18th ECOOP Doctoral Symposium and PhD Student Workshop

Part of ECOOP'08

Paphos, Cyprus

July 8, 2008

Table of Contents

Symposium Organization

Program Chair

Mark Hills, University of Illinois at Urbana-Champaign

Program Committee

Marwan Abi-Antoun, Carnegie Mellon University
Eric Bodden, McGill University
Giovanni Falcone, Universität Mannheim
Mark Hills, University of Illinois at Urbana-Champaign
Haidar Jabbar, Anna University
Ciera Jaspan, Carnegie Mellon University
Romain Robbes, Università della Svizzera italiana
Ilie Savga, Technische Universität Dresden
Michel Soares, Technische Universiteit Delft

Academic Panel

Jonathan Aldrich, Carnegie Mellon University
Erik Ernst, University of Aarhus
Todd Millstein, University of California, Los Angeles
James Noble, Victoria University of Wellington
Jeremy Siek, University of Colorado at Boulder

Additional Reviewers

Donna Malayeri, Carnegie Mellon University
Martin Robillard, McGill University
Volker Stolz, United Nations University

Acknowledgements

We would like to thank AITO for its generous financial support of the students presenting at this year's Doctoral Symposium and PhD Students Workshop. We would also like to thank the organizers of ECOOP 2008 for their logistical support and for providing a web home for this year's event.

Language Features, Patterns, and Models for Interactive Software

Brian Chin

University of California, Los Angeles
naerbnic@cs.ucla.edu

Abstract. Interactive programs, those which continually consume input and produce output, in contrast to the classic batch-processing method, have become increasingly common. Yet most modern programming languages are derived from older ones, which were designed to program towards a classic model. While well understood, programming interactive software in these languages tends to force the programmer to manually manage things which the language would otherwise handle automatically. My dissertation research attempts to find new language features, patterns, and programming models to aid the development of interactive software.

1 Problem Description

A variety of application domains are *interactive* in nature: These programs continuously take in a stream of input, and produce a stream of output in return. These programs do not necessarily have an obvious start and end condition. In fact, they may intentionally not halt at all. I call these sorts of programs examples of *interactive software*. Common examples of these would be computer games, programs with user interfaces, servers and web applications.

However common interactive programs are, they are generally coded in languages which are derived from the classic batch-processing model of computation. That model is exemplified by programs which take in one set of data at program initialization and produce a set of output at program completion. At the core of this model is procedural abstraction. Procedures also take a set of input at the beginning (as arguments) and return output at the end (the return values). Most current languages uses some form of procedure as the primitive unit of abstraction within a program.

Procedures are an excellent way of hiding complexity from the programmer. Local variables allow a programmer to use an arbitrary amount of non-persistent data without having to allocate or deallocate resources for them. They are guaranteed not to change unpredictably even when nested procedures are called (modulo taking addresses of variables, e.g. C). Local program state, in the form of the current program counter within a procedure, is also maintained automatically; a procedure can be called within any control structure, and the current state of the execution is preserved.

When implementing interactive software in procedural languages, the previous advantages are lost. Often such software is driven by a main loop, which gathers data from the input sources (e.g. mice, keyboards, network, etc.) and passes this data to the program to be processed. The fact that the main loop exists implies that the main logic of the interactive program (contained in the input handling code) must yield control back to the main loop each time new input is desired. This alone removes many of the mentioned advantages of procedural abstraction. State stored in local variables is not preserved, and thus must be manually stored. Nested logic, such as code which would be in a while loop, would have to have its logic extracted from familiar control structures, and manually implemented so that the logic can be re-entered "in the middle." The programs are more complicated than necessary, and more effort is required on the part of the programmer.

The programming community has noticed this problem, and various mechanisms have been developed to manage the complexity of such systems. For instance, the state design pattern [4] is a standard object-oriented pattern for interactive software. It forms a type of state machine, where each state is implemented using a class. It provides a common interface for all of the states, and makes it easier to add new states in the future. A state machine object keeps a reference to the current state, providing a place for future computation to continue from. Despite these nice features, it does not change the fact that all of the same issues mentioned previously must still be handled manually by the programmer.

2 Goals

My core goal, first and foremost, is to make programming interactive software easier. This can be done in a number of ways, including adding new features to existing languages, finding new patterns in existing languages, and creating entirely new languages. There are a several properties that would be desired in any such approach:

1. *Simplicity*: A new approach should be simple and easy for a programmer to use and read, and further minimize their workload.
2. *Modularity*: Normal procedural logic allows for complicated pieces of logic to be separated into smaller pieces. A new approach should provide some way of similarly dividing interactive logic into smaller pieces.
3. *Extensibility*: A new approach should provide a way to reuse old code to implement new code.
4. *Compatibility*: A new approach, if interfacing with an existing language, must not interfere with its model of computation. Thus there should exist a mechanism for the extension to interact with the language as a whole.
5. *Minimality*: Any change to a language causes additional difficulty for a programmer, because they have to re-learn a portion of a language which they already understand. Thus new approaches should minimize the necessary changes to any underlying language.

Clearly, not all of the above properties are achievable at the same time, but any approach should try to provide as many of them as possible.

3 Approaches

I have investigated three different approaches to improving programming languages for interactive applications. The first is an extension to Java which adds new language constructs to allow logic to be executed and paused separate from the main program. The second describes an extension of the state design pattern which in exchange for adding additional constraints provides several useful sorts of extensibility. I have developed and published papers on these approaches. The third and final approach is a new language which utilizes new observations I have made on the nature of interactive software.

3.1 ResponderJ

The first approach is to modify existing languages (such as Java). As a brand new language feature changes to Java are necessary, but I still need to ensure the language feature is compatible with Java. My solution is an extension of Java called ResponderJ [2], which adds a new language feature called responders. Responders allow for objects to maintain the state of a piece of interactive logic, along with their normal behavior. Responder code is written in a natural procedural style. When necessary, its execution can be temporarily paused and resumed later, passing control back to the main program in the interim. The resource management in between is managed by the language, introducing some of the conveniences of procedural programs to interactive ones. I adapted this model to the Java object model, thus making responder code compatible with vanilla Java code.

3.2 The Extensible State Machine Pattern

The second approach is to find solutions in existing languages using only well-understood language features to approach the same problem. This requires more work on the part of the programmer than ResponderJ, but by creating a general approach which assumes minimal changes to the language, I hope to reduce the mental strain on the programmer. My solution here was a modification of the state design pattern that added a number of types of extensibility without adding any language features [1]. It does this by placing a few coding constraints on state machines, which allows my extensibility mechanisms to be used reliably. With the addition of delimited continuations, an existing language feature in the functional literature [3], yet another form of extensibility was added, making the language capable of supporting many of the features of ResponderJ.

3.3 Interaction State Diagrams

My final approach is a full change of model. If procedural languages are ideal for classical programs, then similarly there may be a model of execution and abstraction which will provide many of the same advantages to interactive languages. I have been exploring this particular topic, with special focus on a mechanism for interfacing interactive components together which I have named *interaction state diagrams*. These provide a mechanism for complicated pieces of interactive logic to communicate in a regular, predictable way, while also indirectly providing a natural documentation for the communication. This interface is impressively flexible, allowing for both synchronous and asynchronous communication with minimal additional architecture needed to support it.

This interface method also hints at ideas for a new language model, which I intend to explore. For example, unlike method calls, protocols defined by interaction state diagrams are not inherently nested, thus would operate best in a language where such nesting is unnecessary. I am considering a number of techniques to implement such a language.

3.4 Evaluation

I have evaluated ResponderJ and my extension to the state design pattern by implementing several pieces of example code using the respective approach, and comparing the new code against the old method. I also altered a portion of the XML parsing library JDOM [5] to use each of these approaches, demonstrating their utility. As for the interaction state diagrams, I have shown a few real-world examples where such diagrams can describe complicated interaction which would be difficult to show using existing techniques. The diagrams themselves must follow a small number of rules to avoid invalid behavior. I will want to ensure these rules guarantee that such behavior is avoided.

The evaluation of this sort of project is difficult. Ease of use is based around programmer preference which can vary wildly from person to person. I hope to show the community that my ideas are themselves easy to understand, and thus would be easy to implement and use for future projects. I can also perform user studies in the future to evaluate exactly how much benefit programmers gain from these approaches.

References

1. B. Chin and T. Millstein. An extensible state design pattern for interactive applications. In *Proceedings of European Conference on Object-Oriented Programming*, July 2008.
2. B. Chin and T. D. Millstein. Responders: Language support for interactive applications. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 255–278. Springer, 2006.
3. M. Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190, 1988.
4. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.
5. JDOM home page. http://www.jdom.org.

Concern-Sensitive Heuristic Assessment of Aspect-Oriented Design

Eduardo Figueiredo and Alessandro Garcia (Supervisor)

Computing Department, Lancaster University,
InfoLab21, South Drive, United Kingdom
e.figueiredo@lancaster.ac.uk, garciaa@comp.lancs.ac.uk

Abstract. Recent empirical studies of aspect-oriented design have stressed that the inaccurate modularisation of some concerns potentially leads to a plethora of non-obvious modularity flaws. Nowadays, modularity assessment is mostly supported by design heuristics rooted at conventional attributes such as module coupling, module cohesion, and interface complexity. However, such traditional module-driven assessments cannot be tailored to the driving design concerns, thereby leading to recurring false positives and false negatives in design evaluation processes. Our goal is to promote concerns as explicit abstraction in the design assessment process. We propose an assessment technique composed of (i) a concern-oriented measurement framework to support the instantiation and comparison of concern metrics, (ii) a set of concern metrics instantiated and formalised according to our measurement framework, and (iii) a representative suite of concern-sensitive heuristic rules for detection of design flaws. To evaluate our concern-oriented assessment technique, we are conducting a number of empirical studies which encompass a plethora of crosscutting and non-crosscutting concerns.

1 Problem Description

With the emergence of Aspect-Oriented Software Development (AOSD) [9], there is an increasing awareness that some system concerns might be the key factors to the deterioration of system modularity and the main cause of faults [2, 6]. A concern is any consideration that can impact the implementation of a program [11]. AOSD aims to enhance design modularisation through new composition mechanisms, such as pointcut-advice and inter-type declarations [9]. Aspects are new units of modularity for encapsulating crosscutting concerns, i.e. system properties that naturally affect many system modules.

However, the achievement of modular aspectual designs is far from being trivial. Recent studies have shown that inaccurate separation of certain concerns with aspects leads to multiple concern-specific flaws [3, 8]. Even the “aspectisation” of conventional crosscutting concerns, such as exception handling [1, 6], concurrency control [8], and the Observer design pattern [8], might impose negative effects on the system modularity, including (i) increase on the number of undesirable concern couplings [3], and (ii) decrease on the cohesion of modules realising a certain concern

[8, 12]. These concern modularity flaws make the change and removal of the target concerns error prone [1, 6] and lead to the manifestation of ripple effects [3, 8].

The recognition that concern identification and analysis are important through software design activities is not new. In fact, there is a growing body of relevant work in the software engineering literature focusing either on concern representation and identification techniques [2, 11, 12] or on concern analysis tools [5, 11]. However, there is not much knowledge on the efficacy of concern-driven assessment mechanisms for design modularity and error proneness. Even though some works have started to define concern metrics [2, 4, 11, 12], there is a lack of design heuristics to support concern-sensitive assessment.

Design heuristics are metrics-based rules that capture deviations from good design principles [10]. Concern heuristics would lead to a shift in the assessment process: instead of quantifying properties of a particular module, they would quantify properties of one or multiple concerns with respect to the underlying modular structure. To the best of our knowledge, there is no systematic study that investigates whether this category of heuristic rules enhances the process of (i) evaluating design modularity attributes, (ii) identifying error-prone realisations of design concerns, and (iii) detecting design anomalies, such as bad smells [7]. A bad smell is any symptom that indicates something may be wrong and it generally indicates that overall design should be re-examined [7].

In fact, the area of concern-oriented evaluation is still in its infancy and it also suffers from not subsuming to a unified terminology and formalisation of concern measurement. Even the terminology used in existing definitions of concern metrics is diverse and ambiguous by nature [4]. Hence, it is not straightforward the elaboration of concern heuristics relying on those poorly defined metrics. For instance, it is not clear in definitions of concern metrics the granularity of artefacts, ranging from architecture [12] to implementation-specific artefacts [2, 4], and the target concern modularity property (e.g., coupling, cohesion, or tangling). The terminology of concern relies on the jargon of specific research groups, thereby hampering: (i) the process of instantiating, comparing, and theoretically validating concern metrics, (ii) their adoption in academic and industry settings, (iii) independent interpretation of the measurement results, (iv) ways of composing concern metrics to systematically boost heuristic rules, and (v) replication of empirical studies using concern metrics.

2 Goal Statement and Expected Contributions

Our main goals are *(i) to promote concerns as explicit abstraction in the heuristic design assessment*, *(ii) to provide formal definitions and automated support for concern-based heuristic rules and their composing metrics*, and (iii) *to empirically assess the efficacy of the concern-sensitive heuristics when compared with conventional ones*. In the context of the previously described problems, the expected contributions of our PhD research are summarised as follows.

1. A survey and critical review of (i) existing measurement frameworks, (ii) existing concern-oriented metrics, and (iii) metric-based heuristics rooted at conventional modularity attributes, such as coupling and cohesion.

2. A concern-oriented measurement framework [4] to support the instantiation and comparison of concern metrics.
3. A set of concern metrics formalized according to our measurement framework and empirically evaluated whether they are useful to detect design flaws (and error-prone code).
4. A representative suite of concern-sensitive heuristic rules for (i) assessing the overall modularity of design-driven concerns and (ii) detection of specific design flaws including well known bad smells [7].
5. A fully implemented and documented tool for concern-oriented design assessment to support all elements of our heuristic assessment technique.
6. A complementary set of empirical studies to evaluate different usefulness and usability facets of our concern-oriented heuristic technique for assessment of design modularity and error proneness, including:
    a. Evaluation of the measurement framework's generality through the instantiation of concern metrics and their application in empirical studies.
    b. Systematic investigation on the accuracy of the concern-sensitive heuristics in statistically relevant studies encompassing heterogeneous forms of crosscutting and non-crosscutting concerns.
    c. Quantitative and qualitative assessment of the positive and negative impact of aspect-oriented composition mechanisms and crosscutting concerns on evolving applications.

3 The Proposed Solution

Concern-oriented measurement framework. In order to address the formalisation of concern metrics, our PhD research proposes a concern-oriented measurement framework that supports the instantiation and comparison of concern metrics. The proposed framework [4] subsumes a unified concern terminology and criteria in order to lay down a rigorous framework to foster the definition of meaningful and well-founded concern metrics. In the definition of the measurement framework, we undertook an extensive survey of the state-of-the-art on concern measurement [4].

Concern-oriented metrics and heurists. In addition to a concern measurement framework, we are revisiting existing metrics-based heuristic rules and proposing innovative concern-sensitive design heuristics based on a suite of concern metrics. The proposed design metrics and heuristics have the distinguishing characteristic of exploiting concerns as explicit abstractions in the design assessment process. The goal of concern metrics is the association of quantification with concern properties. Besides, concern-sensitive heuristics targeted at enhancing the modularity assessment process by detecting two overlapping categories of modularity flaws, namely crosscutting concerns and classical bad smells [7]. Furthermore, we are investigating concern-sensitive heuristics as mechanisms to support the detection of (i) faults related to crosscutting concerns [1] and (ii) refactoring opportunities [13].

Automated support. To effectively employ our concern measurement approach, it is imperative to provide automated support to reduce the burden on identifying and representing design concerns as well as applying concern metrics and heuristic rules.

An initial architecture model and a prototype tool [5] to support the proposed assessment technique have already been defined. The final implementation of our tool might provide the following functionality: (i) concern representation in a hierarchical model (*Concern Model*), (ii) means to project concerns onto artefacts of detailed design and implementation (*Concern Manager*), (iii) a reasonable set of concern metrics instantiated by our framework (*Metric Collector*), and (iv) an extensible set of concern-sensitive heuristics (*Rule Analyser*).

Empirical evaluation. To evaluate the generality of the proposed concern measurement framework, in the first year of our research we demonstrated the framework instantiation and extension of a number of concern metrics [4]. We also discussed how the proposed measurement framework can help to point out limitations on the used metrics, and assist the planning of new experimental replications. Now (second year), this Doctoral research is focusing on a statistically relevant evaluation on the accuracy of concern-sensitive heuristics in a number of empirical studies. We have already selected eight representative applications [3, 6, 8] which encompass heterogeneous forms of crosscutting and non-crosscutting concerns. For example, in one application [3] we target at systematically verifying the suitability of aspect-oriented composition mechanisms for designing stable and modular software product-line designs. Software product lines [3] represent a common and important technology to support the derivation of a wide range of applications. In the last year of our research, we expected that the collected data might be relevant to draw conclusions regarding the usefulness of concern-sensitive heuristic assessment to detect design anomalies.

References

1. Cacho, N., Filho, F., Garcia, A., Figueiredo, E.: EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In: Int'l Conf. on Aspect-Oriented Soft. Development (AOSD). Brussels (2008).
2. Eaddy, M. et al.: Do Crosscutting Concerns Cause Defects? IEEE Trans. on Soft. Eng., 2008 (to appear).
3. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: International Conference on Software Engineering (ICSE). Leipzig (2008).
4. Figueiredo, E., Sant'Anna, C., Garcia, A., Bartolomei, T., Cazzola, W., Marchetto, A.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In: 12th European Conference on Software Maintenance and Reengineering (CSMR). Athens, Greece (2008).
5. Figueiredo, E., Garcia, A., Lucena, C.: AJATO: an AspectJ Assessment Tool. In: European Conference on Object-Oriented Programming (ECOOP), demo section. Nantes, France (2006).
6. Filho, F., Cacho, N., Figueiredo, E., Maranhao, R., Garcia, A., Rubira, C.: Exceptions and Aspects: The Devil is in the Details. In: Int'l Symposium on Foundations of Software Engineering (FSE), (2006).
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, US, 1999.
8. Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP Conference. Berlin (2007).
9. Kiczales, G. et al.: Aspect-Oriented Programming. In: ECOOP Conference, p. 220-242. Finland (1997).
10. Marinescu, R.: Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In: International Conference on Software Maintenance (ICSM), p. 350-359, Chicago (2004).
11. Robillard, M., Murphy, G.: Representing Concerns in Source Code. ACM TOSEM, 16, 1, (2007).
12. Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C.: On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. In: European Conf. on Soft. Architecture (ECSA), (2007).
13. Silva, B., Figueiredo, E., Garcia, A., Nunes, D.: Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics. In: Int'l Workshop on Software Quality and Maintainability (SQM). Athens (2008).

A Metadata-Based Components Model

Eduardo Martins Guerra[1] and Clovis Torres Fernandes[1] (advisor)

[1] Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acácias - CEP 12.228-900 – São José dos Campos – SP, Brazil
guerraem@gmail.com, clovistf@uol.com.br

Abstract. Metadata-based component is a component or a framework that processes its logic based on the metadata of the class whose instance it is working with. Many frameworks use this approach to increase the flexibility and the reuse in applications. But all those frameworks are created by the experience of the developers, because there are not documented techniques or best practices for the development of this kind of component. This work proposes the development of a model for metadata-based components, to ease the development of flexible software components and increase the application reuse degree.

1 Problem Description

There are many ways for a component or a framework to be reusable and adaptable. Specialization of the component classes and implementation of its interfaces, allows the application to extend the behavior and to adapt it to its needs [1]. For instance, the use of dependency injection, allows the client to insert the instances that composes the component [2], changing the behavior based on the class and information of the instance inserted. Combining these techniques, the developer combines composition and inheritance to enable flexibility and reuse in the component. These techniques are used in design patterns to document common solutions for some recurrent problems in object-oriented applications [3].

The attribute-oriented programming is a program-level marking technique that allow developers to mark programming elements (e.g. classes and methods) to indicate application-specific or domain-specific semantics [4]. Metadata-based components can be defined as components that use class metadata in runtime to process its logic. The metadata can be stored not only using annotations, but also in external files (usually as XML documents), databases or programmatically. The metadata can also be configured implicitly using name conventions [5].

Metadata-based components and attribute-oriented programming had some intersection but they are not the same thing. In metadata-based components the metadata can be stored in other ways and in attribute-based programming the use of the attributes can also be done to generate code or modify classes [6].

The following are examples of metadata-based frameworks and APIs: Hibernate [7] and JPA [8] uses metadata to create an object-relational mapping and to manage events in different phases of an instance persistence process; EJB [8] uses metadata to configure how the container is going to control some non-functional features, like transaction strategy and access control policy; JUnit [9] identifies test and life cycle methods of a test class by metadata.

The use of metadata allows the component to use different rules for objects of different classes. This way, the component can replace behaviors usually implemented manually in the application. This increases the application's amount of reuse and allows the component to be used in different situations. The use of metadata can save a great amount of hand-written program code without losing the semantics of the application [10].

All existent metadata-based components are created based only on the experience of its developers, because there are not documented techniques or best practices for dealing with metadata. This reduces the opportunities of using metadata in situations that it is appropriate.

2 Goal Statement

The thesis statement of this proposal is: "Create a metadata-based components model, to ease the development of flexible software components and increase the application amount of reuse". This model must be a reference for solving design problems about how to deal with metadata in this kind of component. The model must be also the basis for other studies about metadata-based components.

This work investigates the following key questions about the metadata-based components: (1) What are the characteristics that metadata-based components had in common?; (2) What kind of problems can be addressed using metadata-based components?; (3) What are the design requirements that may appear in this kind of components because of the use of metadata?; (4) How are the existent metadata-based components designed? How do they deal with class metadata?; (5) How much amount of reuse can an application get using a metadata-based component?; (6) How much easier and faster is to develop an application using a metadata-based component?

3 Approach

To investigate about metadata-based components, seeking for an abstraction of its characteristics and concepts, three different strategies are being followed, namely:

1. The existent metadata-based components will be analyzed, looking for what problem is solved with metadata, how it is designed and how it is internally structured. The abstraction of those solutions will lead the research for suitable patterns for solve metadata problems.
2. There are two open source metadata-based frameworks, SwingBean [11] and Esfinge [12], developed by the author of this proposal. They are currently used by some large and medium scale applications in production. There is not metrics to measure the result of their use in practice, but qualitatively can be affirmed that there was an improvement in productivity and flexibility. Those frameworks will be refactored to study the implementation of the patterns found and to analyze how their use influence the application and framework flexibility.

3. Other minor case studies of metadata-based components are also going to be developed. These case studies are intended to explore the concepts of metadata-based components and to experiment the patterns found in different situations.

Using the knowledge obtained by these tree different strategies, some design patterns for application in metadata-based components will be identified and documented. These patterns are going to be presented as a pattern language that addresses the most important aspects about dealing with metadata in components. The structure of this pattern language will address patterns for metadata creation and management, logic processing based on metadata and for metadata use applicability. This pattern language is considered the main contribution of this work.

To formalize the identified structural characteristics of this kind of component, the internal architecture will be documented using an Architecture Description Language. The objective is to create a formal description of the internal components and their relationship.

4 Validation

The work will be validated both quantitatively and qualitatively. The qualitative validation will occur during the research, comparing the patterns and the model to existent frameworks and using them in case studies. The case studies will validate if the proposed patterns fulfils some flexibility requirements about metadata manipulation. To validate that the application of the proposed model in a component really improve the reuse degree of the software that uses it, an experiment will be accordingly defined.

In this experiment, the same problem will be addressed using tree kind of approaches: (a) without frameworks; (b) with a framework without the use of metadata (traditional framework) and (c) with a metadata-based framework. The goal of the experiment is to have different solutions with the same interface and behavior developed in a controlled environment, which can be used to do a comparative study. There will be at least tree teams, each one with a different problem. The experiment will use groups of undergraduate students of the object orientation course. A larger-scale experiment would be better, but that would not be really practical. The following phases will be considered:

- **Phase 1:** The teams have to solve the problem without using frameworks. There will be an interface that they must use in the implementation. Some automated integration tests will be developed to ensure that the behavior is correct.
- **Phase 2:** The teams have to develop a traditional framework to help someone to solve similar problems that they solve in Phase 1. The documentation for this framework must be created.
- **Phase 3:** The teams have to exchange the problems and implement the same problem of Phase 1 with the framework developed in Phase 2. The same tests created in Phase 1 must execute successfully.
- **Phase 4:** The team that developed the framework in Phase 2 has to evolve it by using the metadata-based approach. The metadata-based component model developed must be used. The team must suitably document the framework.

- **Phase 5:** A new team that did not already dealt with the problem have to implement it using the metadata-based component created in Phase 4. The same tests created in Phase 1 must execute successfully.

A comparative study will be made based on the solutions created by the students in the experiment. Some metrics will be used to analyze quantitatively each approach. Examples of metrics are amount of reuse [13] and dynamic coupling [14]. The students will also answer some questionnaires for a qualitative analysis of the experiment. It will address issues about the application and framework development, like the difficulty to understand the API and to develop the solution.

The main goal of the quantitative study is to show that applications that use metadata-based components created using the proposed model have a greater amount of reuse compared to those created with classical frameworks techniques. The qualitative study should also detect that the implementation with metadata-based components is more productive and easier than using traditional frameworks.

References

1. Fayad, M. E., Schimidt, D. C. and Johnson, R. Building application frameworks: Object-oriented foundations of framework design. John Wiley & Sons, 1999.
2. Fowler, M. Inversion of Control Containers and the Dependency Injection pattern. Available on http://www.martinfowler.com/articles/injection.html, 2004.
3. Gamma, E. Helm, R. Johnson, R. Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1994.
4. Wada, H.; Suzuki, J. Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Sytems (MoDELS/UML 2005), 2005.
5. DOV, A. B. Convention vs. Configuration. Available on http://www.javalobby.org/java/forums/t65305.html, 2006.
6. Schwarz, D. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5, In onjava.com, O'Reilly Media, Inc., Junho 2004.
7. Bauer, Christian and King, Gavin. Hibernate in Action. Manning Publications, 2004.
8. JSR 220. JSR 220: Enterprise JavaBeans 3.0. Available on http://www.jcp.org/en/jsr/detail?id=220, 2006.
9. JUnit, Available on http://www.junit.org/, 2008.
10. Rouvoy, R. Pessemier, N. Pawlak, R. e Merle, P. Using attribute-oriented programming to leverage fractal-based developments. In Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06), Nantes, France, Julho 2006.
11. SwingBean, Available on http://swingbean.sourceforge.net/, 2008.
12. Esfinge, Available on http://esfinge.sourceforge.net/, 2008.
13. W. Frakes and C. Terry, Software Reuse: Metrics and Models, ACM Computing Surveys, vol. 28, 1996. http://citeseer.ist.psu.edu/frakes96software.html
14. Arisholm, E.; Briand, L.C.; Foyen, A. Dynamic coupling measurement for object-oriented software, IEEE Transactions on Software Engineering, Volume 30, Issue 8, Pages 491 – 506, Aug. 2004.

Formalising Dynamic Languages

Alex Holkner and James Harland

RMIT University
Melbourne, Australia
{alexander.holkner,james.harland}@rmit.edu.au

Abstract. Relatively new dynamic languages such as Python, Ruby and ECMAScript have become popular for rapid prototyping and general-purpose development. These languages share many similarities but are developed independently of each other, and little is known about the extent to which their object models are compatible. We already know that the object models differ greatly from those used in traditional languages such as C++ and Java. They also differ enough from the formal theory of object-oriented languages that makes formal analysis of programs written in these languages next to impossible. We propose extending the existing object-oriented formalisms to cope with the dynamic features used in these languages, as well as delimiting the extent to which these features are not compatible with current analysis techniques.

1 Problem

Functional languages such as Scheme and Haskell are underpinned by formal theoretical frameworks that describe their operation [1–3]. Formalisations of languages are useful for analysing the complexity and equivalence of programs in those languages; making judgements about the security, safety, and correctness of programs; comparing the relative expressiveness or power of languages; making guarantees about the correctness of various program transformations; and so on.

While some progress has been made towards formalizing Java [4, 5], to date most theoretical work on object-oriented languages has proceeded without encoding any particular language, instead focussing on formal type theories of an abstracted "ideal" object-based language.

This has resulted in two related areas of research remaining unexplored. Firstly, existing formalizations are unsuitable for use as a base calculus for developing a virtual machine for executing object-oriented programs, due to their use of constructs not seen in existing languages. Secondly, some features of modern dynamic object-oriented languages, such as changing the class of an object, or replacing a method on a class, cannot be encoded in the existing formalisations (without significantly complicating the encoding of a simple object).

For example, Python [6] appears to be a suitable language to be modelled by the ς-calculus [7]. Objects are simple "dictionary" mappings of names (labels) to attributes (methods). Classes are merely another type of object, encoded in

```
class P(object):
    m = 0

# Create an object ’p’, an instance of ’P’
p = P()

# Prints ’0’
print p.m

# Mutate the ’P’ class
P.m = 1

# Prints ’1’
print p.m
```

Fig. 1: Python code demonstrating global update

the ς-calculus as a collection of “pre-methods”. However, the ς-calculus fails to capture the side effect of mutating a class after objects have already been instantiated from it.

Figure 1 lists a simple program written in Python that demonstrates this. Accessing `p.m` after mutating `p`’s class `P` returns the mutated attribute value.

It is clear from the observed behaviour (and the language specification) that the `p` object delegates to the `P` class for the `m` attribute. The attribute is not simply copied onto the `p` object when it is created, as in the class encoding suggested by Abadi and Cardelli [7].

Python is not a unique language in this respect: several modern dynamic programming languages including Ruby [8] and ECMAScript (JavaScript) [9], and older languages such as Smalltalk [10] and Self [11], exhibit this “global update” behaviour.

2 Goal

We will develop one or more formal object calculi that describe the behaviour of modern dynamic object-oriented languages such as Python. We expect to achieve several outcomes:

Formal description and comparison of the dynamic languages. It is generally understood that the object models of Python, Ruby, Smalltalk and other languages are quite similar. No literature exists that comprehensively documents and compares the object models, irrespective of the other language features.

Reduction of the dynamic object model into existing object formalisms. We would like to find out to what extent the dynamic languages can be encoded simply into the object formalisms such as the ς-calculus. We already know that some features, such as global update, must be dropped; but some core amount of the languages' object models should be representable.

Extension of an object formalism to support dynamic object features. We would like to discover new formalisms that describe as many features of the dynamic languages as possible, without requiring complex or indirect encodings, yet while still remaining susceptible to analysis.

An overriding theme in this research is that the current generation of dynamic languages have an "ad-hoc" object model that is next to impossible to reason about, short of executing the code. Besides the global update problem described earlier, these languages allow objects to completely override the default message dispatch mechanism. This can introduce arbitrary side-effects into a program that appears on the surface to evaluate something as simple as an attribute lookup.

Our goal, then, is to find these areas of difficulty where the languages become "too dynamic" for analysis, and also to extend existing object formalisms to describe as much as possible about the languages without reducing their expressiveness.

3 Approach

The development of a formal system that caters to dynamic languages requires an incremental approach. We intend to extend Abadi and Cardelli's typed object calculus [7] with features for implicit and explicit delegation, mode switching and overridable delegation. We have already developed a functional delegating object calculus that handles the global update problem, and proven the subject reduction theorem for a first-order type system in the calculus.

After prototyping these features in a functional setting, we will formalise them for imperative use, which is necessary if we are to realistically model the current generation languages.

Our other goals, describing the dynamic language object model and reducing it into existing formalisations, can be achieved in concert with this development. We expect that the findings in one calculus will reveal subtle points of difference in others.

We will convince ourselves and others of the correctness of each formalism we develop by continuing to prove standard theorems such as subject reduction and type soundness.

References

1. Sperber, M., Kelsey, R., Clinger, W., Rees, J., Findler, R., Matthews, J.: Revised^6 Report on the Algorithmic Language Scheme. Technical report
2. Sussman, G.J., Steele, G.L.: Scheme: A Interpreter for Extended Lambda Calculus. Higher-Order and Symbolic Computation **11**(4) (1998) 405–439
3. Jones, S.P.: The Haskell 98 Report. Technical report (1999)

4. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3) (2001) 396–450
5. Alves-Foss, J., Lam, F.S.: Dynamic Denotational Semantics of Java. Formal Syntax and Semantics of Java (1999) 201–240
6. Van Rossum, G., Drake, F.L.: Python Language Reference Manual. Network Theory (2003)
7. Abadi, M., Cardelli, L.: A theory of objects. Springer (1996)
8. Thomas, D., Hunt, A.: Programming Ruby. Addison-Wesley Reading, MA (2001)
9. Specification, E.L.: Standard ECMA-262. ECMA Standardizing Information and Communication Systems **3**
10. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1983)
11. Agesen, O., Bak, L., Chambers, C., Chang, B.W., Hlzle, U., Maloney, J., Smith, R.B., Ungar, D., Wolczko, M.: The Self 4.1 Programmer's Reference Manual

A Metrics Based Approach to Evaluate Design of Software Components

Kuljit Kaur, Hardeep Singh,
Department of Computer Science and Engineering,
Guru Nanak Dev University,
Amritsar, India-143005
kuljitchahal@yahoo.com,hardeep_gndu@rediffmail.com

Abstract. Component based software development approach makes use of already existing software components to build new applications. Software components may be available in-house or acquired from the global market. One of the most critical activities in this reuse based process is the selection of appropriate components. Component evaluation is the core of the component selection process. Component quality models have been proposed to decide upon a criterion against which candidate components can be evaluated and then compared. But none is complete enough to carry out the evaluation. It is advocated that component users need not bother about the internal details of the components. But we believe that complexity of the internal structure of the component can help estimating the effort related to evolution of the component. In our ongoing research, we are focusing on quality of internal design of a software component and its relationship to the external quality attributes of the component.

Keywords: Software Components, Component Based Software Development, Component design, Component quality,

1 Introduction

Component Based Software Development (CBSD) paradigm is based on reuse of already existing components to produce new applications. In a Component Based system, many different types of components are integrated such as in-house developed components or third party components. Third party components exist in two forms – commercial off the shelf (COTS) or Open Source. New components may need to be developed for the application in hand and then added to the component library for future use. It is believed that such an approach can reduce the development effort and time, and increase the productivity and quality of the software. On the other hand maintenance of component based systems is still a major challenge. Such systems comprise of components from many different sources. Development and up gradation of these components is not in the control of the (system) development team. In the short run, we may be able to reduce the development effort and hence the cost, but in the long run may end up in increasing the maintenance cost.

This approach is different from the traditional way of software development. In this approach, the development process has two sides: Development of software components for reuse and development of software with reusable components as the building blocks.

The main steps in development for reuse process are:

1. Perform domain analysis
2. Identify the components to be developed
3. Develop the components
4. Evaluate the components so that they can be added to the library
5. Package the components and add to the library.

The main steps in development with reuse [10] are as follows**:**

1. Retrieve components from library (in house or third party) according to some need of the application under development,
2. Evaluate the quality and appropriateness of the components.
3. Adapt a component, if it cannot be reused as-is.
4. Assemble the application
5. Test the integrated assembly

We observe in the above discussion that component evaluation takes place at two stages: when components are added to the library of reusable components and when they are selected for use in an application. In the latter case, context of use is also important for evaluation.

Evaluation of software components requires answers to the following questions:

1. What to evaluate? – To begin with, we have to determine the set of characteristics/attributes/properties of software components that we want to assess. In order to quantify these characteristics, each of them has to be decomposed into sub characteristics, which can be measured directly. For example maintainability expands to extensibility, modifiability.
2. How to evaluate? – For every sub characteristic identified, determine the set of metrics that we can use to measure it.

We need to define a component quality model that categorizes the properties of interest of the components, and provides a set of metrics that can quantify these properties. A review of the research literature reveals that several component quality models have been proposed [2], [8], [22], [24]. But they are very general in nature. None of them gives a detailed view of the quality attributes and related metrics.

2 Problem Description

Software component design has two perspectives- external or interface design that is visible to the component user (component assembler), and internal design that is initially visible to the component developer only and later to the component maintainer too. It is a known fact that effort of software maintenance depends largely upon the internal structure of the software. If internal design of a software product is not good, more cost (in terms of effort and time) will be involved in updating the product to meet changed requirements.

In a component based software system, evolution of individual component becomes difficult because of lack of access to internal details of a component e.g. design, source code etc. Component user has to depend upon component developer to carry out the changes. This is the reason that vendor/developer support is very important in successful implementation and usage of a software component. Component user relies on the vendor or other users of the same component, for information regarding its functionality/quality features. Unfortunately, this information is not readily available.

There is huge gap in the information provided by the component suppliers and information required by the component users [7]. Both the parties have not yet agreed upon a common framework for evaluation of software components.

Several component quality models have been proposed in order to lay down the criteria to evaluate software components. But all except [24] do not address this issue of internal design. Internal design cannot be simply ignored, since some internal attributes of a component may provide an indirect measurement of its external characteristics.

Metrics to evaluate internal design of the software components exist in the research literature [13], [14], [16], [21]. Most of them deal with coupling, cohesion, and complexity features only. We intend to prepare a framework of metrics to measure internal design of a software component from several other perspectives such as abstraction, Information hiding, polymorphism. The focus of this study is the components designed using object oriented design methodology.

3 Related Work

Metrics based evaluation of components and component based systems has been a topic of research in the recent past. Metrics for component based systems are presented in [3], [4], [17], [20], and [23]. Component oriented metrics can be studied across two dimensions – internal metrics and external metrics. Internal metrics measure the internal structure of the components and require access to the internals of the software component such as design or source code. Such information is not available to component users. External metrics are based on whatsoever information is available regarding the components such as interface, component documentation etc.

Interface based metrics are discussed in [9], [15]. Bertoa et al has presented metrics to evaluate usability of software components in [6]. These metrics are based on information made available by the component vendors. Another metric set in this category is presented in [25] to evaluate reusability of software components.

Internal metrics require access to the design or source code. Some of the metrics in this category are related to coupling and cohesion only [16], [21]. Complexity of the components can be measured using metrics available in [13], [14].

4 Significance of the Study

Quality is customer satisfaction, and the customer of a software component is interested in external product attributes like functionality, reliability, verifiability, usability, integrity, reusability, maintainability, portability, and interoperability. Therefore, quantifying quality attributes requires external measures of a product. But the development team, responsible for the development of the software component , has access to and can control over the internal measures such as the software development process followed, the resources used(including human expertise) etc. Internal product attributes include size of the software component, abstractions used, information hiding, modularity (even distribution of responsibilities to classes), level of reuse, intra – component coupling, level of cohesion etc. The development team may paint rosy picture of the otherwise low quality component.

Take an example of a component, which is developed –

- With no sub classing
- High levels of coupling

Now this can be interpreted as -

- The size of the component has increased, if it is measured in Lines of Code (LOC), because of less use of sub classing. This will lead to high price tag of the component, if size is taken as a factor for price determination.
- At the time of maintenance, the change effort will increase because of (unnecessary) coupling, so cost of maintenance will also increase.

Metrics can be used to check as to up to which level a particular object oriented software component follows the principals of a good object oriented design. Good design leads to the ease of maintenance of the software. Poor quality comes from poor design, where internal structures and methods are exposed, resulting in complicated inter-dependencies that grow worse over time. The bad design choices may be made because of time to market pressure. A lot of work has been reported in the research literature that maps the internal measures of the object oriented designs to the external attributes of the software products [5], [12].

5 Aim of the Study

Object oriented methodology presents various features, which help to meet the challenges of building complex and maintainable applications. Such features as inheritance, encapsulation, and abstraction can facilitate the developer to develop easily understandable and modifiable solutions for complex problems. Our goal in this research is to identify metrics that can quantify the usage of basic elements of object oriented design methodology in the internal design of software components.

Several metrics to evaluate object oriented design has already been proposed by various researchers [1, 11]. Many of them are empirically as well as theoretically validated too. Cross validation studies of these metrics also exist. Our aim is not to add new metrics to this already existing volume of metrics. We intend to choose from this collection of metrics, a set of metrics which measure the usage of necessary elements of an object oriented design.

6 Research Methodology

Research will be carried out in following stages:

1. Identify the basic elements of an object oriented design.
2. Prepare a list of metrics, to measure different aspects of object oriented design.
3. Select a set of independent metrics from this list.
4. Validate this set of metrics in the context of software components.

7 Results Achieved so far

We have studied the basic elements of the component based software development approach [18]. In this paper, several points of difference of the traditional software development from the modern component based software development are identified. Software development processes with new sets of activities for this paradigm are discussed.

We applied CK-Metric suite [11] and Abreu's MOOD [1] Metric suite to a model software component. It was found that the internal design of the software component lacks quality [19]. Designers of the component have not made use of the features of the object oriented methodology. In future any change or extension of the component will require more effort.

References:

1) Abreu, F.B., Goulão, M. and Esteves, R.: Towards the design quality evaluation of object oriented Software Systems, Proceedings of the 5th International Conference on Software Quality, Austin, Texas, (1995).
2) Abreu, F.B., Goulão, M.,: Towards a Component Quality Model, Work in Progress Session of the 28th IEEE Euromicro Conference, Dortmund, Germany, (2002).
3) Ali, S.S. & Ghafoor, A., and Paul, R.A.: Software Engineering Metrics for COTS-Based Systems, IEEE Computer, (2001).
4) Ali, S.S. & Ghafoor, A., and Paul, R.A.: Metrics Based Framework for Decision Making in COTS-Based Software Systems, Proceedings of the 7th International Symposium on High Assurance Systems Engineering (HASE'02),(2002).
5) Basili, V. R., Briand, L.C., and Melo, W.L.: A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 22(10), (1996).
6) Bertoa, M.F. & Troya, J.M., and Valleceillo, A.: Measuring the usability of Software Components, The Journal of Systems and Software, (2006).
7) Bertoa, M.F. & Troya, J.M., and Valleceillo, A.: A survey on the Quality Information Provided by Software component Vendors, Proceedings International Workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE'2002), (2002).
8) Bertoa, M., Vallecillo, A.: Quality Attributes for COTS Components, In the Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering *(QAOOSE)*, Spain, (2002).

9) Boxall, M., Araban, S.: Interface metrics for Reusability Analysis of Components, Australian Software Engineering conference (ASWEC'2004), Melbourne, Australia, (2004).

10) Butler, G., Li, L., and Tjandra, I.A.: Reusable Object-Oriented Design, available at http://citeseer.ist.psu.edu/butler95reusable.html accessed on 11.03.2008.

11) Chidamber, S.R. and Kemerer, C.F.: A Metrics Suite for Object Oriented Design, IEEE transactions on Software Engineering, vol. 20, no 6, (1994).

12) Chidamber, S.R., Darcy, D.P., and Kemerer, C.F.: Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis, IEEE Transactions on Software Engineering, vol. 24(8), (1998).

13) Cho, E.S., Kim, M.S., and Kim, S.D.: Component Metrics to measure Component Quality, the 8th Asia Pacific Software Engineering Conference (APSEC), Macau, (2001).

14) Dumke, R., Schmietendorf, A.: Possibilities of the Description and evaluation of Software components, Metrics News 5(1), (2000).

15) Gill, N. S., Grover, P.S.: Few Important considerations for Deriving Interface Complexity Metric for Component Based System, Software Engineering Notes, 29(2), (2004).

16) Gui, G., Scott, P.D.: Coupling and cohesion measure for evaluation of component reusability, International conference on Software Engineering, Proc of the 2006 International workshop on Mining Software Repositories, Shangai, China, (2006).

17) Hoek, A. v. d., Dincel, E. and Medvidovic, N.: Using Service Utilization Metrics to Assess and Improve Product Line Architectures, Proceedings 9th International Software Metrics Symposium (Metrics'03), Sydney, Australia, IEEE Computer Society Press. (2003)

18) Kaur, K., Kaur, P., Bedi, J., Singh, H.: Towards a systematic and suitable approach for component based software development, proceedings XXI International Conference on computer, Information and Systems Science and Engineering, Enformatika Computer society, Vienna, Austria, June, (2007).

19) Kaur, K., Singh, H.: Metrics to evaluate Object Oriented software Components, Computer Society of India (CSI) Communications, Special Issue on Object Oriented Technologies, volume no. 31, issue 11, February, (2008).

20) Narasimhan, V. L. and Hendradjaya, B.: Theoretical Considerations for Software Component Metrics, Transaction on Engineering, Computing and Technology, v10, (2005).

21) Pilskalns, O. and Williams, D.: Defining Maintainable components in the Design Phase, Proceedings 21st international Conference on Software maintenance, ICSM'05, (2005).

22) Simão, R.P.S., Belchior, A.: Quality Characteristics for Software Components: Hierarchy and Quality Guides, Component-Based Software Quality: Methods and Techniques, Lecture Notes in Computer Science (LNCS) Springer-Verlag, Vol. 2693, (2003).

23) Seker, R.: Assessment of Coupling and Cohesion for Component-Based Software by Using Shannon Languages, South African Institute of Computer Scientists and Information Technologists, Stellenbosch, Western Cape, South Africa. (2004)

24) Meyer, B.: The Grand Challenge of Trusted Components, in Proceedings of the 25th International conference on Software engineering (ICSE'03), Portland, Oregon, (2003).

25) Washizaki, H., Yamamoto, H., and Fukazawa, Y.: A Metrics suite for Measuring Reusability of Software Components, Metrics'03, (2003).

Methodology for Requirements Engineering in Model-Based Projects for Reactive Automotive Software

Niklas Mellegård and Miroslaw Staron

Department of Applied IT
IT-University of Gothenburg
{niklas.mellegard,miroslaw.staron}@ituniv.se
http://www.ituniv.se

Abstract. With increasing demands on vehicle safety together with the steep growth in software-controlled functions in contemporary vehicles, the demand for a software-focused development model becomes ever more apparent. There are a number of automotive-domain-specific obstacles that prevent development methods from keeping up with the current trends in more traditional software intensive areas. This paper outlines an empirical research project that focuses on introducing relevant concepts from MDSD to the automotive software development process applied at Volvo Car Corporation in order to raise the level of abstraction during software development. The anticipated outcome of this project is a new method for working with model-based software projects with particular focus on non-functional safety requirements.

Key words: Automotive; Model Driven Engineering; Requirements Engineering

1 Introduction

Future vehicles are predicted to contain increasing amounts of software that will provide unique functions and features built using parts common to several manufacturers. Vehicle safety systems are no exception, and since the reliability of software components has different characteristics than the reliability of hardware (both electronic and mechanical) components, new challenges arise.

Vehicle safety measures have been an ongoing research area since the 1950's with the introduction of seat belts as a standard feature. Such passive, or secondary, safety features [1] aim at minimizing the damage once an accident is considered unavoidable. The introduction of software-controlled functionality in vehicles opens up possibilities for primary safety [1] functions that, in more or less autonomous ways, assist the driver in preventing accidents from happening.

The ASIS (Algorithms and Software for Improved Safety) project at Volvo Car Corporation (VCC) intends to introduce systems for sophisticated primary safety, such as it is described in [2], by combining many different sources of information to create a higher fidelity perception of the current situation in order to provide accident detection and prevention subsystems with higher detailed data for more reliable decision-making.

An active safety system will to a large extent be controlled by software [3–6], which in turn will put high demands on the development models used. These demands include qualities such as:

- Allowing for modelling of functionality at a high abstraction level with a clear and preferably automatic mapping to code such as ones introduced in Model Driven Software Development (MDSD) [7]
- Allowing for continuous forward and backward traceability throughout the process, from requirements all the way to testing and back

This paper is structured as follows: the next section will outline some of the challenges apparent in the current development process and also briefly suggest expected benefits, should the research find alternative ways of handling those challenges. The paper will then present the approach we intend to take and also some expected results.

2 Problem Description - Challenges and Research Focus

It has been estimated that about 10% new functionality is introduced in each new car model. 80% of the new functionality is realized through software, still only 10% of the software is reused between car models [5]. There are many reasons behind the low degree of code reuse, e.g. suppliers of car parts (like ABS) develop most of the code themselves and usually keep the intellectual property (IP), the final code is often heavily optimized for the hardware used, changes in electrical architecture may make the code obsolete.

The overall goal and main focus for research is how to increase the reuse of software and enable using software as the main factor adding functionality to new cars as opposed to today where software is to a large extent developed and delivered bundled with the hardware. We intend to apply concepts from MDSD [7] to raise the abstraction level and introduce automated model transformations refining the code for various platforms. In our research up to the point, we have identified the following issues which we intend to address:

- **Development and integration of heterogeneous subsystems**: The diversity of types of software in cars is growing, the usual types are: infotainment, hard real-time and safety relevant systems [4]. To make these systems operate and interoperate seamlessly and safely, a joint information model and common non-functional safety requirements are needed.
- **Shared data model**: One obstacle in raising the abstraction level and also separating different types of subsystems is there is no central, shared and stable data model, although data is produced by many different subsystems and also used by different parts of the vehicle. The lack of common data model and central data storage has the effect that it may require major

architecture changes even for minor changes in functionality. The ongoing AUTOSAR [8] initiative intends to address this problem.

- **Suppliers and tier X components**: Software is to a large extent written by suppliers and often delivered as black-box components together with the complete mechanical/electric parts. The proprietary software sealed in parts makes it hard to specify common and reusable non-functional requirements (as these are usually proprietary to the supplier too). In consequence, this adds yet another factor making reuse harder.
- **Price-per-unit cost model**: The standard cost model used in the automotive industry in general, and VCC is no exception, has the manufacturing price-per-unit as a central measurement [5]. The effect of having a huge number of units produced, which dwarfs the cost of engineering efforts in comparison [4], is that any cost savings in material and manufacturing that can be made, even at the expense of good software engineering practice will likely be justified. There is a need to illustrate how additional spending on vehicle computer resources can be justified, not only through lower future software engineering costs, but also through increased possibilities for the development of more sophisticated functionality with less engineering efforts.
- **Conceptual gap between model and implementation**: Often qualities such as maintainability and reusability are outweighed by a combination of extreme ones like reliability and performance/cost. This has the effect that software is developed at a low level, close to the hardware, which in turn makes the implementation susceptible to any future low level changes and also makes reuse of components hard. This becomes apparent when the model of a component needs to be changed or replaced during the lifecycle of a vehicle model. There is a need to explore a framework that binds high level object-oriented models tighter together with the mostly procedural implementation.
- **Challenging life cycle**: A car model is typically in development for 7-8 years, in production for an additional 6 years and expected to get service and support by the aftermarket for another 10-15 years [5]. During the life cycle of a car model, much of the electronics in the car will be updated or replaced. These are problems that are present throughout the life cycle of a vehicle model:

  – The hardware that is used in the development of the vehicle model may not be the same as the one used in production.

  – During the production of the car model, parts of the hardware may be discontinued for many reasons; e.g. new generations of the same parts emerge, the supplier may go out of business etc.

  – During the very long aftermarket period, hardware parts are very likely to be discontinued or at least changed.

  This calls for more effective software development methods providing possibilities for "quick" porting of software into new platforms. The notion of model transformations and model-independence (central in Model Driven Architecture [9–11]) will be used to achieve such goals.

The above points are planned to be addressed in the ASIS project as part of this doctoral thesis.

3 Research Approach

As this project is conducted in cooperation with VCC, we will follow an empirical software engineering approach. In particular we plan to conduct a series of interviews in order to identify the source of underlying problems at the company, develop solutions and experiment to validate our solutions. We intend to follow the conceptual research process as presented in Fig. 1.

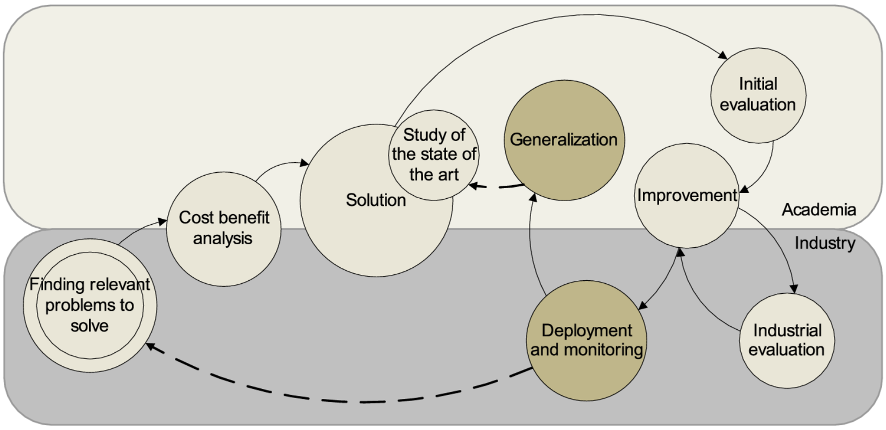

Fig. 1. Conceptual research process ([12, 13])

During this process we expect to incorporate several empirical research methods: case studies (including prototyping), experiments, and surveys. In the initial phase, we will conduct a series of case studies to identify and prioritize improvement issues in the development of software-intensive primary safety systems. By combining the principles of MDSD and requirements engineering methods, we intend to address the problems of low software reuse between car types. By enabling traceability of non-functional requirements in models, we intend to decrease the effort required for developing safety-critical car systems.

As a first step a study of the current requirements engineering procedures will be conducted. The objective with this study is to map the requirements flow and refinement from top to bottom in order to identify key activities to use as basis for how MDSD concepts can be applied. As part of that study the requirements refinement process within each development level will also be examined.

4 Expected Results

The expected result of this project is new methodology for working with non-functional safety requirements in future software development projects within the automotive industry. The methodology is required to have characteristics such as:

- Object-oriented: The object-orientated paradigm is arguably required for modelling at a higher abstraction level, especially when reusability of proprietary code is involved.
- Model driven: models and model transformations are intended to increase portability of software among various hardware platforms.
- Safety focused: the safety requirements are intended to be in the centre of development efforts.

Validation of the results will be done through empirical studies throughout the work in the project. We plan to use experiments whenever possible, both in academia and in industry.

5 Related Work

The ATESST (Advancing Traffic Efficiency and Safety through Software Technology) project [14–17] examined ways of coping with with the complexity of modelling requirements using UML as a notational language.

The AUTOSAR (Automotive Open System Architecture) project [8, 18] aims at standardizing the architecture and the way components communicate with eachother using a service oriented architecture.

Other related studies on requirements management include [19].

6 Summary

The last 20 years has shown a trend of steep increase in the amount of software that is used for realizing functions in a vehicle, a trend which is expected to continue. To cope with not only increasingly complex functions, but also integration of multiple subcomponents to synthesize new functionality without compromising the very strict non-functional requirements that govern features in the automotive industry, improved formal software development models are required.

The research outlined in this paper intends to explore areas where concepts from state-of-the-art software development models will have an improving effect on development in the automotive domain.

Acknowledgement

The ASIS project is partially sponsored by VINNOVA [20] under the IVSS [21] programme. It is actively run and partially sponsored by VCC. We would like to thank the involved managers at VCC for their support.

References

1. EEVC WG19, Primary and Secondary Safety Interaction, EEVC, 2006
2. M. Murphy, Passenger car driver assistance systems, technologies and trends to 2015, Automotive World, 2007; `http:\\www.automotiveworld.com`
3. M. Broy et al., Engineering Automotive Software, Proceedings of the IEEE, vol. 95, 2007, pp. 356-373.
4. N. Navet et al., Trends in Automotive Communication Systems, Proceedings of the IEEE, vol. 93, 2005, pp. 1204-1223
5. C. Salzmann and T. Stauner, Automotive software engineering: an emerging application domain for software engineering, Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL'03, Kluwer Academic Publishers, 2004, pp. 333-347
6. A. Sangiovanni-Vincentelli and M. Di Natale, Embedded System Design for Automotive Applications, Computer, vol. 40, 2007, pp. 42-51
7. T. Stahl and M. Völter, Modeldriven software development: technology, engineering, management, Chichester, England ; Hoboken, NJ: John Wiley, 2006
8. AUTOSAR Webpage; `http://www.autosar.org`
9. J. Miller and J. Mukerji, MDA Guide, Object Management Group, 2003.
10. S.J. Mellor et al., ModelDriven Architecture, ObjectOriented Information Systems, Z. Bellahsene, ed., Montpellier: SpringerVerlag, 2002, pp. 290-307
11. A. Gerber et al., Transformation: The Missing Link of MDA, Graph Transformation: First International Conference, ICGT 2002, A. Corradini et al., ed., Barcelona, Spain: SpringerVerlag, 2002, pp. 90-105
12. Staron, M.: Improving Modeling with UML by Stereotype-based Language Customization. Department of Systems and Software Engineering, Doctoral Blekinge Institute of Technology, Ronneby, Sweden (2005)
13. Gorschek, T.: Software Process Assessment & Improvement in Industrial Requirements Engineering. Department of Systems and Software Engineering, Blekinge Institute of Technology, Ronneby, Sweden (2004)
14. ATESST Webpage; `http://www.atesst.org`
15. P. Cuenot et al., Managing Complexity of Automotive Electronics Using the EAST-ADL, Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), IEEE Computer Society, 2007, pp. 353-358; `http://portal.acm.org/citation.cfm?id=1270390.1271079`
16. C. Sjöstedt et al., Mapping Simulink to UML in the Design of Embedded Systems: Investigating Scenarios and Structural and Behavioral Mapping, OMER 4 Post Workshop Proceedings, 2008.
17. F. Törner et al., Supporting an Automotive Safety Case Through Systematic Model-based Development, Proceedings of the SAE World Congress, 2008
18. S. Furst, AUTOSAR for Safety Related Systems: Objectives, Approach and Status, Automotive Electronics, 2006. The 2nd IEE Conference on, 2006, p. 3
19. Almefelt et al., Requirements management in practice: findings from an empirical study in the automotive industry, Research in Engineering Design, vol. 17, Dec. 2006, pp. 113-134
20. VINNOVA Webpage; `http://www.vinnova.se/`
21. IVSS Intelligent Vehicle Safety Systems - Web Site; `http://www.ivss.se/`

An Exception Handling Framework

Nikolas Nehmer and Andreas Reuter (advisor)

University of Kaiserslautern, 67663 Kaiserslautern, Germany
nnehmer@informatik.uni-kl.de,
WWW home page:
http://lgis.informatik.uni-kl.de/cms/index.php?id=nnehmer

Abstract. Today's exception handling mechanisms are insufficient for meeting the dependability requirements of large and complex software systems. In this paper a novel exception handling framework is introduced. The framework includes a tool supporting developers in reasoning about exception flow. Based on the exception flow analysis a novel fault containment approach is proposed restricting the impact of uncaught exceptions on the overall system.

1 Problem Description

Complex object-oriented software systems have to cope with an increasing number of exceptional conditions, raised by the environment or by faults persisting in the software itself. Developing fault-free software is nearly impossible and it's unwise to assume that the environment in which software operates always functions correctly [1]. Incorporating fault tolerance mechanisms into the system architecture is essential to meet dependability-related system requirements. Exception handling is one of the most important fault tolerance mechanisms for detecting and recovering from errors, and for structuring fault tolerance activities in a system by separating normal and exceptional control flow structures.

In complex object-oriented systems often more than two-thirds of the application code is devoted to detecting error conditions and to handling these erroneous situations [2]. Unfortunately, experience shows that especially exception handling code carries a high risk of being erroneous. The reason is complex. It can be partially attributed to human behavior and error-prone exception handling mechanisms [1]. Furthermore tools supporting developers in reasoning about exception flow are missing. "Real world" examples such as the crash of the Ariane 5 [3] missile illustrate the possible impact of inappropriate exception handling on the overall system dependability.

Although Java is closely related to the ideal exception handling model by Garcia et al. [4] a closer look reveals many issues being the potential source of problems. Uncaught exceptions are a main issue. Furthermore exception handling effectiveness highly depends on the programming discipline, i.e. no compulsory specification of all exceptions possibly propagated by a method, issues related to type subsumption (e.g. implicit catches), long exception propagation paths and sloppy handler design (e.g. empty handlers).

This clearly illustrates the need for a systematic and defensive approach to exception handling tackling two main aspects:

- Missing development tools supporting developers in reasoning about exceptions and exception flow
- Potentially catastrophic impact of uncaught exceptions

2 Goal Statement

The goal of this research is to tackle the two aspects stated above. Although in programming languages such as Java compilers help in reasoning about exception flow, this support is insufficient. Many problems related to type subsumption or uncaught exceptions have to be detected and resolved by the developers manually during the development process. Even in deployed software systems these problems still persist undetected until they raise system failures. Especially in todays large and complex systems, systems of systems, component architectures or service oriented architectures reasoning about exception flow without proper tool-support seems to be unmanageable. Tools supporting software developers in this reasoning process by deriving exception propagation information from component code are strongly required and help in designing robust software systems by establishing sound exception handling mechanisms. Furthermore system runtime mechanisms have to be made available to provide means for automatic fault containment by building quarantine areas around "infected" data structures during runtime. Exception propagation beyond certain system boundaries has to be prohibited to contain the impact of uncaught exceptions. Components and data structures affected by exceptions not handled appropriately have to be identified and isolated during runtime. Access to the functionality provided by these data structures has to be restricted dynamically to reduce error propagation.

Accordingly, an exception handling framework including two highly interdependent and closely related approaches is proposed:

- Development of a graph-based static exception analysis tool for Java integrated into the Eclipse development environment
- Development of a runtime system mechanism for Java containing the impact of uncaught exceptions by prohibiting exception propagation and gracefully degrading system functionality

3 Approach

Graph-based Exception Analysis – Supporting developers by analyzing component code during compile-time to detect exception handling related problems is the basic goal of the graph-based exception analysis approach. Static code analysis based on the abstract syntax tree (AST) of component code is applied. A transparent representation of all exceptions possibly encountered within an arbitrary system scope is the outcome of such analysis. In object oriented programming languages methods are a reasonable scope for exception handling analysis.

Exceptions encountered within a scope comprise exceptions explicitly raised by "throw"-statements, exceptions raised as the result of system operations, exceptions explicitly propagated from method calls and the set of uncaught exceptions implicitly propagated from enclosed scopes. This approach is based on existing exception flow analysis approaches [5–9].

To support static and dynamic reasoning about exception flow in the context of object oriented software development, a general model of exception handling structures is mandatory – the general exception model (GEM). The model correlates exception handling structures in object-oriented languages that define exceptions as objects to their occurrence in a program's calling hierarchy. The goal of the model is to provide a unified basis for discussing problems related to the design, implementation, and maintenance of exception-handling structures, and for the analysis that can help alleviate these problems. The model is focused on the description of possible exception flows in a program. The general exception model is easily applicable to the Java programming language. The GEM is adapted from work by Robillard/Murphy [7] and Schaefer/Bundy [8]. A GEM-instance represents the information required to identify a program's exception flow. A data structure intuitively suitable to represent this information is a directed labeled graph – the exception propagation graph (EPG). The graph can be derived from a program's abstract syntax tree. The EPG represents a program's calling hierarchy and correlates exception structures (i.e. possibly raised exceptions and exception handlers) to their occurrence in the calling hierarchy. To properly reflect exception flow in the graph structure the graph is annotated. Nodes and edges are extended with exception path information (EPI) identifying all possible exception propagation paths from lower level nodes to the root node (system entry point).

Most integrated development environments such as Eclipse use abstract syntax trees (AST) as internal object representations of program code. Navigation and editing program code, for example, are performed based on this AST-representation. The proof of concept for the approach briefly described above is based on Eclipse framework's DOM representation (Eclipse's AST) of Java code. The prototypical exception analyzer is realized as an Eclipse plug-in and is tightly integrated into the Eclipse platform providing precise exception flow information for program code. The exception analyzer prototype is presently being implemented.

Fault Containment and Graceful Degradation in the Presence of Uncaught Exceptions – Todays exception handling mechanisms exclusively located at the application level are highly error-prone. A default exception handling mechanism located at the runtime level protecting the application from the impact of uncaught exceptions is missing. An exception handling framework (exception guard) prohibiting exception propagation beyond well-defined system points and gracefully degrading system parts affected by uncaught exceptions is proposed. The framework is based on the EPG gained from the static code analysis.

Components are a reasonable scope for prohibiting exception propagation. A vital feature of any sound exception handling mechanism is the differentiation between internal exceptions to be handled inside the scope and the external exceptions which are propagated outside the scope [10]. For every exception raised during runtime, a system's EPG can be used to determine if an appropriate handler exists within the same scope without propagating the exception up the call stack. Internal exceptions identified as uncaught exceptions are converted into default external exceptions, declared in the component signature by default. Component parts affected by uncaught exceptions have to be gracefully degraded. Access to these component parts are either restricted or prohibited. "Erroneous" object instances are identified by combining static information represented by the EPG with runtime information. Exception raising system states are analyzed by investigating call stack information and object states. This information can be used to identify potential error-raising conditions and prevent future exception occurrences. A prototypical implementation of the exception guard is subject of ongoing work. Systematic fault injection into sample applications will be used to evaluate the framework comparing application behavior with and without support by the framework.

References

1. Romanovsky, A., Sandén, B.: Except for exception handling . . . . Ada Lett. **XXI**(3) (2001) 19–25
2. Cristian, F.: Exception Handling and Tolerance of Software Faults. In: Software Fault Tolerance. John Wiley & Sons (1995) 81–107
3. Jézéquel, J.M., Meyer, B.: Design by contract: The lessons of ariane. Computer **30**(1) (1997) 129–130
4. Garcia, A.F., Rubira, C.M.F., Romanovsky, A., Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software. The Journal of Systems and Software **59**(2) (2001) 197–222
5. Malayeri, D., Aldrich, J.: Practical exception specifications. In Dony, C., Knudsen, J.L., Romanovsky, A.B., Tripathi, A., eds.: Advanced Topics in Exception Handling Techniques. Volume 4119 of Lecture Notes in Computer Science., Springer (2006) 200–220
6. Sinha, S., Orso, A., Harrold, M.J.: Automated support for development, maintenance, and testing in the presence of implicit control flow. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 336–345
7. Robillard, M.P., Murphy, G.C.: Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. Softw. Eng. Methodol. **12**(2) (2003) 191–221
8. Schaefer, C.F., Bundy, G.N.: Static analysis of exception handling in ada. Softw. Pract. Exper. **23**(10) (1993) 1157–1174
9. Chang, B.M., Jo, J.W., Yi, K., Choe, K.M.: Interprocedural exception analysis for java. In: SAC '01: Proceedings of the 2001 ACM symposium on Applied computing, New York, NY, USA, ACM (2001) 620–625
10. Romanovsky, A.B.: Exception handling in component-based system development. In: COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, Washington, DC, USA, IEEE Computer Society (2001) 580

First Class Relationships for OO Languages

Stephen Nelson, David J Pearce (Advisor), and James Noble (Advisor)

{stephen,djp,kjx}@mcs.vuw.ac.nz
Victoria University of Wellington, New Zealand

Abstract. Relationships have been an essential component of OO design since the 90s and, although several groups have attempted to rectify this, mainstream OO languages still do not support *first-class* relationships. This requires programmers to implement relationships in an ad-hoc fashion which results in unnecessarily complex code. We have developed a new model for relationships in OO which provides a better abstraction than existing models provide. We believe that a language based on this model could bring the benefits of relationships to mainstream languages.

1 Problem Description

Object-oriented practitioners are frequently faced with a dilemma when they design and implement object-oriented systems: modelling languages describe object systems as a graphs of objects connected by relationships [2, 9], but most object-oriented languages have no explicit support for relationships. This results in a trade-off between high-level models which are de-coupled from their implementations and low-level models which are confusing to use as they contain irrelevant detail.

There have been many proposals for adding relationship support to object-oriented languages. Rumbaugh proposed a language with relationship support in 1987 and there has been a recent resurgence interest with proposals for language extensions [1, 3, 10] and library support [6, 7].

The potential benefits of such support are clear: improved traceability between design and implementation, reduced boilerplate code, better program understanding by programmers, and the opportunity to introduce new high-level language features such as queries, relationship (function) operations, and program structure constraints.

In spite of the clear advantages of relationship support none of the solutions proposed so far have achieved widespread use, and while this may be due to limited time and exposure we believe that existing solutions fail to capture the intent of object-oriented models and so fail to realise those advantages.

We intend to address the disconnect between object-oriented design and implementation by rethinking the way object-oriented languages are structured. Rather than adding relationships to existing language models [1, 3, 6, 7, 10] we propose that existing language models should be re-factored to support relationships as a primary metaphor.

2 Goal Statement

Our goal is to address the disconnect between object-oriented design and implementation by rethinking the way object-oriented languages are structured.

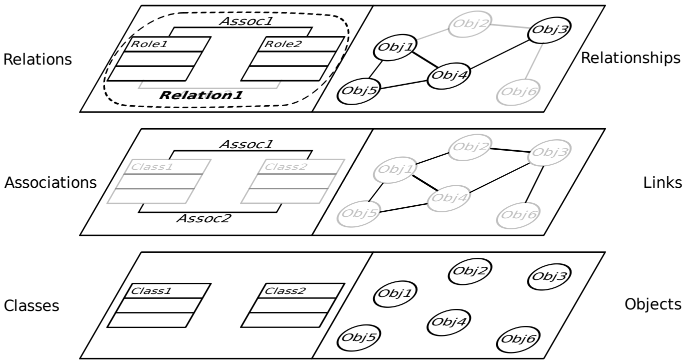

Fig. 1. The three tiers in our relationship system. The bottom tier (object tier) describes the objects in the system. The middle (link tier) adds the links between objects, and the top (relationship tier) describes properties of groups of objects and links, including roles, relationship constraints, and collections.

This will allow us to realise the advantages long promised by relationships and introduce new high-level language features such as relationship queries, relationship (function) traversal operations, and program structure constraints. We have developed a new model for the object-oriented paradigm which focuses on relationships rather than objects.

3 Relationship Model

Typical relationship systems introduce relationships or associations (class level) which describe links, or groups of links in implementation (object level)[1, 3, 7, 6, 8, 10]. We believe that these systems do not adequately describe relationships because they either fail to consider groups of links at the implementation level or focus on groups of links (relationships) at the expense of the individual link at the modelling level. This introduces confusion in the literature because it is not clear whether a relationship is a link or a group of links, and whether the term can be used at the modelling or the implementation level, or both.

We believe that by modelling objects, links, and relationships separately as classes, associations and relations, the resulting system will be easier to understand and consequently easier for programmers to use. To clarify the distinctions between the model elements we introduce a three tiered system for describing programs which is shown in Figure 1.

Object Tier The object tier consists of objects (modelled by classes), similar to the objects defined in traditional object-oriented models. Objects may have state and behaviour, but they may not communicate with other objects on this tier unless there is a strong composition relationship between them. This ensures that more complex relationships are moved up into the relationship tier.

Link Tier The next tier of our relationship system builds on the objects and classes defined in the object tier by adding connections (links) between objects described as associations between classes. An association between classes in this system indicates that the classes are related in some way. For example, an association between the classes Person and Course indicates that people may be linked with courses.

Links are immutable tuples containing one instance (object) from every class in their association, similar to Vaziri *et al.*'s *relations* [10]. Unlike Vaziri *et al.*'s relations, our links may not have any state or behaviour associated with them. This removes a lot of the complexity of associations in UML and other systems which makes them easier to describe and understand. For example, there is no distinction between sub-typing and subsetting of associations in this system: an association between subtypes is naturally a subset of the related association between super-types because all possible tuples in the subtype association are also tuples in the super-type association. Instead of having complex associations, we introduce an additional modelling element.

Relationship Tier Relationships, introduced at the third tier of our system, are sets of links which have type and identity. For example, we can describe the notion of people attending courses as a subset of people and a subset of courses which are linked by some subset of the possible links between the set of people and the set of courses. Relationships exist as runtime entities modelled by 'relations'.

Unlike links and associations, which have the same type if they have the same type parameters (participants), relationships have a type defined by their relation in the same way as objects have a type defined by their class. For example, there may be two relationships between People and Courses, one of type "Attends" and the other of type "Teaches". These can even contain the same links but are not equivalent because they have incompatible type. There may even be two instances of type "Attends" which contain the same links and are not equivalent: relationships have independent identity in the same way as objects.

In addition to defining relationships between objects, relations may also define roles for the objects and associations which participant in them. Roles are like dynamic mixins; they define state and behaviour for objects (or links) separate from the object's class which is added to the object at runtime. The role is associated both with the class of objects (e.g. Person) and the relation of relationships (e.g. Attends) and adds functionality to the objects which participate in the relationship at runtime. For example, a person who is attending courses is a student. This does not subsume the identity of the person — the person may exist in other relationships which are unrelated to their role as a student. Rather, the role of a person as a student adds more functionality to the person: in the context of the person as a student they may have a student id, enrolment details, and functionality for calculating fees. These details are irrelevant outside of the student context and they do not affect the identity of the person however, so it does not make sense to include them in the standard definition of a student. Roles may also be defined for links and relationships (if the relationship is a participant in this relationship).

The three levels of abstraction which we identify are typically collapsed into a single level in the object-oriented paradigm. This causes interwoven, fragile code which is confusing to create and to maintain. The dependence of some object state and behaviour on relationships is seldom recognised because object-oriented programming languages obscure relationship concerns by describing their implementation rather than their intention. Even languages which do address relationships as separate entities fail to separate the behaviour associated with relationships from the classes [8, 3], or lose the association between the state and behaviour, and the objects with which it is associated [5, 7]. We believe that our relationship system simplifies the categorisation of behaviour as object, link, or relationship (role) behaviour and provides a more natural way to describe the related behaviour than Aspect-Oriented programming [4].

4 Validation

To validate our model we plan to use it to develop a language. We will present a grammar for the language, provide typing rules and semantics, and we will provide proofs of type safety, soundness and correctness. We will also provide an implementation of the language. This will highlight any practical problems with the language which are not apparent from the theoretical model. The implementation will consist of a compiler and a library system which will allow programs to be written in our language. Finally, we will conduct various case studies; implementing different systems in our language and others so that we can make comparisons between them.

References

1. Balzer, S., Gross, T. R., and Eugster, P. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP* (2007).
2. Beck, K., and Cunningham, W. A laboratory for teaching object oriented thinking. In *OOPSLA* (1989), ACM, pp. 1–6.
3. Bierman, G. M., and Wren, A. First-class relationships in an object-oriented language. In *ECOOP* (2005), pp. 262–286.
4. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP*. 1997.
5. Noble, J., and Grundy, J. Explicit relationships in object oriented development. In *TOOLS* (1995), Prentice-Hall.
6. Østerbye, K. Design of a class library for association relationships. In *LCSD* (2007).
7. Pearce, D. J., and Noble, J. Relationship aspects. In *AOSD* (2006), ACM Press, pp. 75–86.
8. Rumbaugh, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA* (1987), ACM Press, pp. 466–481.
9. Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
10. Vaziri, M., Tip, F., Fink, S., and Dolby, J. Declaritive object identity using relation types. In *ECOOP* (2007), pp. 54–78.

Towards a Formal Diagrammatic Framework for MDA

Adrian Rutle

Bergen University College, p.b. 7030, 5020 Bergen, Norway aru@hib.no

Abstract. Since the beginning of computer science, raising the abstraction level of software systems has been a continuous goal. One of the newest developments in this direction has lead to the usage of models and modeling languages in software development processes since software models are abstract representations of software systems which can be used to describe different aspects of the software systems at a higher abstraction level. Currently, in addition to documentation purposes models are increasingly used to generate and integrate parts of the software systems that they describe. In model-driven engineering these processes are referred to as model transformations and are executed by model transformation tools. As a consequence of the usage of models as input to model transformation tools, formal modeling languages and formal transformation definition techniques are needed in order to automatically translate between (and integrate) models. Therefore, a major focus of our research is on the formalization of modeling and model transformation in the generic formalism, Diagrammatic Predicate Logic (DPL) which is based on graph theory and category theory. This paper provides an overview of the state-of-the-art of our ongoing research on analysis of modeling and model transformations based on the DPL framework.

1 Introduction and Motivation

Models, model transformations and automatization of model transformations are key issues in the emergent approach of software development process, which is standardized by the Object Management Group (OMG) as Model Driven Architecture (MDA) [4]. In the MDA approach, building an application starts with a (set of) platform-independent models (PIM) in which the structure, logic and behavior of the application are specified. The PIMs are then transformed by transformation tools to a set of platform-specific models (PSM). These PSMs are used as input to code-generation tools which automatically create software systems based on the input models [3].

In MDA, model transformation is the generation of a target model from a source model [3]. These transformations are executed in transformation processes. Each transformation process is described by a transformation definition, which in turn consists of a set of transformation rules. The transformation definition is written in a transformation definition language. The transformation rules define how (and which) constructs from the source model are transformed to (which) constructs in the target model.

Several languages, approaches and tools for the definition of models and model transformations have emerged in the last years due to their usage in model-driven engineering. However, many of these modeling and transformation languages are either not sufficiently formalized or very complicated (text-based) or both, which makes writing formal models difficult and error-prone [2]. Moreover, since software models are graph-based, modeling languages which use string-based logic (e.g. first order logic) may fail to express all properties of software systems in an intuitive way without flattening the so-called conceptual two-dimensionality of the domain that they specify. Examples of these domains are entities and relationships, objects and links, states and transitions etc. Thus, diagrammatic modeling seems to be a reasonable approach for modeling software systems since it is graph-based – making the relation between the syntax and semantics of models more compact –, and nonetheless, it is easier for domain experts to understand [9].

However, diagrammatic modeling languages are considered more difficult to formalize than text-based languages, therefore, diagrammatic languages often use text-based languages to define constraints and system properties that are difficult to express by their own syntax and semantics, e.g. the combination of UML and OCL which is suggested by OMG as a specification formalism for the definition of PIMs [3]. This turns models to a mixture of text and diagrams which is often difficult for non-experts to evaluate and understand, i.e. the models loose their simplicity and high level of abstraction which are the most appealing features of modeling.

2 The Goals and Approaches of our Project

Our general idea is to develop and use a diagrammatic formalism to define and reason about models and model transformations. Therefore, a major focus of our research is on the analysis of diagrammatic modeling, MDA and model transformations based on a clean mathematical foundation. This includes the analysis of some of the existing approaches, languages and tools which are used in model-driven engineering. Further, based on this analysis and due to the needs arising during the development of a well-structured and well-founded approach to diagrammatic modeling and model transformation, we adapt and further develop the theoretical foundations.

Our project involves an inter-disciplinary research where both theoretical computer scientists and software engineers are working together to solve practical and theoretical problems. The challenge is to establish a common conceptual platform between the theoreticians and the software engineers to enable discussions about these problems. We have made a stepwise progress which started by studying the basics of Category Theory, Graph Theory, MDA and model transformations. Now we are studying some existing tools and approaches that are directly related to MDA, especially tools that are based on graph transformations and OMG standards. After understanding the principles of these approaches, we compare them with each other and find the commonality and differences between them. At the same time, we study some typical cases where model transformations are used and we have decided to use class diagram to database scheme transformation in our case study. Based on our preliminary investigations and analysis, we decided to use the Eclipse Modeling Framework (EMF) as the basis for our implementations. We chose EMF because of the clear semantics of Ecore, which is the metamodel of EMF models. Moreover, EMF is an open source initiative and many of the existing tools are implemented as plug-ins to Eclipse. Currently, we are working on the design of a toolset that will exploit the proposed mathematical foundation which is intended to be our formalization approach.

Our approach is based on the generic formalism Diagrammatic Predicate Logic (DPL) [9,8][1]. We consider DPL a suitable specification formalism to define diagrammatic modeling languages with a strong mathematical foundation, first, because models and metamodels in MDA are graph-based, and also because DPL is based on Graph Theory and Category Theory (CT), which is the mathematics of diagrammatic notations. Further adaptations of DPL have shown that it is also a promising approach for the formalization of model transformations [5]. Graph Theory is already used as a basis for modeling and model transformations. The concepts of typing and typed graphes are used frequently in software engineering communities to talk about metamodels and their instances, respectively. In addition, graphical model transformations are often linked to graph transformations and several implementations of triple graph grammars (TGG) are discussed in the literature.

DPL is a graph-based specification format that takes its main ideas from both categorical and first-order logic (FOL), and adapts them to SE needs [9]. In DPL, software models are represented by diagram specifications. These diagram specifications are structures which consist of a graph G and a set of diagrams in G which are labeled with predicates from a predefined signature [6]. Each labeled diagram corresponds to a constraint. The notion of labeled diagrams is nothing but a graph-based analog of a logical formula whose arity is a graph instead of a set. In DPL, each modeling language corresponds to a modeling formalism which consists of a signature and a diagram specification; where the signature corresponds to language constructs and the diagram specification specifies the metamodel of the language. The DPL formalism is a generalization and adaptation of the categorical sketch formalism where signatures are restricted to a limited set of predicates: limit, colimit and commutative diagrams [7]. This generalization is necessary to make DPL suitable for use in SE.

Research in the field of model transformations has shown that model transformation approaches are categorized into relational, where logic programming is used to express relations between models; graphical, which is based on the theory of graph-transformations; and hybrid, which is a mixture of both [1]. DPL can be seen as a natural enhancement of the graphical approaches which extends

[1] The DPL framework is called Generalized Sketches in previous publications, however, since the concept of "sketch" is misleading in SE, the name of the formalism is changed to DPL.

these approaches by a graph-based specification formalism and a corresponding graph-based logic, which is also part of our ongoing project.

The goals of our project are summarized as follows:

- development and adaptation of the theory of DPL and exploit these theories for the definition of diagrammatic specifications,
- analysis of the metamodel of EMF and relational database systems as diagram specifications in the DPL formalism and the transformation between them as case-studies, and,
- design and implementation of a framework for model-driven development that uses properties and capabilities from DPL.

The framework will consist of modeling tools, code-generation facilities and transformation definition tools. The modeling tools will be used to specify both signatures and diagram specifications, i.e. both models and modeling formalisms. The code-generation and model transformation tools will be used to define and execute model-to-code and model-to-model transformations, respectively. By developing tools that support DPL as a generic pattern for specifying and developing diagrammatic specification techniques and transformations between them, we can evaluate and exploit the practical values of DPL in many aspects of modeling and MDA.

References

1. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In OOPSLA 2003, editor, *Generative Techniques in the context of MDA*, 2003.
2. Zinovy Diskin. *Mathematics of UML: Making the Odysseys of UML less dramatic. Practical foundations of business system specifications*, chapter 8, pages 145–178. Kluwer Academic Publishers, 2003.
3. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: practice and promise.* Addison-Wesley, 1 edition, April 2003.
4. OMG. *OMG Model Driven Architecture Web Site*, June 2007. Object Management Group, http://www.omg.org/mda/index.htm.
5. Laura Rivero, Jorge Doorn, and Viviana Ferraggine. *Encyclopedia of Database Technologies and Applications*, chapter Mathematics of Generic Specifications for Meodel Management I and II, pages 351–366. Information Science Reference, 2005.
6. Adrian Rutle, Uwe Wolter, and Yngve Lamo. Diagrammatic software specifications. In Iceland Reykjavik University, editor, *The 18th Nordic Workshop on Programming Theory (NWPT'06)*, october 2006.
7. Adrian Rutle, Uwe Wolter, and Yngve Lamo. Generalized sketches and model driven architecture. Technical Report 367, Department of Informatics, University of Bergen, Norway, 2008. Presented at CALCO Young Researchers Workshop 2007.
8. Uwe Wolter and Zinovy Diskin. The next hundred diagrammatic specification techniques: A gentle introduction to generalized sketches. Technical Report 358, Dept of Informatics, University of Bergen, July 2007.
9. Uwe Wolter and Zinovy Diskin. Generalized sketches: Towards a universal logic for diagrammatic modeling in software engineering. 2008. Proceedings, ACCAT 2007, ENTCS, Accepted for publication.

Author Index