Automated Software Engineering

98

13th IEEE International Conference

Doctoral Symposium Proceedings

October 13, 1998

Sheraton Princess Kaiulani
Honolulu, Hawaii, USA

Contents

About the Doctoral Symposium

The Doctoral Symposium at ASE'98 is intended to bring together PhD students working on foundations, techniques, tools and applications of automated software engineering technology and give them the opportunity to present and to discuss their research in a constructive and international atmosphere. The goals of the symposium are:

- To provide a setting for mutual feedback on participants' current research, and guidance on future research directions
- To develop a supportive community of scholars and a spirit of collaborative research
- To contribute to the conference goals through interaction with other researchers and conference events.

The main part of the Doctoral Symposium was held on October 13, 1998, the day before the main conference. This day took the form of a one day workshop, in which selected students each presented their work, with constructive feedback from one another, and from a panel of advisors. The workshop also included two invited talks on topics relevant to the process of completing a PhD and writing a thesis. In addition to the one-day workshop, participants of the Doctoral Symposium were encouraged to present their work as posters.

Twenty PhD students from nine different countries submitted papers to the symposium. The submissions were all of an excellent quality. Of these, seven students were invited to present their work at the symposium. Due to the high quality of all the submissions, all twenty students were invited to participate in the symposium and have their paper printed in the proceedings, whether they were presenting or not. By including all the students, we hope to foster a community of research students, and to continue interaction beyond the conference itself.

I would like to thank the members of the doctoral symposium panel for their work in reviewing the students abstracts, and for participating in the symposium and providing feedback to the students. The panel members were Perry Alexander (University of Cincinnati), John Penix (NASA Ames), Michael Lowry (NASA Ames), and Louis Hoebel (GE Research & Development Center).

Steve Easterbrook
Doctoral Symposium Chair
October 6, 1998

Doctoral Symposium Program

8:30 Welcome and introductions

9:00 Rogelio Adobbati, “Software Understanding through Automated Visual Presentations”

9:40 Peter Krogsgaard Jenson, “Automated Modeling of Real-Time Implementation”

10:20 Break

10:40 Kim Mens, “Towards an Explicit Intentional Semantics for Evolving Software”

11:20 *Reflections from a recent PhD “if only I had known...” – by Dr John Penix*

12:00 ***Lunch***

13:00 Jeffrey Smith, “UML Formalization and Transformation”

13:40 *“How to write a PhD thesis” – by Dr Steve Easterbrook*

14:20 Discussion Session.

15:00 ***Break***

15:20 Judy Stafford, “Dependence Analysis for Software Architectures”

16:00 Tarja Systa, “Dynamic Modeling in Forward and Reverse Engineering of Object-Oriented Software Systems”

16:40 Discussion and Wrap-up Session

17:00 ***Close***

Software Understanding through Automated Visual Presentations

Rogelio Adobbati
Computer Science Department / Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292-6695
+1 310 822 1511
rogelio@isi.edu

Abstract

It is well known that visual presentations can facilitate the understanding of software. However, effective visual presentations can be difficult to generate and maintain. In this paper I describe my work on PESCE [Presentation Engine for Software Comprehension and Explanation], a system that addresses this problem via automatic generation of visual explanations of software. The system uses a model of what the user knows about the system, the user's task, and a set of visualization rules to build consistent visual presentations about software objects.

1. Introduction

The importance of visual representations in understanding complex systems has been well established [19]. In the particular case of software systems, conceptual visualization becomes critical due to the absence of physical parts. As such systems grow in complexity, textual explanations get more difficult to understand; here is where graphical representations prove their worth.

Static, predefined diagrams have been used as documentation for complex software systems. The dynamic nature of software, and the different characteristics of users trying to perform software understanding conspire to lessen the utility of static diagrams. Dynamically generated presentations are therefore highly desirable.

2. The problem

The task of generating presentations can roughly be broken down into two steps: select what information to show (*content selection*), and how to show it (*presentation generation*) [22]. The focus of my work is the latter step; I am not investigating the content selection problem, but instead make use of content selection developed by other members of our research project at ISI.

Dynamically generating presentations for software artifacts represents a difficult challenge. First of all, software artifacts are complex objects of arbitrary dimension [18], and software understanding requires the ability to understand these objects from different views and the ability to map between these views (multiple dimensionality) [12,15]. Moreover, any selected information about these artifacts needs to be tailored to fit different user levels of expertise and tasks [10]. The visual component adds extra complexity to the problem since the mechanisms for conceptual comprehension of graphical depictions are not well understood [17]. In addition, visually displaying the information introduces extra implementation constraints due to the limited amount of graphical resources available at any given time [11].

To address these problems, I have identified certain key components to automatically generate visual explanations of software systems [1]. These are:

- *Relevant information about software objects*
- *A model of user knowledge of the system and current user task*
- *A repository of visual presentation methods*
- *A presentation engine/planner to coherently apply those methods*
- *Heuristics and rules for the layout of visual presentations*

In order to generate a presentation for a particular user, the problem has been divided in two main stages:

- *Spatial layout* of the presentation (the diagram itself)
- *Temporal layout* (the animation and diagram view transitions)

The resulting presentation must show the information relevant to the user in a series of steps in correct logical order. Furthermore, it needs to comply with pedagogical prerequisites and user model restrictions, to achieve a series of communicative goals.

3. Related work

Several systems have been developed that address the problem of automatically generating visual presentations. Some of these systems deal with the visualization of mainly quantitative information in the form of tables, graphs, etc. (e.g. SAGE [13], BOZ [5]). These scientific visualization systems do not provide the adequate techniques to represent abstract relationships between concepts, a feature that proves critical in software understanding.

Other systems generate planned multimodal presentations from some underlying representation (e.g. WIP [3], COMET [8]). These systems have been successfully used to generate instructions for technical devices, a task that is similar to the explanation of software artifacts. On the other hand, they have not been used in the domain of software engineering, and it is not clear that they could provide the multiple integrated views needed for a clear understanding of conceptual relations between software objects.

Stasko's work to visualize program execution through automated animations provides tools for understanding and debugging programs (LENS[14], GROOVE[9]). Nonetheless, the presentations generated are designed towards program debugging; their focus has not been to provide high-level, abstract visualization of the different components of a complex software system.

Another example is the RIGI system [15], a visual software understanding tool that provides different conceptual views of complex software systems. It does not, however, allow different conceptual views of a complex system to be shown at the same time and, more importantly, does not provide the display over time of multiple diagrams and animated presentations.

Zhou and Feiner's IMPROVISE [23] is a knowledge-based system that can automatically generate coherent visual discourse using a top-down, hierarchical-decomposition, partial-order planner. Their approach seems to be suitable to be applied to the software understanding problem, even though they have not tried such an application.

4. Direction of my work

To provide a solution for the presentation generation problem, I am currently working on PESCE (***P***resentation ***E***ngine for ***S***oftware ***C***omprehension and ***E***xplanation). PESCE is a component of MediaDoc [7], a software engineering tool being developed at ISI that uses both textual and graphical presentations for software explanation. I have completed a first implementation of PESCE that includes a few simple visualization rules and presentation methods to generate software explanations for different users.

I am also developing a formal framework for solving the spatial and temporal problems mentioned in section 2. For that purpose, I am investigating what characteristics make a visual presentation clear and useful; this is a difficult task since visual representation has been traditionally more of an art than a science. Nevertheless, I have been testing different rules for proper presentation generation based on the work of Tufte [19, 20, 21], Albers [2], and Bertin [4], and specific methods for graph layout inspired by [16, 6].

5. Current implementation of PESCE

The core components of PESCE are a repository of visualization rules for software objects and relationships, a presentation engine that applies those rules to generate visual directives to display some given information about a software system, and a diagram generator that realizes those directives on the user's screen (see figure 1).

5.1 The presentation engine

The main component of PESCE is the presentation engine; its first implementation is written in Perl. This module receives relevant content information from MediaDoc's explanation engine in response to a particular user query. That information is in a SGML-like format that can be easily parsed into individual objects and relationships; it also has the advantage of making it easier to interface PESCE to other software engineering systems besides MediaDoc.

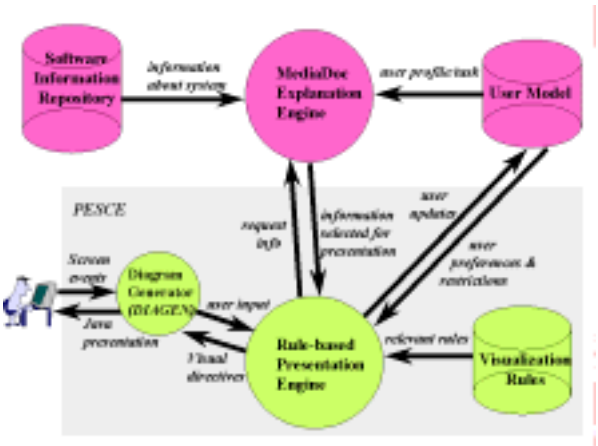

Figure 1: The MediaDoc Architecture

From the information given, PESCE builds a data structure that will be searched to solve the presentation problem. For each object (or relationship) type, a list of

visualization methods and their constraints is retrieved from the rule repository. An element is added to a working memory structure containing descriptive information about the object, the list of methods that may be used to visualize it, and the constraints inherent to each of the methods.

Besides constraints related to presentation methods, PESCE relies on several global constraints to tailor a visual presentation to the current user. MediaDoc's user model is accessed by PESCE and the appropriate global constraints are pushed into a constraint stack, e.g., color-based coding should be avoided when generating diagrams for color-blind people, etc. The user model stores a set of global constraints for each particular type of user and task; this information is represented in an SGML-like format, giving PESCE the potential to be easily interfaced to different user models outside of MediaDoc.

Once this linked data structure is completed, it has to be traversed to generate the visualization of all its components, paying special attention to their constraints. This is done through a forward-chaining mechanism that backtracks when a method sets a conflicting constraint. After all the elements in the structure are realized successfully, the resulting set of visual directives is specially formatted and sent to a diagram generator (Diagen) for graphical display.

In choosing a traversal sequence, a heuristic is needed to try to minimize backtracking. I have chosen a heuristic based on the complexity of the data elements and their connections; in my experience, more densely connected objects will lead to more spatial constraints, which are difficult to resolve once other constraints related to several simpler objects have been instantiated. By instantiating the more complex objects first, the level of backtracking is reduced up to the simpler ones. The content selection system can override this heuristic by providing a particular order of rethorical importance for the software objects to be visualized, in which case the given order is used to guide the search.

5.2 The visualization rules

Visual rules are used by the presentation engine to generate graphical representations of an object or relationship (figure 2). Each rule has 3 main components:

- The object/relationship type it realizes
- The presentation method to be called in order to display the object/relationship (including any required arguments)
- One or more links to spatial (size, position), temporal (order, duration), or style constraints for the presentation method

Methods that have been implemented already or that are in the process of being implemented include: null(), circular_node(), square_node(), star(), arrows(), network(), state_machine(), venn_diagram(), top_down(), animated_message(), nested(), and animated_sequence(). A fair amount of software explanations can be generated from this set of methods, including data flows, control flows, software dependencies, general architecture of a system or part of it, functional diagrams, message passing between objects, state diagrams, object and class hierarchies, etc.

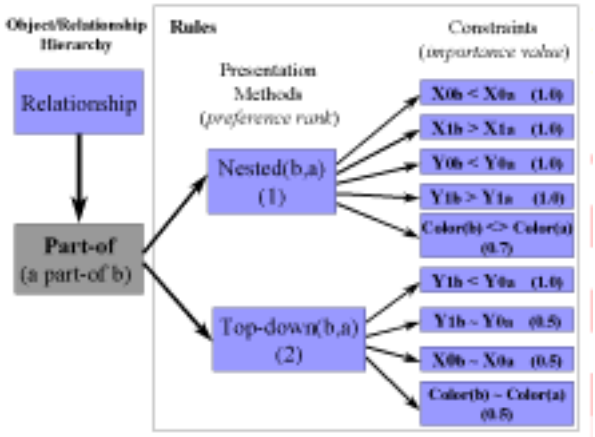

Figure 2: PESCE's internal representation of the Part-of relationship and two of its visualization rules

Rules and constraints have values related to them to help in the traversal and backtracking process. Rules for a particular object type are ranked by some arbitrary style preference; during the forward chaining, rules are tried in the order assigned. Constraints have an associated importance value to select which one to relax when a conflict arises; it ranges from 0 (irrelevant constraint) to 1 (mandatory constraint). I currently treat that value as a binary (1.0 is mandatory, anything else is non-mandatory), since I am still investigating a principled way to assign the right values to non-mandatory constraints.

5.3 The diagram generator

The Diagram Generator, or Diagen, is a Java applet that represents objects and relationships through a graphical layout on a web page. Diagen is used to provide a graphical element to MediaDoc through PESCE, but has also been interfaced to several other packages.

The diagram is generated from an SGML-like description (MAP – *M*arkup language for *A*uthoring *P*resentations-) provided by the applet server's machine. The applet requests the file from the server (PESCE in MediaDoc) and interprets the MAP description to create and display the diagram at the client site. Foreground objects and actions can be specified on top of the graph to create time-sequenced animations; the user can

interactively control these animations.

6. Future work and conclusions

I am building a system that automatically generates a series of visual representations to form a coherent explanation about the components of a software system and their underlying relationships. For that purpose, I have integrated into the system a user model and a diagram visualization tool, both developed for MediaDoc, a set of visual rules derived from current literature, and an algorithm to instantiate these rules and check their inherent constraints. I have been testing several examples of visual presentations in response of user queries about a software system.

Currently, I am working on defining a larger, more general set of visual rules to address a wider range of software visualization cases, and on testing different heuristics to efficiently instantiate those presentation rules and their corresponding constraints for every object and relationship. I am also trying to define an evaluation plan to measure the usability of the visualization rules, and the scalability of the presentation generation algorithm.

References

1. R. Adobbati, Towards the Automated Generation of User-Tailored Visual Representation of Complex Software Systems. Poster Presented at the California Software Symposium, University of California, Irvine, CA (1997).
2. E J. Albers, Interaction of Color. Yale University Press, New Haven, CT (1975).
3. E. Andre, W. Finkler, W. Graf,, T. Rist, A. Schauder and W. Wahlster, WIP: The Automatic Synthesis of Multimodal Presentations. M. Maybury, ed., Intelligent Multimedia Interfaces, AAAI Press, Cambridge, MA (1993) 75-93.
4. J. Bertin, Semiology of Graphics. University of Wisconsin Press, Madison, WI (1983).
5. S.M. Casner, A Task-analytic Approach to the Automated Design of Graphic Presentations. ACM Transactions on Graphics 10(2), (1991) 111-151.
6. G.Di Battista, P. Eades, R. Tamassia, I. G. Tollis, Algorithms for Drawing Graphs: an Annotated Bibliography. Journal of Computational Geometry (1994).
7. A.Erdem, W.L.Johnson and Stacy Marsella, Task Oriented Software Understanding. To be presented at ASE98, Hawaii (1998).
8. S. Feiner and K. McKeown, Automating the Generation of Coordinated Multimedia Explanations. IEEE Computer 24(10), (1991) 33-41.
9. D. Jerding, J. Stasko, and T. Ball, Visualizing Interactions in Program Executions. Proceedings of the 1997 International Conference on Software Engineering (ICSE-97), Boston, MA, May 1997, pp. 360-370.
10. W.L.Johnson and A.Erdem, Interactive Explanation of Software Systems. Automated Software Engineering 4, (1997) 53-75.
11. Z. Kulpa, Diagrammatic Representation and Reasoning. Machine GRAPHICS & VISION 3(1-2), (1994) 77-103.
12. Lakhotia, Understanding Someone Else's Code: Analysis of experiences. Journal of Systems and Software 20, (1993) 93-100.
13. V. Mittal, S. Roth, J. Moore, J. Mattis and G. Carenini, Generating Explanatory Captions for Information Graphics. Proc. of the Fourteenth IJCAI, Montreal, Canada (1995).
14. Mukherjea, Sougata and J. Stasko, Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger. ACM Transactions on Computer-Human Interaction 1(3), (1994) 215-244.
15. H. Muller, K. Wong And S. Tilley, Understanding Software Systems Using Reverse Engineering Technology. Colloquium on Object Orientation in Databases and Software engineering, The 62nd Congress of "L'Association Canadienne Francaise pour l'Avancement des Sciences", Montreal, Canada (1994).
16. K. Ryall, J. Marks, and S. Shieber, An Interactive Constraint-Based System for Drawing Graphs. Proc. of UIST 97, Banff, Alberta (1997) 97-104.
17. M. Scaife and Y. Rogers, External Cognition: How Do Graphical Representations Work? Int. Journal on Human-Computer Studies 45, Academic Press Limited (1996) 185-213.
18. M. Petre, A.F. Blackwell and T.R.G. Green, Cognitive Questions in Software Visualization. J.Stasko, J. Dominguez, B. Price and M. Brown, eds. Software Visualization: Programming as a Multi-Media Experience, MIT Press (1997).
19. E.R. Tufte, The Visual Display of Quantitative Information. Graphics Press, Cheshire, CT (1983).
20. E.R. Tufte, Envisioning Information. Graphics Press, Cheshire, CT (1990).
21. E.R. Tufte, Visual Explanations. Graphics Press, Cheshire, CT (1997).
22. M. Vossers, Automatic Generation of Formatted Text and Line Drawings. Master's Thesis, University of Nijmegen, The Netherlands (1991).
23. M.X. Zhou, and S.K. Feiner, The Representation and Usage of a Visual Lexicon for Automated Graphics Generation. Proc. IJCAI '97 (1997 Int. Joint Conf. on AI), Nagoya, Japan, (1997).

Odyssey: A Reuse Environment Based on Domain Models

Regina M. M. Braga Cláudia M. L. Werner Marta Mattoso
{regina, werner, marta}@cos.ufrj.br
Computer Science Department- COPPE/UFRJ
Federal University of Rio de Janeiro – Brazil

KeyWords: Reuse Software Development Environments, Component-based development, Domain Analysis, Object-Orientation, Software Architecture, Frameworks, Patterns, Mediators.

1. Introduction

Reuse is a promising way to help improving software development. One of the most encouraging reuse techniques available is the component-based software development. The component-based software development employs interrelations between preexisting components and the reuse of components that have been exhaustively tested to reduce complexity and costs of software development [10].

To meet this requirement, reuse must be applied to all phases during the development process. Therefore, the domain concepts that were considered as reusable in the initial development phases must be closely related to the code components that will be used during application implementation. A reuse environment based on abstract domain concepts can help in the effective application of reuse during software development, since it can provide methods, tools and procedures for the specification of domain models and applications. There is no environment to our knowledge that is capable of addressing all these aspects together. The works found in the technical literature [9], [12], [13], generally concentrate on one aspect or another.

In order to increase the productivity and reduce the cost of software development, we propose a reuse environment, named Odyssey [17]. The main feature of Odyssey is its ability to encompass the whole cycle from conceptual models to component implementation.

The main contribution of the Odyssey environment is the combination of concepts found in component-based development and domain engineering.

2. An Overview of Odyssey

The main goal of Odyssey is to provide mechanisms for software development based on the concept of reuse. To attain this goal, Odyssey has been conceived as a framework where *conceptual models*, *software architectures* and *implementation models* are specified for previously selected application domains. These domain models are specified and further modified according to the activities defined in the DE method, based on the DE process. To accomplish this, we use *domain agent tools* and *domain models specification and evolution tools* for specification and knowledge evolution of domains. The domain models are presented to users using a hypermedia interface. Also, all domain models should be stored in a distributed and heterogeneous way using the mediation technology.

The main users of the environment are the domain engineer, the domain specialist and the software engineer responsible for the development of applications within that domain. The domain engineer and the specialist use the environment mainly to specify and enlarge the concepts of the domain. The software engineer uses it to gain an understanding of the application domain and to reuse this understanding in the specification of his/her application. The software engineer interacts with Odyssey through the *Information Agent* – that aids the him/her while getting familiar with the domain – and through the *Architectural Transformation Agent* – which allows the transformation of the initial domain concepts, selected by the information agent, to a specific architectural model. These tools use the services of the mediation layer to access the domain models stored at the domain sources. The mediation layer plays a key feature in this environment since it provides a uniform representation and manipulation for all the domains, therefore facilitating the encompassment of the whole development cycle. The main objective of this layer is to allow the integration of information from various domains that are stored in heterogeneous and distributed data sources, in a way that the user of this information has access to it in a transparent and uniform way. In this aspect, Odyssey presents an advantage when compared to similar structures, such as KBSs, that generally use file systems to store data, resulting in redundant information storage and poor performance.

3. Representation of Domain Models

3.1 - Representation of conceptual models

In the representation of conceptual models, we must pay special attention to the understanding and recognition of concepts and functionalities by the Odyssey users. Thus, the form of expressing the domain concepts and functionalities are important. The functionalities are important because they are the base for reusable components creation. The domain concepts are equally

important since they provide users with an understanding of the domain as a whole, besides facilitating the understanding of the interaction between reusable components internal types[1]. The main conceptual models used by Odyssey are:

- Domain Context Diagram: The context diagram situates the domain in relation to its scope, limits, relationships with other domains and main actors involved. Its objective is to provide a general overview of the domain and situate it in the organization context;
- Domain Use Cases and related OO models: Shows how the domain concepts are represented in domain applications. Use Cases are used mainly to capture the main functionalities of the domain in a way that should be possible to derive other OO models. Use Cases can also help in the identification of the reusable components that are described in more detail in other OO models (class diagrams, interaction diagrams, etc). Several use cases are created in a domain, some are generic enough and others are specific to certain domain applications, many are very similar to each other, other may have some inconsistencies. So, it is necessary to abstract and merge these use cases, generating the domain use cases.
- Feature Diagram: presents, in an abstract level, the relationships among the functionalities and concepts of the domain, trying to explain what are the meanings of the main concepts of the domain and its relationships, that facilitates the understanding of the domain as a whole, because the OO (types) models related to domain use cases don't provide this general vision. It only provides a snapshot of the domain. However, for a complete understanding of the domain concepts, its synonyms, restrictions, among others, the feature model by itself is not enough. Thus, it is necessary that we have some construction that permits the linking of these terms and other related issues. This complete understanding is an essential characteristic to the development of domain applications. In Odyssey we use, for this detailed description of the domain concepts, a structured template that describes the domain concepts in more detail. This structure is denominated Ontological Pattern. It is also important to point out that all these models are connected through a trace relationship, i.e., if the user is examining a certain model, this model has connections with the other models that describe the same subject, thus the user can examine the other related models. Odyssey provides automated support for this " traceability ".

3.2 - Representation of architectural models

We also use the concept of patterns in the representation of architectural models. The generic architectures – the architectural styles that are relevant for the domain – are represented by structures similar to Bushmann's [2] architectural patterns and also by Gamma's design patterns [7].

Based on the conceptual domain use cases, OO models, and on the advice given by the architectural and design patterns, the architectural diagrams are composed. The main characteristic of this model is that it is partitioned by components. Each component specifies a domain task and how this task can be architected. The connections between the components are also described. So, the main architectural models are:

- Services (interfaces) Model of the components: This model presents each component as a type that possesses a series of services that are visible by the other components.
- Architectural Collaboration Model between domain components and support components: This model is mainly worried with the definition of a global architecture of the domain, including the interaction among the components of the domain and support components that deal with issues such as persistence, distribution, parallelism, among others. These support components can be acquired by vendors and could be shared by several applications of several domains.
- Classes model and state diagram for each participant type of components: the definition of the internal component design deals with the definition, in greater detail, of the internal structure of each component. All the conceptual collaboration models related to the component are refined. This stage tries "to improve" the conceptual collaboration models in the architectural sense, taking into consideration performance and optimization issues, among others. For that, new types can be added to the components and the modeling of the types can also be modified for the production of more robust, flexible and extensible models. For this, a base of design patterns can be consulted " to improve" the modeling of component internal types. In Odyssey, a tool that uses Case-Based Reasoning (CBR) to aid in this activity is used.

3.3 - Representation of implementation models

This model is formed by a set of code components that are related based on a CORBA protocol. The components are more general, but they can be specialized by using techniques such as parametrization, class specialization, etc. We use a CORBA protocol for the interoperability between components. For that, we have two strategies that can be followed: i) codification of components in an OO programming language; ii) use of legacy components.

4. Specification and Use of Domain Models (Domain Engineering)

Along with the representation of domain models,

[1] We used the type notion, as it is proposed by the OMG and ODMG models, as a reference to an object or class of an OO model. The type is the object in a high abstraction level.

Odyssey must provide tools that allow users to specify and use these domain models. In this sense, tools for eliciting requirements, pattern management, reusable components management and others should be provided.

We briefly describe below each tool used in Odyssey:

- The pattern and component management system are generic tools that are responsible for the creation, deletion and modification of the pattern and components. The components could also be generated by some type of automatic code generators.
- Requirements Elicitation Tool: This tool is responsible for the acquisition of domain information. The information can be knowledge of domain expert, domain documentation and domain applications (legacy domain applications must first be submitted to a reengineering process). The main model that is used in the acquisition process is the use case model. The use case template will guide the acquisition, helping in the organization of the other models. The domain engineer helps organizing the information in a better way.
- Information Agent Tool: This tool serves as a guide to the search for specific domain information. The models are interrelated as a hypermedia web. When the user notifies some interest in a concept, the tool seeks the related concepts and other related information such as use cases, OO models, etc. Thus, besides using a hypermedia interface, there should be a way to dynamically guide the navigation. This guidance should show the best paths for navigating and the type of knowledge that better suits the needs of the user. It should be based on the initial requirements of the user[2]. This tool uses the concept of intelligent link, where invoking the link provides additional knowledge (as in rule-based expert systems or a set of related cases) embedded within the information space to guide the selection of destination data. New data can be added to the domain repository and the intelligent link will be capable of referencing the new data. In addition, the intelligent links can be invoked based on the user objectives.
- Architectural Transformation Tool: This tool, considering the advice of architectural and design patterns, helps in the transformation of the conceptual models to architectural components models. Once more, the technique used is CBR. Despite of CBR use, the software engineer has an active role in the transformation, since this is not a trivial taks.

5. Using a Mediator Layer to Store Domain Models

One of the key questions in our project is how to enable the management of domain models, according to the activities defined in the DE method, in an integrated and efficient way. Therefore, for the effective implementation of the technologies involved in the specification of Odyssey, we need a component that allows for the integration of concepts, preserving the semantics. However, what we can notice is that, in general, the information is stored in a great variety of data sources, using the most varied data models, access mechanisms and platforms. Further, most times, the domain information is distant geographically, resulting in a difficulty on its manipulation.

Thus, a possible solution of access to the domain information is the use of a software layer that allows the integration of different domain databases (distributed and/or heterogeneous). A mediation technique [2] may be used in this case. Mediators are programs that make the connection between distributed data bases, heterogeneous data models, and the users of these data, providing the information in an adequate format to the user.

In the context of Odyssey, where the access to domain information is an essential requirement, the use of mediators allows that this access to information can be carried independent of the format and the operational platform where this information is stored.

Another interesting feature of the mediators use in Odyssey is that the reusable information is naturally divided by domain, which facilitates the search for domain concepts, since only the domain data will be accessed in a search. Moreover, the use of mediators in the environment context allows the aggregation of information already stored in legacy databases, without the necessity of transformations in the original database format. In order to facilitate the correct choice of mediators for a given domain, the Odyssey mediation layer provides specific ontologies for each domain. These ontologies are specified by domain specialists [18], facilitating thus the searching for specific components, since the ontology definition is directly connected to domain specific concepts. This structure conforms to the Wiederhold latest idea of intelligent mediators partitioned by domains [2].

6. Related Works

Related works can be found in the technical literature that have something in common with ours. Nonetheless, most of them deals with only a few aspects of Odyssey. None of them treats with the same emphasis each one of the several activities and technologies that are important for the development of component-based software, as we do in our work.

Regarding the specification of environments to support component-based software development, there are some interesting approaches. The domain modeling in [5] uses AI techniques. By comparing this work with Odyssey, we observe that it is mostly concerned with the representation of conceptual models. However, no attention is paid to the description of a detailed method for structuring its knowledge base. Moreover, all the design environments reported until now are specific to predefined domains, and cannot be used to store knowledge about other application domains.

The work of Gomaa et al [9] focus on the creation

[2]Since this kind of guidance is directed to application developers.

of a reuse environment based on the automation of its DE method, the EDLC (*Evolutionary Domain Life Cycle*), thus creating a generic environment named KBSEE (*Knowledge Based Software Engineering Environment*). KBSEE has some points in common with our work, such as: the adoption of a method to systematize the domain engineering; and the specification of domain models in various abstraction levels, using mainly the object-oriented paradigm. However, when we consider the aspect of data storage, KBSEE requires transformations between different representations and databases. This results in redundant information storage and poor performance. Moreover, the semantic gap is increased. In this aspect, our proposal differs from Gomaa's, since it is based on the use of a generic and standard model to store domain models. This standard is compatible with UML, using the structure of mediation layer, that leads to a better performance.

7. Final Considerations

In this work, we presented some requirements to support CBD, through the specification of a reuse environment based on domain models (Odyssey).

The implementation of Odyssey environment is a great effort. Therefore, there are several people who are involved in this project (i.e., two PHD students, three master students, and one undergraduate student). Currently we have an operational prototype of the environment that provides some basic functionalities, such as an OO diagram editor, designed specifically to deal with Odyssey models and their traceability, and a tool that helps to configure DE acquisition process to specific projects. The mediation layer is under development and the information agent tool is also being specified.

Odyssey environment brings some interesting contributions, mainly in the following points:

- Identification of technologies and specification of components capable of addressing various stages involved in CBD;
- A DE method to support all phases of the process, including a viability analysis stage, the purpose of which is to validate the viability of applying model oriented reuse in that domain;
- Systematic use of high-level OO constructs, such as patterns, and its insertion into a DE method;
- Systematization of the transition between conceptual and architectural models.

Approaches that are similar to ours [5] [9] [8] [12], although presenting concrete results, do not specify all the aspects addressed by Odyssey when supporting the component-based development. Such proposals present results in certain aspects of the component-based development, but these results are isolated from a wider context. The innovative approach of Odyssey reduces the semantic gap between the specification and the software development. Therefore, with the help of Odyssey, the software developer is able to apply reuse techniques from the early stages of the application development process. The other approaches instead, generally put emphasis on just one of the phases of the process.

References

[1] Bosh, J. - Reusable Specification of Architectural Fragments - University of Karlskrona/Ronneby, Sweden, 1997- at http://www.pt.hk-r.se/~bosh

[2] Buschmann F. et al - Pattern-Oriented Software Architecture - A system of patterns - John Wiley, 1996

[3] Cohen, S; "Feature-Oriented Domain Analysis: Domain Modeling", Tutorial Notes; 3rd Int. Conference on Software Reuse, Rio de Janeiro, November 1994.

[4] Fayad, F., Douglas Schmidt, Object-Oriented Application Frameworks, Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[5] Fischer, G. - "Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments", IFIP WG 8.1/13.2 Joint Working Conference, A. Sutcliffe, D. Benyon and F. van Assche (eds): "Domain Knowledge for Interactive System Design", Chapman & Hall, pp 1-16, May 1996.

[6] Fowler, M. - Analysis Patterns - Reusable Object Models - Addison Wesley, 1997

[7] Gamma E. et al - Design Patterns: Reuse of Object Oriented Design Addison Wesley, 1994

[9] Gomaa, H et al - A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures - Automated Software Engineering 3(3/4): 285-307, August 1996

[10] Jacobson, I.; Griss, M.; Jonsson, P. , "Software Reuse: Architecture, Process and Organization for Business Success", Addison Wesley Longman, May 1997

[11] Klingler, C. D.,Schwarting, D. - A Practical Approach to Process Definition, Proceedings of the Seventh Annual Software Technology Conference, Utah, April 1995

[12] Lowry, M., Van Baalen J., "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems", Automated Software Engineering, 4, pp. 199-241, 1997.

[13] Euzenat J., Corporate Memory through cooperative creation of knowledge based and hyper-documents, Proocedings of KAW'96, 1996

[14] Object Management Group Adopts Unified Modeling Language and Meta Object Facility Specifications at http://www.omg.org/news/pr97/umlpr.htm

[15] Tracz, W., Batory, D. David McAllester, Lou Coglianese, Domain Modeling in Engineering of Computer-Based Systems. In Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tucson, Arizona, February 1995.

[16] Studer, R.; Angele J. Fensel D.: Domain and Task Modeling in MIKE. In: A. Sutcliffe, D. Benyon, F. van Assche (Eds.): Domain Knowledge for Interactive System Design, Proceedings of IFIP 8.1/13.2 Joint Working Conference, Geneva, May 1996.

[17] Braga, Regina; Werner, Claudia; Mattoso, Marta – A Reuse Infrastructure Based on Domain Models, Proceedings of ICCI'98, Canada, June, 1998

[18] Oliveira, K; Towards a Domain-Oriented Software Development Environment for Cardiology, Proceedings of CaiSE'98, Italy, 1998

Integrating Automated Verification into Interactive Systems Development

José C. Campos*
HCI Group, Department of Computer Science, University of York, York, UK
Jose.Campos@cs.york.ac.uk

Abstract

Our field of research is the application of automated reasoning techniques during interactor based interactive systems development. The aim being to ensure that the developed systems embody appropriate properties and principles. In this report we identify some of the pitfalls of current approaches and propose a new way to integrate verification into interactive systems development.

1. Introduction

The widespread use of computers puts increasing demands on user interfaces. On the one hand, systems must be intuitive and easy to use, on the other hand they must ensure safety and avoid risk. Due to their increasing complexity, reasoning about systems behaviour has become increasingly hard. This raises the question of how to ensure quality during development.

The use of formal methods has long since been proposed as a solution to this problem. The advantages are two-fold: they enable better design understanding and communication; and mathematical reasoning can be used to validate the design. This last point is especially useful when we think of ensuring system quality, as it allows us to assess the system from early stages in the development process.

Because reasoning about specifications of complex systems will be a complex and error prone exercise in itself, ways of automating the reasoning process have been sought. Two well established approaches to automated reasoning are model checking [7] and theorem proving. While these techniques have been used mainly in the field of hardware verification [8], their application to the verification of reactive systems in general is also being studied [18].

Despite being a particular case of reactive systems, interactive systems have specific concepts and concerns. So, novel approaches have been sought. In this context, the notion of Interactor [12, 20] has been introduced as a way to structure specifications of interactive systems.

Our field of research is the application of automated reasoning techniques during interactor based interactive systems development. The aim being to ensure that the developed systems embody appropriate properties and principles.

2. Review

Four major approaches to the formal (automated) verification of interactive systems have been identified so far [4, 6]. Three use model checking: Abowd, Wang & Monk [1] use SMV [19], Paternó [20] uses the Lite tool-set [17], d'Ausbourg, Durrieu & Roche [9] use a model checking related technique based on Lustre; and one uses theorem proving: Bumbulis [3] uses the HOL theorem prover.

In order to better compare these approaches we have defined a framework with which to compare them [6]. It identifies three entities involved in interaction: *User*, *User Interface*, and an *Underlying System*. Interaction proceeds through interaction mechanisms: *Events* and *Status Phenomena* are atomic, *Task*, and *Mode* are used to structure the user interface. The framework identifies also three basic types of properties to be verified: *Visibility*, *Reachability*, and *Reliability*. Table 1 summarises the results of the review in terms of what each approach addresses (✓), partially addresses (∼), or does not address (×).

The conclusions drawn from the review are two-fold. At the technological level, it was seen that both model checking and theorem proving have difficulties when dealing with the added complexity introduced by interactive systems.

At the methodological level, there is a need to further investigate what should/can be proved of interactive systems using automated reasoning tools. Previous approaches have tried to map what could be expressed in traditional verification tools into the *interactive systems space*. In order to make the most of automated reasoning we must try to do the opposite: identify what properties are interesting and map them into automated verification tools.

If we combine the above two concerns, we can identify a third issue that needs addressing: when should we do the

*José Campos is supported by Fundação para a Ciência e a Tecnologia (FCT, Portugal) under grant PRAXIS XXI/BD/9562/96.

Table 1. Summary of the comparison

		SMV	Lotos	Lustre	HOL
Entities	Users	×	×	×	×
	User Interf.	∼	✓	✓	✓
	Underl. Sys.	∼	×	×	×
Inter. Mech.	Events	∼	✓	✓	∼
	Stat. phenom.	∼	×	✓	∼
	Modality	×	✓	×	×
	Task	∼	∼	×	×
	Mode	×	×	×	×
Prop.	Visibility	×	✓	✓	×
	Reachability	✓	✓	✓	×
	Reliability	✓	∼	∼	∼

proofs? — i.e., at what level of abstraction, and at what stage of development should we be working? Traditionally, verification has been used to assess design against absolute measures of quality. Regarding HCI, matters are not so clear cut. Furthermore, if we are using principled design, it would be useful to test the design decisions against the appropriate principles as soon as possible.

The challenge, then, is trying to make the best of the available verification technology by means of defining an appropriate methodological framework which will allow us to identify how and when verification should be applied.

3. The Thesis

In view of the complexity of the systems, and of the limitations of the available technology, the best approach to achieve the goals set forth above is to allow for a flexible scheme of verification. With this in mind we established the following objectives:

- non commitment to a specific technique — we want to be able to use model checking and theorem proving as appropriate, and not to be tied to a particular verification strategy.
- use of partial models — models that try to address all relevant aspects of an interactive system are too complex; instead, we want to use partial models, each model focusing on different design aspects (cf. [13]).

These two points, together with the observation that identifying (let alone proving) interesting properties in "finished" models becomes difficult, lead us to the realization that instead of being used as a *post facto* check on the quality of the specification, verification should be used to inform design decisions during development [5]. This can be done by using partial models that highlight the design features under consideration, and allows us to use the most appropriate verification technique for each model.

All the above leads to the definition of four lines of work:

- verification as a support to design — verification should be used to inform design decisions rather than to check the final design;
- understanding properties — we need to establish a framework that enables us to reason about how to go from design principles to verifiable properties;
- model checking for interactor specifications — we need to determine how model checking can be applied to interactor based specifications;
- theorem proving for interactor specifications — similarly, we need to determine how theorem proving can be applied to interactor based specifications;

and the definition of the central proposition of the thesis as: *Formal verification techniques (automated reasoning tools in particular) can be used to inform design decisions during interactive systems development.*

A novel approach to the integration of automated verification into interactive systems development will be proposed, and it will be shown how model checking and theorem proving can be used in the context of the approach.

4. Progress

In this section, we briefly describe the work done so far.

4.1. The role of formal verification

We propose that verification should be used to inform design choices during development, and not only as a check on the correctness of the specified system. The complete rationale behind this proposal is presented in [5]. Some of the points that are made are: that the role properties play depends not only on the system under consideration, but also on the particular specification that is adopted; that it is difficult to base design decisions on prescriptive theories alone, so the possibility of early assessment of design decisions would be useful; that seeing the verification step as a final step in the development process, and trying to use *off the shelf* properties, might lead us to end up looking at properties of the specification instead of the system; and finally, that the particular specification style adopted influences which verification tools can be used.

The use of verification to inform design can be achieved by using, not a monolithic specification which tries to encompass all of the system, but a set of models each focusing on particular features of the system. This type of approach has a number of benefits. Namely: we use verification to validate the choices that are made in relation to what is important of the system, not its specification; we are able to

apply the most appropriate verification technique in each case; conversely, we can develop each model in the most suitable way, regarding the tool that will be used; also, using models that focus on properties means we will be able to verify properties that otherwise would be too difficult to check; finally, we might be able to reuse the proofs when thinking of related properties of different systems.

4.2. Using model checking

We are exploring the use of model checking in the verification of interactor based specifications. This is being done at two levels: using a traditional model checker (SMV [19]), and using the μ-calculus model checker in PVS.

4.2.1 SMV

A compiler has been developed (see [5]) that enables us to analyse Interactors specifications in SMV. For an introduction to Interactors see [12]. In short, interactors are objects which allow their state to be perceived through some presentation (cf. **visible** clause below). Interactors provide a framework for specification and do not prescribe a particular notation.

In the present case, we are using Modal Action Logic (MAL) [21] to specify interactors behaviour. In the input language accepted by the compiler, an interactor describing whether a window is mapped on the screen looks like this:

interactor $window$
attributes
$mapped : boolean$
visible
$mapped$
actions
$map, unmap$
axioms
1. $[]\neg\ mapped$
2. $\neg\ mapped \Rightarrow [map]\mathsf{next}(mapped)$
3. $mapped \Rightarrow [unmap]\neg\ \mathsf{next}(mapped)$

Besides the clauses shown in the example, the interactor notation allows for three additional clauses: **importing** (allows inheritance), **fairness** (allows the definition of a fairness expression to be used by SMV), and **define** (enable us to give names to expressions as can be done in SMV). Multiple interactor specifications can be written by organising interactors in a hierarchy. In order to translate these hierarchies of interactors into SMV, we use the notion of module. So, each interactor will be a module in SMV.

To test properties of the specification, a further clause was introduced in the language: **test**. It is used to specify a CTL formula whose validity is to be verified by SMV.

In [5] it is shown how the compiler and SMV can be used to reason about different design possibilities in the development of an e-mail client.

4.2.2 PVS

PVS comes with a theory that defines the CTL operators in terms of the μ-calculus. Alternatively we can define temporal operators for other logics. In [6, Appendix C] we have defined the operators for ACTL.

In order to use the model checker, the specification needs to be structured as a predicate over pairs of states, where the state type must be finite. We can then use the temporal operators to write putative theorems. PVS performs BDD simplification over the finite-state machine defined by the predicate over pairs of states, rewrites the temporal operators in terms of μ-calculus, and runs the resulting state machine and μ-calculus predicate in the model checker.

The present approach to model interactors and properties in this way is still tentative. We plan to expand on it in order to explore how the combination of theorem proving and model checking can be used to enhance the analytic power of both techniques.

4.3. Using theorem proving

While theorem provers do not have facilities to perform temporal reasoning, they are better than model checkers when it comes to reasoning over more information oriented features of systems. At the moment, three possible uses for theorem proving are envisaged: the validation of the adequacy of perceptual operators as suggested in [11] (see below), using it as an additional layer over model checking, and embedding a temporal logic in PVS (c.f. [16]).

4.3.1 Perceptual operators

The type of analysis described in [11] has to do mainly with symbolic manipulation of expressions in order to prove their equality. The basic idea is that properties that are proved of an abstract specification must also be shown to held at the level of the concrete presentation of the system. To do this we represent both a model of the abstract specification of the system and a model of the concrete presentation that is proposed as PVS theories, and then use PVS to determine if predicates over the abstract model are equivalent to corresponding predicates over the presentation model.

In [10] we apply this line of reasoning to the analysis of an aircraft air speed indicator, regarding its fitness to assist the pilot in the task of maintaining the correct aircraft configuration during landing (cf. [15]). Three PVS theories were developed. One to specify the logical model of the air speed indicator as well as the logical operators that support the task; another to specify the concrete circular air speed indicator, with its needle and speed bugs (which indicate at which air speeds the aircraft configuration should be changed), and the mapping from the logical to the perceptual level; and finally a third theory which introduces the

conjectures to be verified. As an example we present here one of the conjectures which is analysed in [10]:

$$\begin{aligned}&\mathsf{configuration\_change\_task}: \text{ CONJECTURE}\\ &\mathsf{configChangeCheck}(\mathsf{abs\_asi}) =\\ &\mathsf{asiConfigCheck}(\rho(\mathsf{abs\_asi}))\end{aligned}$$

What this conjecture expresses is that checking for the need to change the aircraft configuration should yield the same result regardless of the check being done at the logical level (configChangeCheck) or at the perceptual level (asiConfigCheck). ρ is the mapping from the logical to the perceptual level.

In [10] we show how performing this type of consistency check improves our understanding of the specification, and allows us to identify assumptions about the system which are embedded in the representation but not made explicitly represented anywhere. As an example, during the proof of the conjecture above, we were led to realize how, at the presentation level, the speed bugs implicitly acquire the function of indicating the aircraft current configuration.

5. Conclusion

We have motivated the field of formal (automated) verification of interactive systems, and identified the main approaches to the area (see [4] for a more detailed review).

We have identified some of the pitfalls of the current approaches and proposed a new way to integrate verification into interactive systems development.

We have also briefly described the work done so far (see also [6, 5, 10] for more details).

References

[1] Gregory D. Abowd, Hung-Ming Wang, and Andrew F. Monk. A formal technique for automated dialogue development. In *Proceedings of the First Symposium of Designing Interactive Systems - DIS'95*, pages 219–226. ACM Press, August 1995.

[2] F. Bodart and J. Vanderdonckt, editors. *Design, Specification and Verification of Interactive Systems '96*, Springer Computer Science. Springer-Verlag/Vien, June 1996.

[3] Peter Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, 1996.

[4] José C. Campos and Michael D. Harrison. Formal verification of interactive systems: A review. In Harrison and Torres [14], pages 109–124.

[5] José C. Campos and Michael D. Harrison. The role of verification in interactive systems design. In *Design, Specification and Verification of Interactive Systems '98*, Springer Computer Science, pages 155–170. Eurographics, Springer-Verlag/Wien, 1998.

[6] José Creissac Campos. Formal verification of interactive systens. 1st year qualifying dissertation, Department of Computer Science, University of York, June 1997.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[9] Bruno d'Ausbourg, Guy Durrieu, and Pierre Roche. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. In Bodart and Vanderdonckt [2], pages 105–122.

[10] G. Doherty, J. C. Campos, and M. D. Harrison. Representational reasoning and verification. In *Proceedings of the BCS-FACS Workshop: Formal Aspects of the Human Computer Interaction*, pages 193–212. Computing Research Centre, Sheffield Hallam University, September 1998.

[11] Gavin Doherty and Michael D. Harrison. A representational approach to the specification of presentations. In Harrison and Torres [14], pages 273–290.

[12] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[13] Bob Fields, Nick Merriam, and Andy Dearden. DMVIS: Design, modelling and validation of interactive systems. In Harrison and Torres [14], pages 29–44.

[14] M. D. Harrison and J. C. Torres, editors. *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science. Springer-Verlag/Vien, June 1997.

[15] E. Hutchins. How a cockpit remembers its speed. *Cognitive Science*, 19:265–288, 1995.

[16] Pertti Kellomäki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, 1997.

[17] José A. Mañas et al. *Lite User Manual*. LOTOSPHERE consortium, March 1992. Ref. Lo/WP2/N0034/V08.

[18] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[19] K. L. McMillan. *The SMV system*. Carnegie-Mellon University, draft edition, February 1992.

[20] Fabio Paternó. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995.

[21] Mark Ryan, José Fiadeiro, and Tom Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer-Verlag, 1991.

Automated Modeling of Real-Time Implementation

Peter Krogsgaard Jensen
Aalborg University
Department of Computer Science
Fr. Bajersvej 7E, 9220 Aalborg Ost, Denmark
pkj@cs.auc.dk

Abstract

This paper describes ongoing work on the automatic construction of formal models from Real-Time implementations. The model construction is based on measurements of the timed behavior of the threads of an implementation, their causal interaction patterns and external visible events. A specification of the timed behavior is modeled in timed automata and checked against the generated model in order to validate their timed behavior.

1. Introduction

When developing a Real-Time application it is a problem to obtain precise information about how much CPU time is needed to complete the jobs of the application. A widely used way to make schedulability analysis is to use an offline worst case execution time (WCET) calculation. However, when several processes (or threads) interact via shared data, this calculation often becomes extremely complicated. This problem has been addressed in [9] using a framework where the offline analysis is extended with some application dependent knowledge and use of priority inheritants protocol. The work described in this paper is directed towards automatic collection of application dependent knowledge, resulting in less manual (and error prone) work to do schedulability analysis.

Besides schedulability, the logical correctness is also important to Real-Time applications. A number of formal tools are already available to support the correctness analysis during the design phase of such applications, but there is still a gap between design and implementation - and this may cause human errors. One approach is automatic code generation, but often the formal method is used only for essential algorithms and to model parallel composition. This makes it impossible to auto-generate the complete implementation. The work described here attempts to bridge the gap from implementation to design, by automatic synthesis of a formal model, which incorporates the actual (measured) timing behavior of the application. The model can then be fully analyzed by existing automated tools like e.g. UPPAAL [5].

In this work, we suggest a semi-automated iterative way to attack the above problems: First an initial implementation is developed and instrumented with the logging of relevant events; then a series of runs are logged and three different models are synthesized - including a timing diagram; finally the models are analyzed by using an automated real-time model checker and timing errors are corrected in the next iteration. The corrections may be validated by a new iteration.

It is our plan to implement tool support for the above method at a prototype level and to evaluate its feasibility through realistic case studies. In the present paper we present an preliminary result, i.e. we present our event logging tool and the tool for generating timing diagrams. Also, we sketch how to derive the models to be used by the model checker, and we present the preliminary experiences on a non-trivial case study. The prototype tool does not support testing, but assumes that the log stream it experiences is sufficient for creating a complete model.

A result obtained with the prototype tool, is the automatic calculation of the average case execution time (ACET). ACET is calculated as the average of a set of execution times, between f.ex. job start and job end. The ACET therefore becomes a number for how much CPU a particular job needs. Another result is the deducting of timed behavior which is pictured in a timing diagram, called execution time graph (ETG), this is done to present an overview of the interaction pattern between threads. The ETG is in fact an annotated message sequence diagram, where both synchronous and asynchronous interaction is pictured. This is the current state of the prototype tool.

The work done in by Havelund, Skou & Larsen in [3] indicates that it is possible to verify time requirements in a single processor interleaved system, and in this way an alternative to offline schedulability analysis is obtained.

The work described here is based on a series of tests of an industrial process control application, performed on a single CPU system using the RT-Mach micro kernel. The soft- and hardware system is described in section 2. In section 3 is a description of the necessary analysis to generate the implementation model, which still remains to be fully defined. Finally, in section 4 is located a plan for the future work.

2. The RT Test System

The target system is a single CPU running an RT-Mach micro kernel extended with event logging on both kernel and user level. The software system consist of four parts: event logging subsystem, submarine test application with testbed, ACET analysis prototype tool, and the UPPAAL model checker. The event logging subsystem, the prototype tool and the UPPAAL model checker are application independent, and will analyze any instrumented application running on RT-Mach. Figure 1 shows the data flow in the software system, but before going through this the main components are described individually.

The event logging subsystem is a part of the RT-Mach micro kernel, and can log scheduling- and user-events with a time stamp local to the machine. The event logging subsystem is developed by the RT-Mach group at CMU, and has been used by several tools. The system has been customized by the author to connect it to the prototype tool. The logging is fast and best effort, but congestion and packet loss is handled by dropping affected sub-traces during analysis.

The submarine test application is a 4000 lines multi threaded program set. It is a small process control system, where an unmanned submarine is directed from a ship. The submarine system handles a variety of periodic jobs automatically, and it receives sporadic commands from an operator at the command bridge. The application has been instrumented with a small number of system calls for logging. *A testbed* controls the input to the application under test, making it possible to simulate different types of situations and errors in the submarine environment. This work is trying to improve test analysis, by automating critical tasks, therefore we assume that the application is put through a sufficiently thorough test. The log information we analysis is then assumed to come from such test. In out example the testbed is used to drive the application through a realistic series of runs, such that all types of jobs are executed, and the different input is impressed on the application many times with random intervals. We will not further elaborate on what a sufficient test is, as this is not the scope of the paper.

The ACET analysis prototype tool is used to observe the system. Job knowledge, originating from the log information, is used to deduct job-patterns and to book the time spent to the correct job. How to produce a formal implementation model is described in section 3.

The UPPAAL *model checker* is an automatic model checker working on timed automata (TA). It incorporates Real-Time clocks as well as discrete analysis. This highly specialized tool is described in [5].

In figure 1 the dashed line divide application dependent and independent parts of the system. The dotted line is the network boundary, where the left part is executed on the target computer, the right part is spread on the adjacent network. All arrows are dataflow. The application exchanges directions, commands, information and alive signals with the testbed. The testbed is controlled by an operator, either interactively or it can be programmed to operate automatically. Via the system calls made by the test application, the kernel generates logging events and ship these of the local host. The stream of log events are received by the prototype tool which can store, calculate and display information about the logging events received. The tool can run in both automatic and manual mode. In manual mode a designer can take interactive control and generate ETG diagrams, and job- and thread-level models. The job- and thread-models are combined with the selected platform model and an operator defined requirement model in UPPAAL. The complete model can now be checked against its requirements by the analyst which is interactive with UPPAAL.

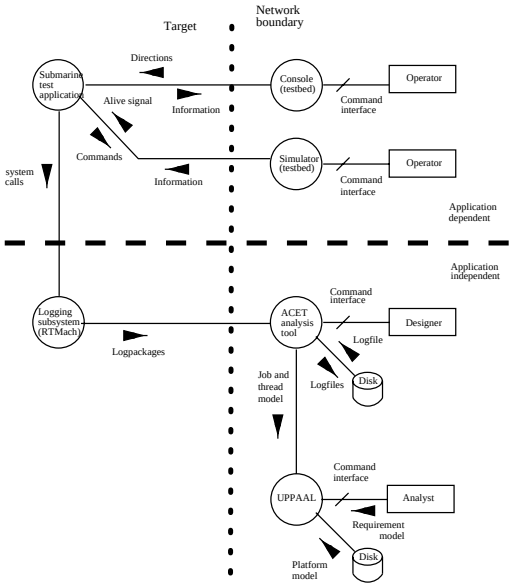

Figure 1. Dataflow in the experimental system.

3. Building the model

The complete model consist of a requirement, an implementation and a platform-model. The requirement model contains external observable events and their timing constrains. An implementation model has two levels: job- and thread-level. The job level maintains information about the period or mean arrival time (MAT) for each job executed during test. The thread level describes the ACET and causal interaction patterns. The platform level contains the scheduling algorithm if needed.

The practical analysis performed in the prototype tool is divided in two layers, job analysis where job behavior is described in the job model, and ETG analysis where threads and interaction patterns are described in the thread model. The requirement- and platform-model are more static and will be created manually, once for each application/platform.

3.1. Application assumptions

In order to make the analysis we must assume that the application is a set of threads each responsible for one or a set of clearly defined task(s) - like “listen on network”, “transmit on network”, or “do calculation A”. We also assumes that the application will solve a job, by using the same threads in the same sequence for each repetition of the job. Furthermore we assume that a thread, which uses a resource, will use the same resource for each repetition of the job. These assumptions enables us to view the work done by a Real-Time application as a set of skeletons, and the analysis described here will synthesis these skeletons. Further it will calculate how often a skeleton is used, how much CPU it consumes, and what resources it accesses.

3.2. Job Model

To describe the job behavior of an arbitrary Real-Time application, a connection must be established between the threads of the implementation and the specification defining the job requirements. This is done by instrumentation, such that a thread, during execution, will state which job it is working on, and further log important (external observable) events. A job trace is created when events are assembled from all the threads participating in the job execution. For each job type the job model must know the frequency, and it is found by calculating the period, or MAT and standard deviation from time stamping of the job traces. This is enough information to produce a job model which will reflect the series of runs the application experienced.

3.3. Thread Model

To describe each thread of the application, its ACET and interaction with other threads must be modeled. The ACET is needed to model the CPU consumption, and the interaction patterns between threads are needed because they will restrict the computation. From a job trace a skeleton of the interaction can be extracted, be examining the use of mutexes and semaphores, the message passing, and the IPC. All job traces with the same skeleton are concentrated into one ETG, using the ACET - in place of the WCET - as the measure for how much CPU a certain job needs. It is now possible to create an automaton for each thread (in the ETG), and the set of automata will describe the interaction of the threads when they are working for a certain job.

When this is done for all jobs in the application, the behavior of each thread is completely described, and the thread model will constrain the model checking such that only the implemented behavior is possible.

3.4. Model checking

To complete the description of our Real-Time system, a platform model is needed. It will be application independent, but must incorporate the scheduling algorithm. With this method it is possible to use different scheduling algorithms and even verify the implementation model on a non-existing platform.

The model checking is done on a requirement model, consisting of a set of timed automata which define the end-to-end time requirement with respect to the external observable events. Figure 2 shows an example model, where a sporadic event must be answered within 1.0 second. A question to the model is whether it is possible that MSG-OUT is not done before t equals 1.0 - or even worse is it possible that MSG-IN can happen without MSG-OUT happens afterwards.

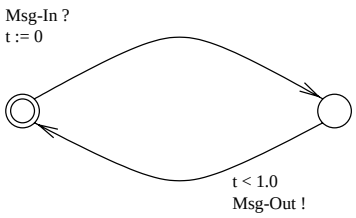

Figure 2. Requirement model expressed as a timed automaton for a time requirement where a sporadic event MSG-IN must be answered with MSG-OUT within 1.0 second. The implementation model is responsible for generating the matching events as the model is synchronous.

During model checking the job model is responsible for initiating jobs, the thread model restrict sequences of interaction, the platform model restrict CPU usage, and the requirement model defined the questions that must be examined. Finally it is left to the model checker to go through all allowed computations, and possible finding erroneous, or perhaps more efficient computations, that those actually seen during test.

4. Future work

Work is currently done, to automate the generation of the implementation model. The logging and analysis of traces is completed, while the interaction patterns remains to be incorporated. The logging subsystem must reveal detailed information about mutex access and the type of thread-to-thread call. In particular the thread-to-thread call is interesting because several different types of synchronous and asynchronous call/messages are possible. A plausible solution is to create a piece of middleware through which the applications must call to interact with each other. This enables an application independent logging.

Having seen that it is feasible to log information from a running Real-Time application, we must address the question of how our observation changes the original system. It is changed in two ways: extra code complexity during development of the application, and extra CPU cycles during execution. The overhead added to the design phase is small calculated as extra lines of code. The CPU overhead still remains to be measured, as we are still making changes to the RT-Mach kernel.

5. Acknowledgement

The described work is going on at Aalborg University under supervision of Professor Arne Skou.

References

[1] C. M. Chen Lee, Katsuhi Yosida and R. Rajkumar. Predictable communication protocol processing in real-time mach. *Proceedings of Real-Time Application Symposium*, 1996.

[2] D. Haban and K. G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transaction on Software Engineering*, 16(12), December 1990.

[3] K. Havelund, A. Skou, and K. G. Larsen. Formal verification of an audio/video power controller using the real-time model checker UPPAAL. *Work in progress*, 1998.

[4] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. *Proceedings of Real-Time Systems Symposium*, December 1995.

[5] K. G. Larsen, J. Bengtsson, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, October 1995.

[6] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[7] J. E. Sasinowski and J. K. Strosnider. Artifact: A platform for evaluating real-time window system designs. *Proceedings of Real-Time Systems Symposium*, December 1995.

[8] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(7), 1990.

[9] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1), January 1994.

[10] H. Tokuda and P. Rao. Real-time mach: Towards a predictable real-time system. *Proceedings of the USENIX Mach Workshop. Burlington, Vermont: The USENIX Association*, 1990.

[11] A. Wellings and A. Burns. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. ELSEVIER, Amsterdam, Netherlands., 1995.

Tool Support for Requirement Level Change Management and Impact Analysis

Author: Simon Lock (s.lock@lancaster.ac.uk)
Affiliation: Lancaster University, UK

Abstract

Change management is an important yet often problematic stage of the software development lifecycle. Even with substantial knowledge of a system, managing it's change and evolution is by no means straightforward. This is particularly true for requirement level entities which are by necessity expressed in an abstract manner. For this reason, most current research has concentrated on later design and implementation level artifacts where more concrete information is plentiful. This paper considers an approach and support tool for performing change management at the requirement level and focuses particularly on the identification and visualisation of change impact.

1 Introduction

It is generally recognised that the process of managing requirements change can be expensive and time-consuming [1]. Indeed, it has been shown that the largest proportion of requirement costs can often be traced to change management [2]. Current change management techniques have focused largely on design and code level artifacts, rather than requirements level entities [3,4,5,6,7,8]. The main reason for this is that the artifacts from the later stages of development are more concrete and provide developers with more information required for change management. Ignoring requirement change management often leads to systems that fail to meet the real business needs of the system procurer.

The main purpose of this work is to develop a requirement centred impact analysis technique that allows engineers to rapidly and accurately enact and assess proposed changes. In summary, this work aims to:

- Develop an interactive technique for visualising requirement change impact
- Investigate how previous change knowledge can be accommodated in the approach and used to inform on intended changes
- Develop a mechanism for adapting the technique to existing requirements engineering methods
- Produce tool support for the technique

2 The problem

Change is inherent in the development of most software systems. Software requirements change and evolve even as they are formulated. These changes can affect both the system function and the wider business goals of the organisation for which the software is being developed. For these reasons it is important that changes in requirements are carefully traced, analysed and their effects on the system operation, and the wider business goals properly assessed. Change results from the need to take into account new or altered requirements caused by the following factors:

- Changes in operational environment/application domain
- Improved client understanding of the domain or system
- The introduction or installation of the system into its operational environment
- Changes in the business objectives of the client

As a result of the large amount of information and complex relationships involved, tracking and assessing the effect of a requirement change can be expensive, time consuming and error-prone. Current techniques for change management have aimed to minimise the occurrence of change rather than accommodate it [9]. By delaying or "freezing" change out of the development process it is hoped that the problems of change assessment and integration can be avoided [10]. This has resulted in systems that do not adequately address real user needs.

An approach that integrates traceability extraction and visualisation may provide a powerful alternative to the current techniques for assessing the impact of change, a number of which are discussed in the next section.

3 Current impact analysis techniques

In order to perform impact analysis, extensive traceability information regarding the system must first be obtained. Traceability links indicate a potential relationship between the components which make up a system. These relationships can cause a change in one component to be propagated to another. By collecting these traceability relationships is should be possible to identify the paths of impact propagation within a given system.

The following list gives a brief description of some of the techniques currently employed for extracting traceability information from software systems:

i) Pre-recorded traceability analysis approaches - Pre-recorded traceability information consists of the accumulated details of relationships between the various components which make up a system. Such traceability information is termed 'pre-recorded' because it is manually identified and collected over the entire development life cycle of the system and documented in

an appropriate manner by the developers.

ii) Dependency analysis approaches - In dependency analysis approaches, the relationships between system components are extracted by analysing pre-existing development artifacts (e.g. source code, system models, formal specifications etc.) [11].

iii) Knowledge based approaches - The aim of a knowledge based approach is to extract traceability information about system components by analysing the impact effects of previous change enactions. In order to achieve such evaluation, data concerning changes made to a system and their impact effects must first be recorded. Once this has been done it is then possible to identify traceability relationships within the system by analysing this information.

iv) Probability based approaches - Probability based approaches aim to provide additional information about existing component links. These approaches depend upon estimated probabilities of traceability relationships within a system. We can assign traceability relationships a 'conductivity' (or impact strength [4]) value which represents the probability that the target of the relationship is traceable from the source [3].

4 Limitations of current practice

Many of the current approaches to traceability extraction have significant flaws. The following list briefly outlines some of those deficiencies:

i) Dependency based methods provide detailed analysis for formalised information, but have little support for informal, natural language documents (e.g. requirement definitions) [12].

ii) Although pre-recorded traceability analysis provides support for all levels of formalism, it does not provide as in-depth analysis as dependency based approaches due to the generally vague nature of the pre-recorded relationships. [12] This is because the informal and loose definition of much of this observed traceability data makes important relationships hard to distinguish. Additionally, the transitive closure algorithm used to identify possible impacted components in pre-recorded traceability approaches is inefficient for most non-trivial systems. This is due to the potentially huge number of components and vast number of pre-recorded traceability links between them.

iii) Matrix structures often used to collect traceability information quickly become very large and are thus impractical for most non trivial systems. In addition, matrices do not provide the requirements engineer with an obvious mechanism for assessing the impact of proposed changes, but only the propagation of existing impacts [13].

iv) During the learning stage of knowledge based approaches, little traceability analysis can take place. This is because a knowledge base of previous change integrations must first be built before full knowledge based analysis can take place.

v) The propagation values used by probability based approaches are inherently inaccurate due to the fact that they must either be estimated or are calculated from metrics with uncertain reliability. Due to the fact that probability techniques only provide additional information about identified impacts, they can not be used on their own for impact analysis.

vi) No individual method is guaranteed to identify every single traceability link within a system. In addition to this, all methods will identify traceability links which do not necessarily imply an impact propagation path.

Taking a wider view of the entire change management and impact analysis processes, many of these techniques suffer from the following limitations:

i) The techniques cannot be introduced until the system integration phase of system development [14]

ii) In a significant number of methods alterations are restricted, rather than supported by, the change management scheme [9]

iii) Most of the techniques are dependent on formal, low level artifacts for impact analysis

iv) They provide little support for managing the evolution of proposed changes themselves

v) The support provided for visualisation of traceability and impact information is often minimal

5 Proposed solution

The following sections describe a tool based approach that is currently under development which supports the ideas presented in this paper. Figure 1 shows the general structure of the tool, incorporating all the main operational components.

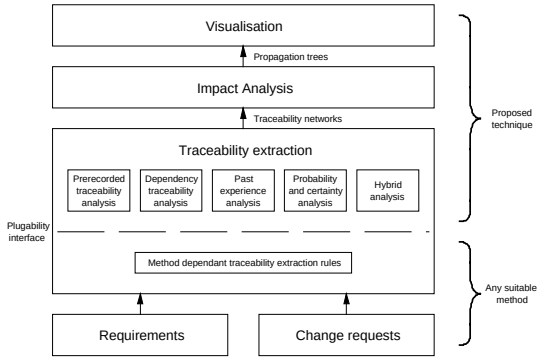

Figure 1 Structure of the support tool

The impact analysis capabilities which form the core of the tool are able to operate in any situation where traceability rich information is available. This implies that, provided suitable traceability extraction rules are at hand, the approach described in this paper can be integrated into any existing requirement elicitation technique.

5.1 The change management process

The availability of a change management mechanism for use at the requirements stage is particularly useful because it allows change management to be introduced very early on in

the development lifecycle. This means that support is available for change integration plus conflict and impact identification while the system is still very young. Such support will help to prevent the introduction of mistakes and omissions which could be expensive to fix at a later date. In addition, continuity is offered by the application of a change management mechanism that may be extended for use throughout the entire development life cycle.

Rather than restricting the evolution of components, a more suitable approach is to allow fluid alterations. This is supported by a change control scheme which works in the background assisting, but not constraining evolution. By allowing finer grained change management we can allow the system to evolve more naturally, rather than in quantum version steps.

The approach described in this paper uses the notion of viewpoints [15] as a mechanism for classifying and managing both requirements and changes. Fundamentally, viewpoints allow the explicit identification of the different perspective of a system from the point of view of interactors, stake holders, domain entities or other interested parties.

5.2 Assessing proposed changes

An important aim of the tool is to provide a mechanism for analysing the propagation of change impact through the set of components representing the system. To facilitate this, the tool supports the collection and management of many items of information which may be of use in performing such analysis. Once this has been done, a number of traceability extraction mechanisms can be used to identify the potential propagation paths of the proposed change. Finally, the results of this analysis can then be graphically visualised to assist the engineer in assessing the full potential impact of proposed changes.

Due to the fact that changes are usually enacted in batches, there is often a time delay between the initial impact assessment and final enaction of a particular change. In such a situation, it is probable that the system on which the change is to be enacted will have altered during the delay. Thus the system which the proposed change is actually enacted upon is not identical to that against which it was analysed.

To compensate for this phenomenon when the impact of a change is initially assessed, in addition to the system itself, we must also perform analysis of all currently accepted but as yet unenacted changes. By considering these changes as additional system components, it is possible to gain an analysable 'vision' of the state of the system after their enaction.

5.3 Traceability extraction

The tool employs the following mechanisms for extracting traceability information about a particular system:

i) Pre-recorded traceability - The support tool employs mechanisms to allow collection of much pre-recorded traceability information. This includes items such as the relationships between the following:
   - The different versions of each component
   - Conflicting and harmonic requirements
   - The direct impacts of changes
   - Parent and sub requirement relationships
   - Functional requirements with overlapping system functionality
   - Constraining relationships between functional and non functional requirements

ii) Dependency analysis - As an example of the type of requirements specification techniques suitable for impact analysis, the proposed tool employs finite state event scenario diagrams to represent the desired system functionality. These semi-formal representations not only allow developers and potential users to understand and validate the requirements, but they also allow formal dependency analysis to take place while remaining at the abstract level mandated by requirements level analysis.

iii) Past experience analysis - The tool allows the capture of traceability information regarding the effect of past alterations which may then be used to shed light on the potential impact of future changes. The tool supports two types of past experience analysis:
   - Analysis of previously recorded impacts to produce a list of actual components which could be impacted by a particular change.
   - Analysis of 'Inference cube' data structures which record classes of system component and propagation paths between those classes. This type of analysis produces a list of classes of component which could be impacted by a proposed change. Inference cube analysis is considered in more detail in the following section.

iv) Inference cube experience analysis - To try to predict the impact effect of a proposed change, the tool maintains a classification structure of all previous changes. The two main methods which exist for deriving impact predictions from this 'Inference cube' knowledge are:
   - Direct inference - A newly proposed change is first classified and then the classification structure is analysed to identify all of the previous classes impacted by changes of this class. All components which are members of the identified classes are then collected and presented as potential impacts.
   - Fuzzy inference - This approach is similar to direct inference except that it makes use of knowledge relating to the proposal of similar, rather than exactly matching change requests.

   The complete inference analysis technique employed by the tool is achieved by combining the fuzzy and direct inference approaches. This allows us to gain the benefits and overcome the drawbacks of the individual methods. The use of fuzzy inference techniques contributes a large data set, while the employment of direct inference techniques ensures that analysis has sufficient focus.

v) Probability analysis - Each analysis method produces it's own estimates of the probability of the propagation of impact along particular paths based on the individual metrics of that method.

vi) Certainty analysis - In addition the probability of impact propagation, the certainty of propagation can also be

calculated. This value indicates the certainty with which a propagation prediction can be made and is assess by combining the following two metrics:

- Degree of definition - The degree to which the components are specified
- Certainty of definition - An estimated level of confidence in the correctness of the data which is held

These features are included to help combat incompleteness and incorrectness in the specification of the system and proposed changes.

vii) Hybrid analysis - To overcome many of the problems associated with individual methods, it is possible to combine a number of different complementary techniques to produce a hybrid approach.

5.4 Impact visualisation

Once the potential impact propagation paths have been identified, this information can be transformed into one of a number of visualisations. These can then be used to present and examine the effect of the changes in a graphical manner. This has the following advantages:

- System end users, procurers and developers may easily appreciate the full effect of a proposed change. Plain numerical data can often be hard to interpret, therefore this can make the job of assessing impact much simpler.
- It becomes possible to perform fast visual comparisons of alternative evolution paths and change proposals.
- Graphical visualisation allows for direct manipulation and investigation of the proposed system by members of the development team.
- The tool allows the user to experiment with changes proposals and to view their consequences without the overhead of costly implementation.

It is the intention of this project to perform a wide ranging comparison of visualisation techniques in order to try and identify those most appropriate and effective. This will include the exploration of composite technique as well as common single paradigm methods.

6 Expected benefits

This project promises to generate the following potential benefits:

i) Support for the extension of the change management process to the entire system life cycle
ii) A more fluid change management process, allowing flexibility of system evolution
iii) Support for impact analysis and management of changes as well as for requirements
iv) Efficient impact demonstration and change experimentation via graphical impact visualisation
v) 'Plugability' of both method and support tool facilitated by requirements elicitation method independence
vi) Enhanced hybrid traceability extraction mechanism incorporating:
    - Dependency analysis based approaches
    - Pre-recorded traceability based techniques
    - Knowledge based analysis and prediction incorporating fuzzy inference mechanisms
    - Supplementary information depth provided by probability and certainty measures

7 Current state of research

A survey of existing techniques has been performed to identify useful approaches and techniques which could be employed in an improved change management process. These included change management processes, traceability extraction techniques, impact assessment approaches and visualisation schemes. Once this survey was complete, the most appropriate methods were identified for inclusion in the new process.

At the present time, work on the project is currently concentrating on the primary implementation of the change management tool. To assist with implementation, a series of worked examples and test applications will be used to assess tool practicality. The results of these studies will allow us to incrementally improve and enhance both the tool and method during the development process.

References

1. G. Kotonya; I. Sommerville "Viewpoints for Requirements Definition", *Software Engineering Journal*, Vol. 7 (6), Nov 1992, pp375-387
2. S. J Andriole, "*Managing Systems: Requirements, Methods, Tools and Cases*", McGraw-Hill, 1996
3. S. Ajila. "Software Maintenance: An Approach to Impact Analysis of Objects Change", *Software-Practice and Experience* vol. 25 (10), pp1155-1181
4. T. Goradia, "Dynamic Impact Analysis: A cost-effective Technique to Enforce Error-propagation", *Proceedings of the 1993 Int. Symposium on Software Testing and Analysis*, pp171-181
5. L. Li; A. J. Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software", *Proceedings of the International Conference on Software Maintenance*, pp171-184, Montery, CA
6. D. S. McCrickard; G. D. Abowd, "Assessing the Impact of Changes at the Architectural level: A case study on Graphical Debuggers", *Proceedings of the International Conference on Software Maintenance*, pp59-67, Montery, CA
7. M, Moriconi; T. C. Winkley, "Approximate reasoning about the effects of program changes", *IEEE Transactions on Software Engineering* vol 16 (9), pp980-992
8. J. Han, "Supporting Impact Analysis and Change Propagation in Software Engineering Environments", *Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice*, pp172-182, London
9. S. D. P. Harker; K. D. Eason; J. E. Dobson, "The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering", *Proceedings of the IEEE International Symposium on Requirements Engineering*, Jan 1993, pp266-271, SanDiego, CA
10. I. Sommerville, "*Software Engineering*", Addison-Wesley, 5th edition pp552-554
11. S. A. Bohner; R. S. Arnold, Preface to "*Software Change Impact Analysis*", IEEE Computer Society, pp ix-xii
12. S. A. Bohner; R. S. Arnold, Introduction to "*Software Change Impact Analysis*", IEEE Computer Society, pp1-26
13. G. O. Kotonya, "*Project proposal report: Managing RE Change Through Visualisation*", unpublished report, pp1-8
14. J. K. Buckle, "*Software Configuration Management*", Macmillan, p15 and p55
15. G. O. Kotonya; I. Sommerville, "*Requirements Engineering*", John Wiley and Sons, Chapter 5

Towards an Explicit Intentional Semantics for Evolving Software

Kim Mens
Vrije Universiteit Brussel
Programming Technology Lab (PROG)
Pleinlaan 2, B-1050 Brussel, Belgium
kimmens@vub.ac.be

Abstract

The subject of my PhD work is the study of software engineers' intentions *and the importance of using the information provided by such intentions during the software engineering (SE) process. More specifically, we will study how automated reasoning about explicit software intentions can facilitate many software engineering activities, and software evolution in particular.*

1. Introduction

It is generally acknowledged that a lot of software[1] today is difficult to understand, maintain or adapt, hard to reuse, difficult to evolve, and so on [1, 5, 6]. This is partly due to the fact that most software contains a lot of hidden assumptions. The software reveals only *how* things will work, and (implicitly) *what* will happen, but provides little or no information on the *intentions* of the engineers that built the software (e.g. *why* something was constructed in a certain way). Even when the software does contain such information it is most often implicit or described informally in the software documentation [13].

Our contribution will be to make a first step towards a kind of intentional 'semantics' for software in which this kind of information can be expressed explicitly, preferably in a computable and declarative way, and to show how automated SE tools can use this information to make software more 'manageable'. We do *not* intend to develop a complete formal semantic model, but rather to study the use of intentions in automated SE tools.

To restrict the scope a bit, we focus on the domain of evolution of object-oriented (OO) software[2], and set out to prove the following thesis:

> **Thesis:** Automated reasoning about explicit information on the intentions of software engineers allows to build more powerful tools for software evolution. (More powerful in the sense that they can draw stronger conclusions by reasoning not only about the software but also about higher level conceptual information, i.e. the software intentions.)

We admit that this thesis is still somewhat too broad and needs to be made more precise. For example, the kind of software evolution *tools* we are particularly interested in are tools for detecting evolution conflicts. We will try to show that conflict detection tools using intentional information can be made more powerful in the sense that they can detect more conflicts. Also, we need to make more precise *how* intentional information will *allow to* do this.

2. Intentions

When constructing a software artifact, a software engineer constantly makes important and less important choices and decisions. These decisions are typically based on and motivated by various assumptions about the problem domain, about the software requirements (functional as well as non-functional), about other software artifacts with which the artifact under construction should co-operate or upon which it should build, and so on...

All these assumptions and the associated intentions of a software engineer when making decisions, usually are not captured explicitly in the software. Only the results of the decisions that were made can be found in the software. In the best case an engineer writes down his or her intentions on paper or in the software documentation in natural language, or uses certain conventions, software patterns

[1] We explicitly use the term 'software' throughout this paper instead of the word 'code' or 'program', because we believe the same research problems and solutions are also relevant to artifacts in other phases of the software life cycle such as requirements, architecture, analysis and design.

[2] We choose evolution and OO because of our background in these domains [12] and because they pose some non-trivial and important problems.

or style guidelines from which some intentions can be derived implicitly. (For example, using a strategy design pattern might express a designer's intention to make an important algorithm easily replaceable by a variant [4], or "best programming patterns" might be used to communicate programming intentions [2].) Most intentions however, e.g. *why* software was constructed in a certain way, are difficult or impossible to extract from the software. (As opposed to information on *what* the software does, and *how* it works, which usually can be derived implicitly or explicitly from the software.) Therefore, we think there is a need for making these intentions explicit.

Intuitively, we could define a software intention as any kind of information on the purpose of the software, that is not explicitly contained in the software itself. In other words, an intention is a meta description of the software that motivates why the software is constructed in a certain way. But not any meta description is a software intention: only those meta descriptions that link software artifacts to the 'hidden assumptions' are software intentions.

> **Definition:** A software intention is a meta description of the software that links software artifacts to the 'hidden assumptions' made by a software engineer (about the problem domain, about the software requirements, about the purpose of related and co-operating software artifacts,...).

One of the reasons why software engineers are unable to adequately document their intentions is that SE tools and notations provide insufficient support for expressing intentions in a more explicit, formal and disciplined manner. We feel that such information can play an important role to facilitate SE activities in general, and software evolution in particular. However, although we want to express intentions in a formal way, we want a notation that is simple enough to be used and accepted in practice, and easy to be manipulated in tools. We claim that a need exists for building SE tools that can reason automatically about such explicit intentions.

Our claim is supported, amongst others, by [9], where it is argued that software evolution currently suffers from a lack of intentional information: when the original software engineers' intentions are insufficiently documented, their continued involvement is needed to enable later engineers to learn their way through the software system and to better understand the assumptions behind the system's design. This may be too time-consuming or simply impossible when the original software engineers are not available anymore Lehman[3] also agrees that software engineers' hidden assumptions should be made explicit in the software, preferably in a structured and machine-processable form, to facilitate change management during software evolution [8]. He argues that at all stages of the software life cycle,"attempts must be made to recognize, capture and record assumptions, whether explicit or implicit, in design and implementation decisions, as must any dependencies and relationships between them".

Therefore, we assume the following research hypothesis.

> **Research hypothesis:** Many SE activities (such as software maintenance, adaptation, evolution, reuse, re-engineering, reverse engineering,...) benefit by intentional information of the software engineer.

We motivate this research hypothesis, by arguing that some of the technical problems that hinder these activities could be solved more easily if one would have more intentional information of the software engineer. Some of the technical problems are:

1. understanding[4] the purpose of software artifacts, as well as why they were constructed in a certain way;
2. understanding the dependencies and relationships between different software artifacts;
3. detecting and solving conflicts when changing, adapting, evolving or reusing software artifacts;
4. traceability of software artifacts.

It is clear that the first two problems immediately benefit by more intentional information. Solving the second problem is important to be able to assess the impact of making changes to certain software artifacts on the other software artifacts. The third problem is a special case of the more general problem of *compliance checking*: checking whether some evolved software artifact conforms to what is expected from it, i.e. does it work together correctly with other software artifacts, are the assumptions that it makes and that are made about it valid, does the software artifact respect the original intentions, ... ? It should be at least intuitively clear that compliance checking can benefit by more intentional information. Finally, *traceability* problem comes down to"justifying the existence of a given result by tying it back to the stated goals and objectives" [11]. This information could be expressed by explicit intentions.

We will try to validate this research hypothesis in practice by showing that automated reasoning about software intentions does not only make it possible and easy to build

[3] Lehman studies the *laws of software evolution* and their implications to improve software processes dealing with evolution.

[4] Although we think that software intentions can clearly contribute to the research domain of software comprehension, our focus will be more on the use of intentions to enhance software evolution tools.

automated SE tools (and tools for checking evolution conflicts in particular), but also allows us to draw stronger conclusions than without that information. This immediately proves our thesis as well.

3. Approach

We will follow a "bottom-up approach with a top-down vision". Our ultimate goal is to show that automated SE tools can use explicit intentional information to make software more manageable. However, to simplify things at first, we limit the scope by looking at the problem of evolution of OO software. Later we broaden the scope again and show that the results are also valid for other SE activities (than software evolution) and other programming paradigms (than OO).

Instead of immediately trying to build a general formal model of software intentions, we focus on a particular kind of intentions first and study what extra power they can provide. Although we still have to complete our literature study and make a categorization of the kinds of intentions that are most promising, we think it would be interesting to look at those intentions that can be expressed in terms of *classifications* and relationships between these classifications.

3.1. Classifications

The idea of a *classification* is to group a collection of software artifacts together because they ought to be considered as a whole (from an intentional point of view). All artifacts in a classification typically share some important feature. For example, in a financial application it could be interesting to group all software artifacts dealing with "handling deposits" together in a single classification. This classification expresses the intention that all these software artifacts cooperate in achieving the functionality of handling deposits.

A software artifact can belong to different classifications and a single classification can contain many different kinds of software artifacts. A classification does not necessarily correspond to the classifications that can typically be found in the software. The only requirement is that the software artifacts in a classification share some functional (e.g. handling deposits) or non-functional (e.g. aspects such as "persistency" or "distribution") feature. As such, classifications express part of a software engineer's intentions, because they provide conceptual classifications of software items that may not be found in the software itself. Dependencies and relationships between classifications ("part of", "is a", causal relationships as well as negative relationships stating independencies) can provide even more important intentional information.

Intentional information on which software artifacts are grouped according to which classifications and what the dependencies between the different classifications are could be used in tools for dealing with software evolution conflicts. For example, if there is a conceptual dependency between two classifications, one could expect that this dependency is reflected in some way by the artifacts that are contained in those classifications. If this dependency structure is accidentally invalidated upon evolution, there is an evolution conflict.

3.2. Validation

After having chosen a particular kind of intentions to investigate in more detail, we perform some experiments to validate whether the proposed approach actually works (i.e. that intentional information based on classifications and dependencies between them can really be used to solve new and interesting evolution conflicts). We will build a prototype of an automated SE tool (more specifically, a tool for detecting evolution conflicts) and apply it to an industrial case study. We will try to merge our theory and tool with the existing reuse contracts methodology [12, 10], which is a proven methodology for dealing with evolution conflicts in OO software. We plan the following validation experiment:

1. identify and declare some classifications as explicit intentional tags about the case;
2. identify and declare dependencies between classifications as intentional information about the case;
3. implement and test conflict detection and compliance checking rules based on this information;
4. analyze how this approach extends the reuse contracts model (i.e. how it makes it more powerful).

Whereas the purpose of this experiment is to show that software evolution tools benefit by more intentional information, we also need to investigate what happens when the intentions themselves evolve.

3.3. Generalization

To generalize the obtained results we will study which other kinds of intentions can be expressed and how they can be used to build more powerful software evolution tools. Next we broaden the scope and try to show that the results are also valid for other SE activities (than software evolution) and other programming paradigms (than OO). To conclude the thesis we hope to be able to show the generality of our research results by showing that existing "hard" semantic techniques which also declare a kind of intentional semantics, can be expressed with our approach as well.

4. Related Work

4.1. Program Comprehension Research

Program comprehension research results might provide interesting clues as to which kinds of intentions are useful to enhance the evolvability of software. Although current program comprehension research fails to provide a clear picture of comprehension processes with respect to specialized tasks such as software evolution, some existing research results do indicate which kind of information is considered important by engineers when trying to understand software constructed by other engineers [14]:

- *Software-specific knowledge* relating to functionality, software architecture, the way algorithms and objects are implemented, and so on.
- Information on the *what*, *how* and *why* of software artifacts.
- Used *styles* and *conventions* ('rules of discourse') such as coding standards, algorithm implementations, expected use of data structures, and so on.
- Information on the *control flow* and other dependencies (e.g. data flow) in the software.

4.2. Intentional Programming

The concept of 'intentional programming', seems closely related to our work, as it is also based on making intentions explicit in the software. [13] agrees with us that "much of what makes programming[5] costly and time-consuming, including the declaration of design intentions, the identification of invariants, the alternatives which were not chosen, the overall structure, the dependencies ... and so on are either not encoded at all, or not encoded in a machine understandable form". However, whereas we see intentions as a kind of meta description on top of the software, [13] introduces intentions as a new programming abstraction which can actually be executed.

4.3. Features

We informally defined *classifications* as collections of software artifacts that need to be considered as a whole, because they share an important 'feature'. So classifications can be identified by identifying the important features. [6] provides some examples of (functional) features and defines a 'feature' as "any distinguishing characteristic of a software system that customers or reusers can use to select between available options". The FODA methodology [7] considers distinct types of features: operational, non-functional, development,... [3] defines a feature as "the difference that makes the difference" and provides some guidelines for identifying features.

[5] Note that we did not focus on the programming level only, but also on the other phases of the software life cycle.

4.4. Other Related Work

In the research areas of program understanding, design theory and knowledge based SE, many systems have been described that represent programming knowledge in one way or another. We need to investigate how these kinds of knowledge relate to software intentions.

References

[1] M. Aksit, B. Tekinerdogan, L. Bergmans, K. Mens, P. Steyaert, C. Lucas, and K. Lieberherr. Adaptability in object-oriented software development. In *ECOOP'96 Workshop Reader*, pages 5–52. dpunkt.verlag, 1997.

[2] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[3] R. Creps, C. Klinger, M. Simos, L. Lavine, and D. Allemang. Organization domain modeling (odm) guidebook version 2.0, 1996. Technical report for Software Technology for Adaptable, Reliable Systems. STARS-VC-A025/001/00.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.

[6] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addisson-Wesley, 1997.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[8] M. Lehman. Software's future: Managing evolution. *IEEE Software*, January/February:40–44, 1998.

[9] K. Lieberherr. Workshop on adaptable and adaptive software. In *Addendum to the OOPSLA'95 proceedings*, pages 149–154. ACM Press, 1995.

[10] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1997.

[11] K. Rubin and A. Goldberg. Object behaviour analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[12] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings OOPSLA '96*, pages 268–285. ACM Press, 1996.

[13] C. Symonyi. Intentional programming — innovation in the legacy age, 1996. Presentation notes IFIP WG 2.1 meeting.

[14] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

Real-Time Reactive System Development - A Formal Approach Based on UML and PVS*

D. Muthiayen
Department of Computer Science
Concordia University
Montréal, Québec H3G 1M8, Canada
d_muthi@cs.concordia.ca

Abstract

The maturation of a methodology for formal development of Real-Time Reactive Systems of industrial scale broaches issues including automated development of software specification, design, analysis, and synthesis. Automated software engineering methods should be grounded on rigorous principles and not on ad hoc approaches. Our approach integrates the recently published industrial standard graphic notation UML (Unified Modeling Language), for object-oriented modeling, and PVS (Prototype Verification System), for automated reasoning.

1. Introduction

This paper proposes a methodology that synthesizes object-oriented and real-time technologies for reactive system development. We first formalize UML [7] semantics, and embed the notation in PVS [6]. We then develop methods for consistency checking across design specifications and for verifying system properties in a design. The formalization of UML is undertaken to fulfill the need for a sound foundation for requirements modeling and rigorous design analysis in the context of safety-critical systems. The methodology forms the basis for the process model for reactive system development shown in Figure 1. In this iterative process, we develop a UML model from system requirements, translate the graphical design into PVS theories, analyze the design for consistency, simulate the design specifications for validation, and verify desired system properties in the design, before proceeding to an implementation. The motivation for this work comes from two fronts: (i) the wide acceptance of UML in industry, as a unified modeling notation applicable in a broad spectrum of domains, and (ii) the use of PVS for formal design analysis of large scale applications, as reported in NASA guidebooks [4, 5].

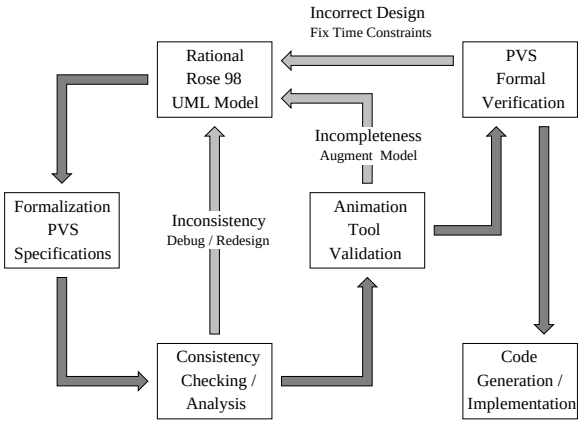

Figure 1. Iterative process model.

2. Research Goals

The main goal of this research is to develop a methodology for rigorous software development in industrial context. Figure 2 shows major aspects of a specification and verification environment based on the methodology. Rigorous modeling and analysis methods can only be established after providing formal semantics, and instituting mechanisms for checking design *completeness* and *consistency*. Formalizing the modeling technique involves the following steps.

1. Select components of UML notation suitable for specification of real-time reactive systems, and relate these components in a consistent way.
2. Provide formal semantics for the components and their relationships using PVS specification language, with focus on application to reactive systems.

*This work is supported by a fellowship from Natural Sciences and Engineering Research Council, Canada. I acknowledge the support of my thesis advisor, Prof. V.S. Alagar, in conducting this research.

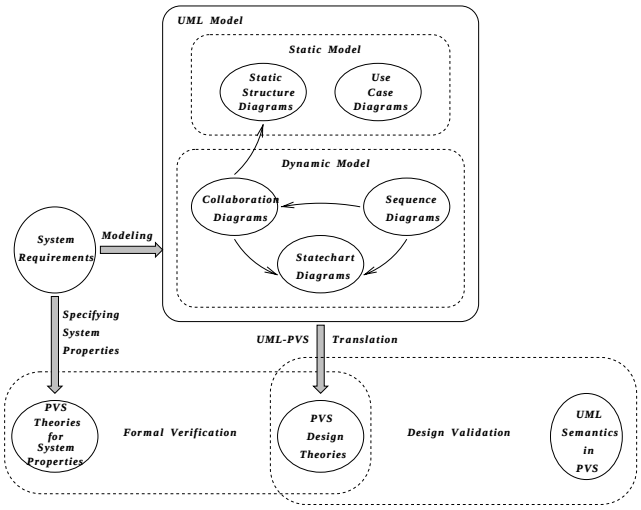

Figure 2. Reactive system development.

3. Develop a formalism incorporating the components for modeling objects and subsystems.

Milestones in the research work are:

1. Formalization of UML notation. In developing UML formal semantics, we specify PVS type definitions for UML model elements, based on the abstract syntax available in UML class diagrams. We then specify PVS predicates and lemmas for (i) constraints and well-formedness rules on UML model elements, available in OCL (Object Constraint Language), (ii) UML semantics, available in natural language, and (iii) relationships among UML components.

2. Adaptation of the verification methodology described in [3] for formally verifying safety and liveness properties in design specifications within the PVS verification environment.

3. Proposed Methodology

Easterbrook et al. [2] give an extensive experience report on requirements modeling and analysis based on a methodology incorporating OMT (Object Modeling Technique) and PVS. This approach does not integrate the graphic notation of OMT with the formal specification language of PVS; it uses the notations to complement each other. While the OMT model provides a high level structural view of requirements, the PVS model gives a detailed view and supports rigorous behavioral analysis. It is not apparent how correspondence is established between the OMT diagrams and the PVS specifications. Our primary goal is to provide precise methods based on formal semantics to translate UML design models into PVS theories.

3.1. Components of UML Notation to Formalize

We focus on a subset of UML notation suitable for modeling objects, subsystems, their static structure, and their dynamic behavior, in the context of real-time reactive system development. *Static structure diagrams* describe the object model; object and class diagrams capture relationships among objects and classes in a system. In *use case diagrams*, use cases give abstract descriptions of tasks performed by cooperating objects, and actors symbolize roles played by external objects interacting with a system. *Sequence diagrams* capture sequences of messages exchanged among objects in an interaction, as well as timing constraints on responses to stimuli. In *collaboration diagrams*, a collaboration describes associations among cooperating objects, showing the context for the purpose of the cooperation; messages exchanged among the objects constitute an interaction that is superimposed on the collaboration. *Statechart diagrams* specify the states in which an object can be, possible transitions between states, the event labeling each transition, entry and exit points for complex states, and certain timing constraints on transitions.

3.2. The Choice of PVS - Justification

There is an increasing demand on the construction of provably correct software systems in strategically important areas, such as the aerospace industry and NASA projects. The current status of formal method integration in industrial software development includes application in areas such as avionics, telecommunications, and nuclear power plants. PVS is being groomed for use in the integration of formal methods in the development process of mission critical systems. Experience gained from these studies are reported in two NASA guidebooks [4, 5].

PVS consists of a specification language based on higher-order logic, and an interactive proof checker that uses powerful arithmetic decision procedures. The language allows the definition of predicate subtypes, and dependent types, with constraints attached to type definitions. Specifications can be written as parameterized theories, with constraints on the parameters. PVS supports specification of abstract data types in a concise and efficient way, with automatic generation of axioms and functions capturing intended properties of the data types. The higher-order logic and its type system bring lot of expressive power to the specification language. This makes it suitable for formally describing semantics of complex structures, and the abstract syntax and well-formedness rules of UML.

PVS implements a set of powerful primitive inference rules, and a mechanism for composing proof strategies based on frequently used patterns of inference steps. The reasoning system supports a wide range of decision proce-

dures, provides an extensive set of proof commands classified as *primitive rules*, *defined rules*, and *strategies*, and supports interactive proof construction. These features make PVS well-suited for verifying the inherence of properties in design specifications. For instance, Shankar [8] gives a theory of time, and a computational model for specification and verification of real-time systems.

3.3. Methodology for Formalizing UML Notation

Providing formal semantics implies identifying attributes and properties of model elements relevant to the application domain, and describing their meaning in a mathematical notation. The semantics must ensure *completeness* in the sense that sufficient number of axioms describing attributes and properties of the model are included for a precise understanding of expected behaviors. This may involve the construction of a formal object constraint language.

We adopt the following procedure in providing a formal semantics for UML notation. We identify the model elements described in UML class diagrams comprising metaclasses and their relationships, and specify each model element in PVS. We give a PVS specification for each well-formedness rule for the metaclass describing a model element, and formulate invariants and constraints on the model element as lemmas. These need to be proved in checking the well-formedness of a diagram including instances of the model element. We flatten the class hierarchy to obtain all the attributes and well-formedness rules for a model element. We translate the informal description of the semantics for each UML logical package into PVS predicates and lemmas. We specify relationships and constraints identified among UML components as lemmas involving predicates on instances of model elements.

3.4. Static and Dynamic Models

In modeling system requirements, the first step is to capture the static structure of the system by abstracting objects, and their relationships in collaborating to perform a task. For instance, entities may require complex data structures to capture their functionalities. Relationships among objects and classes include associations, aggregations, composition aggregations, generalizations and specializations. These features of a model must be properly specified before describing the dynamic behavior of entities.

The two types of object interaction relevant to the design of embedded systems are *sequential composition*, and *concurrency*. We use UML icons [1] for *arrival pattern* and *synchronization pattern* to indicate different kinds of message flow between interacting objects. An arrival pattern icon can be combined with a synchronization pattern icon to capture two orthogonal dimensions.

A collaboration represents a set of objects and relationships among the objects; the relationships shown are those that are meaningful to the purpose of the collaboration. We define a *projection* of a collaboration C as a representation of a subset of the objects in C, and the relationships present in C, among the objects in the subset. For a collaboration C, and an operation *op*, there exists a collaboration diagram C_{op}, such that (i) C_{op} is empty, or C_{op} is a projection of C superimposed with a message sequence, and (ii) it can be *proved* that the effect of an interaction based on C_{op} is the performance of operation *op*.

3.5. Requirements Modeling and Design Analysis

Achieving design consistency is a major issue when using a notation with several interleaving components. It is imperative that consistency is obtained within diagrams to determine the satisfaction of system properties, as well as across diagrams to ensure that components of the notation are compatible with each other. We identify relationships among components of UML notation, and describe corresponding constraints in specifying reactive systems. These constraints are formally stated, so that if a property exists at one level, we can conclude whether it exists at another level. Satisfaction of invariants capturing these relationships implies that semantics of constructs in different components of the notation are consistent with each other.

Completeness in Data Type Specifications The PVS specification of an abstract data type is concise, with a set of *constructors* along with associated *accessors* and *recognizers*. When the data type is type-checked, a new theory is generated, providing axioms relating the constructors, accessors, and recognizers, as well as induction principles needed to ensure that the data type is the initial algebra defined by the constructors. For instance, *extensionality* and *eta* axioms are generated to define equality on instances of the data type. Other axioms define well-foundedness rules, and support well-founded subterm ordering relations and strong forms of induction. The functions *every* and *some* are generated to establish the truth value of a predicate in existentially and universally quantified formulas on the data type. Functions are included to define a generic *map* on the data type. These generated axioms and functions must often be augmented with additional ones capturing other properties of the data type to ensure *completeness* of the specification. A data type specification is *complete* if every intended property of the data type can be deduced from the axioms.

The rich type system of PVS is supported by the generation of proof obligations, called *type constraint conditions (TCC's)*. In proving a theorem, subproofs may require discharging some of these obligations. This can be achieved by invoking predicates used in subtyping, axioms generated for

abstract data type definitions, and user-defined axioms capturing additional properties of these data types. If a proof cannot be discharged, it is due to the incompleteness of the specification.

Consistency in Design Specifications For each UML design specification, we use the formal semantics to formulate a corresponding PVS specification. A relationship R between two UML design components is stated in the form of a set of theorems in a parameterized theory T_R. The theorems in theory T_R instantiated with two actual design specifications must be proved in order to establish *consistency* between the two designs. Checking for consistency of design specifications may not be possible without sufficient axioms capturing properties of data types in the specifications. Consequently, consistency cannot be assured without completeness of abstract data types.

Consistency: Let d_1 and d_2 be two design specifications in UML; let p_1 and p_2 be their corresponding PVS specifications. There exists a parameterized theory T_R, corresponding to the relationship R between the designs d_1 and d_2. If a proof can be constructed for every theorem in the instance $T_R(p_1, p_2)$ of the theory, then design specifications d_1 and d_2 are consistent.

3.6. Verifying System Properties

The goal of verification is to establish that a design specification satisfies properties included in requirements specifications. Specifications for a reactive system and its environment are given in terms of *axioms* and *inference rules* for reasoning. These specifications can be used to verify properties, such as *safety* properties and *liveness* properties. To accomplish this analysis task, we need a *logic*, such as temporal logic. Requirements can be formally described within the logical semantics, providing a *Behavioral Specification (BS)* of the system. Static and dynamic properties of the design are stated as axioms and rules of inference, giving a *Design Specification (DS)* of the system. For instance, a design specification can include a description of the state transitions due to the occurrence of internal and external events. We need a verification methodology based on these specifications. To demonstrate that a design satisfies the requirements, it is sufficient to show that every formula in *BS* is a logical consequence of *DS*.

In the PVS model of a system, we include axioms for transitions, timing constraints, and arrival and synchronization patterns, based on the formal semantics. Each axiom expresses either a system status or a time constrained activity causing a change in system status. The property to be verified in the design is stated in a semi-formal notation, which is translated into a set of invariance assertions. Based on the UML model, each invariance assertion is translated into a PVS formula. We use Shankar's *since* operator [8] to specify durational expressions in axioms and invariance assertions. The given property is formally proved in the design if we can construct a proof for each lemma specifying an invariance assertion in design theories.

A straightforward approach to formal verification is to follow Shankar's methodology [8]. We intend to investigate an approach in which formulas involving the *since* operator over state predicates are transformed into linear inequalities. Axioms and lemmas are transformed into a system of linear inequalities involving logical variables denoting absolute times. Proving a lemma is thus reduced to proving the consistency of a set of linear inequalities. Since PVS has a rich set of rewrite rules for inequalities over reals, we expect the verification process to be less complex than the straightforward approach. Certain aspects of this methodology, such as formally specifying invariance assertions from the semi-formal description of a property, cannot be fully automated. However, several steps such as transforming a formula using the *since* operator over state predicates into a linear inequality over logical variables, can be mechanized.

References

[1] B. P. Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Reading, MA, 1998.

[2] S. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.

[3] D. Muthiayen. Animation and formal verification of real-time reactive systems in an object-oriented environment. Master's thesis, Department of Computer Science, Concordia University, Montréal, Canada, October 1996.

[4] NASA. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Vol. 1: Planning and Technology Insertion*. Report NASA-GB-002-95. NASA Office of Safety and Mission Assurance, Washington D.C., 1995.

[5] NASA. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Vol. 2: A Practitioner's Companion*. Report NASA-GB-001-97. NASA Office of Safety and Mission Assurance, Washington D.C., 1997.

[6] S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[7] Rational Software Corporation. *UML Notation Guide, and UML Semantics, Version 1.1*, September 1997.

[8] N. Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Proceedings of Computer Aided Verification, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, June 1993. Springer Verlag.

Automating migration of Fortran programs*

Christophe Roudet
INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
Christophe.Roudet@sophia.inria.fr

Abstract

We present TrfL, a language independent transformation system based on syntactic rewriting rules working on abstract syntax trees, with semantic constraints. The aim of TrfL is to ease the maintenance of large systems by automating transformation tasks such as restructuring, porting, documenting. We describe the integration of TrfL in Foresys, a Fortran engineering environment developed by SIMULOG.

1. Identification of the problem

The Year 2000 problem has posed significant problems that are being addressed using automated software engineering tools. The current legacy systems and the large software systems of the future will be too large to maintain without program transformation tools [3]. A first level of user assistance is provided by simple textual tools, like the search/replace tool available in all interactive text editors, or batch tools like *sed* or *awk*. Even very simple transformation like swapping array indices for a given common variable requires some understanding of the language syntax and typing rules. The difficulty of performing semantic analysis using character-based tools is the most serious limitation of the Unix shell script approach. Transformation systems rather work with abstract syntax trees that make analysis easier and can also be used to hook results on relevant nodes.

We propose a program transformation system, TrfL, that emphasizes the expressiveness of tree transformations, can be easily connected to other tools and provides a framework to build and apply transformations. Our description will be based on a Fortran environment. In the first section, we try to classify program transformation systems. The solution described in sections 3 and 4 is a language independent transformation system based on syntactic rewriting rules working on abstract syntax trees, with semantic constraints. Section 5 describes the integration of TrfL in Foresys[1] (a Fortran engineering environment), developed by SIMULOG, then section 6 points out one simple example of transformation. Section 7 deals with the different approaches to re-structure Fortran code.

2. Related work

Program transformation systems can be found in several systems and domains and here is a tentative of classification:

- Environment generators. These kinds of tools generate syntax directed editor for a special language from formal specifications of this language. Centaur [4] and the Cornell Synthesizer Generator [26] are in that family. These tools give a transformation system to edit and manipulate interactively programs by means of menus. Transformations are syntactic, expressed with two patterns (the source and target patterns) based on the abstract syntax of the language.
- Program synthesis tools. Purpose of these tools is to obtain efficient code from formal specification by successively applying semantic preserving transformations. KIDS [27], CIP [2, 19], PROSPECTRA [21], ZAP [12], and POPART [10] are systems of this domain. This was the first domain where program transformations were used. These systems come with a base of predefined transformations.
- Compiler tool kits. These systems include several tools such as parser and lexer generators, attribute grammar evaluators and attributed tree transformation system. The transformations involved are optimization, code generation and translation. Puma [17, 18], Gentle [30, 29], FNC-2 [23] and Optran [25] fall in this category. These tools work in batch mode and only accept a restricted form of pattern matching to some

*This work is partially supported by SIMULOG

[1] http://www.simulog.fr/foresys

fixed region near the root of terms. Backend generators (Beg [11], Twig [1], Burg [15], Iburg [16]) can be included in this domain. These tools are dedicated to generating machine code and even offer mechanisms to specify registers allocation.

- Functional programming languages. Most of the modern functional languages (Hope, SML, Caml, Miranda) allow pattern matching. So these languages can be used as a transformation language. Trafola-H [20] was designed to express complex tree transformations in a short and suggestive formulation. It enhances pattern matching in allowing distributed patterns [14].
- Transformation systems. TXL (Turing eXtender Language) [9] and TAMPR [5, 28, 6] are special purpose languages to express transformations. These two tools work in batch mode and patterns are expressed on the abstract syntax of the manipulated language. TrfL falls in this category.

All these systems have restrictions: only accept basic forms of pattern matching, work only in batch mode or interactive mode, do not accept non semantic preserving transformations, do not use contextual information (data flow graph, symbol table, ...), can not extend the transformations base, some of them use a wide spectrum language and programs must first be translated in this language before applying transformations, ... These restrictions were kept in mind in the design of TrfL.

3. TrfL scope and applications

TrfL is a generic rule-based **Tr**ans**f**ormation **L**anguage. TrfL is generic, as it is not dedicated to one language. TrfL is a language to assist engineers in maintaining programs. TrfL can cover several tasks of the maintenance process: documenting, restructuring, translating or porting programs. For instance, TrfL has been used to document Lisp programs and TrfL is bootstrapped in TrfL. For that purpose, TrfL provides an expressive pattern language, facilities to be connected with analysis tools and a collection of tools to build and perform transformations. We focus in this paper on using this language in a Fortran environment. We want an open restructuring tool, where transformations can easily be added and performed. TrfL intends to be used by both Fortran programmers and Fortran transformation experts (who know typing and contextual informations, Fortran and TrfL). A user can specify simple transformations with little effort, since transformation like adding/removing a parameter in function definitions/calls doesn't require a great level of knowledge. More complex transformations like moving from a programming convention to another or from Fortran77 to Fortran90 have to be specified by an expert.

4. Description of TrfL

The first goal was to design a language to express complex tree transformations, and the second to provide an environment with a wide collection of tools. These tools can be divided in two. The first ones are to build and modify transformations in an easy way by using structured manipulation to build patterns, and the second ones are in the application of transformations. The system should support both interactive and batch transformations with different strategies, and functional or side-effect (translations) transformations. A TrfL specification is a set of transformation rules. Roughly, a transformation consists of a *source pattern*, a *target pattern* and an *application condition*. Source and target patterns[2] are abstract syntax trees (AST) with metavariables (place-holders that represent arbitrary pieces of source code). An abstract syntax tree is a structured representation of a program that omits all unnecessary information like keywords. The application condition is a predicate that must be true in order for the rule to be fired. The target pattern is a replacement for the source code when the source pattern has matched the source code, the metavariables have been instantiated and the application condition verified. Compared to other transformation languages which generally allow a very restricted form of pattern matching, TrfL enhances this mechanism (feature found in [14]). In these transformation languages, a pattern can only match near the root of a tree term or it can only select fixed number of items in list structured trees. The user only describes the shape of the subtree he wants to find, and not where and how such a subtree should be found. Thus it becomes easier to describe a pattern that find nested loops for example. It is a fact that non-trivial program transformations require program analysis. For example a transformation that deletes unreachable code, needs control flow analysis information in order to identify code fragments that can never be reached. So most transformation systems work with analysis tools. Patterns represent the syntactic part of the rule and the application condition can refer to non local and semantic information (symbol table, control flow graph, etc) in order to restrict the applicability of rules. This information is called contextual information. A TrfL rule can be thought of as a function that takes as input a tree and contextual information, and returns a transformed tree or fails. A rule is applicable when both the pattern matches the input tree and the application condition is evaluated to true. Side effects are allowed, for example to update contextual information or emit an error message to the user. The TrfL system (Fig. 1) consists of an *engine* and an *environment*. The engine takes a source program, searches for pieces of code

[2]The pattern are presented here in concrete syntax for reason of readability (the system is able to display the patterns in both abstract or concrete syntax).

that match a transformation, and checks the application condition of that rule. If the condition is verified, the selected code is replaced with the instantiated target pattern. If the condition is not, the current transformation rule is not used and the engine has to find a new transformation rule for the same piece of code, look for a new piece of code, etc. The engine has several decisions to make during the transformation process: How the source code should be searched?, Which rule should be fired if more than one is applicable?, Should rules be re-applied?, etc. A program transformation system can be fully automatic, or semi-automatic (with somebody guiding the transformation process). This is why we want to provide an environment. The environment supports two functions:

1. it embeds the engine. One may want to perform the transformation process in batch mode without interaction, in this case some parameters (to control the strategies listed above) must be set before starting the process and the engine can work alone. In another way the user can interact on the process, 1) by guiding or controlling the engine, 2) by choosing a piece of code and then the engine gives the list of valid transformations, 3) by choosing a transformation and then the engine shows where this transformation can be applied.
2. it provides a framework to build or modify transformations. Patterns are abstract syntax trees, but a user is more familiar with the concrete syntax than with the abstract syntax and then building transformations could be a hard task. This can be made easier with an interactive building of a transformation from a template code, and pattern can be derived from a source code.

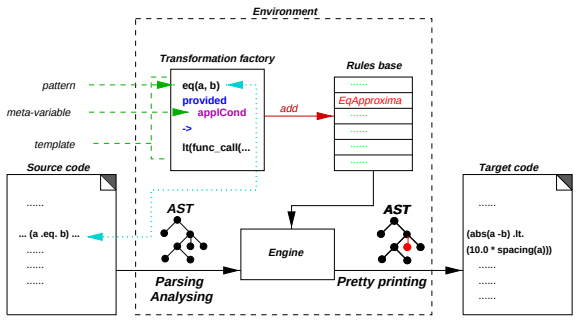

Figure 1. The transformation system

5. Transforming Fortran programs

The need of transformations in Fortran is a recurrent subject, and tools that automate (fully or partially) the process are welcome and useful. A list of tasks that a Fortran transformation system can support includes parallelization, instrumentation, restructuring, optimization and documentation. A text editor with a string regular expression "query-replace" function or a script (Perl, awk, ...) is often not enough and is somehow error prone. The simplicity of regular expressions often does not allow a programmer to express the desired query and does not exploit the underlying tree structure of the program (on the other side, processing comments or non syntactically correct code is easier). More over program analysis is not performed on the textual representation but rather on the abstract syntax tree. Foresys is an engineering system dedicated to Fortran. The Foresys package provides project engineers with tools that allow to apply all modern development rules to legacy Fortran code (re-engineering, parallelization, maintenance, quality assurance, etc). Foresys comes with a full Fortran90/95 environment, including parser, several analyzers, and a pretty-printer. Foresys analyzers run on any size of Fortran source code to automatically create a global information database called ForLib. The first goal was to provide a fully automatic transformation system to automate the porting of Fortran70 code to Fortran90. The integration of TrfL in Foresys is to instantiate the TrfL system for Fortran, that is provide an API to access the structures that analyzer tools produce (symbol table, typing information, control flow graph) and derive an engine to perform transformations on generated ForLibs. The four steps of the transformation process are:

1. Foresys parses the Fortran program, it gives a syntax abstract tree on which transformations can be performed,
2. then it runs analysis tools and creates a ForLib. This gives the contextual information.
3. transformations can be performed using the TrfL engine and finally
4. the transformed tree is pretty-printed.

The engine only works in batch mode and offers several tree walks : *bottom-up*, i.e. the tree is visited in postorder, or *top-down*, i.e. the tree is visited in preorder. In addition to the vertical direction, the user can choose the horizontal direction: left-to-right or right-to-left. This determines in which order the children of a given node are visited. For example, in case of a bottom-up left-to-right traversal, the tree is walked through inspecting the leftmost leaf first. The tree walk can be declared *monotonous* or *strictly monotonous*. In the first case, it guarantees that, after a transformation application, the next entry point in the tree is the transformed code, in the second case the root of the transformed code is not searched for a new transformation.

When the engine is running, it must find the current valid symbol table in order to pick up some contextual informations. Some of the analysis results are global, i.e. they are

valid for the whole program, and some are local to a special fragment of the program. This is the case for symbol tables. The scope of symbol tables is restricted to precise pieces of program. So symbol tables are hooked on the root the tree where they are valid.

6. Example

This simple transformation replaces an equality test where the two operands are of floating type with an approximation equality test. For instance a small sample code:

```
if (a .eq. b)
 ...
```

and the transformed code:

```
if (abs(a - b) .lt. (10.0 * spacing(a)))
 ...
```

The source (or left) pattern matches equality tests and bounds the left and right subtrees that represent the operands of the test, respectively to X and Y, two metavariables. This transformation is mostly syntactic, although typing information is needed to check the applicability of the rule. The application condition checks that both operands are of floating type, calling the predicate *isFloating* that returns true or false. This predicate is part of the API that gives an access to contextual information computed by Foresys analyzers. When both the left pattern matches the input tree and the application condition is evaluated to true, the matched tree is replaced with the target pattern.

```
-- the source pattern: (X.eq.Y)
eq(X, Y)
 provided #:fsys:isFloatingType(X) &
        #:fsys:isFloatingType(Y)
 ->
-- the target pattern: (abs(X - Y) .lt. (10.0 * spacing(X)))
lt(func_call(name "abs", l_exp(sub(X, Y))) ,
   mul(real_cst "10.0", func_call(name "spacing", l_exp(X))))
```

7. Discussion

Program transformations is still largely confined to research laboratories. In the past, the main interest in program transformation has been the generation of programs from specifications [13, 21]. But since the Year 2000 problem and legacy code problem, industrial use of program transformation has emerged mostly in software maintenance applications [22, 28].
In our case, i.e. the restructuring of Fortran programs, we can find several points why the use of a program transformations system is relevant. What are the questions to ask when you want to transform a program: How many time will it take to design a transformation ?, How many time will it take to apply this transformation ?, Can we re-use this transformation ?, Will it be easy to design and then apply a transformation ?, etc. An user has three possibilities: 1) he can use a text-based editor with a string-search-replace capability or tool like awk or sed, 2) he can develop his own ad hoc transformations in his favorite programming language or 3) he can use a special purpose language for transformation. Let's list the advantages and disadvantages of the three approaches:

1. *text-based tools*
   Advantages: Regular expressions are easy to use and since these tools are based on text processing you don't need a parser that builds an abstract syntax tree; also you can process non correct programs. Text editors or shell scripts are suitable for small low level syntactic transformations which need to be performed efficiently.
   Drawbacks: With text-based patterns you don't see the underlying tree structure of the program, and so specifying a complex pattern may be difficult and error-prone. Complex transformations require contextual information; with an editor, the user is the only source of contextual information (at the condition that he understands the program!), and if the user needs to confirm the application of the transformation, it may be time consuming and error-prone (the user may loose vigilance on a repetitive task). With a script, he must program the analysis. Transformations specified using these tools are not really re-usable. It will be difficult to derive a new transformation from the formers.
2. *hand-coded transformations*
   Advantages: the user uses his favorite language, he does not need to learn a new language. The transformation will be really adapted to the user needs and optimized.
   Drawbacks: the time of design and programming may be important if the user needs a parser, abstract tree manipulation functions and analysis tools. These tools may be built once, but in this case let's built a program transformation system. The code of the transformation may not be re-usable if the programmer is the only one who understands what the transformation does.
3. *Domain Specific Language approach*
   Advantages: the language is a special purpose language dedicated to the specification of transformations, it has qualities and facilities to build and apply transformations: high-level patterns description, high-level transformations description, integrated environment (analysis tools, parser, pretty-printer), batch and interactive transformation engines, re-usable libraries

of transformations.
Drawbacks: the user must learn a new language and get familiar with the environment. For complex transformations, the help of an expert is required.

The first two approaches can not be applied in an industrial context, since they do not fit software engineering criteria: minimize time and cost of development and maintenance, enhance modularity and re-usability. The third approach can be used in any process of the software life cycle : from requirement analysis you obtain specifications, from specifications you derive code, and then, code can be documented and maintained.

8. Conclusion

Software maintenance is the first widespread use of program transformations technology [7, 8, 24]. Scientific Fortran codes live for decades and so Fortran code to transform can be counted in millions of lines; a manual process is inconceivable in terms of time and reliability. Moreover, Fortran programmers are in general, not from the software engineering community. The need of transformation tools that automate or semi-automate the transformation tasks is obvious. Our contribution is TrfL, a high-level language to specify program transformations and a batch oriented transformation process that can be integrated in other environments. Future work includes:

1. providing an interactive engine, to build and apply transformations. Some transformations, like optimization, had to be done in an interactive way, so that the user can proceed by trial and error, in order to choose the right transformation. The features that will help will be undo/redo, history of applied transformations, and user customizable engine. Some experiences have been carried out in building interactively transformation rules in the Centaur system, and we planned to soon implement this feature for TrfL;
2. build up a larger transformation library. A large Fortran program has many authors and a lifetime of many years. Code can be inherited from other projects. Differences in style, in management and in the available programming tools lead to the creation of a dusty code. So other transformations of interest are code instrumentation, change of programming style and convention, and documentation. Re-structuring and documentation are closely linked, since one may change the other.

In the next few years, transformation technology will integrate many software engineering processes.

Acknowlegments: I would like to thank Laurent Hill (from SIMULOG) and Isabelle Attali (from INRIA) for their help and support.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[2] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L*. Springer LNCS 183, 1985.

[3] I. Baxter and C. Pidgeon. Software change through design maintenance. In I. Press, editor, *Proceedings of International Conference on Software Maintenance '97*, 1997.

[4] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the System. In *The Third Symposium on Software Development Environments (SDE3)*, Boston, 1988. ACM SIGSOFT'88. Also appears as Inria Research Report no. 777 (December 1987).

[5] J. Boyle. A transformational component for programming language grammar. Technical Report Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[6] J. Boyle. Abstract programming and program transformations - an approach to reusing programs. In T. Biggerstaff, editor, *Software Reusability*, pages 361–413. ACM Press, 1989.

[7] J. M. Boyle and M. N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5):574–588, Sept. 1984.

[8] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software reengineering. In *Proceedings. Fourteenth Annual International Computer Software and Applications Conference*, pages 314–22, Chicago, IL, 31 Oct.–2 Nov. 1990. IEEE.

[9] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of The International Conference of Computer Languages*, pages 280–285, Miami, FL, Oct. 9-13 1988.

[10] D. W. D. Popart: Producer of parsers and related tools system builders'manual. Technical report, USC/Information Sciences Institute, Nov 1993.

[11] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG - A generator for efficient back ends. In B. Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pages 227–237, Portland, OR, USA, June 1989. ACM Press.

[12] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, Jan. 1982.

[13] M. Feathers. A survey and classification of some program transformation approaches and techniques. In L. Meertens, editor, *Program specification and transformation*, pages 165–195. North-Holland, 1987.

[14] C. Ferdinand. Pattern matching in a functional transformation language using treeparsing. In I. P. Deransart and J. Mauszynski, editors, *Programming Language Implementation and Logic Programming*, pages 358–371, Aug. 1990. Linköping, Sweden.

[15] C. Fraser, R. Henry, and T. Proebsting. BURG – fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[16] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[17] J. Grosch. Puma - A generator for the transformation of attributed trees. Compiler Generation Report No. 27, GMD, Forschungsstelle an der Universität Karlsruhe, Nov. 1991.

[18] J. Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 26, GMD, Forschungsstelle an der Universität Karlsruhe, July 1991.

[19] T. C. S. Group. *The Munich Project CIP, Vol. 2: The Program Transformation System CIP-S*. Springer LNCS 292, 1987.

[20] R. Heckmann. A functional language for the specification of complex tree transformations. In *2nd European Symposium on Programming, Nancy*, pages 175–190. Springer-Verlag, New York, NY, 1988. Lecture Notes in Computer Science 300.

[21] B. Hoffmann and B. Krieg-Brueckner. *Program Development by Specification and Transformation*. Number 680 in LNCS. Springer-Verlag, Berlin, 1993.

[22] S. S. Inc. Coboltransformer–peek under the hood, 1997. Available at http://www.siber.com/sct/tech-paper.html.

[23] M. Jourdan and D. Parigot. Internals and externals of the fnc-2 attribute grammar system. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 485–504. Springer-Verlag, New York–Heidelberg–Berlin, June 1991. Prague.

[24] G. B. Kotik and L. Z. Markosian. Program transformation: the key to automating software maintenance and reengineering. In *IEEE Trans. Software Eng.*, volume 16(9), pages 1024–1043, 1990.

[25] P. Lipps, U. Möncke, and R. Wilhelm. OPTRAN: A language/system for the specification of program transformations—system overview and experiences. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1988. Berlin.

[26] T. Reps and T. Teitelbaum. *The Synthsizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, NY, 1988.

[27] D. R. Smith. Kids: A knowledge-based software development system. In E. M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[28] P. M. T. Harmer and J. Boyle. Using knowledge-based transformations to reverse engineer cobol programs. In *In 11th Knowledge-Based Software Engineering Conference*. IEEE-CS-Press, 1996.

[29] J. Vollmer. *The compiler construction system GENTLE – manual and tutorial*. GMD – Bericht 508, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1991.

[30] J. Vollmer. Experiences with Gentle: Efficient compiler construction based on logic programming. In J. Maluszyński and M. Wirsing, editors, *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP 91)*, number 528 in Lecture Notes in Computer Science, pages 425–426. Springer Verlag, Aug. 1991. system demonstration.

UML Formalization and Transformation Dissertation Proposal Abstract

Jeffrey E. Smith
Northeastern University, Boston, MA
jsmith@coe.neu.edu

Abstract

Using computer-aided, formally developed specifications to build and verify software leads to provably correct code, deeper consistency checking and specification reusability. The problem with applying this approach is its rift with mainstream commercial software engineering tools and development/specification methodologies. The primary goal of this research is to bridge this rift. The purpose of my research is to construct formal and CASE development methods by formalizing the common CASE graphical specification language, i.e. the Unified Modeling Language (UML), and automating the transformation from UML diagrams to a formal representation.

1. Introduction

It has been shown that the Transformational Programming Paradigm (TPP) of developing, verifying and maintaining software at the specification level leads to the development of provably correct code, deeper consistency checking and specification reusability. The problem with applying this approach is its rift with mainstream commercial software engineering tools and development/specification methodologies. The primary goal of this research is to bridge this rift by mapping from a popular Computer-Aided Software Engineering (CASE) tool modeling language to a formal methods language. Formal methods involves the specification of a formal syntax and semantics to specify system behavior so that consistency, completeness and correctness of complex systems can be assessed systematically.

Prior research has demonstrated the possibility of automating the TPP by deriving a technique to translate OMT (Object Modeling Technique) diagrams to a formal representation that lends itself for use with tools that support theorem proving and automated code generation. Some of this OMT formalization research was category theory and algebraic based, and drew from diverse theoretical foundations. The purpose of my research is to construct a new TPP by changing the paradigm for the more comprehensive UML and developing a path to automating the UML to formal representation transformation.

2. Problem

There are two primary computer-aided software development techniques in contemporary use for developing robust applications. One technique supports and enforces modern graphical-based software engineering methodologies with a CASE tool. While CASE tools do help with the construction of diagrams associated software engineering development methodologies (e.g. OMT, UML, Booch, Statecharts, etc.), library level code generation and syntactic level error checking, they come with disadvantages that preclude their exclusive use in critical software development. These disadvantages include ambiguous semantics and syntax, inconsistency between different diagrammatic views of the same system and the inability to generate more than header file software.

The other primary, but less used, technique is computer-aided support of formal methods. This technique overcomes many of the ambiguity and inconsistency problems associated with the CASE-based technique. There have been many impediments to this approach, despite its potential for more verifiable software development. These problems include the 1) Tower of Babel of supported logics, tools and formal languages, 2) lack of developers and end-users trained in formal methods, 3) rift between modern object-oriented architecture/design and formal methods based code generation and 4) apparent non-scalability of current formal methods based software successes compared to the size of large-scale applications associated with mission critical software.

Fortunately, CASE tool vendors are converging on a graphical software engineering methodology standard

that encompasses the previously mentioned popular diagrammatic methods. This standard is UML. If a formal UML syntax and semantics were defined, one could bridge the gap between the described computer-aided software development techniques by translating between the well defined CASE tool front-end to a formal software development environment, creating the best of both approaches. My research will take a major step in building this bridge.

3. Current Solutions

Methods for deriving algebraic specifications from object model diagrams have been described based on OMT diagrams. Bourdeau and Cheng (Figure 1.a) used 1) instance diagrams, formalized as algebras, to provide a graphical definition of semantics for object models and 2) the original object models, formalized as algebraic specifications, to provide an algebraic definition of semantics.

It is argued that either method for deriving semantics of the object model will yield the same set of algebras, since Figure 1.a commutes. Wang, Richter and Cheng later expanded this research to include dynamic and functional models, defining a set of semantics for the complete OMT model.

DeLoach (Figure 1.b) continued a similar reasoning process, using categories as the specification representation. His notion was to use category arrows to map from the internal structure between category objects (objects in the category of interest), so specification morphisms can map the axioms in a specification to theorems of the derived specification. His research used a similar commutative diagram method to check the consistency of results. In his research, DeLoach had shown the completeness and efficacy of his OMT formalism approach through formal rules describing how to map between a derived generic OMT AST, an O-Slang AST (O-Slang is an object-oriented variant of Slang, the representation compatible with Specware©, developed to capture classes as algebraic specifications and class relationships as category operations) and OMT semantics. The OMT semantics DeLoach described was based on formal approaches described for the static object, functional model and dynamic model.

Fraser described a framework of organizing and classifying strategies for incorporating formal specifications into the software development process for the purpose of:

1. identifying commonalities and differences between strategies,

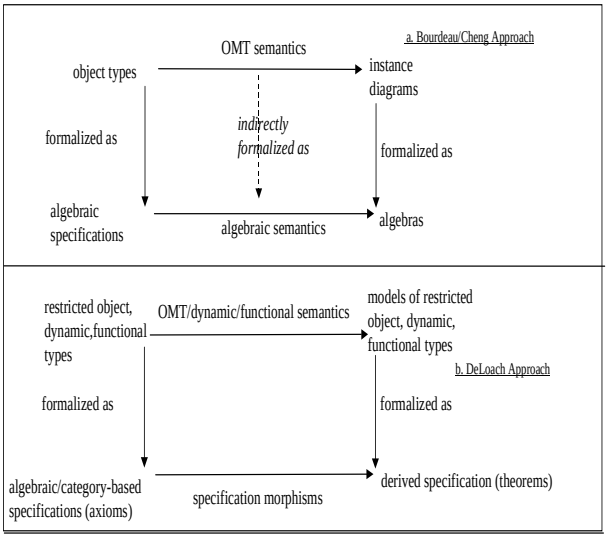

Figure 1. Specification Formalization Methodologies

2. assessing their applicability in different contexts,

3. making sense of competing proposals for using formal methods in the software development process,

4. identifying the advantages and disadvantages of each proposal and

5. identifying the gaps in proposed strategies.

Fraser's two-dimensional framework identifies four generic strategies: direct unassisted, direct computer assisted, transitional unassisted and transitional computer assisted. The direct category, i.e. where software developers move directly from informal to formal specifications without going through any intermediate semiformal representation, is contrasted with the transitional category, that does use an intermediate representation. Fraser defines two subtypes of transitional categories, sequential and parallel successive refinement. In a sequential transitional approach, the semi-formal specifications are fully defined and then transformed into a formal specification. In the parallel successive refinement approach, the semi-formal and formal specifications are produced and refined simultaneously. The unassisted category, where all the translation is performed manually, is contrasted by

Formalization Process		Formalization Support: Unassisted	Formalization Support: Computer assisted
Direct		Kemmerer Wing	Miriyala & Harandi
Transitional	Sequential	Andrews & Gibbins Kung	Babin, Lustman & Shoval Fraser, Kumar & Vaishnavi Cheng et al Robbins et al Paredes et al DeLoach Proposed Research
	Parallel Successive Refinement	Conger et al	

Figure 2. Formal Methods Strategies

the computer-assisted category, where computer-based transformation tools help the developer.

I use Fraser's framework to categorize my research so I can compare the most applicable of portion this broad base of research. This research falls into the border between the transitional-sequential computer-assisted and the transitional-parallel successive refinement computer-assisted categories. Computer assisted because of tool support with the translation of a formalized, constrained UML to a formal language and with stepwise refinement, composition and theorem proving. My work is transitional because of the use of several levels of intermediate representations, viz. UML → UML AST/BNF → UML-Slang → Slang. It is sequential in the sense that the process begins with a semi-formal UML specification, but also successive refinement in that I do provide for consistency checking between parallel UML views of software design. Figure 2 shows examples of Fraser's classification, with pointers to comparable research in my self-assessed classification.

4. Approach

A UML formalization process overview is portrayed in Figure 3. The abstract syntax of both UML and formal language domain theories will be mapped to derived UML Formal Semantics to provide a formal system that well-formed notations can be proved in. Showing the mapping rules are consistent will be accomplished by demonstrating that center portion of Figure 3 commutes.

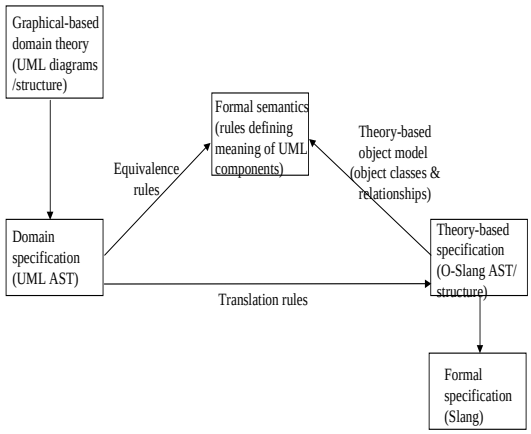

Figure 3. UML Formalization Process

The UML to formal methods mapping will be accomplished by the steps listed in Figure 4.

5. Objectives/Contributions

As mentioned earlier, the goal of this research is to formalize both UML, and the software derived from UML, in an extensible and automatable method that supports specification composition and CASE tool interoperability. In particular, the objectives that follow from this goal are enumerated in Figure 5, along with my proposed contributions for each of the enumerated research objectives.

6. Summary

My dissertation (at ftp.coe.neu.edu, /incoming/users/jsmith/paper.ps) proposal represents my plans, and associated examples, that I propose to complete for my actual dissertation. My primary motivation is to build that framework that combines the advantages of CASE tools with the advantages of formal methods systems by:

Task Name	Task Description
1. UML Syntax	Construct ASTs of the constrained UML, developed in the UML Semantics Task, to build a concise, unambiguous, text form of generic UML, independent of any specific CASE tool
2. UML Semantics	Construct the formal semantics of UML, derived from the semi-formal semantics and relevant UML formal representation research
3. Theory-based Intermediate Representation	Construct an object-oriented, theory-based intermediate representation, compatible with tools that support theorem proving, software generation composition and refinement
4. Mapping Rules	Derive formal rules that permit mapping between UML syntax/semantics and the theory-based intermediate representation, serving as a completeness check and verification of each of these three UML representations
5. Automation	Define and implement the automation process, i.e. translate exemplary UML forms of a software specification, to representations compatible with a software tool that utilizes the theory-based intermediate representation

Figure 4. Research Summary Descriptions

- formalizing the popular CASE tool graphical specification language, UML,
- translating the formalized UML to a formal language use by a tool supporting theorem proving, software composition, code generation and refinement and
- proving that the translation between the graphical specification and the resulting formal translation is viable.

This research will contribute to future evolutions of UML and help bring formal methods into the mainstream of common software engineering practice.

Objective	Contribution
1. Development of method to support composition of software specifications, logical theories and formalizations so that larger entities may be constructed from smaller components and checked for consistency, completeness and redundancy	Development of techniques to ensure consistency of object-oriented specification composition (based on forming colimits of the category-based components derived from UML components)
2. Development of method to support interoperability with other CASE tool modeling languages and other formal methods systems	Formalization of the translation from UML to an algebraic specification
3. Construction of framework to permit formalizations to be extended when new features are added to the CASE tool modeling language	Incorporation of object-oriented and type extensions to the theorem prover/code generation tool
4. Minimization of the human effort needed to create formal models & to an algebraic specification	Automation of translation from UML
5. Improvement of UML understandability through unambiguous syntactic diagrams and explicit, verifiable semantics	Construction of formal UML syntax and semantic contributions to future versions of UML
6. Development of a technique to reduce errors in the translation from graphical specification to software	Development of a technique to formally prove correct translation of CASE specified models to formal methods systems

Figure 5. Contributions Associated with Each Objective

Dependence Analysis for Software Architectures

Judith A. Stafford
Software Engineering Research Laboratory
Department of Computer Science
University of Colorado, Boulder
judys@cs.colorado.edu

1. Introduction

Software architectures model systems at high levels of abstraction. They capture information about a system's components and how those components are interconnected. Some software architectures also capture information about the possible states of components and about the component behaviors that involve component interaction; behaviors and data manipulations internal to a component are typically not considered at this level.

Formal software architecture description languages allow one to reason about the correctness of software systems at a correspondingly high level of abstraction. Techniques have been developed for architecture analysis that can reveal such problems as potential deadlock and component mismatches [2, 9, 12, 17].

In general, there are many kinds of questions one might want to ask at an architectural level for purposes as varied as reuse, reverse engineering, fault localization, impact analysis, regression testing, and even workspace management. These kinds of questions are similar to those currently asked at the implementation level and answered through static dependence analysis techniques applied to program code. It seems reasonable, therefore, to apply similar techniques at the architectural level, either because the program code may not exist at the time the question is being asked or because answering the question at the architectural level is more tractable than at the implementation level.

This research introduces *chaining*, a dependence analysis technique for software architectures. In chaining, links represent the direct dependence relationships that exist in an architectural specification that, when collected together, produce a chain of dependencies that can be followed during analysis. The traditional view of dependence analysis is based on control and data flow relationships associated with functions and variables [1, 5, 8, 15, 19, 20]. This research takes a broader view of dependence relationships that is more appropriate to the concerns of architectures and their attention to component interactions. In particular, both the structural and the behavioral relationships among components expressed in current-day formal architecture description languages, such as Rapide [11] and Wright [2] are considered.

2. Related Research

This work builds on previous and related work in three primary areas: traditional dependence analysis techniques, novel approaches to slicing, and applications of static concurrency analysis tools to architecture descriptions.

ProDAG [21] is a program dependence analysis tool set that performs statement-level dependence analysis. ProDAG allows one to create and access various predefined relationships originally identified by Podgurski and Clarke [20]. The technique of chaining, the dependence analysis technique proposed as a focus of this research, raises these ideas to the architectural level, as well as incorporating the notion of structural dependence.

Chaining is similar in nature to Weiser's concept of program slicing [24]. Korel and Rilling recently proposed slicing at the module level as an aid to program comprehension [10]. Tip has provided a survey of traditional program slicing techniques [23]. Sloane and Holdsworth [22] suggest advanced applications for slicing, in which the basis for analysis includes aspects other than traditional data and control flow. They present a concept of syntactically based generalized slicing for use in slicing of non-imperative programs. I agree with the spirit of this work and, in some sense, am pursuing a similar goal, but in the particular context of software architectures.

Oda and Araki [18] first introduced the concept of static specification slicing for specifications written in Z. Chang and Richardson [4] extend this work with the introduction of techniques for creating dynamic slices. Both these ap-

proaches use traditional slicing criteria, whereas this work involves exploring relationships at the architectural level, where the concept of a variable is abstracted away.

Zhao, Cheng, and Ushijima [25] propose the system dependence net (SDN) as a representation of concurrent object-oriented programs. The SDN is used to find slices of CC++ (Concurrent C++) programs.

Zhao [26] has recently begun work in the area of dependence analysis of formally described software architectures. The work described in this initial paper is similar in nature to my proposal but is preliminary and the details are unstated.

Naumovich et al. [17] apply INCA and FLAVERS, two static concurrency analysis tools used for proving behavioral properties of concurrent programs, to an Ada translation of a description of the gas station problem that was written in the Wright ADL [3]. Their approach is to create a concurrent program that can simulate the intended concurrent behavior of the system. My work is aimed at developing general dependence analysis techniques that may, in fact, contribute to the enhancement of the static analyses already provided by these tools.

Other work relevant to my thesis work includes the work by Medvidovic [13] on the classification of ADLs, Craigen, Gerhart and Ralston [7], that of Clarke and Wing [6] on the state of acceptance of formal methods, and work in the area of software architecture recovery (e.g. Murphy, Notkin and Sullivan's [16] work on reflection models, and the work being done by Mendonca and Kramer [14] on software architecture recovery).

3. Methodology

Initial research has been done in the areas of identification of dependence relations among components and questions that are appropriate and interesting at an architectural level as well as the development of an architectural level dependence analysis technique called chaining.

Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions. These relationships may involve some form of control or data flow, but generally they involve *structure* and *behavior*. Examples of structural relationships *Includes*, *Import/Export*, *Inheritance* and examples of behavioral relationships are *Temporal*, *Causal*, *Input,Output*. Both structural and behavioral dependencies are important to capture and understand when analyzing an architecture. I am investigating the ways in which architectural dependencies are influenced by the primitive features of the architecture description language.

There are a variety of questions that should be answerable by an examination of a formal architecture description. Several architecture-level dependence related questions have been identified and will be used as a basis for validating this work. As examples:

1. Are there any components of the system that are never needed by any other components of the system?

2. If this component is communicating through a shared repository, with what other components does it communicate?

3. If the source specification for a component is checked out into a workspace for modification, which other source specifications should also be checked out into that workspace?

4. If a change is made to this component, what other components might be affected?

5. If a change is made to this component, what is the minimal set of test cases that must be rerun?

6. If a failure of the system occurs, what is the minimal set of components of the system that must be inspected during the debugging process?

These questions share the theme of identifying the components of a system that either affect or are affected by a particular component in some way.

In chaining, chains represent dependence relationships in an architectural specification. Individual chain-links within a chain associate components and/or component elements of an architecture that are directly related, while a chain of dependencies associates components and/or component elements that are indirectly related. To build a chain one determines a component or element to use as the origin of the chain and a relationship type that will help answer the question at hand. For instance, if the analyst is trying to discover why an error message was incorrectly emitted, then the chain would be constructed based on the event that generates the error message and the caused-by relationship. The chain that is produced will contain the reduced set of elements that could have been involved in the generation of the error message.

A language independent tabular representation for architectures has been developed to capture the relationships among architectural elements. The chaining algorithm is applied to this representation in order to discover chains of related component. Chaining has been used to help localize faults and discover anomalies in descriptions of a version of the well known gas station example as well as IBM's ADAGE avionics system. Both of these descriptions were written in the Rapide ADL. The gas station example is quite simple while the ADAGE example is large and complex.

Language constructs of various ADLs will be investigated in order to determine their implications for

architecture-level dependence analysis. The chaining technique is being implemented in an analysis support tool called Aladdin. At the highest level of abstraction, Aladdin's architecture is composed of three components, the language specific table builder, the language independent chain builder and the user interface. The table builder must be constructed for each ADL in order to determine the relationships that exist among the architectural elements. The table builder maps the elements modeled in the specific language to relationships known to Aladdin. The chain builder performs a transitive analysis over the table. The technique and the tool will be refined over the course of the next year. Work will also continue in the evaluation of dependence relationships that exist in architectural descriptions and the effectiveness of chaining in educing these sets of related elements from systems. Other issues to be studied are interlevel mappings, as well as scalability, modularity and incrementality of chaining.

As with any software engineering research, the ultimate usefullness of the technique is always a question. With respect to this work, in fact, historically subprogram slices, which are based on a concept similar in nature to chaining, have not been shown to be significantly smaller than the original program. Computing statement level slices for large systems is impractical, whereas architecture-level slicing may prove a good alternative. I intend to compare the characteristics of large and small systems to determine whether the level of coupling tends to be lower in larger systems. If this is true, the savings from applying such techniques will be greatest when applied to large systems.

Historically testing has concentrated on the implementation of the system, which has meant that it is considered fairly late in the development process. Eventually I intend to incorporate chaining into a complete life cycle software analysis and testing environment. My goals for this research are less ambitious however and include only the building of Aladdin which is intended to be used to support other automated analyses.

4. Evaluation of Results

Evaluation of this work will be accomplished as follows:

I will choose three languages that contain a variety of the previously identified dependence relationships. I will determine mappings between relationships modeled in each language and the relationships known to Aladdin, then apply chaining to specifications written in each of the languages.

The questions listed in the Methodology section are important questions for software engineers. I will obtain, or create, formal architectural descriptions of systems for which architecture based questions of these types is appropriate. Specific questions will be formulated for each architecture and Aladdin will be used to answer the questions. When selecting systems for this part of the evaluation, I will balance size against complexity so that the chosen systems are small enough to be understandable and for me to described formally in a reasonable amount of time if need be, yet complex enough to demonstrate the effectiveness of the tool.

Finally, an experiment will be performed involving the use of chaining to perform analyses of an architecture of an implemented, industrial system. Some companies' software development process involves a lengthy specification and peer review process before implementation is begun. The goal of this process is to uncover requirements and design flaws prior to the implementation phase. I will evaluate the effectiveness of automated dependence analysis in accomplishing this same goal. The purpose of the experiment is to show that time can be saved in the development process through the use of automated dependence analysis. The steps of the experiment are 1) Write a formal description of the system architecture based on the requirements documents provided for the system. 2) Determine what types of problems might be expected of the particular system based on a) prior experience of the company, or other companies who have had experience in the development of similar systems and b) expected functionality stated in the requirements documentation. 3) Determine which types of chains would reveal expected faults. 4) Apply Aladdin to the formal description of the system. 5) Compare the faults that are discovered automatically with those that were discovered by the design team during peer review.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, pages 21–26, May 1991.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, May 1994.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] J. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *Proceedings of the Fourth Irvine Software Symposium*, Irvine, CA, April 1994.

[5] J. Cheng. Slicing Concurrent Programs — A Graph-Theoretical Approach. *Lecture Notes in Computer Science, Automated and Algorithmic Debugging*, pages 223–240, 1993.

[6] E. Clarke and J. Wing. State of the Art and Future Directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, August 1996.

[7] D. Craigen, S. Gerhart, and T. Ralston. Formal Methods Reality Check: Industrial Usage. 21(2):90–98, February 1995.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Prog. Lang. Syst.*, 22(1):26–60, January 1990.

[9] P. Inverardi, A. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 46–63. Springer-Verlag, Sept. 1997.

[10] B. Korel and J. Rilling. Program Slicing in Understanding of Large Programs. In *Sixth International Workshop on Program Comprehension*, pages 145–152, Ischia, June 1998.

[11] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr. 1995.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, Sept. 1995.

[13] N. Medvidovic. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 60–76, Zurich, Switzerland, September 1997.

[14] N. Mendonca and J. Kramer. Developing an Approach for the Recovery of Distributed Software Architectures. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 28–36, June 1998.

[15] C. Moore, T. O'Malley, D. Richardson, S. Aha, and D. Brodbeck. ProDAG: A Program Dependence Graph System. Arcadia Technical Report UCI-90-25, Department of Information and Computer Science, University of California at Irvine, 1990.

[16] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, October 1995.

[17] G. Naumovich, G. Avrunin, L. Clarke, and L. Osterweil. Applying Static Analysis to Software Architectures. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[18] T. Oda and K. Araki. Specification Slicing in Formal Methods of Software Development. In *Proceedings of the Seventeenth Annual International Computer Software and Applications Conference*, pages 313–319. IEEE Computer Society Press, November 1993.

[19] H. Pande, W. Landi, and B. Ryder. Interprocedural Def-Use Associations for C Systems with Single Level Pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[20] A. Podgurski and L. Clarke. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sept. 1990.

[21] D. Richardson, T. O'Malley, C. Moore, and S. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119. ACM SIGSOFT, Dec. 1992.

[22] A. Sloane and J. Holdsworth. Beyond Traditional Program Slicing. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 180–186. ACM SIGSOFT, Jan. 1996.

[23] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[24] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society, Mar. 1981.

[25] J. Zhao. Static Slicing of Concurrent Object-Oriented Programs. In *Proc. 20th IEEE Annual International Computer Software and Applications Conference (COMPSAC'96)*, pages 312–320, Seoul, Korea, August 1996.

[26] J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, September 1997.

Dynamic Modeling in Forward and Reverse Engineering of Object-Oriented Software Systems

Tarja Systä
Department of Computer Science
University of Tampere
P.O. Box 607, FIN-33101 Tampere, Finland
cstasy@cs.uta.fi

Abstract

A prototype tool called SCED is used for modeling the dynamic behavior of object-oriented software as scenario diagrams and state diagrams. In SCED state diagrams can be synthesized automatically from scenario diagrams. When reverse engineering existing software, a parser and a debugger are used for extracting static and dynamic information, respectively. The parsed information is viewed as a nested graph using a reverse engineering environment Rigi. The debugged information is shown as SCED scenario diagrams and state diagrams. Static and dynamic views to the software can be improved and insured by comparing partly overlapping information generated by the parser and the debugger.

1. Introduction

Object-Oriented Analysis and Design (OOAD) methodologies support the designer in designing, visualizing, and documenting artifacts in object-oriented software systems. These methodologies provide notations and guidance to model both the static structure of the program and the dynamic behavior of the objects.

Variations of scenario diagrams and finite state machines are used in several OOAD methodoligies for dynamic modeling. In The Unified Modeling Technique (UML) [1] the corresponding diagrams are called sequence diagrams and statechart diagrams, respectively. In The Object Modeling Technique (OMT) [2] they are called event trace diagrams and state diagrams. A scenario diagram shows an object interaction arranged in time sequence during a particular execution of the system. Participating objects or classes are drawn as vertical lines and the interaction between them with horizontal arcs. A scenario diagram can also have participants outside of the system border, for example, a user giving inputs to the system. A state diagram shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

SCED [3] is a prototype environment built to support the dynamic modeling of object-oriented applications. SCED uses the OMT methodology as a guideline, although the resulting system could be useful for other methods as well, particularly for methods with a scenario driven approach. Despite the different purposes of scenario diagrams and state diagrams, they share common information. Hence, constructing one from the other can be partly automated. One of the basic observations behind SCED is that constructing scenario diagrams and fusing them into a state diagram can be supported by automated tools far more than is currently practiced. In [4], it has been demonstrated how a minimal state machine, which is able to execute all the given scenarios with respect to a certain object, can be synthesized automatically. This algorithm with few modifications has been implemented in SCED. On the other hand, scenario diagrams can be generated by animating the interaction of objects through a set of collaborating state diagrams. However, in contrast to conventional animation systems, in SCED one can add new behavior to the system during the animation process. This technique could be characterized as *design-by-animation*. By using state diagram synthesis and design-by-animation techniques in turns, the dynamic model can be refined semi-automatically: each iteration gives a more comprehensive set of scenario diagrams and more complete state diagrams.

Several tools are available for reverse engineering the dynamic behavior and static structure of existing software. Tools that focus on static aspects of the target system usually use parsers for extracting the software artifacts and their dependencies. Rigi [5] is an extensible and tailorable reverse engineering environment. The parsing system of Rigi supports several programming languages, and new parsers can

easily be added to it. The parsed information can be viewed as a directed graph using Rigi editor. Rigi also supports program slicing and building abstractions for the static views. These features are used for increasing the understandability and readability of the views.

The dynamic behavior of software can be extracted, e.g., by using a debugger, a profiler, or instrumenting the source code. Typical behavioral aspects to be extracted are: object and thread interaction, exceptions and errors, garbage collection, memory leaks, etc. A scenario is a natural, descriptive, and powerful way to record the object interaction. However, scenarios tend to grow rapidly when the target system gets more complicated. One way to deal with the scenario explosion is to detect behavioral patterns from the event trace. The automatic state diagram generation property of SCED provides another efficient way to deal with large event traces, to view the total behavior of a class of interest in a single view, and to examine its run-time behavior separately from the rest of the system.

2. Forward engineering

Most user interaction with SCED involves two independent editors: a scenario diagram editor and a state diagram editor. At any time while editing the scenario diagram, the user can select one participating object and synthesize a state diagram automatically for this object using a single menu command. The synthesis can be done for one scenario diagram only or for a specified set of scenario diagrams. Moreover, scenario diagrams can be synthesized into an existing state diagram.

When synthesizing a state diagram for an object, each scenario diagram is traversed from top to bottom from the point of view of a participant corresponding to that object. Each received event is mapped to a transition in the state diagram. Sent events are regarded as primitive actions that are associated with states. The synthesis algorithm attaches states to all actions and places at most one action into a single state. This is a restriction when OMT state diagram notation is considered. Hence, SCED provides algorithms for generating OMT state diagram notation for a synthesized and/or edited state diagram to simplify the state diagram while preserving its information. The generated OMT state diagram allows several actions placed into a state, actions attached to transitions, entry and exit actions of states, etc.

SCED scenario diagram notation differs slightly from UML or OMT ones. Some new concepts have been added in order to make SCED scenario notation more expressive. Like subroutines, a scenario diagram may consist of parts that have their own aims and characterizations. Such parts can be placed into a separate *subscenario* in SCED. These subscenarios can then be "called" instead of repeating their contents. SCED scenario notation also lacks some UML sequence diagram concepts, and some of the concepts have different graphical representations. For example, focus-of-control regions and timing constraints are not included in SCED scenario notation. The extended scenario diagram notation of SCED does not cause any major changes to the synthesis algorithm.

While the state diagram synthesis technique uses a set of scenario diagrams for generating a state diagram, design-by-animation technique uses a set of state diagrams for generating a scenario diagram. The state diagrams can simulate system behavior, sending events to each other and changing states according to received events. As long as external stimuli is not needed and the state diagram set represents a complete system, the event trace can be automatically generated. If that is not the case, the event tracing process halts whenever such undefined events are needed. In these cases the user helps the event tracing process to go on by providing the unknown behavior. Hence the resulting scenario diagram contains both automatically generated events and user defined events.

By using the state diagram synthesis and design-by-animation techniques in turns, a powerful design tool can be achieved. The dynamic modeling process is smoothly changed from a "water fall" type of modeling (first scenario diagrams, then state diagrams) to a more spiral and incremental way of modeling; the dynamic models for objects can be constructed semi-automatically by refining the state diagrams using a growing set of scenario diagrams and extending the scenario diagram set based on communicating state diagrams. Each iteration hence gives a more comprehensive set of scenario diagrams and a more complete state diagrams for the objects to be modeled. Each iteration also increases the degree of automation. The method is especially suited for modeling the behavior of one new component using the known behavior of other, predefined, and presumably correctly implemented components. For example, such predefined components could be classes belonging to a graphical user interface framework.

As a counterbalance to the state diagram synthesis property, scenario diagrams can also be desynthesized from the state diagram: the state diagram is updated by removing parts that correspond only to the scenario diagram to be desynthesized. Some support for consistency checking between scenario and state diagrams is available as well.

3. Reverse engineering

For fully understanding existing software both static and dynamic information need to be extracted. Static information includes typically software artifacts and their relations. In Java, for example, such artifacts could be classes, interfaces, methods, variables etc. The relations might include extending relationships between classes or interfaces,

method calls between methods, containment relationships between classes and methods or variables etc. Dynamic information contains software artifacts as well. In addition, it contains sequential information, information about concurrency and code coverage, etc.

Reverse engineering is not only applied to old legacy systems, it always needs to be part of forward engineering as well. In software development, reverse engineering the current static structure of the software helps the engineer to insure that the architectural guidelines are followed, to get an overall picture of the software, to document the implementation steps and so on. Reverse engineering the run-time behavior during the software development phase is essential. If the system seems to be irregularly unstable, tracing the bugs is possible only if the history and order of occured events is known.

The extracted information is not useful unless it can be shown in a readable and descriptive way. There are basically three kinds of views that can be used: static views, dynamic views, and merged views. The extracted information often consists of a large amount of detailed and low level software artifacts. Hence good views for showing the information is not usually enough, abstractions need to be build for making the views more clear and understandable. Figure 1 shows the main aspects of the problem.

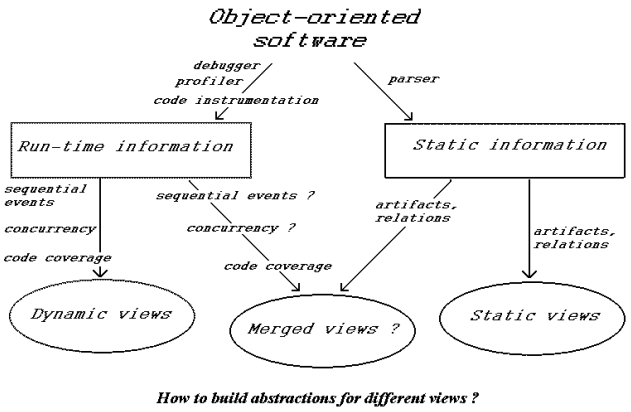

Figure 1. Different aspects of reverse engineering object-oriented software.

Rigi views static information in a form of a (nested) graph. It offers a graph editor and provides an extensible set of layout algorithms and algorithms used for program slicing and analyzing the software. Nested graphs enable showing the static structure of the whole system in a single view, and provide flexibility in browsing between different levels of abstractions built for the graph. This is an advantage compared to traditionally used class diagrams.

The state diagram synthesis facility of SCED provides an efficient way to view the total run-time behavior of one participant in a single view. The resulting state diagrams can then be used in design-by-animation approach when extending the design of the target system or designing another system that partly uses same classes as the current system.

When both static and dynamic information is extracted, also merged views can be used. Such views are usually formed by extending the static view by adding dynamic information into it. For example, code coverage information can be shown against a static view by giving weights for the corrensponding parts in the static view according their run-time usage. Merging static and run-time information has several advantages. First, the quality of the view can be insured by combining static and dynamic information. If both the parser and the debugger produces same source code artifacts and relations, the engineer can be quite confident that the artifacts are the right ones and the parser and the debugger works correctly. Second, the differencies between static and dynamic artifacts can be used to improve the views. For example, the parser cannot generate all default constructors if they are not explicitely written in the source code. The debugger can provide this piece of information. Third, merging information provides extended ways to do program slicing. For instance, based on dynamic information parts that are not used at run-time can be filtered out from the view. Slicing can also be made according to example scenarios.

Building abstractions for the views can be partly but not fully automated. Object-oriented languages support encaptulation. Such language structures can be used to build static abstractions automatically. For example, examing Java software by observing classes and their relations might clear the overall structure of the software, compared to studing it at the level of class members. In Rigi such abstractions can be built by collapsing all class method and variable nodes into a single class node, hence making the graph considerably smaller. Examing the structure in the class level might still contain too detailed information. The next step could be collapsing all classes and interfaces into packages, etc. Object-oriented metrics can also be applied for reasoning potential subsystems. Such subsystems could be groups of classes that are highly cohesive and have low coupling with other classes.

Dynamic abstractions typically differ from static ones. Building dynamic abstractions usually focuses on defining behavioral patterns and use cases. For example, initialization of a dialog might contain a sequence of events that are executed in a row every time the dialog is opened. Instead of repeated the whole event sequence, one single “dialog initialization” event could be considered. An example of a use case might be “withdrawing money using an ATM”. Such abstractions simplify sequence diagrams vertically: the number of events is decreased. Sequence diagrams can also be simpified horizontally by decreasing the

number of participants. This can be done by using the abstracted static model; sequence diagrams could show interaction between high level static compenents.

Building abstractions for merged views can be difficult, since the differencies between the static and the dynamic artifacts and their relations are not always complementary. For example, when overriding super class methods, polymorfism causes different method to be called than is actually written in the source code. Sequential information is often difficult to show in the same view with static information. In UML, collaboration diagrams can be used but the diagram gets difficult to read when the target software gets bigger. In general, the more information is added into a single view, the less readable it gets loosing its descritive power. Finally, building abstractions for merged views gets ambigious because dynamic and static abstractions usually differ considerably. When dynamic abstractions usually are behavioral patterns or use cases, static abstractions are subsystems. For example, most of the classes used by two use cases "withdrawing money using an ATM" and "paying a bill using an ATM" are the same and may belong to a single subsystem "ATM".

4. Current state of the research and future work

SCED is used for dynamic modeling in forward engineering of object-oriented software. The state diagram synthesis and design-by-animation features raise the level of automation in construction of the dynamic model. A prototype environment for reverse engineering Java software has been built. A Java source code parser is used for extracting static code artifacts and their relations for the target Java application or applet. The extracted information is viewed with Rigi editor. Rigi environment is also used for building abstractions and for program slicing. A Java source code debugger produces event traces consisting of basic object interactions. These event traces are shown as SCED scenario diagrams. The total behavior of an object can be viewed as a synthesized state diagram. Synthesized state diagrams can then be used in design-by-animation approach when designing new features for the target system. Some dynamic information is added to the static Rigi graph as well. The graph is extended with code coverage information and artifacts generated by the debugger but not recognizeded by the parser. Figure 2 shows the overall structure of the current system.

The emphasis in the future work is on examing how the dynamic and static views could contribute each other and when merged views could be used. Furthermore, the functionality of the Java debugger needs to be extended. Currently, the debugger produces method calls, constructor invocations, and thrown exceptions. However, the user should be given an option to choose information to be generated

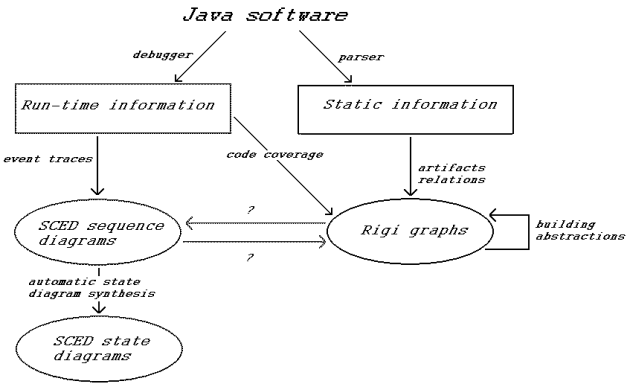

Figure 2. Current solution for reverse engineering Java software.

into the scenario diagrams. For example, in addition to currently generated events, the set of options might include: class variable assignments or accesses, if-else structures, repetition structures etc.

SCED has been built in a research project in co-operation with the University of Tampere, Tampere University of Technology, and several Finnish industrial partners. It is freely available at http://www.uta.fi/∼cstasy/scedpage.html or via ftp (ftp.cs.uta.fi, in directory /pub/sced). Rigi has been conducted by researchers in the Department of Computer Science at the University of Victoria. Rigi can be downloaded from http://www.rigi.csc.uvic.ca/.

I wish to thank Kai Koskimies and Hausi Müller for supervising my work. The SCED project has been financially supported by the Center for Technological Development in Finland (TEKES), the Nokia Research Center, Valmet Automation, Stonesoft, Kone, and Prosa Software. My current research is financially supported by Tampere Graduate School and Academy of Finland.

References

[1] Rational Software, *Unified Modeling Language, version 1.1*, [http://www.rational.com/uml/documentation.html], 1998.

[2] J. Rumbaugh et al, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[3] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "Automated Support for Modeling OO Software", *IEEE Software*, **15**, 1, January/February 1998, pp. 87–94.

[4] K. Koskimies and E. Mäkinen, "Automatic Synthesis of State Machines from Trace Diagrams", *Software—Practice & Experience*, **24**, 7, pp. 643–658, 1994.

[5] H. MÏller, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology", In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

Vorlon: A Visual Object-Oriented Approach to Parallel Application Development

Jim Webber (James.Webber@ncl.ac.uk)
Department of Computing Science,
The University of Newcastle upon Tyne, UK

Abstract

Software engineering is about to undergo a fundamental change. Compelled by ever-expectant users and the impending problem of Moore's Second Law, the reliance on increasingly powerful hardware to support increasingly demanding software must soon end. To keep abreast of the demand for increasingly responsive and functionality rich applications, software engineers must look to ways of creating programs which are insulated from the effects of Moore's Second Law through the application of parallelism. The Vorlon programming language conceals the detail of complicated parallel hardware by using an abstract, machine independent, visual approach to developing applications. Development is underpinned by the object-oriented methodology and supported by a CASE-style tool that bears the responsibility for producing parallel code from Vorlon graphs. Vorlon sets out to demonstrate the combination of a powerful programming paradigm (object-oriented) together with the application of computer graphics can enable engineers to design and build parallel applications to the same level of engineering that sequential applications enjoy.

1. Introduction

As the field of computing evolves, society has come to expect ever more ambitious and powerful software to support its activities. From the user's perspective, there should be no reason why software should not continue to evolve in terms of functionality and responsiveness. Software itself has become reliant on increasingly powerful hardware to deliver functionality in a timely fashion. Presently, software practitioners are supported in meeting user requirements by continual substantial increases in performance from hardware.

Unfortunately, this almost euphoric atmosphere within which modern software is developed and used may soon end. Already hardware manufacturers have begun to experience the effects of Moore's Second Law as the price of chip fabrication facilities increases with each decrease in component size. If hardware follows an evolutionary rather than revolutionary development path, it is a reasonable assumption that the economic repercussions which prevent an continuous improvement in hardware performance must impact software at some point in the near future. In effect, software will be stretched between ever more demanding users and a reduction in the rate of increase in power from hardware.

It is paramount that the end user is not aware of the existence of the "software stretch" and can continue to enjoy the benefits of increased application functionality and response. As hardware seems destined to be unable to provide continual performance improvement to the user, the burden must fall to the software community. Whilst at first glance the future for software practitioners looks somewhat austere, there is hope in that although hardware will continue to evolve upwards at a greatly reduced rate, single processor performance gains are not the only way to achieve faster computation. One possible solution for counteracting the software stretch would be the deployment of multiprocessor computer systems, a vision already embraced by some within the scientific computing field[1].

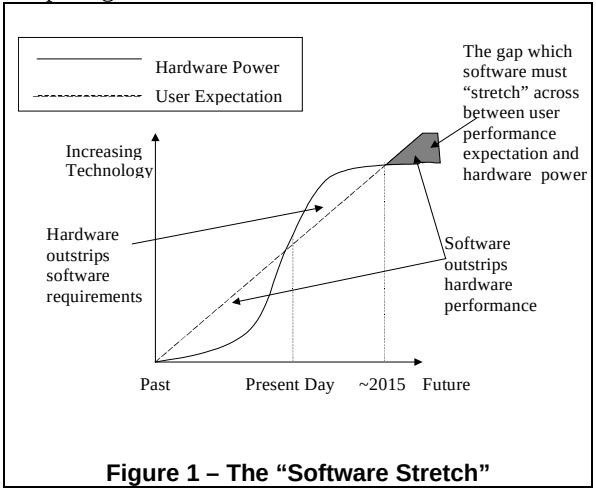

Figure 1 – The "Software Stretch"

To date, the majority of parallel computing applications have not been developed using a software process. Correspondingly, little parallel technology has

been adopted by mainstream software engineers, though the field itself remains an active research topic [2].

The present low level of software engineering techniques applied to traditional parallel computing problems is excusable, in that the majority of parallel computing platforms are currently used to solve computationally intensive problems whose functionality is often limited to a single, highly domain-specific problem. It is entirely feasible that for such projects, the application of any kind of software engineering technique would be unnecessary because the problem domain is well known and solutions can be largely implemented immediately. Furthermore, as the goal of such applications is to solve a problem as rapidly as possible, low-level programming techniques which contravene "best software practice" are often used to optimise programs. Moreover, the developers of such software are rarely interested in creating well-engineered software. For such users the program is merely a means to an end, and not an end in itself.

Similarly, software which is rich in functionality has tended to be computationally less intensive often performing computation only in response to user requests, and thus largely obviating the need for explicit parallelism. This is a situation which will not continue. As user expectation continues to grow at a constant rate, the software which users run will become increasingly computationally intensive. Indeed there are already several classes of application, notably image processing, CAD, and gaming which are already able to utilise vast amounts of processing power.

It is an obvious question then, "Why has software engineering failed to embrace key parallel technologies?" The answer is twofold and simple: Up to now there has been little need to utilise parallel processing to deliver required functionality to the user, and perhaps most obviously that parallel applications are complicated to build, particularly in the absence of any high-level development models for parallel applications.

2. Developing Parallel Applications with the Vorlon Programming Language

Firstly, and probably most fundamentally, the underlying object-oriented paradigm empowers Vorlon above levels of its contemporaries [3-6] by offering a structured method of analysis, design, implementation and code re-use. In particular, re-use in parallel applications is of even greater importance than in sequential applications due to the high costs and potentially higher failure rate of re-writing parallel code. To date, no similar system has been built on such a powerful development paradigm. Instead, visual systems for developing parallel applications have been based upon paradigms that are aimed entirely at supporting the implementation phase of the development lifecycle, concentrating largely upon on low-level aspects whilst ignoring higher level activities.

The Vorlon approach to parallel application development relies on a visually based environment within which applications are developed from inception to release. Vorlon takes a dual-level approach to developing object-oriented parallel applications, where each level utilises a hybrid graphical-textual language for modelling and programming respectively. At the higher level, there is the class model, loosely based upon the UML class model, which provides a repository of types that will be used within programs. At the lower level, there exists a method graph that stipulates method functionality for each method declared in the class model. In effect, the class model provides type declarations, and the method models provide the definitions of those types.

The visual nature of Vorlon enables developers to visualise parallel execution in a straightforward way. Programming in two dimensions, as opposed to the single dimension offered by textual programming languages, naturally suits parallel programming where there may be more than one concurrent flow of control at any given moment. In addition, Vorlon also separates control flow and computation components of an application. Unlike textual parallel programming languages where complicated parallel control flow mechanisms reduce the clarity of code, all control flow in Vorlon is expressed graphically. The dichotomy between computation and control flow ensures that each remains uncontaminated by the other.

2.1 Analysis and Design: Vorlon Class Models

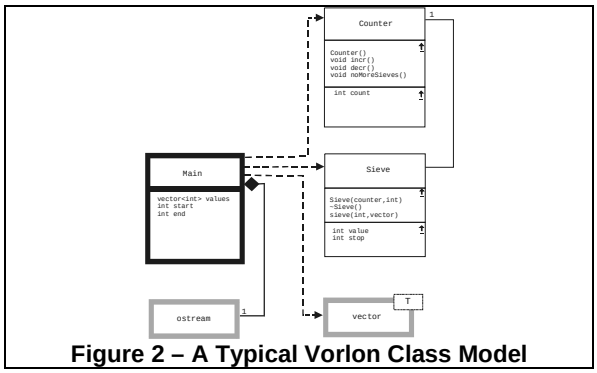

Figure 2 – A Typical Vorlon Class Model

The class model in Vorlon is where the developer performs problem domain analysis and high-level design stages of the software lifecycle. At this level,

there is no mention of parallelism, the user merely analyses and designs software in accordance with appropriate object-oriented analysis and design strategies.

Relations for inheritance, dependency, composition and uses are available to the developer. Though based upon the UML class model, Vorlon's class model more appropriately reflects the fact that one of the main axioms underpinning development with Vorlon is to produce high-performance, parallel applications, and as such its expressiveness is slightly reduced. The inclusion of a richer set of modelling primitives would certainly jeopardise this one fundamental goal and there has thus been a trade-off between expressiveness and run-time efficiency. However, it is believed that the level of support provided will be sufficient for the majority of applications as it is as expressive as previous work in the area of object modelling, such as the Coad-Yourdon, and Booch notations. In addition to the fact that Vorlon class models are somewhat keener than their UML equivalents, aspects which reflect the executable nature of Vorlon programs, such as a main *class-and-object* [7, 8] , are also included.

2.2 Implementing Vorlon Methods

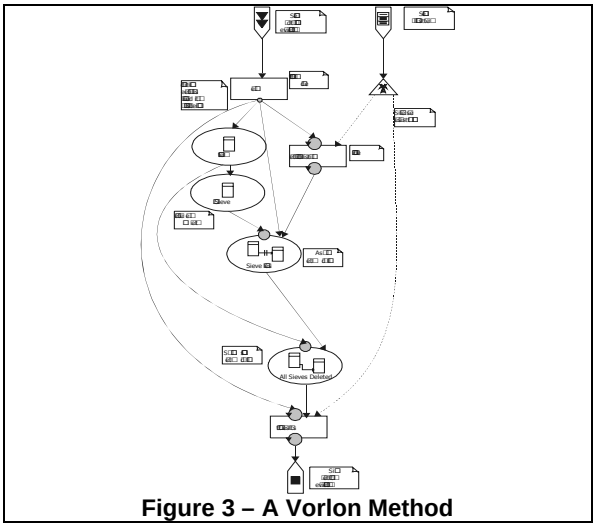

Figure 3 – A Vorlon Method

The Vorlon method model is somewhat more intricate than the class model. Method graphs are superficially similar to dataflow graphs, though unlike dataflow graphs it is not values that are transmitted along arcs, but object handles, which may be used to call methods on objects, and as parameters to method calls. As several handles to one object may be present on the graph at any one time, there are concurrency issues pertaining to the simultaneous method calls on that object. Such issues are resolved automatically by the object itself according to a relaxed version of the active-object model [9]. The relaxation adopted by Vorlon is that methods which do not update object state (those identified as **const** on within the class model) are free to execute in parallel, whilst methods which may update object state must run sequentially.

Each method graph consists of a set of special and general-purpose nodes connected by arcs which demonstrate control flow dependencies between those nodes. Parallel activity, including pipelined and task parallelism is implicit within the structure of the graphs, thus freeing the developer from the task of explicitly instigating parallel activity and synchronising concurrent processes. The only exception to this is data-parallel activity which must be explicitly programmed by the developer using the appropriate nodes types. In practice, the explicit data-parallel programming may not prove to be a great burden on the programmer, as when data-parallelism is required it is often obvious.

Method graphs are read top to bottom and may be hierarchically decomposed in a top-down fashion to aid navigation and limit information density on any particular graph. Hierarchical decomposition is not a means by which design is carried out, unlike contemporary systems which rely only upon top-down design, top-down decomposition is used only as a readability aid.

Like the data-flow model, it is the arrival of appropriate object handles at a node that cause its execution to begin. Vorlon clarifies exactly which of the handles arriving will cause the node to execute by insisting on the simplest of firing rules: all arcs must present an object handle before execution will commence. This has the effect of simplifying the semantics of the nodes, and clarifying the graph in that conditional execution is based purely on the arrival or non-arrival of handles, and not on some arbitrary combination of the two which has been to the detriment of other similar languages.

Once a node fires, the handles present at the head of the arcs involved are consumed. Node operation is generally synchronous in that for each set of input handles consumed there will be one output produced after computation has completed, though there exists an asynchronous method call facility which can be used when results are not required from method invocations. Correspondingly, there exists and event wait node which offers synchronization at the sub-graph level. The event wait node will delay the progress of any object handles which flow into it until a condition on a local data member is met.

Within each of the nodes, there exists a piece of sequential C++ which determines the actual functionality of the node. Depending on whether the node is general or special purpose, the developer is at

liberty to change certain aspects of that code. Within a general-purpose node, any sequential C++ can be embedded, whilst in the special purpose nodes, only parameters can be changed. In both cases, Vorlon automatically binds graph components to objects in the textual source code, which seamlessly integrates the textual and graphical components of the application.

3. Conclusions and Further Work

The application of object-oriented programming to parallelism is not new, nor is the concept that visual programming can reduce the inherent complexity of parallel programs. However, supporting the entire parallel application development process with object-oriented technology, using visual programming techniques to abstract away the complexities inherent within parallel architectures, and wrapping the whole system within an automated CASE environment *is* novel. It is believed that only by simplifying parallel programming and absorbing it into a software engineering process that its use will become accepted, and Vorlon provides a first attempt at satisfying these goals. It is the intention to develop Vorlon to a fully working prototype system and investigate the feasibility of the approach by tackling complicated applications normally not undertaken by visual parallel programming languages whose usage largely concentrates on algorithms already satisfactorily built with textual programming languages. Vorlon's success thus depends not on the attainment of speedup per se, but on the more subjective goal of supporting the developer in constructing properly engineered parallel software.

Less visibly, there are several ideas for altering the underlying execution model. One method of achieving greater speedup relies on the fact that Vorlon subgraphs declare which local data members are to be used within their scope. Using the information on scoping, it may be possible to execute several write-methods concurrently, without causing corruption. Under this model, methods that are independent in terms of the state that they access would be able to execute in parallel. Whilst the concurrency mechanisms are more complicated under these circumstances, there may be improvements in performance where the level of parallelism exploited is greater than the cost of implementing those concurrency mechanisms. Verification of this would require experimentation, though to the developer, there is the advantage that the language itself remains unchanged.

The method model component of the language could also be made to support the development of fault tolerant parallel applications without change to syntax or semantics. Arcs connecting nodes could be made to be transactional, and nodes themselves could be automatically replicated without explicit programmer intervention [10]. Furthermore, as well as describing normal control flow dependencies between nodes, a second view could be provided which would show exceptional program behaviour. The syntax and semantics of the exceptional behaviour would be identical to the standard control flow, thus supporting the construction of robust applications with a single programming paradigm.

Whilst Vorlon is unlikely to envelop fault-tolerance and optimistic method execution strategies in the prototype version, its development model could clearly assimilate them. It is hoped that future research based on Vorlon will lead to the development of a robust, general-purpose language for developing high-performance applications, which will enable future software to bridge the user-hardware gap.

4. References

[1] K. Kennedy et al, "A Nationwide Parallel Computing Environment," *Communications of the ACM*, vol. 40, pp. 63-72, 1997.

[2] I. Jelly et al, *Software Engineering for Parallel and Distributed Systems*: Chapman and Hall, 1996.

[3] R. S. Allen, "A Graphical System for Parallel Software Development," Department of Computing Science, The University of Newcastle upon Tyne, 1998.

[4] J. W. Harley, "Dataflow Development of Medium Grained Parallel Software," Department of Computing Science, The University of Newcastle upon Tyne, 1993

[5] J. C. Browne et al, "Visual Programming and Debugging for Parallel Computing," *IEEE Parallel and Distributed Technology*, pp. 7583, 1995.

[6] P. Newton and J. Dongarra, "Overview of VPE: A Visual Environment for Message-Passing Parallel Programming," The University of Tennesse, Knoxville, Knoxville, Technical Report ut-cs-94-261, 1994.

[7] P. Coad and E. Yourdon, *Object-Oriented Design*: Yourdon Press, 1991.

[8] P. Coad and E. Yourdon, *Object-Oriented Analysis*: Yourdon Press, 1991.

[9] R. G. Lavender and D. C. Schmidt, "Active Object: An Object Behavioural Pattern for Concurrent Programming," in *Pattern Languages of Program Design 2*, J. Vlissides, J. Coplien, and N. Kerth, Eds.: Addison-Wesley, 1996.

[10] O. Babaoglu et al, "Paralex: An Environment for Parallel Programming in Distributed Systems," presented at 6th ACM International Conference on Supercomputing, 1992.

Improving Reusability in the Process of Method Engineering

Zheying Zhang
Department of Computer Science and Information Systems
University of Jyväskylä, PL 35, FIN-40351 Jyväskylä, Finland
Email: zhezhan@cc.jyu.fi

Abstract

This research proposal examines how to improve method component reuse in a customizable CASE environment. Software reuse is an old topic that began from the end of the 1960s, while the component reuse in method engineering is relatively new. In my research work, the major objective is to study method component reusability in method engineering rather than general software component reuse. The research begins with an exploratory qualitative analysis of possibility and necessity of component reuse from both managerial and technical viewpoints based on a literature review. Afterward, it deals with method component reuse in a specific customizable CASE environment called MetaEdit+. The results highlight the importance of component reuse in customizable CASE environment.

1. Introduction

Software reuse was first touted as an approach to overcoming the software crisis at the end of 1960s [1, 2]. At the beginning, the interest in reusable software stemmed from the realization that one way to increase productivity during the production of a particular system is to produce less software for that system while achieving the same functionality [2]. In recent years, software reuse supported by object-oriented programming techniques and network techniques has become a technology whereby proven components can be cataloged, identified for reuse to improve system reliability and to reduce system cost.

Although the study of software reuse covers a wide research area and has achieved few promising results, current CASE tools and method engineering environment rarely provide a methodical and systematic approach to reuse in information system development. In order to improve component reusability in a customizable CASE environment, this study takes MetaEdit+, a configurable CASE and CAME environment [3, 4], as a platform to study component reusability in the customizable CASE environment.

This paper is arranged as follows. The necessity of reuse in customizable CASE environment is first presented in section 2; the possibilities for reuse are then analyzed based on four facets of reuse techniques in section 3; finally, the proposed research questions, feasible research approaches and expected contributions are outlined in the last section.

2. Necessity of reuse in CASE environment

Information system development (ISD) is a change process taken with respect to an object system in an environment by a development group using tools and an organized collection of methods to produce a target system [5]. Normally, the set of methods is integrated into the computer aided software engineering (CASE) environment to support ISD. A method is a set of steps or rules that define how a representation of an information system (IS) is derived and handled. To be able to successfully specify and present methods, we need tools and techniques on another level called method development level to describe the method's conceptual structures, which form a computer aided method engineering (CAME) environment. Method engineering is a discipline to design, construct and adapt methods, techniques and tools for systems development [6]. It needs support on two levels: ISD level and information system development methodology (ISDM) level. CASE and CAME are two corresponding environments to support these two levels of method engineering.

A customizable CASE environment provides facilities for method engineering process to construct homegrown methods or adapt existing methods to cater for specific ISD requirements. Normally, a method is made up of several components describing its structure, function, behavior or other unfunctional aspects. It is obvious that such a customizable environment stores a large number of components to specify diverse methods based on the same semantics. Furthermore, the number of components grows relatively large due to the continuous method development. Reapplying existing components to support the method specification processes is a way arising spontaneously, which introduces the concept of reuse. In general,

reuse is the reapplication of various kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of the other system. In a customizable CASE environment, reuse can be understood at least on two levels. On ISD level, the different system development applications can be reused by reapplying the knowledge such as architecture structures, requirements, code, and so on; on ISDM level, the method components can be reused by adapting their semantic specifications. In this study, I will make researches on component reusability on ISDM level.

3. Possibility of component reuse

A major impediment to reuse of software has been a mindset of always thinking in terms of new development [7]. The organizations do not take reuse into account in software process and method engineering process as well, although it is clear that reuse is an effective way to improve the quantity and quality of software product. The major reason is the deficiency in commercial environments where class libraries and efficient tools such as intelligent browsers and application generators can be integrated to effectively support the reuse process.

Although component reuse is not widely used in CAME environment, possible techniques supporting reuse exist. The diverse approaches and techniques can be found from the literature or industrial applications. Although most reuse processes belong to the system development process rather than the process of method engineering, the notions and techniques are still useful. The different reuse processes share a commonality that is they all have four basic facets: abstraction, selection, specialization and integration [1]. If possible tools can be provided to support the four facets of reuse, method component reuse is not a so difficult process. In the following, the possible techniques supporting component reuse are discussed based on these four facets.

3.1. Abstraction

Abstraction is closely related to software reuse. Every abstraction describes a related collection of reusable components and every related collection of reusable components determines an abstraction. Besides providing a concise description to each component, the issues regarding component management are too important to be ignored. Method component categorization and identification is related to the conceptual framework for IS and IS modelling [8-12]. In this proposal, I review briefly to the conceptual framework for IS and reference models [11, 12].

Conceptual Model for IS

The framework developed by Iivari [8-12] is a typical conceptual framework for IS. It is based on three levels of abstractions: organizational, conceptual /infological and datalogical/technical level. These three levels can be regarded as the requirement specification, system analysis and design, and implementation process in ISD. On each level, the abstractions describe its feature from three aspects such as structure, function and behavior.

The similar description for IS also appears in studies of ontological foundations for IS modelling which describe the system using things and properties (structure), systems (function) and dynamics (behavior) [13]. These three concepts in turn correspond to the three aspects of abstraction in Iivari's conceptual framework. Therefore, it is reasonable and feasible that an information system is viewed from these three aspects. They can accordingly be taken as three main viewpoints to category the components in the repository. Besides, other more detailed features describing each level of IS modelling [9], such as the goal structure, environment interaction, allocation aspect and so on, can be taken as the complementary facets for method component specification.

Reference Model

Besides using conceptual frameworks to specify IS modelling, reference models are widely used in understanding the business process. It describes a frame of reference for one or more standards. A frame of reference can be thought of as a set of conceptual entities, and their relationships, plus a set of rules that govern their interaction [14]. Reference model is usually not a standard model implemented by each user without modification, but a data model that usually requires some adaptation to the specific situation. For example, the reference model is a blueprint to describe the business process in SAP R/3 system [11, 12]. It is a description of a business domain acting as frame of reference for one or more standards. There are five kinds of models: process model, interaction model, data model, organization model and component model [11, 12]. The various models or viewpoints contained in the reference model address specific aspects of a company's real situation which supports the system development more easily and makes the interactions of people within an organization more effective. The reference model for business process provides detailed descriptions on the level of system development, but it lacks descriptions on the conceptual level, which limits its applications to a special system. Anyway, the basic concepts of reference model, combined with conceptual framework, can be applied to model the framework to support method engineering process. It should improve component reusability in the method engineering process.

3.2. Selection

Abstracted reusable components are typically stored in a repository for future locating, comparing and selecting. Techniques supporting component retrieval have proliferated in recent years. The most widely used are keyword

search, full text retrieval, structured classification schemata, and hypertext [15]. The first three are traditional approaches based on the classification of repository objects. Hypertext is a revolutionary technique based on navigational metaphors. Component selection can be achieved in conjunction with a variety of techniques such as visual presentation, which are useful references to component reuse in method engineering process.

3.3. Specialization and integration

Often, potentially reusable components only match partially to the functionality required. Although the intuitive approach is to adapt the component found, one might end up by doing far too much work for adaptation and integration. There are normally three techniques to handle the selected components to the proposed requirements [16-18]: composition-based reuse, generation-based reuse and derivation-based reuse. These techniques provide feasible ways to reuse the revised components.

In the four facets of software reuse described above, abstraction plays a central and oftentimes limiting role in each of the other facets. In this study, I will concentrate on constructing conceptual framework and managing the diverse method components to support method engineering process by reusing existing method components.

4. Improving component reuse in MetaEdit+

In this section, the research questions, approaches and expected contributions will be outlined on the basis of a specific customizable CASE environment, MetaEdit+ [3].

4.1. MetaEdit+: a configurable CASE and CAME environment

Due to diverse requirements for method development, improving the quality and productivity of methods has become an important issue. Accordingly, many meta-CASE environments, such as MetaView [19], Tool-Builder [20] and MetaEdit+ [3], have been developed to aid the process of method engineering and ISD. For example, MetaEdit+ is a configurable multi-method and multi-tool platform for both CASE and CAME. As a CASE tool it establishes a versatile and powerful multi-tool environment which enables flexible creation, maintenance, manipulation, retrieval and representation of design information among multiple developers. As a CAME environment it offers an easy-to-use yet powerful environment for method specification, integration, management and reuse [3].

As shown on the left side of Figure 1, MetaEdit+ is based on three levels of abstraction: ISD level, ISD meta level and ISD meta-meta level. The most abstract and highest level is meta-meta level that contains a set of primitive types needed as a language to specify methods on meta level. Different methods are specified and presented using the metamodeling language. Each method is made up of several method components, for example, an object diagram to specify the static objects and their relationships, a state transition diagram to present the behavior of a system in a time dimension, or others to specify the features of information systems. Different methods can be selected to support project modelling.

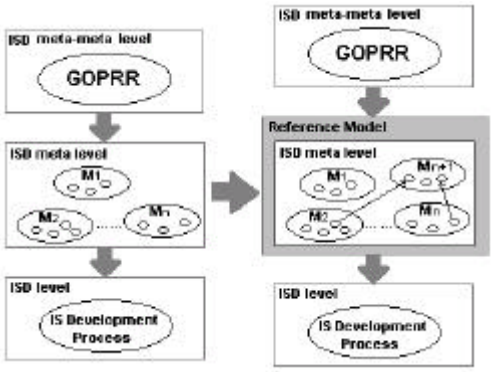

Figure 1 Reuse architecture for method components in MetaEdit+

4.2. Component management and reuse

As discussed, component reuse will improve the productivity and quality of system development. In MetaEdit+, a new method can be constructed and represented by reusing other methods' components instead of beginning from scratch. As shown on the right side of Figure 1, a new method M_{n+1} is made up of three components, two of which are adapted from the components in M_2 or M_n directly. Such reuse process reduces the efforts and improves the efficiency in method engineering process. Besides, the issue regarding component management should be taken as a foundation for efficient component reuse. Component management is a corner stone for the whole reuse process as discussed above. Being inspired from the reference models in business engineering process, we propose categorizing the method components into several typical sets and constructing reference models to specify the sets of components, the related components and their relationships to support method engineering process. As shown in Figure 1, the reference model takes a role to manage the components on the meta level of ISD.

Using reference models to support method engineering is a relatively new topic. The key issue concerns how to categorize the method components to effectively support component retrieval and further reuse. The reference models should present the way by which various models interact in the process of method engineering. They should guide engineers from the beginning, including component selection, evaluation and analysis, to the final stages of integration. And also, they should provide a comprehensive view of all the components and their in-

teraction in the repository. The followings are the research questions:

- How can important features of each component be identified for classification?
- What are the relationships among components?
- What is the suitable way to harmonize the notational conflicts among the components from different methods by using reference models?
- How can user's requirements be identified to conduct the whole reuse process?

4.3. Research approaches

In this study, it is indispensable to take system development as one of the research methods. Research questions should be formed in the course of observation such as survey and case studies, and then to be confirmed and generalized through analysis, which is called theory building. Both observation and theory building are the necessary parts to support the research work. Especially, theory building takes a central role to guide the prototyping in system development. In my research work, three approaches will be applied: observation, theory building and system development and evaluation.

4.4. Expected contributions

The research work would increase the reuse potential of previously developed specifications and specialize them for new system requirements. The proposed framework will organize the method components based on several aspects of the component abstraction; the reference model will present the relationship and interaction between different component models and their semantic features. Based on a detailed categorization and description, the components in the repository can be easily retrieved and adapted for reuse, which will decrease the effort to new method development.

It should be noted that although the research work will be carried out on the base of a specific customizable CASE environment, the principles and theory can be generalized and applied in the field of software engineering and method engineering.

Acknowledgments

I wish to express my sincere gratitude to my supervisor Professor Kalle Lyytinen and the members of MetaPHOR project for their encouragement, and guidance to this work.

References

[1] Krueger, C.W., *Software Reuse.* ACM Computing Surveys, 1992. 24(2): p. 131 - 183.

[2] Neighbors, J.N., *Draco: A Method for Engineering Reusable Software Systems*, in *Software Reusability*,1989. p. 295--319.

[3] Kelly, S., K. Lyytinen, and M. Rossi. *MetaEdit+: a Fully Configurable Multi-User and Multu-Tool CASE and CAME Environment*. in *Advanced Information Systems Engineering*, 1996: Springer-Verlag.

[4] Lyytinen, K., *et al.*, *MetaPHOR: Metamodelling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7*, 1994, University of Jyväskylä, Finland.

[5] Lyytinen, K., *A Taxonomic Perspective of Information Systems Development: Theoretical Constructs and Recommendations*, in *Critical Issues in Information Systems Research*, 1987, John Wiley & Sons Ltd. p. 3 - 41.

[6] Brinkkemper, S., *Method engineering: engineering of information systems development methods and tools.* Information & Software Technology, 1996. 38(6): p. 275--280.

[7] Hooper, J.W. and R.O. Chester, *Software Reuse: Guidelines and Methods*. Software Science and Engineering, ed. R.A. DeMillo. 1991, Plenum Press. 180.

[8] Iivari, J., *Levels of abstraction as a conceptual framework for an information system*, in *Information System Concepts: An In-depth Analysis*, 1989, Amsterdam North-Holland. p. 323 - 352.

[9] Essink, L.J.B., ed. *A Conceptual Framework for Information Systems Development Methodologies*, 1988, Elsevier Science Publishers B.V.

[10] Basili, V.R., G. Caldiera, and G. Cantone, *A Reference Architecture for the Component Factory.* ACM Transactions on Software Engineering and Methodology, 1992. 1(1): p. 53-80.

[11] Curran, T. and G. Keller, *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. 1998: Prentice Hall. 288.

[12] Scheer, A.-W., *Business Process Engineering: Reference Models for Industry Enterprises*. 1994: Springer-Verlag Berlin. Heidelberg. 770.

[13] Wand, Y., *Ontology as a Foundation for meta-modelling and Method Engineering.* Information and Software Technology, 1996. 38(4): p. 281 - 287.

[14] Averill, E., *Reference Models and Standards.* StandardView, 1994. 2(2): p. 96 - 109.

[15] Isakowitz, T. and R.J. Kauffman, *Supporting Search for Reusable Software Objects.* IEEE Transactions on Software engineering, 1996. 22(6): p. 407 - 423.

[16] Ransom, K.J. and C.D. Marlin, *Supporting software reuse within an integrated software development environment.* Proc. ACM SIGSOFT Symposium on Software Reusability, 1995: p. 233 - 237.

[17] Wohlin, C. and P. Runeson, *Certification of Software Components.* IEEE Transactions on Software Engineering, 1994. 20(6): p. 494 - 499.

[18] Biggerstaff, T.J. and A.J. Perlis, eds. *Software Reusability*, 1989, ACM Press: New York.

[19] Sorenson, P.G., J.P. Tremblay, and A.J. McAllister, *The MetaView System for many Specification Environment.* IEEE Software, 1988. 14(3): p. 30 - 38.

[20] Alderson, A., *Meta-CASE Technology*, in *Software Development Environments and CASE Technology*, 1991, Springer-Verlag: Berlin. p. 81 - 91.