Extended OCL for Goal Monitoring

William Robinson
Computer Information Systems Department, Georgia State University
Atlanta, Georgia USA
wrobinson@gsu.edu

Abstract. Monitoring human-computer interaction aids the analysis for understanding how well software meets its purpose. In particular, monitoring human-computer interactions with respect to a user's goal model helps to determine user satisfaction. By formalizing a goal model, runtime monitors can be automatically derived.

The REQMON system monitors the satisfaction of goal models. Recently, an OCL compiler was developed for REQMON. The OCL was extended slightly to address temporal and real-time constraints. Now, goal models can be represented in the extended OCL, from which runtime monitors can be compiled. The resulting REQMON system appears to be easier to use.

1 Introduction

Software systems are becoming increasingly complex. It is difficult to know if we have "built the system right"; that is, the system has been verified to meet its specification. It is also difficult to know if we have "built the right system"; that is, the system meets the user's needs. Software verification has improved with models, formalization, and testing, the latter of which is now creeping into runtime. Similarly, software validation occurs mostly towards the end of development. However, validation during 'natural' usage and continuous validation are gaining importance as developers are under increasingly competitive pressures to evolve software to satisfy changing customer needs. Generally, system behavior monitoring is growing in importance, whether it is for verification or validation.

1.1 Monitoring home healthcare

As an illustration of software monitoring, consider home health care. The population of those requiring personalized healthcare is increasing. Cognitive impairments, for example, are expected to grow substantially over the next decades. These include autism and various forms of dementia, such as Alzheimer's and trauma-induced brain injury. Many of the cognitively impaired (CI) can only function well when assisted in certain activities such as communication, travel, and taking medications. Computer-supported monitoring provides a cost-effective means to assist those requiring personalized health care.

In our recent study, software monitoring was used to assess the satisfaction of clinical goals for a small group of CI patients[6]. As part of cognitive rehabilitation, the patients were given goals of communicating through a very limited and personalized emailing application[21]. Software monitoring was used, in part, to track the clinical goals.

The cognitive rehabilitation field uses a goal attainment scale to evaluate goal satisfaction[20]. Each goal is refined into a set of attainment levels, or milestones, to provide a measure of attainment. The goal of communicating through email can be refined as follows:

- *Level 1* (not attained): will not be able to learn how to use email.
- *Level 2*: can email, but only with lots of prompting and help.
- *Level 3*: can email, with some prompting and help.
- *Level 4*: can email with no prompting and help.
- *Level 5* (fully attained): can teach others how to email

These goal attainment levels can be measured through more refined subgoals, which include the following:

- $G_{presence}$: *The period between viewings of the email in-box shall be no more than k days.*
- G_{read}: *After noticing a new email, a user shall read the email, within k hours.*
- G_{reply}: *After receiving an email, a user shall read and reply to the sender, within k days.*

Clinicians want to see: (1) a good success-to-failure ratio over sessions, and (2) a constant or improving trend of this ratio. This leads to define ratio goals, such as the following:

$G_{reply\text{-}ratio}$: *The ratio of successes vs. attempts for email replies shall be ≥ 75%, with any two-week period.*

Representing and monitoring such goals at runtime is a goal of our research. We have achieved some successes by applying a goal-oriented requirements engineering approach. Using our monitoring system, called REQMON, goals are represented a variant of the UML Object

Constraint Language (OCL)[16] and monitored at runtime.

1.2 Goal-oriented requirements engineering

We use goal modeling to describe and explain behaviors. Other models are useful during monitoring—for example, cost models or diagnostic models. Initially, however, we must describe the goals of the software system, and later explain the software behavior. Goals support description and explanation by providing[24]: criterion for sufficient completeness[29]; criterion for requirements pertinence[29]; rationale—particularly traceability—for requirements[3, 28]; a natural mechanism for structuring complex requirements documents; abstractions for defining alternatives, detecting and resolving conflicts[19]; and a means to drive the identification of supporting requirements[22].

"Goal-oriented requirements engineering (GORE) is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements" [24]. Goal modeling is central to GORE:

> *Goals are prescriptive statements of intent whose satisfaction requires the cooperation of agents (or active components) in the software and its environment. Goals may be organized in AND/OR structures that capture how they are being refined or abstracted. Such structures form the skeleton of goal models; goals there range from high-level, strategic objectives to fine-grained, technical prescriptions that can be assigned as responsibilities of single agents. The latter may be requirements on the software-to-be or expectations on its environment.—[24]*

GORE modeling includes three major phases: (1) identifying goals, (2) refining and formalizing goals, and (3) deriving and assigning operations to agents[25]. Software specifications in various formats—describing agents, their operations, and the data model—can be automatically generated from a GORE model[24].

An analyst can apply refinement techniques to derive operational descriptions, including pre- and post-conditions, from goals[4, 13, 14, 23]. Throughout the process, agent and object models are refined. The resulting specification can be represented in the UML. Table 1 illustrates the correspondence between the GORE elements and the UML elements.

Table 1 Correspondence of GORE elements to the UML elements.

KAOS	UML
Agent	Class
Class	Class
Operation	Operation
Goal	Extended OCL constraints

The research described here shows how certain properties, derived from goals, can be represented within a variant of the OCL. Once represented in the OCL variant, the properties can be monitored at runtime.

2 The monitoring system

A *monitor* is a software system that observes and analyzes the behavior of another (target) system, determining qualities of interest, such as the satisfaction of the target system's requirements. A *monitor* determines the requirements status from a stream of inputs ($\mathbf{IN_{mon}}$). A monitor can be characterized as a function that processes its input data stream to determine the status of requirements.

$$\mathbf{MON(IN_{mon})} \rightarrow \mathit{Sat}(\mathbf{REQ})$$

In practice, the monitored event stream is comprised of complex objects, such as the XML objects produced by event management and logging frameworks, such as Common Base Event (CBE) or log4j.

2.1 Monitoring components

A two-component monitor architecture can be inferred from the preceding characterization. An event listener acquires events from the stream of inputs ($\mathbf{IN_{mon}}$). The requirements analyzer reviews the events to determine requirements satisfaction $\mathit{Sat}(\mathbf{REQ})$. An intervening event repository simplifies event acquisition and analysis. Additionally, a user interface presents the results of analysis.

Figure 1 illustrates the main REQMON components[18]. In the figure, each box is a software component, which may be network distributed; alternatively, the whole system can be deployed as one embedded program. Figure 1 illustrates component interactions that occur when a monitored event is observed. The shaded portions toward the right indicate typical process boundaries; thus, the monitored program and event sink typically comprise one process, the event listener and repository comprise another process, finally the analyzer, presenter and reactor each have their own processes. REQMON defines the components from the event sink through the reactor.

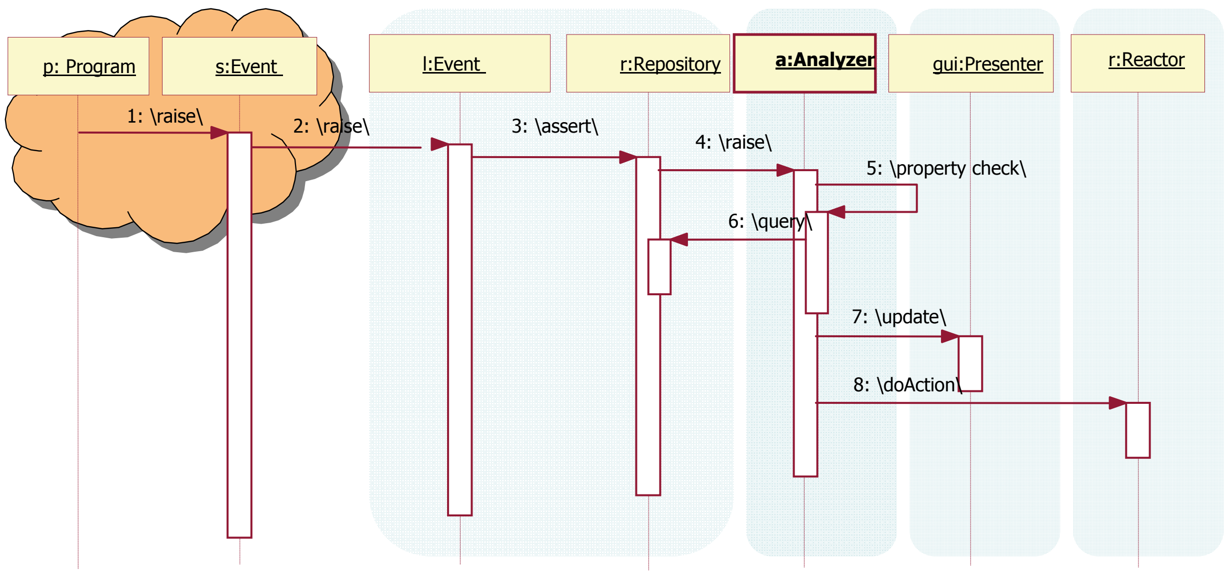

Figure 1 Illustration of ReqMon component interactions.

In this article, we are concerned with the analyzer. That is, we assume that an event stream updates the repository. As events arrive, the analyzer executes in response. The analyzer contains requirements monitors, which are compiled from a variant of OCL 2.0.

2.2 Language considerations

The requirements language has been a consideration for the REQMON monitoring system. The REQMON monitor system is language neutral; it only requires a translator from a requirements language to its evaluation subsystem. (REQMON includes a Jess rule-based evaluation subsystem.) The REQMON research project aims to support GORE. This suggests the following expression needs:

1. Matching and filtering expressions for selecting relevant events from the input stream, $\mathbf{IN_{mon}}$.
2. Object expressions for representing the goal, agent, and object models of GORE.
3. Relational, temporal, and real-time expressions for precisely describing goal, agent, and object models.
4. Meta-requirements for representing expressions about property satisfaction.

Additionally, practical considerations are important:

5. User-defined libraries for extending the language.
6. Well-defined, documented, syntax and semantics that facilitate understanding and external tool development.
7. A large user community, which can apply the monitoring tools.

Initially, the REQMON requirements language was defined with macros over its implementation language (e.g., Jess assertion macros)[6, 17, 18]. This addressed the preceding expression needs 1 – 5; however, considerations 6 and 7 were not met. Therefore, the we explored compilers for other languages, including Tropos[11] and KAOS [26]. Both met considerations 1 - 4. One could argue that the remaining needs were only partially met or described.

REQMON now supports a variant of the OCL 2.0 as a requirements language. The OCL meets needs 1 - 7 with the exception of 3. In particular, temporal and real-time expressions are not native to the OCL. To fulfill those needs, we have provided language extensions.

2.3 OCL temporal assumptions

As a requirements language, REQMON supports a variant of OCL 2.0, which we call OCL_{TM}—meaning OCL with Temporal Message logic. The OCL standard has been extended to include temporal operations based on state[8, 9] and event[2, 12, 30, 31] semantics. Flake[7] addresses temporal expressions over the events of sending and receiving messages. The OCL 2.0 specification allows for the specification of sent messages, but not received messages. Moreover, the syntax can be considered confusing: `^message()` returns true if the message is sent, where as `^^message()` returns a `Sequence` of `OclMessage` objects. Such syntax has led to errors—for example, in the examples of the OCL 2.0 specification itself[7].

Flake's message notation simplifies the message syntax and allows for the specification of received messages. His definitions include `sentMessages` and

```
OclExpression::          Temporal-scoped-pattern
Temporal-scoped-pattern:: [Temporal-scope] Temporal-pattern
Temporal-scope::         after TSE [until] TSE | before TSE | between TSE | global TSE
Temporal-op::            next | prior | eventually | previously | constantly
Temporal pattern::       Temporal-op TE | always TE [unless | until] TE | Response-exp | Precedence-exp
Response-exp::           response [#] [@timeout+] '(' expression (, expression)* ')'
Precedence -exp:         precedence [#] '(' expression (, expression)* ')'
TSE:                     [#] [@timeout] expression
TE:                      [#] expression
Expression::             OclExpression-primitives | '(' OclExpression ')'
```

Dywer specifies synonyms terms, some of which are used here: global is the default scope; absence becomes never, universal becomes always, existence becomes eventually and is the default pattern; Dywer's modifying term chained is unnecessary because response and precedence accept sequences; Dywer's modifying term bounded is addressed by the standard OCL size() operation on collections.

Figure 2 Syntax of extended OCL 2.0 expressions.

`receivedMessages` defined for the general type, `OclAny`.

In OCL_{TM}, we apply Flake's approach to messages[7]. Both `sentMessages` and `receivedMessages` return a `Sequence` of `OclMessage` objects, and `sentMessage` and `receivedMessage` returns the last, most recent message in the sequence.

Over the Flake OCL messages, we apply linear temporal logic semantics. The temporal operators include[3, 15] the following (the OCL_{TM} keyword is bold):

o (the **next** state) ● (the **prior** state)
◊ (some time in the future, **eventually**) ◆ (some time in the past, **previously**)
□ (**always** in the future) ■ (always in the past, **constantly**)
$\mathcal{W}$ (**always** in the future **unless**) $\mathcal{U}$ (**always** in the future **until**)

Using the operations, one can express "eventually class object *obj* will receive message *msg*". In OCL_{TM}, this would be as follows:

```
context Class
  inv:  msgArrives:
         eventually(self.receivedMessage(msg()))
```

2.4 Temporal patterns

Generalized from an empirical study[5], Dwyer *et. al.* defined five temporal scopes over eight temporal patterns.

In all, we collected 555 specifications from at least 35 different sources. The specifications collected were in many forms. … The specifications came from a wide variety of application domains, including: hardware protocols, communication protocols, GUIs, control systems, abstract data types, avionics, operating systems, distributed object systems, and databases. …Of the 555 example specifications we collected, 511 (92%) matched one of our patterns.–Dwyer et. al.*[5]*

Their property patterns include `universal`, `absence`, `existence`, `bounded existence`, `response`, `precedence`, `chained precedence`, and `chained response`. Their scope patterns include `global`, `before R`, `after Q`, `between Q and R`, and `after Q until R`. Distinguishing scoping properties from other properties seems to simplify property specifications through modularization. These patterns have been formalized in linear temporal logic (LTL) and other logics.

OCL_{TM} includes standard temporal operators, the Dwyer patterns, and timeouts. Figure 2 shows these extensions. The expressions extend `OclExpression` of the OCL 2.0 specification [16]. In so doing, these extensions allow for nested expressions.

2.5 Timeouts

Timeouts are associated with scope, as Figure 2 shows. The `@` character precedes a sequence of comma separated timeouts. Thus, `after@0d:3h:0m:0s (Q) P` specifies a timeout that begins with an `after(Q)` scope activation. The scope activation closes early if the timeout occurs before the satisfaction of `P`, the scoped property.

Timeouts are also associated with `response` property sequences. Consider, for example, the expression: `global response@1s,2s (A,B,C)`. The response property has global scope, and is satisfied when properties `A`, `B`, and `C` are satisfied in sequence. Moreover, the two response timeouts specify, in order, that maximum time between property satisfactions: 1 second between `A` and `B`, and 2 seconds between `B` and `C`. If property `A` is satisfied, and subsequently either timeout occurs before the associated property is satisfied, then the whole response property is violated.

2.6 Event and scope sharing

Input data sharing is a monitor design issue[1]. By default, a single event can satisfy multiple properties, unless *no sharing* is specified. Sharing applies to both properties and scopes; by default, a single event, property evaluation, or scope activation can satisfy multiple properties, unless *no sharing* is specified.

As an illustration, consider the following constraints on a Buffer class, where S_i specifies the sharing, with either `#` or nothing.

```
-- Event and property sharing affects matching.
context Buffer
  def: clear : OclMessage =
self.receivedMessage(clear())
  def: addObject : OclMessage =
       receivedMessage(addItem(?: Object))
  def: addTransaction : OclMessage =
       receivedMessage(addItem(?: Transaction))
  inv: eventualObject:
    after S1(clear) eventually S2(addObject)
  inv: eventualTransaction:
    after S3(clear) eventually S4(addTransaction)
```

If *sharing* is allowed by both S_2 and S_4, then a single message (`addItem(? :Transaction)`) will satisfy both `eventually` clauses, because `Transaction` is a subclass of `Object`, and thus both `addObject` and `addTransaction` will be satisfied. Conversely, if either S_2 or S_4 specify *no sharing* (denoted by `#`) then it will require two `addItem` messages to satisfy both `eventually` clauses. Similarly, a single scope activation can be shared by both invariants if S_1 and S_3 specify sharing; conversely, *no sharing* requires two `clear` messages. Although it is possible to specify sharing relationships directly in the property expression (e.g., `addObject <> addTransaction`), like Bates[1], we have found it useful to support sharing directly in the property language.

2.7 Compiling monitors

Compilation is outside the scope of this article. However, a concise overview may be helpful. In short, each monitor specification is compiled into an property evaluation tree, where each node is a rule set[10]. The compiler is written using Antlr 3.0 (a parser generator) and StringTemplate (a template engine). The resulting monitor rules run in Jess 7.0[10].

Consider the following simple property as an illustration.

```
context ContextClasss
  def: m1 : OclMessage =
            self.receivedMessage(message1())
  def: m2 : OclMessage =
            self.receivedMessage(message2())
  inv: prop: after(eventually m1) eventually m2
```

Each temporal expression of `prop` is compiled to one main rule, and possibility some auxiliary rules. A simple compilation of `prop` generates three main rules: (1) *evaluate*`(eventually m1)`, (2) *evaluate*`(eventually m2)` and (3) the root of the tree: *evaluate*`(after(eventually m1) (eventually m2))`.

As events arrive on the input stream, rules evaluate node satisfaction in the property evaluation trees. For example, for the leaf node `eventually m1`, a rule's left-hand-side (LHS) matches a repository assertion representing a received `message1` by an instance of the `ContextClass`; other LHS expressions may further constrain the evaluation. As nodes are satisfied, their values are propagated up the tree, until finally the entire `prop` expression is evaluated.

3 Discussion

Rather than defining our own custom monitoring language we have chosen to (slightly) extend the standard OCL. The language supports typical GORE models. Additionally, specification and tool support for monitoring has been simplified. Moreover, simplified usage has been an unanticipated benefit.

3.1 The OCL for Requirements

To illustrate usage, consider the following expressions, which include two invariants (`readEmail` and `replyEmail`) that represent the G_{read} and G_{reply} goals introduced in section 1.1. Additionally, two meta-goals track the number of their satisfied evaluations during a two week window, represented by `readEmailProp` and `replyEmailProp`. Finally, `goodReadReplyRatio` represents the $G_{reply\text{-}ratio}$ goal of section 1.1. These expressions illustrate how typical GORE goals can be represented in the OCL_{TM}.

```
context EmailClient
  def: eArrival: OclMessage =
            receiveMessage(NewEmail)
  def: eReads: OclMessage =
            receiveMessages(ReadEmail)
  inv: readEmail:
     after@8h(a = eArrival)
     eventually (eReads->select(m |
     m.arguments('ID') = a.arguments('ID'))
            .size() > 0)
  inv: replyEmail:
     response@8h(cSends->select(m |
     m.arguments('ID') = a.arguments('ID'))
            .size() > 0,a = eArrival)
-- evaluations is the collection of all
-- properties, access from  the Property class
  def: readEmailProp:
     Property->evaluations(p| p.name='readEmail'
     and p.satisfied = true
     and (new Date()
     .difference(p.dateTime,DAYS) <= 14))
  def: replyEmailProp:
     Property->evaluations(p|p.name='replyEmail'
     and p.satisfied = true
     and (new Date()
     .difference(p.dateTime,DAYS) <= 14))
  inv: goodReadReplyRatio:
     always((replyEmailProp.size() /
     readEmailProp.size()) >= 0.75)
```

The logical expressions of the OCL_{TM} are similar to other languages that support some form of predicate calculus and temporal logic over an object model (e.g., Tropos[11], KAOS [26]). This kind of model is an improvement over prior REQMON expressions, based on Jess macros (cf. [6]). Moreover, the OCL_{TM} makes use of the OCL library mechanism. Object models can be referenced, including the Java and other runtime models. For example, the preceding `Date` class is defined in the Java runtime, and loaded into REQMON for property evaluation.

3.2 Monitor hierarchies

Goal hierarchies are the core modeling approach in GORE, as introduced in section 1.2. Ideally, the monitoring model mirrors the goal model. Thus, monitored properties should be specified in a hierarchy. This approach is supported in the OCL_{TM} through the standard UML class inheritance mechanism.

Inheritance can simplify the expression of monitors. Consider two classes, where Child is a subclass of Parent. Each class has an associated property as illustrated below.

```
context Parent
  inv: propA: -- ...
context Child
  inv: propB: -- ...
```

Because of inheritance, both child and parent properties apply to child objects. The monitoring system supports such inheritance. Each property is individually compiled to a property evaluation tree. The evaluation system, running as Jess rules, matches objects according to the class hierarchy. Thus, when instances of the child object are observed, then its properties and ancestor properties are evaluated. Inheritance simplifies specification and compilation of requirements monitors.

Analysts benefit from requirements on abstract classes. Subclasses can be checked for requirements compliance, with little additional effort. Of course, subclasses can add specialized requirements, in which case both the abstract and specialized requirements are checked. More generally, a requirements annotated class hierarchy provides a means to describe requirements at multi-levels of abstraction. Thus, hierarchical requirements support the definition, refinement, and analysis of requirements monitors.

3.3 Tool Support

Simplified tool support is a consequence of fulfilling the practical considerations of §2.2. Early development of REQMON required custom tool support for a custom language (c.f. [18]). Although OCL_{TM} is yet another custom language, it is only a slight extension of the standard OCL. Thus, it has been relatively simple to extend existing OCL tools to support it. In particular, we have developed plugins for the Eclipse platform. Currently, three basic plugins types are being developed.

- An editor, which supports OCL_{TM}
- A compiler, which translates OCL_{TM} into a rule-based runtime system
- A pattern library, which supports instantiation and transformations of OCL_{TM} properties

Each of these plugins is based on an existing plugin for the OCL or the UML.

3.4 Simplified usage

REQMON has a small user base (about 10 off-site users). In reviewing our communications over the past few years, we find that since the introduction of an OCL compiler have been a decreasing number of requests for clarification. This supports our practical considerations of §2.2—although, the user base provides little statistical significance. By minimally extending a well-defined, documented, and widely used language, we gained many of the advantages of the language itself (as well as its shortcomings[27].) On balance, we are encouraged that our OCL_{TM} gains from the OCL (semantics and user community). Alternatives, such as KAOS and TROPOS, include temporal semantics, but have smaller user communities, and less development of associated documentation, tutorials, and tools.

Simplified validation is a consequence of simplified usage. Of course, we have not solved the software validation problem. However, use of the OCL has simplified the formal expression of user and system goals, which has simplified the runtime monitoring provided by REQMON.

4 Acknowledgments

This work was supported in part by the National Science Foundation grant CCF-0613698.

References

1. P. C. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *Acm Transactions on Computer Systems,* vol. 13, pp. 1-31, Feb 1995.
2. S. Conrad and K. Turowski, "Temporal OCL: Meeting Specifications Demands for Business Components," *Unified Modeling Language: Systems Analysis, Design, and Development Issues,* pp. 151–165.
3. A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition.," *Science of Computing Programming,* vol. 20, pp. 3-50, 1993.
4. R. Darimont and A. van Lamsweerde, "Formal

Refinement Patterns for Goal-Driven Requirements Elaboration," in *Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, 1996, pp. 179-190.

5. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Twenty-First International Conference on Software Engineering, pages*, Los Angeles, 1999, pp. 411-420.
6. S. Fickas, W. Robinson, and M. Sohlberg, "The Role of Deferred Requirements: A Case Study," in *International Conference on Requirements Engineering (RE'05)*, Paris, France, 2005.
7. S. Flake, "Enhancing the Message Concept of the Object Constraint Language," in *In Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, Banff, Canada, 2004, pp. 161-166.
8. S. Flake and W. Mueller, "An OCL Extension for Real-Time Constraints," in *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. vol. 2263: Springer Berlin / Heidelberg, 2002, pp. 150–171.
9. S. Flake and W. Mueller, "A UML Profile for Real-Time Constraints with the OCL," *Proceedings of the 5th International Conference on The Unified Modeling Language,* pp. 179-195, 2002.
10. E. Friedman-Hill, *Jess in Action*: Manning Publications Co, 2003.
11. E. Kavakli, "Goal-Oriented Requirements Engineering: A Unifying Framework.," *Requirements Engineering Journal,* vol. 6, pp. 237-251, 2000.
12. M. Kyas and F. de Boer, "On message specification in OCL," in *UML 2003 Workshop on Compositional Verification of UML Models*, San Francisco, CA, 2003, pp. 73-93.
13. E. Letier and A. v. Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration," in *Proceedings ICSE'2002 - 24th International Conference on Software Engineering*, Orlando, FL, 2002.
14. E. Letier and A. v. Lamsweerde, "Deriving Operational Software Specifications from System Goals," in *FSE'10 - 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering,*, Charleston, NC, 2002.
15. Z. Manna and A. Prueli, *The Temporal Logic of Reactive and Concurrent Systems*: Springer-Verlag, 1992.
16. O. M. G. OMG, "Object Constraint Language Version 2.0," OMG, Object Management Group May 1 2006.
17. W. N. Robinson, "Implementing Rule-based Monitors within a Framework for Continuous Requirements Monitoring, best paper nominee," in *Hawaii International Conference On System Sciences (HICSS'05)*, Big Island, Hawaii, USA, 2005.
18. W. N. Robinson, "A requirements monitoring framework for enterprise systems," in *Requirements Engineering Journal*. vol. 11: Springer, 2006, pp. 17-41.
19. W. N. Robinson, S. Pawlowski, and V. Volkov, "Requirements Interaction Management," *ACM Computing Surveys (CSUR),* vol. 35, pp. 132 - 190, June 2003.
20. M. M. Sohlberg and C. A. Mateer, *Cognitive rehabilitation: An integrated neuropsychological approach*. New York: Guilford Publication, 2001.
21. A. Sutcliffe, S. Fickas, and M. M. Sohlberg, "Personal and Contextual Requirements Engineering," *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on,* pp. 19-30, 2005.
22. A. van Lamsweerde, "From System Goals to Software Architecture," Springer, 2003, pp. 25–43.
23. A. van Lamsweerde, "From System Goals to Software Architecture," *Formal Methods for Software Architectures,* pp. 25–43, 2003.
24. A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice," 2004, pp. 4-8.
25. A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt," in *IEEE, Second International Symposium on Requirements Engineering*, 1995, pp. pp. 194-203.
26. A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," in *IEEE Transactions on Software Engineering*. vol. 26, 2000, pp. 978-1005.
27. M. Vaziri and D. Jackson, "Some Shortcomings of OCL, the Object Constraint Language of UML," *TOOLS: 34th International Conference,* pp. 555–560.
28. E. S. K. Yu, "Modeling organizations for information systems requirements engineering," in *roceedings of IEEE International Symposium on Requirements Engineering, 1993*, San Diego, CA, USA, 1993, pp. 34-41.
29. K. Yue, "What does it mean to say that a specification is complete?," in *4th International workshop on software specification and design*, Montery,CA, 1987, pp. 42-51.
30. P. Ziemann and M. Gogolla, "An Extension of OCL with Temporal Logic," *Critical Systems Development with UML,* pp. 53–62.
31. P. Ziemann and M. Gogolla, "OCL Extended with Temporal Logic," *Perspective of System Informatics*.