research directions.

II. Background and Related Work

As we consider requirements as one of the core artifacts within the development process, a question arises regarding the requirements engineering technique we should adopt. Alexander [12] presented various requirements engineering approaches including: stakeholder analysis, goal-oriented analysis, scenario-based analysis, and event-driven analysis. In this work we attempt at unifying two major approaches, namely, goal- and scenario-oriented. These approaches have already been integrated under the User Requirement Notation (URN) standard [13], which include GRL and UCM. In this paper, we refer to a subset of the concepts presented within the standard.

Once requirements have been identified, whether upfront in a waterfall fashion, or iteratively in an agile fashion, there comes a need to create and maintain links to other development artifacts. This introduces a number of needs and challenges as follows:

- Requirements should be traceable to verify coverage, avoid redundancies, or to assess the impact of change.
- Traces need to be maintained as the system evolves.
- Traces should be available and accessible at run-time.

- Traces should be viewable at variable levels of abstractions.

Early attempts to connect requirements to code have already been done. For example, Hirschfeld et al. [1] propose the representation of use cases in modern object-oriented languages. The premise is that by making such requirements entities explicit, developers and maintainers can better associate their units of work to the user needs. However, the traceability from the code to use cases is not clear and there is considerable effort required to manage such traceability [15].

There has been some attempts to link together MDA and requirements. The work of Baudry et al. [18] makes use of two concepts from MDA domain, namely, meta-modeling and model transformation. Baudry et al. leverage these concepts and presented an executable Requirement Engineering language. Similarly, Koch and Kozuruba [19] presents a Requirement Engineering language designed as an extension to UML-based Web Engineering (UWE) profile. The approach utilizes model transformation technique to generate draft design models. Such approaches, however, still treats requirements as separate from executable artifacts.

Requirements manifest themselves across multiple parts of the system. This cross-cutting nature means that a single requirement may be related to multiple components. As an example, consider the work of Cleland-Huang [14] that proposes an approach where a probabilistic model is used to automatically retrieve requirements links from the concerned classes to the goal model. The probabilistic model, however, requires frequent user inspections.

Computational reflection is a well established technique that gives a software system the ability to dynamically observe and possibly modify its behavior [17]. One way to achieve such run-time adaptation is by making requirements represented explicitly in the code. This has the potential to enable software systems to reason about and adapt to requirements at run time.

III. Requirement-Oriented Programming Language

In an attempt to address the traceability challenges as described in the previous section, we introduce a Requirement-Oriented Modeling and Programming Language (ROMPL). ROMPL, being a modeling and programming language, is inspired by UMPLE [5] and Alf [4]. Both Umple and Alf attempt to bridge the model-code divide by raising the abstraction level of modern programming languages by incorporating model-level abstractions. Key modeling abstractions are represented textually and are incorporated in the language itself. The premise is that both developers and modelers can manipulate both the code and model abstractions of the system. The objective is to minimize or eliminate the need for round-trip engineering [16]. ROMPL attempts to incorporate, not only modeling elements, but also requirements elements, into modern object oriented languages.

In the following, we first present the language, the metamodel, and the concrete syntax. We then demonstrate the language by representing an example system from the healthcare domain highlighting key requirements and modeling elements of the language.

A. The ROMPL Syntax

The core motivation behind ROMPL is to facilitate requirement traceability. We start with a set of concepts that combines two major approaches related to requirements engineering, namely, scenario and goal based. We then incorporate additional implementation-oriented concepts. The proposed language combines requirements abstractions, such as those of URN, and modeling abstractions, such as those from UML class, state machine, and activity diagrams. The language also supports explicit definition of goals and non-functional requirements (NFRs). To handle algorithmic calculations (for example, to define Key Performance Indicators (KPIs)) the language makes use of imperative code. Figure 1 presents the various concepts used within the language. It is presented as a partial meta-model of the language. In the following, we elaborate on these concepts and their semantics.

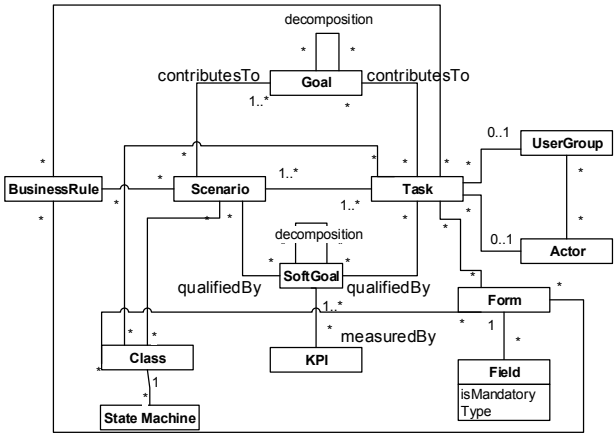

Figure 1: The ROMPL meta-model

- A *Goal* represents a functional requirement (or a set of) to be achieved by the software. A goal can be further decomposed.
- A *SoftGoal* represents a non-functional requirement. In case the *SoftGoal* can be measured quantitatively, it is linked with a KPI. Otherwise, such a *SoftGoal* is expected to be further decomposed, until its components (other *SoftGoals*) are linked with KPIs.
- A *KPI* represent a measurable quality within the system or it environment.
- A *Task* is an action that can be physical (i.e., beyond the system boundaries) or logical (i.e., part of the system). In case it is a human-oriented task, a *Form* representing the UI is attached. An *Actor* (or a *UserGroup*) can also be associated to a task. A task may contribute to one or more goals. However, sometimes this can be explicated indirectly via a scenario.
- A *Scenario* is a collection of tasks that should be executed in a certain order. Scenarios should contribute to a Goal; otherwise their presence is questionable.
- A *BusinessRule* is a constraint that can be associated with various elements presented above.

- The *Class* and *State Machine* metaclasses are adopted from UMPLE and these represent a large portion of syntax constraints and semantics.

B. ROMPL Example

To illustrate the usage of ROMPL, we present an example from the healthcare domain in which a patient is admitted to obtain treatment. The requirements include the support for an admission nurse to enter the patient's vital signs as she or he enters the hospital. Vital signs include blood pressure, and other basic health measurements. The patient's clinical pathway and priority is based on these vital signs. For example, if the patient is suffering from severe pain, then he is given high priority and is required to be re-examined on specified intervals. One of the goals is to minimize patient wait time, measured by comparing actual patient wait time to a specified duration for each clinical pathway. Reducing patients' wait time contribute positively to patients' satisfaction levels. However, satisfaction is also affected by other goals, such as re-admission rates, which the hospital attempts to minimize.

The nurse requires forms to update patients' details and the system must conform to some business rules. For example, if the cost of the treatment exceeds a certain amount, then it must be approved by an accounting representative. As a patient arrives to the hospital, he is *Triaged*, *Admitted*, and then *Discharged* at the end of the hospital visit. Patients who suffer from minor issues can go through a quick assessment process which shortens the duration of their hospital visit significantly. Other patients may need to be admitted and examined before being recommended for discharge.

An excerpt of the system is implemented in ROMPL in Listing 1. For brevity, we represent a subset of the textual system specifications (including requirements, models, and code).

```
1  Goal AdmitPatient{}
2
3  Class Patient {
4    Integer age;
5    Name;
6    1 -- * Registration;
7
8    patientStates {
9      Admitted { .. }
10     Re-Admitted { .. }
11     Discharged { .. } } }
12
13 Form Registration {
14   Patient.age mandatory;
15   Patient.name mandatory;
16   symptom optional;
17
18   //state machine to define
19   //behavior of the form.
20   status {
21     Open {
22       submit [complete] ->
23 Submitted
24       close -> Closed; }
25
26     Closed {
27       entry/ {saveFormData();} }
28
29     Submitted {
30       reOpen -> Open; } }
31
32   calculatePriority {
33     // Algorithmic code to
34 calculate
35     // priority of patients. } }
36
37 Actor Nurse {..}
38 Actor Clinician {..}
39 UserGroup Accountants {..}
40
41 Task PatientRegistration {
42   Actor Nurse;
43   Form Registration;
44   KPI patientWaitTime;
45   BusinessRules CostReimbursement;}
46
47 BusinessRule CostLimit {
48   // Definition of Business Rule..
49 }
50
51 Scenario PatientAdmission {
52   ContributesTo AdmitPatient;
53
54   Triage ->
55   Admit ->
56   Discharge;}
57
58 Scenario PatientRegistration {..}
59
60 Scenario PatientDischarge { .. }
61
62 KPI PatientWaitTime {
63   // Algorithmic code .. }
64
65 SoftGoal WaitTime {
66   patientWaitTime;}
67
68 SoftGoal ReAdmission {..}

   SoftGoal PatientSatisfaction {
     WaitTime & ReAdmission; }
```

Listing 1. The Patient Admission Example in ROMPL

C. Visual Representation of the System

ROMPL's textual representation is not intended to replace visual representations of the requirements. ROMPL treats both the visual and textual representations as two manifestations of elements of the system under development. In the example system we introduced, there are four modeling notations

involved. These modeling elements are represented textually (as shown above), or can be represented visually (as typically done in the majority of IDEs).

1) Class Diagram Models of the System Entities. ROMPL adopts a notation similar to that of Alf and Umple for defining entities and associations. Lines 3 to 6 define a *one-to-many* relationship between *Patient* and *Registration*. *Registration* is a class of type *Form*. This would instruct the compiler to generate a template form that includes fields for all the class properties. The Registration form has fields of its own (symptom) and fields from another class (i.e., patient age and name). The symptom field is optional, while the patient age and name are mandatory fields.

2) State Machine Model Describing the Behavior of the Registration Forms. Lines 18-29 define three states for the registration form, *Open*, *Closed*, and *Submitted*. The syntax is similar to that of Umple for defining states and transitions. The example in Listing 1 illustrates some transitions between those states. For example, if the form is *Open*, and the event *submit* occurs, a transition to *Submitted* takes place. The guard condition (*complete*) must be true for the transition to take place. The square brackets notation defines a transition guard. In fact, a business rule can also be used as a guard condition, since a business rule must evaluate to true or false. Similarly, if the form is *Submitted*, and the event *reOpen* occurs, then the form becomes in the *Open* state. As the form is being closed, the system saves the Form data. This is represented as an *entry* action into the state *Closed*. ROMPL supports the representation of composite state machines. The notation is similar to that of the Umple.

3) Use Case Models Represented by the Scenario of Patient Admission. The language defines Tasks, which are atomic operations performed by some actors in the system. A task may be associated with one or more KPIs, and one or more Business Rules (Lines 39-43). The example shows a task *PatientAdmission*, performed by an Actor *Nurse*, and involves the completion of the form *Registration*. This task's performance is measured by the KPI *PatientWaitTime* and is subject to the business rule *CostReimbursement*. KPIs and Business Rules are defined algorithmically by imperative code.

A scenario is defined as a sequence of tasks (lines 48-54). A scenario may contribute to one of more goals, and may also be associated with one or more KPIs and Business Rules. The logic behind the separation of tasks and scenarios, rather than combining the two by listing tasks from within a scenario, is to allow the reusing of tasks in multiple scenarios. In simple cases as the one listed above, the scenario is composed of sequential tasks. Nevertheless, there is a need to define flow control elements, such as Forks and Joins. These elements are defined as follows:

```
Scenario PatientAdmission {
  Triage;
    if triage.priority > 5
      Admit || Assess;
    PatientDischarge;}
```

When the patient is triaged, if the priority is greater than 5, then the nurse can start admitting procedures for the patient, while also a clinician can start assessing the patient at the same time. This parallel activities is denoted by the "||" symbol, which represents a *Fork*. Only when both branches are complete, can a clinician start discharging the patient. This represents a *Join*.

4) Goal model of the key objectives and KPIs.

Goals and Soft Goals are explicitly represented in the language. Goals satisfactions are measured by KPIs, which are defined algorithmically using imperative code. Goal compositions are also represented explicitly in the language. For example, the soft goal *PatientSatisfaction* has an AND composition with the two goals, *WaitTime* and *ReAdmission*.

IV. Discussion

The approach presented in this paper weaves requirements abstractions into model-oriented programming languages. This approach provides a platform for interplay between models, code, and requirements. For example, KPIs and Business Rules are defined by utilizing algorithmic code elements. Behavior of requirements entities is defined using state machine abstractions. System entities and relationships are represented using class diagram abstractions.

The approach utilizes two types of abstractions to define behavior. System entities' behavior is better represented using state machine abstractions. This is because it is more natural to think of system entities in terms of their states. While scenario-based behavior is better represented using BPMN-like abstractions. Such abstraction is used to represent scenarios, where the focus is on the flow and sequencing activities and tasks. The two behavior definition notations (state machines and BPMN) are interchangeable [7]. ROMPL handles the subtle differences between the two notations as follows. In ROMPL, state machine abstractions are applied to entities only, while the BPMN-like abstractions are utilized within scenarios to define the flow from one task to the other (as we have shown in the example). However, a user can also define states for any task. For example, the *PatientRegistration* task can be 'Started', '*InProgress'*, or '*Completed'*. This flexibility in the language can be useful in the process of requirements elicitation.

ROMPL represents functional requirements as Goals, while non-functional requirements as Soft Goals. While it may be the common practice to associate KPIs to Soft Goals only, ROMPL allows KPIs to be associated to both Goals, and Soft Goals. This design choice aims at reducing restrictions to support quick system development.

Returning to the needs for traceability, the way the requirements are weaved would allow to verify coverage, avoid redundancy and assess the impact of a change, as these are logically connected. Another benefit of the approach presented in this paper is to support the automatic maintenance of the links between requirements, models, and code. Since both modeling and requirements elements are first class entities in the language, there is no longer need for maintaining requirements links to design and development artifacts. In this

approach, visual models become a side product and are updated automatically as the code evolves. This is similar to the approach adopted by Umple [5]. As traces reside within the code/model these can be easily made visible at run-time if needed. Also, one can query the code/model to view the traces at various levels of abstractions, such as, goals, scenarios, tasks, etc.

MDA advocates for numerous development roles, each is concerned about one aspect of the system. For example, a Business Analysts role need not worry about the implementation of the system, and should focus exclusively on requirements. ROMPL's approach brings different views of the system (requirements, design, and , implementation) together into a single artifact. This can negatively affect modularity, and could mean that a typical user of the language may need to be involved with multiple aspects and views of the system. This is a concern we are considering in the design and development of ROMPL. However, one should acknowledge that in many projects, a single developer may play multiple roles. In addition, the language IDE can support multiple views tailored for the user's role. For example, a requirement view can hide away all implementation and design elements and present only requirements-related entities.

V. Conclusion

Requirements elicitation and analysis are typically performed independently of system development. The need for understanding and analyzing links between requirements entities and their corresponding development entities is well recognized. Maintaining such requirements links is challenging and can be time consuming and error-prone. To address this challenge, we propose the integration of key requirements entities into the development artefacts. Specifically, we introduce ROMPL that weaves key requirements and modeling notations in a way that traces are embedded into the code.

In the future, we plan to further refine ROMPL. Such refinements include revisiting the core elements, assigning action semantics, facilitating complete code and prototype interface generation. We also aim at empirically evaluating its usability and effectiveness in developing and maintaining software systems.

References

[1] Hirschfeld, Robert, Michael Perscheid, and Michael Haupt. "Explicit use-case representation in object-oriented programming languages." *ACM SIGPLAN Notices*. Vol. 47. No. 2. ACM, 2011.

[2] Amyot, Daniel. "Introduction to the user requirements notation: learning by example." *Computer Networks* 42.3 (2003): 285-301.

[3] Buhr, Ray JA, Ron S. Casselman, and Ron Casselman. "Use case maps for object-oriented systems." (1996).

[4] Omar Badreddin, Timothy C. Lethbridge, and Andrew Forward. "Investigation and Evaluation of UML Action Languages". MODELSWARD 2014, International Conference on Model-Driven Engineering and Software Development. 2014.

[5] Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Model oriented programming: an empirical study of comprehension." *CASCON*. 2012.

[6] Derivation of Event-Based State Machines from Business Processes

[7] Marat Abilov, Jorge Marx Gómez. "Derivation of Event-Based State Machines from Business Processes". In proceeding of: International Conference on New Trends in Information and Communication Technologies.

[8] Gotel, Orlena CZ, and Anthony CW Finkelstein. "An analysis of the requirements traceability problem." *Requirements Engineering, 1994., Proceedings of the First International Conference on*. IEEE, 1994.

[9] Object Management Group (OMG). "Concrete Syntax for a UML Action Language RFP", accessed 2012, http://www.omg.org/cgi-bin/doc?ad/2008-9-9.

[10] CoEST: Center of excellence for software traceability, http://www.CoEST.org

[11] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering* (FOSE 2014). ACM, New York, NY, USA, 55-69.

[12] Alexander, Ian, and Ljerka Beus-Dukic. (2009) Discovering requirements: how to specify products and services / Chichester, West Sussex, England ; Hoboken, NJ : John Wiley & Sons.

[13] ITU, User Requirements Notation (URN) – Language Definition, Z. 151, 2012.

[14] Cleland-Huang, Jane, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. "Goal-centric traceability for managing non-functional requirements." *Proceedings of the 27th international conference on Software engineering*. ACM, 2005.

[15] Gotel, Orlena CZ, and Anthony CW Finkelstein. "An analysis of the requirements traceability problem." *Requirements Engineering, 1994., Proceedings of the First International Conference on*. IEEE, 1994.

[16] Hettel, Thomas, Michael Lawley, and Kerry Raymond. "Model synchronisation: Definitions for round-trip engineering." *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2008. 31-45.

[17] Bencomo, Nelly, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. "Requirements reflection: requirements as runtime entities." *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010.

[18] Baudry, Benoit, Clementine Nebut, and Yves Le Traon. "Model-driven engineering for requirements analysis." *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007.

[19] Koch, Nora, and Sergej Kozuruba. "Requirements models as first class entities in model-driven web engineering." *Current Trends in Web Engineering*. Springer Berlin Heidelberg, 2012. 158-169.