Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica, Portugal

Technical Report UNL-DI-1-2001

Proceedings of the Workshop on

Formal Foundations of Software Evolution

Tom Mens

Programming Technology Lab
Departement Informatica
Vrije Universiteit Brussel
Brussels, Belgium

Michel Wermelinger

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica, Portugal

Co-located with the
European Conference on Software Maintenance and Reengineering
Centro de Congressos do IST
Lisboa, Portugal

March 2001

This workshop is an official activity of the
Scientific Research Network on "Foundations of Software Evolution",
which is financed by the Fund for Scientific Research — Flanders (Belgium).

Introduction

Numerous scientific studies of large-scale software systems have shown that about 80% of the total cost of software development is devoted to software maintenance. This is mainly due to the fact that software systems are under constant evolution to cope with changing requirements. Today this is more than ever the case, because of the dramatic evolution of technology, the ever changing legislation, etc. Despite this omnipresence of software evolution, existing tools that try to offer support are far from ideal. They are often implemented in an ad-hoc way, are not generally applicable, are not scalable, or they are difficult to integrate with other tools.

The goal of this workshop is to try and find out how formal techniques can alleviate those problems, and how they can lead to tools for large-scale software systems that are more robust and more widely applicable without sacrificing efficiency. Preferably, provided techniques should not be restricted to a particular phase in the software life-cycle, but should be generally applicable throughout the entire software development process.

The various workshop submissions discuss how formalisms allow us to build tools that support software developers with solving typical evolution problems of large and complex software systems. These tools can provide support for different aspects of software engineering, such as:

Forward engineering Techniques to ensure consistency and detect differences between implementation, design, analysis, requirements and software architectures.

Reverse engineering Techniques to extract relevant abstractions from source code in order to improve understanding of the global structure of a software system.

Re-engineering Techniques to restructure software (possibly at run-time) in order to improve reusability, extensibility and maintainability (e.g., refactoring, reconfiguration).

Team Engineering Techniques to support software evolution when multiple developers change software simultaneously (e.g., software merging, versioning).

In total, there were 13 submissions, 3 of which were selected for a long presentation during the workshop. The workshop participants came from 7 different European countries (Belgium, Finland, Germany, Portugal, Spain, Switzerland, United Kingdom), from Argentina, and from Japan.

The proposed formalisms range from transformational to declarative, and from logic to algebraic. Some of the approaches even use a mixture of different formalisms. The focus of the different approaches also varies depending on the kind of software artifacts that are considered: software architectures, analysis models, design artefacts or implementation code.

Contents

Wolfram Kahl
Software Evolution via Hierarchical Hypergraphs with Flexible Coverage

Lina García-Cabrera, M. José Rodríguez-Fórtiz, José Parets-Llorca
Formal Foundations for the Evolution of Hypermedia Systems

Claudia Pons, Gabriel Baum
Software Development Contracts

Meir M. Lehman, Juan F. Ramil, Goel Kahen
Thoughts on the Role of Formalisms in Studying Software Evolution

Timo Aaltonen, Tommi Mikkonen
Software Evolution Based on Formalized Abstraction Hierarchy

Luís Andrade, João Gouveia, Georgios Koutsoukos, José Luiz Fiadeiro
Coordination Contracts, Evolution and Tools

Reiko Heckel, Gregor Engels
Graph Transformation as Meta Language for Dynamic Modeling and Model Evolution

Jianjun Zhao
Change Impact Analysis for Architectural Evolution

Michele Lanza, Stéphane Ducasse, Lukas Steiger
Understanding Software Evolution Using a Flexible Query Engine

Michel Wermelinger, Antónia Lopes, José Luiz Fiadeiro
A Graph Transformation Approach to Architectural Run-Time Reconfiguration

Tom Mens
Transformational Software Evolution by Assertions

Jamal Said, Eric Steegmans
Transformation of Binary relations into Associations and Nested Classes

Software Evolution
via
Hierarchical Hypergraphs with Flexible Coverage

WOLFRAM KAHL
Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg
kahl@ist.unibw-muenchen.de

Abstract

We present a simple, abstract approach to the use of hierarchical hypergraphs in software evolution. Borrowing ideas from graph transformation and attribute grammars, we show how these hypergraphs can be used in a flexible way to cover *all or part of a software development process.*

This unifying framework allows to design a set of tools *based on common data structures and representations and applicable to diverse tasks and settings.*

1 Introduction

When a piece of "software" evolves with *full* formal support, this implies that for all components that belong to that piece of software, such as

- domain knowledge (ideally in the shape of formal theories),
- requirements,
- user documentation,
- design decisions and their motivations,
- design documentation, and
- "source code",

we have the following:

- the formal support is aware of these components and their structure, and
- the formal support is aware of all kinds of relations between these components, down to arbitrary constituent levels.

Furthermore, to support a more abstract view on evolution, we need that

- the formal support is aware of different versions of the system and of the relations between them.

Therefore, to the formal support, the whole system together with its history and variations appears as a single, though highly structured, *hypergraph*.

However, it will be rare to have full formal support for the whole of the software development process. Usually, already the availability of semi-formal approximations to the full hypergraph of formal support will be considered as an improvement in the process. Furthermore, sometimes formal support needs to be added only later in the process, on an existing system, for example in re-engineering projects. There, it will usually not be possible to derive the whole hypergraph from a given system state (repository) without expensive human interaction, since frequently the relations that have to be represented in the hypergraph are not obvious from the system state. For example, it may be (formally) undecidable which requirements specification or domain knowledge formula is reflected in a specific design decision. Therefore,

- this hypergraph will be *incrementally constructed* as one activity of the development process among others, and
- this hypergraph will have to be *maintained* along with the system representation, or better,
- this hypergraph will have to be *viewed as being* the system representation, even if many edges are "missing".

Tools that aid maintenance of such a hypergraph would be easier to implement if no part of the whole system representation could be changed without awareness of the impact on

the hypergraph structure. That approach, however, would imply almost zero interoperability, and may also be a serious impediment to scalability.

External tools, and, to a certain degree, distributed development will always at least locally and temporarily destroy the hypergraph structure.

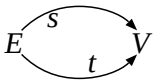
In order to deal with all these facets of software evolution reality, we propose a formalism of *hierarchical hypergraphs with flexible coverage*.

2 Hierarchical Hypergraphs with Flexible Coverage

We propose a hypergraph formalism using hierarchical hypergraphs along the lines of the "higraphs" of Tourlas [1]. Therefore, we first introduce these "higraphs", and then explain how we instantiate this definition for our purposes.

There are many ways to approach the definition of graphs and hypergraphs, and also many ways to specify graph transformations. Because of the high level of abstraction and generality, approaches based in category theory are very prominent, and the basic techniques of the caategorical approach to graph transformation are well-established and accepted.

In category theory, there is one particularly simple and useful approach to what turns out to be a very conventional definition of graphs: One starts by defining a category $\mathbf{G}$ with two objects and two non-identical morphisms, postulationg only the category equations:

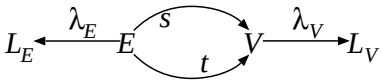


A graph is then a functor from this category $\mathbf{G}$ into the category *Set* of sets and total functions between sets. This means, that for every graph G_i there are

- a vertex set V_i,
- an edge set E_i,
- a total function $s_i : E_i \to V_i$, which is understood to associate every edge with its *source* vertex, and
- a total function $t_i : E_i \to V_i$ which associates every edge with its *target* vertex.

From the definitions of categories and functors one obtains a natural definition of graph homomorphisms (as *natural transformations* between functors). General theory about *set*-valued functors then immediately produces a wealth of results about this category of graphs.

Vertex and edge labellings may be added by extending the base category with additional objects for the label sets, and labelling morphisms, thus obtaining a category $\mathbf{GL}$:



All this is well-established.

Now we come to the idea behind the "higraphs" of Tourlas [1], which is in fact extremely simple: Use the above setting with base category $\mathbf{G}$, but replace the category *Set* of sets and total functions between sets with the category *PO* of partially-ordered sets and order homomorphisms (i.e., monotonic total functions) between them.

Thus, every higraph consists of the same four components as a graph, but the source and target functions now have to be monotonic: whenever two edges $e_1, e_2 : E_0$ in a higraph H_0 are related by the edge ordering, i.e., when we have $e_1 \leq_E e_2$, then the incident vertices have to be related by the vertex ordering, i.e., we also need to have $s_0(e_1) \leq_V s_0(e_2)$ and $t_0(e_1) \leq_V t_0(e_2)$.

Tourlas uses these graphs most notably for representing statecharts, with their hierarchical state transition diagrams and their edges at and between different levels in the hierarchy.

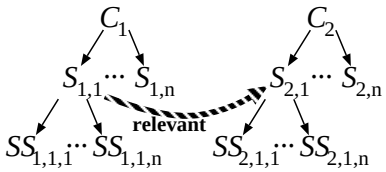
This approach even carries over to the labelled case without problems: We then just need order-preserving labelling functions. A particularly simple instance might, for example, use the trivial identity ordering on the edge label set; this then implies that whenever $e_1 \leq_E e_2$, then their labels conincide: $\lambda_E(e_1) = \lambda_E(e_2)$.

This already shows that this approach to hierarchical graphs is quite flexible. We now propose a hypergraph formalism which is an instance of edge-labelled higraphs in the following way:

- Nodes represent basic items of the system representation, such as formulae, natural language sentences, source code statements.
- Subsystems are, for simplicity, considered as the sets of nodes they contain. Such subsystems are going to be used as the vertices of our higraphs.
- Edges will be hyper-edges with a non-zero number of tentacles attached to them; instead of just two tentacle rôles "source" and "target" we admit an arbitrary number of tentacle rôles, and correspondingly expand the number of morphisms between edge set and vertex set. Since these tentacle rôles will have to be interpreted as *total* monotonic functions, we may choses empty subsustems as targets for rôles where these rôles are not applicable to the edge in question..
- Edges will be labelled, sometimes just with rôles, sometimes e.g. with in addition formal proofs that establish the relation asserted by the edge.

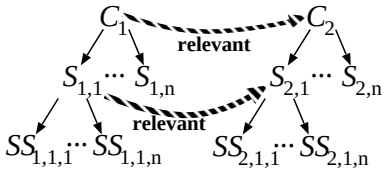
On these hyper-graphs, transformations and transitions are defined as follows:

- *Covering transitions* add edges in a way that roughly corresponds to calculation of attributes in attribute grammars. Consider the following as an example: Assume that a certain section of the specification has a "relevance" edge to a certain section of the user documentation:

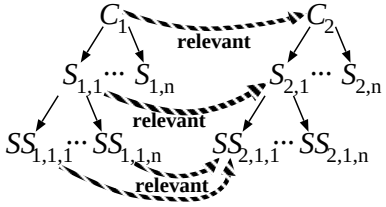
  then:

  - A "relevance" edge will be introduced between the chapters containing those sections:

    This may happen automatically.

  - Edges may be introduced between subsections or formulae in the specification and subsections in the user documentation:

    This may need human assistance.

- *Transfer transitions* add edges in a similar way to relate new versions of parts of the system with the rest of the system. Part of this may be automated, and human assistance may be needed in certain cases as in most version management systems.

- *Lossy transitions* remove edges in response to changes to a part of the system that cannot be assured to have preserved the properties represented by those edges. For example, manual editing of any document will in many cases destroy (or mark as unreliable) most edges incident with that document.

- *Transformations* change the structure of some part of the system, and may add and delete nodes and edges. (The fact that some part of the structure is derived from a transformation will usually be recorded in appropriate edges resulting in "self-covering" transformations.) Transformations can serve the most diverse purposes, and higher degree of formality in the development will usually involve a higher percentage of transformations in the process.

Formally, transitions will usually be described by total single-pushout rules, while transformations will essentially be conventional double-pushout rules.

On such a hypergraph representing a system state, several predicates will be defined, such as:

P1: the hypergraph covers the whole system representation (no covering or transfer transitions can be applied)

P2: the hypergraph covers the whole representation of a specific version

P3: the hypergraph covers all relations between two specific versions

P4: the hypergraph demonstrates that, in a specific version, a specific set of requirements is fulfilled by the implementation

P5: the hypergraph demonstrates that, in a specific version, a specific set of requirements is reflected in the user documentation

...

3 How and why can this formalism be used to provide tool support for evolution?

As documented by the examples given above, a hierarchical hypergraph is a universal framework that can be used to represent and document very different kinds of relations between very different parts and aspects of the system. Some parts of the system, e.g. UML diagrams or finite-state machines, may even be directly encoded as sub-hypergraphs in the same formalism.

The fact that a single formal model stands behind all aspects of the system structure makes it easy to develop a coherent tool set of tools containing special functionality for special aspects of the system, or for special aspects for the interaction with the hypergraph structure:

- Visualisation may be unified, and will automatically be available at all levels of the hierarchy.

- Closure tools will have different derivation components for correctness proofs than for documentation coverage checks.

- Derivation tools may have different instances for different kinds of diagrams and different target paradigms.

Although a unified approach is taken, there is no necessity to use a unique tool, as long as the different tools operate on the same formal model and with compatible representations.

Since not all of the desirable predicates (e.g., **P1**) need to hold all the time during development, tools cannot rely on such assumptions, either, so there is a certain *built-in robustness* in our approach. In particular the possibility to have parts of the system loosing their connections with the rest of the system, or starting their existence in such an isolated state, is the key to interoperability with other tools that are not aware of the hypergraph structure, but only operate on certain (sets of) nodes. Some external tools may still provide some certain kinds of to relevant structure; this can then be used by hypergraph tools e.g. to automate at least certain coverage processes.

4 For which aspects of software evolution can this formalism provide support?

Since our formalism is essentially a meta-formalism, it can be used for all kinds of software evolution and in all parts of the software development process as long as tools with the relevant additional capabilities are available.

It is of course possible to encode even the formulae of the requirements specification as hypergraphs, and similarly the "source-code" of software products, and have hypergraph transformations for the complete development, proof, and maintenance process. However, this will probably be the exception.

More or less at the other extreme, it is also conceivable that a re-engineering project starts out with just nodes, namely the existing source code and documentation, and progressively adds edges as relations between documentation and source code are discovered, and adds nodes as new documents are added.

5 Items for Discussion

Instead of a conclusion, let us raise a few points that might deserve discussion:

- In our examples, edges range from the "soft", such as documentation coverage, to the "hard", such as documenting transformation steps and formal correctness proofs. I would consider this as an advantage, since it gives users flexibility with respect to the degree of formal support they wish to see integrated into their process. Since "hard" edges are usually accessible to automatic proof-checking tools, predicates asserting consistency of proof-carrying subgraphs may be defined and checked automatically.

  Would a more rigorous support of consistency blend in equally well with a potentially mixed environment?

- In the implementation of tools for our hypergraphs, edges will exist outside the linked documents, employing addressing mechanisms such as e.g. XLink. Are there other obvious candidates for standardised representations?

- Fine-grained distributed locking will be necessary to minimise conflicts between concurrent application of hypergraph-aware tools — is this considered problematic?

- We mentioned the possibility to store formal correctness proofs in edges (a variant would be to store them as nodes and just link them via edges) — would other ways of linking in external theorem provers be more attractive?

References

[1] K. Tourlas. Towards the principled design of diagrams in computing and software engineering. slides from a talk given in Birmingham on 27th October 2000, Oct. 2000. URL: http://www.dcs.ed.ac.uk/home/kxt/birmingham_4up.ps.gz.

Formal Foundations for the Evolution of Hypermedia Systems

Lina García-Cabrera*, M.José Rodríguez-Fórtiz**, José Parets-Llorca**
* Depto. Informática. Universidad de Jaén
EPS Avda. Madrid, 35, Jaén, SPAIN
Tel: +34 953 212475 E-mail: lina@ujaen.es

** Depto. L.S.I. Universidad de Granada
ETSII Avda. Andalucía, 38, Granada, SPAIN
Tel: +34 958 243179 E-mail:<mjfortiz, jparets@ugr.es>

Abstract[1]

In this paper, we shall attempt to justify the need for an evolving conception of hypermedia systems and its formalisation. We propose graph theory, predicate logic, temporal logic and Petri nets to support evolution in hypermedia systems. A semantic-dynamic model based on these formalisms is presented. It provides a complete, adaptive and evolving control of development and maintenance of hyperdocuments and an understandable navigation.

1. Introduction

Hypermedia systems are an special kind of Information Systems constructed over a conceptual domain. Because they include the knowledge captured by their authors, they are continuously changing. Changes can be carried out in the concepts offered by them, in the relationships between concepts, in the way of presenting the information and in the documents (information items) which explain the concepts.

Bieber [1] says, "Currently, developers and authors must build all hypermedia representations and navigation using single-step links without semantic or behaviour typing." and "Fourth-generation hypermedia features would provide sophisticated relationship management and navigation support." In our opinion, we must face two challenges. Firstly, we must assume the dynamic and evolving nature of hypermedia systems. A hypermedia system represents some aspects and relationships of a conceptual domain explained by a set of authors. But there are very different ways of representing, structuring and browsing it. Secondly, the bulk of the hypermedia systems, and web in particular, only considers the final hypermedia documents and, sometimes, the navigation performed by the reader. Nevertheless, the design, construction and evolution processes –the whole life-cycle- of hypermedia is not sufficiently considered [10]. However, this development process is very important because it implies a structuring process that is implicit, diluted and unaffordable inside the documents [5].

1.1. Our Approach

In order to provide dynamic, flexible, robust and understandable hypermedia systems we propose an approach based on four main assumptions:

- Following the Theory of the General System [7], a hypermedia system can be conceived as a set of interacting systems in continuous evolution.
- The following elements should be provided: mechanisms for representing the information system; a representation of the conceptual domain or ontology [12] that information belongs to; useful ways of browsing and remembering the memorised knowledge.
- The process of construction of information systems, conceptual domains and routes –ways of navigation- should be flexible.
- Information systems, conceptual domains and navigation routes are exposed to continuous changes and updates which should be integrated in the development process.

In order to provide an operational view of these assumptions, our approach distinguish two abstraction levels in the design of a hypermedia system. The first level, called *memorisation system*, includes the representation and management of information semantics [4], i.e. the conceptual domain. The second level, called *navigation system*, extends this semantics adding dependence and order relationships which allows navigation over the conceptual domain. This distinction is useful because allows a separation of concerns both in the development and the evolution processes. In addition

[1] This research is supported by a project –MEIGAS- by the Spanish CICYT (TIC2000-1673-C06-04) which is a subproject of the DOLMEN project (TIC2000-1673-C06).

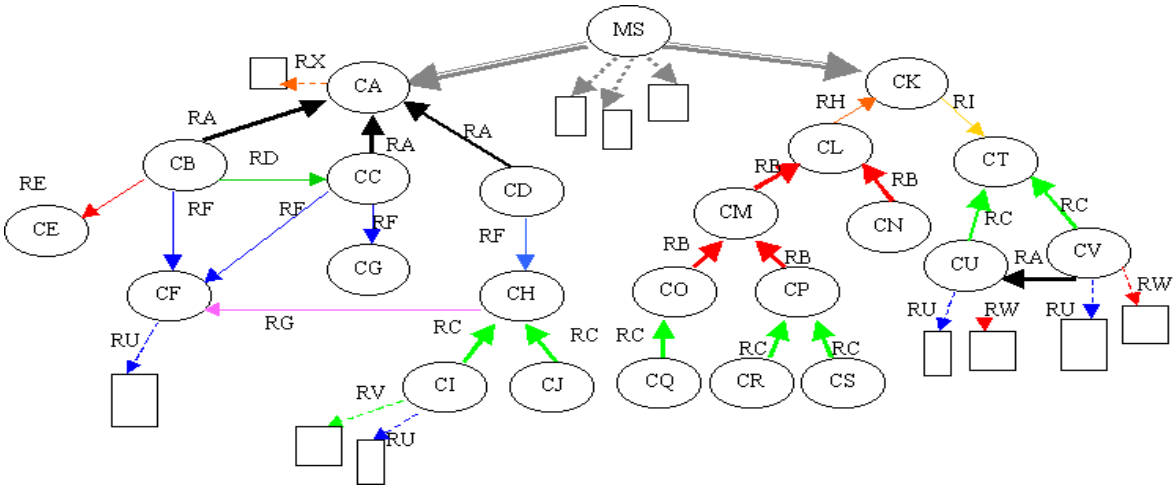


Figure 1. Examples of CSs of the Memorisation System.

different navigation 'styles' can be performed using the same semantic structure.

Different formalisms will be used in representing these systems which will allow to manage the development and evolution in both of them. They will be presented in the next sections.

2. An Evolutionary Model Based on Systems

A Hypermedia System can be conceived as being made up by two interrelated and interacting systems (for a complete description of the model, see [2]):

1. Memorisation System (*MS)* is in charge of the storage, structuring and maintenance of the different pieces of information –pages or documents-. It memorises the knowledge acquired about the information system that is represented. This knowledge will guide the design and structuring processes of the information system. It will determine the possibilities of change in this structure throughout its evolution.
2. Navigation System (*NS)* helps the reader in his interaction with the information system. Using the memorised knowledge and the reader activity over time in a dynamic way, this system determines –firstly- the accessible information and –secondly- the interaction possibilities.

A concrete and complete example of the use of the formalism to specify the structure and evolution of an hypermedia System can be seen in appendix..

2.1. Formalisation of the Hypermedia Systems

As stated above, two systems are distinguished in the model. The formalisms associated and the modelled aspects of each system are summarised in table 1.

In the *MS*, which mainly includes the semantic structure of an information system, graph theory [13] and temporal logic are used. The second system, *NS*, specifies the order relationships between concepts when navigation will be performed. Petri nets and temporal logic are used in this case [8][11].

The *MS* provides the necessary instruments which allows a representation of the information system by means of a directed graph [4], in which, nodes and links are labelled with semantic meanings –a semantic net-. The graph represents the conceptual domain –concepts and relationships between concepts- of the information system, named Conceptual Structure (CS*)*. The different information items –documents- can be associated –labelled- with one or more concepts of the CS. These items are also nodes of the CS. In order to allow provisional and incomplete development, items which are no related to any concept can also be included. Figure 1 shows an abstract example where *MS* is an artificial node which is the root of the represented information systems. Two conceptual structures are included (CA and CK). A conceptual structure for the Solar System is explained in the appendix example.

Therefore CS is defined as: CS = (C, II, A_c, A_i), where C is the set of concepts, II is the set of information items, A_c is the set of labelled conceptual associations, A_i is the set of labelled associations between concepts and information items.

Because CS is constructed by the authors in a dynamic way, some evolution operations as add-concept, delete-association, modify-association, add-item, etc. have to be included. The operations must verify a set of restrictions in order to maintain the consistency of the CS. These restrictions can be basic ones, defined as a functional part of the *MS,* or can also be defined by the author. Some examples of basic restrictions are:

- Each association of the *CS* must connect two concepts or a concept and an item.
- Each arc and node of the *CS* must be labelled.
- Two nodes in a *CS* cannot have the same label.

The author can also include additional restrictions which determine what associations between concepts are possible. In order to represent these restrictions, formulas in temporal logic are used. This formalism also allows to check if the CS is valid at any moment. Some examples are:

- Concept-A can be connected with concept-B by means of the relationship-A.
- The relationship-B must be acyclic.
- Concept-C can be connected with concept-G if concept-C is reached from concept-B.

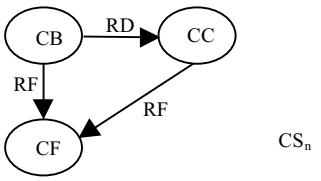
Therefore, the Memorisation System is defined as *MS* = (CS, RT, AC_e), where CS is the previously defined directed and labelled graph weakly connected that represents the conceptual domain of a hypermedia system, RT is the set of restrictions that must verify the CS –defined by the system RT_s and by the author RT_a- and AC_e is a set of evolutionary actions (see next section).

Memorisation System	**Graphs**	**TemporalLogic**
Concept (C)	Labelled node	Proposition
Item (II)	Labelled leaf node	Proposition
Relationship between concepts (A_c) or concepts and items (A_i)	Labelled arc	Formula with temporal and logic operators
Navigation System	**Petri Nets**	**Temporal Logic**
Concept or item	Place	Proposition
Order relationship between concepts or items	Transition and arcs	Formula with temporal operator
Dependence relationship between concepts or concepts and items	Transition and arcs	Formula with logic operator
Navigation	Firing transitions	Instantiation of formulas

Table 1. Formalisms used in specifying the structure of a hypermedia system

The Navigation System, using as basis the CS of the Memorisation System, allows a selection of a subset of the concepts and associations included in CS. This graph, CS_n, being a subgraph of CS, $CS_n = (C_n, II_n, A_{cn}, A_{in})$, will be presented to the reader. In addition, some navigation restriction can be added in order to follow more restricted paths in the subgraph. These restrictions or navigation rules are expressed using temporal logic. Considering the CS_n and temporal restrictions, a Petri net is automatically constructed. As demonstrated in [3] and in [8], Petri nets give an operational semantics to temporal logic formulas allowing an operational navigation. The algorithm which transforms temporal logic formulas in Petri net is explained in [3].

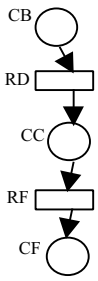
Therefore, the Navigation System is defined as *NS* = (CS_n , RT_n, PN, AC_e), where RT_n is the set of restrictions specified by the author by means of temporal logic, PN is the Petri Net and AC_e is the set of evolving actions to adding, deleting or modifying navigation restrictions (see next section).



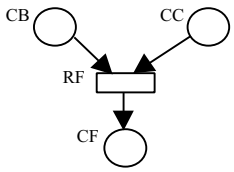
CS_n

From this CS_n two navigation systems examples are constructed:

1) CC ← ◊ CB
CF ← ◊ CC



2) CF ← ◊ CB and ◊ CC



PN are constructed taken into account the logic navigation restrictions.

Figure 2. Construction of the Navigation Paths

An example of the specification of the navigation possibilities is shown in figure 2. It gets a subgraph based on the left CS of the example of figure 1. The appendix presents the navigation system of part of the CS of the Solar System, having only into account the Earth relationships

2.2. Formalisation of the Hypermedia Evolution

Both systems, *MS* and *NS,* include a set of evolving actions, AC_e, that allow to make and propagate changes in the hypermedia system. An evolving action can belong to three different types:

1. Actions that redefine some aspects the system. Obviously the basic restrictions discussed below, RT_s, cannot be changed.
2. Actions that control the propagation of these changes inside of the system itself.
3. Actions that control the propagation of these changes outside the system, i.e. in the other system

When these actions are carried out they change the corresponding elements of the hypermedia system. Because integrity should be guaranteed in any case, these operations should be carried out following a set of meta-restrictions. The specification of these meta-restrictions implies a meta-level in the definition of the *MS* and *NS*.

Formalisms of a higher abstraction level should be used. See figure 3.

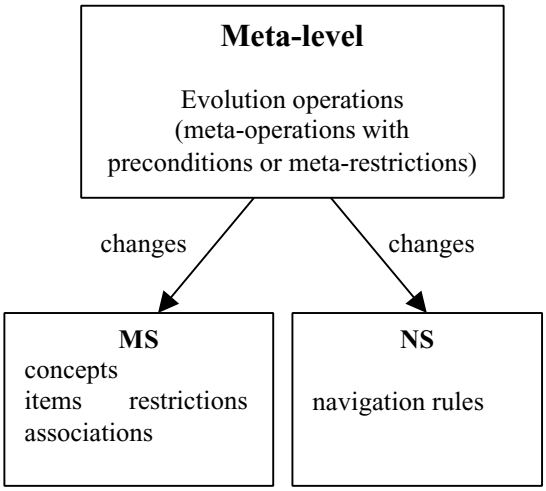


Figure 3. The Meta-level in evolving the Memorisation and Navigation Systems

Table 2 summarise the formalisms used in specifying meta-restrictions in both systems. Lets describe how they are specified for each systemm *MS* and *NS*.

The Memorisation System always must guarantee its consistency. Two aspects of this system can change, the CS –the graph- and the restrictions defined by the author. Graph Theory is used to represent the evolution operations of the graph and their associated meta-restrictions. Changes in restrictions defined by the author, RT_a, must be defined by means of meta-restrictions.

When the author changes the CS –add, delete or modify a concept, item or association- the system must check:

1. CS verifies the restrictions defined by the system and associations satisfy the set of restrictions defined by the author. RT acts as a set of restrictions for the operations, only if the operation match restrictions, it will be carried out (internal propagation of changes).
2. The subgraph used by the *NS*, CS_n, is consistent with changes in CS. If a concept or relationship have been deleted in CS, the *NS* must also delete this concept or relationship in CS_n (external propagation of changes).

When the author redefines –add, delete or modify- one associative restriction RT_a, the system must check:

1. The set of axioms about associations is valid, by means of predicate temporal logic.
2. CS verifies the new set of restrictions, using the graph theory. The system must detect the associations that not satisfy one or more restrictions and delete them (internal propagation of changes).
3. The CS_n verifies the new set of restrictions by means of graph theory. The system must detect the associations that not satisfy these restrictions and delete them (external propagation of changes).

Memorisation System	**Graph Theory**	**Predicate Temporal Logic**
Operation	Set operation	Predicate
Meta-restriction	Reachability function	Temporal formula
Modified aspect	Set	Variable
Navigation System	**Predicate Temporal Logic**	
Operation	Predicate	
Meta-restriction	Formula	
Modified aspect	Instantiation in the variable of a predicate	

Table 2. Formalisms used in specifying the evolution meta-restrictions of a hypermedia system

Navigation System models evolution using predicate temporal logic. It provides a meta-level with evolution operations which manage and change the navigation restrictions. Navigation rules can be added, deleted or modified, and the meta-restrictions of these operations can be established.

In a similar way that the Memorisation System does, the consistency must be guaranteed during the evolution of the Navigation System. In this system, changes can be produced in the subgraph selected CS_n and in the navigation restrictions, RT_n, defined by the author, and therefore, in the PN obtained from them.

When CS_n is changed –the author select another set of concepts and relationships- new navigation possibilities are defined. In this case, the author must define again the navigation restrictions. This change is not a real evolution, the author is designing new navigation possibilities, but if these possibilities are defined in an incremental way, the system can aid the author in the design process.

When the author redefines –add, delete or modify- a navigation restriction, RT_n, the system must check:

1. The set of restrictions that establish the order of navigation is consistent. Predicate temporal logic is used to specify the evolution operations over the restrictions, and their associated meta-restrictions.
2. The navigation restrictions have changed. Changes in a restriction can imply the modification of other restrictions. The PN based in the navigation restrictions must evolve, generating it again (internal propagation of changes).

Figure 4 sums up the evolving changes described above and the interactions between the systems Restrictions defined by the system, RT_s, or by the author, RT_a, are associated to the conceptual structure CS (1). Author selects only a subset of concepts and relationships from the CS in order to establish the navigation routes, creating the CS_n (2). Navigation restrictions, RT_n , are added (3) and a Petri net, PN, is created from them (4).

Evolution can be carried out in the conceptual structure, CS (5), in RT_a by means of predicate logic (6) and in RT_n using predicate temporal logic (8). When RT_a is modified CS could also change (7). PN evolves being reconstructed

from RT_n (4). The evolution in the Memorisation system is also propagated to the Navigation system (2).

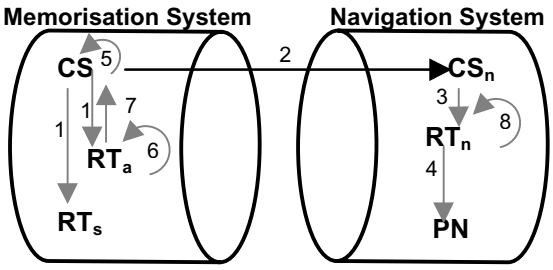


Figure 4. Definition and evolution of a hypermedia system

3. Contributions of the Formalisms

The different formalisms –graphs, Petri nets and propositional and predicate temporal logic- allow to model and distinguish between the information system, the conceptual structure and navigation. The author organises the information of the Memorisation System according to his particular interpretation of the conceptual domain. Therefore, to offer more than one structure –perspective- of the same information is possible. In addition, the model can provide more than one view (CS_n s) of the source CS by means of the Navigation System and different routes of navigation over the same subset of information

In particular, the Memorisation System contains the semantic structure–how knowledge is organised-, therefore, labelled graphs are the more suitable mechanism for representing it. Because restrictions should be also represented, indicating what associations are valid in the CS, temporal logic is a natural way to formulate them.

In the Navigation System, the main objective is to restrict the possible paths that can be followed when information is navigated and the order in which navigation is carried out. Temporal logic allows the specification of order relationships and Petri nets offer an operational formalism which can be executed in order to show these paths and analyse their properties [8][10].

The formalisms used in evolving the systems –graph theory and predicate temporal logic- easily support the changes and its propagation. Changes in the items, the CS, and in the Petri net are possible in an independent way. But, at the same time, the system can propagate these changes in order to maintain the global consistency.

In particular, graph theory is based on set theory, so the evolution operations can be expressed by simple set operations. Predicate temporal logic allows us to modify consistently the restrictions expressed in propositional temporal logic. Predicate temporal logic manage the meta-restrictions treating the propositions of the restrictions as variables, modifying them, and therefore, changing the restrictions.

Predicate temporal logic is used in the Navigation System with the same proposal, but respect to navigation restrictions. Predicate temporal logic is used, as demonstrated in previous papers [8][9], to verify these restrictions and to observe how the evolution is carried out.

The proposal of one such amount of formalisms has a main objective: to represent each evolution problem using the formalism which better fits the evolution possibilities. Obviously these formalism are hidden and the author have not to know them. These formalisms can be hidden inside the tools which implements the *MS* and the *NS* and the author could define its CS and restrictions using a visual -graph- representation of them.

4. The Evolution Formalisms in Other Systems

Although we use the previous formalisms in specifying and evolving hypermedia systems, we consider that they are useful in modelling the functioning and evolution of other types of systems, as reactive systems or temporal databases [8][9].

Graph theory can represent the relationships between agents and their environment in reactive systems. The associations established in the schema of temporal databases can also be defined by means of graph theory.

Due to the nature of both kinds of systems, meta-restrictions about relationships can be expressed using Temporal Logic. The evolution of these relationships and restrictions can be expressed by predicate temporal logic as a meta-level which defines the evolution operations and their meta-restrictions.

5. Conclusions

The separation of hypermedia systems in two abstraction levels allows a specification and management of the semantics of information and its navigation in a separated way, using different formalisms. Evolution operations can be defined independently in each level, but it is possible to determine what changes must be propagated to other components or to the other level.

The most important consideration during evolution is the conservation of the integrity of the system. Each evolution operation must verify a meta-restriction, checking the integrity restrictions associated to it. The meta-restrictions depend on the system (*MS* or *NS*) in which the change will be carried out.

The novelty of our approach about evolution is the incorporation of a meta-level, by means of reflectivity and second order, which allows us to reason about the functioning and structure of an hypermedia system which evolves.

The selected formalisms allow an easy specification and change of the structure of each system. It is very easy to modify a graph, a Petri net or a logic program in order

to change the structure of the system that they represent. Set Theory allows the verification of properties and integrity rules over the graph. Predicate temporal logic represents the evolution meta-restrictions over the memorisation and navigation in a hypermedia system.
These evolution formalisms can also be applied to other kinds of systems with an evolving nature, such as reactive or temporal ones.

6. Appendix

The following example of a hypermedia system shows different concepts related to the Solar System.

First of all the specification and evolution of the Memorisation System will be presented. After that, the Navigation system will be specified and evolved.

6.1. Specification of the Memorisation System.

a) Graph $CS=(C, II, A_c, A_i)$ (See figure 5)

C= *{Solar System, Planets, Stars, Earth, Venus, Sun, Moon, Countries, Oceans, Portugal}*

II=*{P1, M1, C1, C2, Po1, Po2, O1, Su1, Su2, S1, S2, S3}*

A_c= *{<Earth, rotate, Moon>, <Earth, part_of, Countries>, <Earth, part_of, Oceans>, <Sun, rotate, Earth>, <Sun, rotate, Venus>, <Countries, is_a, Portugal>, <Solar_System, part_of, Planets>, <Solar_System, part_of, Sun>, <Stars, is_a, Sun>,<Stars, part_of, Solar_System>, <Planets, is_a, Earth>, <Planets, is_a, Venus>}*

A_i= *{<Moon, photos, M1>, <Countries, list, C1>, <Countries, cities, C2>, <Portugal, map, Po2>, <Portugal, history, Po1>, Oceans, list, O1>, <Sun, photos, Su1>, <Sun, quimical composition, Su2>, <Planets, def, P1>, <Stars, def, S1>, <Stars, nova, S2>, <Stars, supernova, S3>*

b) Temporal logic

Examples of restrictions RT over the associations:

- RT_s : *is_a* association is not recursive.

$<c, is\_a, c1> \leftarrow not \Diamond <c1, is\_a, c> \quad \forall c, c1 \in C$

- RT_a: If an *is_a* association exist previously between any concept and the *Planets* concept, an association *rotate* must be added relating that concept with the *Sun* concept (every planet must rotate around the sun).

$<c, rotate, Sun> \leftarrow \Diamond <c, is\_a, Planets> \quad \forall c \in C$

6.2. Evolution of the Memorisation System.

a) Graph Theory

Example of operation: *add_concept: Saturn.*
The meta-restriction of this evolution operation must hold.

- *Meta-restriction*: $Saturn \notin C$
  Meta-restriction holds, so *Saturn* can be a new concept.
- *Internal propagation of the change:* if a concept is added, it must be associated to other concepts. The evolution operation *add_concep_assoc* must be carried out as consequence of the previous.

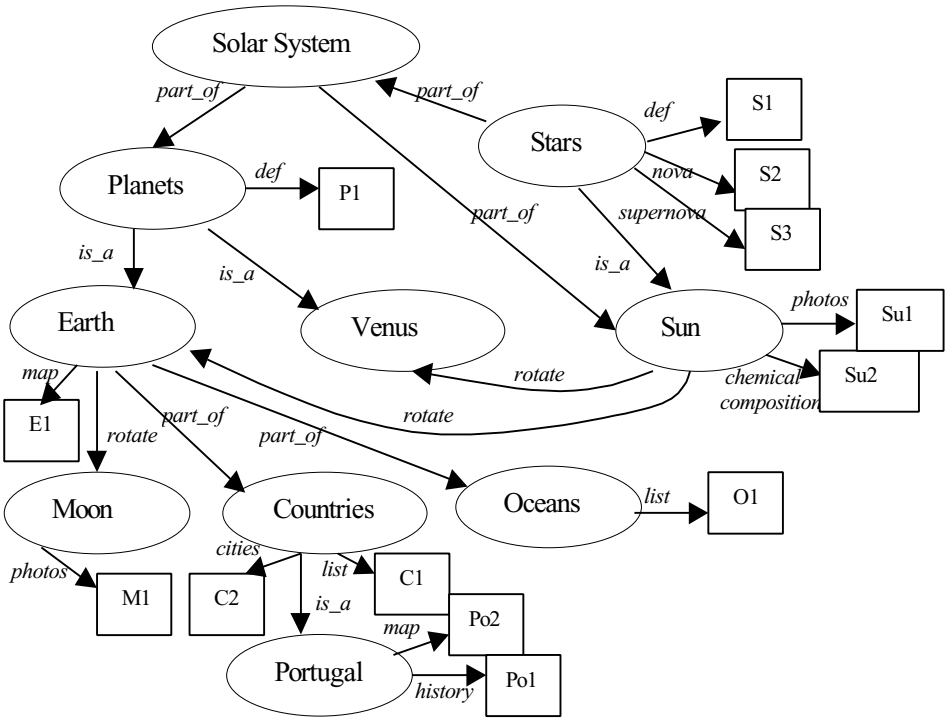


Figure 5. CS from a Solar System hypermedia

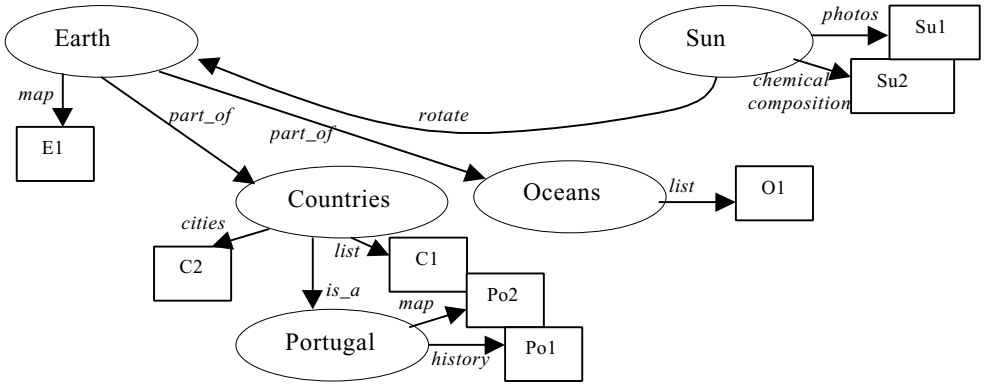


Figure 6. CS_n: selection of CS

In this case the operation: a*dd_concep_assoc: <Planets, is_a, Saturn>* will be carried out. Its meta-restriction must also be verified:

- *Meta-restriction*: *<Saturn, is_a, Planets>* $\notin A_c$

This meta-restriction holds. It can be also verified proving the logic restriction:

<c, is_a, c1> ← *not* ◊ *<c1, is_a, c>* with *c = Planets* and *c1= Saturn.*

After these changes, the graph which represents the Memorisation System has evolved: CS CS'

CS' = (C', II', A_c', A_i') ; C'= C ∪ *{Saturn}; II' = II ;*

A_c'= A_c ∪ *{<Planets, is_a, Saturn}; A_i' = A_i*

b) Predicate Temporal Logic

Restrictions RT over the associations can be also changed. Predicate Temporal Logic is used as a meta-level to manage and evolve these restrictions.

Example:

As previously stated, cycles in concept associations are not allowed. An association *ac2* can be included in the restriction to establish an association *ac1* if previously *ac1* is not included in the restriction to establish the association *ac2*.

The meta-restriction which describes this restriction is:

addRest(ac2, ac1) ← *not* ◊ *isRest(ac1,ac2)*

ac1,ac2 ∈ A_c

This clause can be instantiated:

addRest (<c, rotate, Sun>, <c, is_a, Planets>) ←

not ◊ *isRest(<c, is_a, Planets>, <c,rotate, Sun>)*

If *c* is *Earth*, the restriction can not be added because the meta-restriction does not hold (see 6.1.b)). *Earth* is a planet, and this is the restriction to rotate around sun. If we stated that the restriction of being a planet is that previously it rotate around sun (inverse relationship), a not desired cycle situation is being produced.

6.3. Specification of the Navigation System

A part of the Memorisation System, CS_n , is chosen to navigate (See figure 6). In that case the navigation restrictions are expressed in Temporal Logic.

a) Temporal Logic

Example of definition of navigation restriction:

c.Portugal.map ← ◊ *c.Countries.list and a.is_a*

It expresses that the *map* of *Portugal* can be shown if previously the *list* of *Countries* has been presented and there is an association *is_a* between both concepts. Letters *c* and *a* in the propositions represent the concepts and associations respectively.

Using the previous CS, the rest of the navigation restrictions can be constructed automatically. For example:

c.Portugal.map ← ◊ *c.Countries.cities and a.is_a*

c.Portugal.history ← ◊ *c.Countries.cities and a.is_a*

b) Petri nets

A Petri net can be constructed from the navigation restrictions, as the figure 7 shows.

6.4. Evolution of the Navigation System

Predicate Temporal Logic is used to define the meta-restrictions associated to the evolution operations of this System.

a) Predicate Temporal Logic

Adding, modifying or deleting navigation restrictions is possible if each concept and item selected from the CS can be reached. A navigation restriction can be modified or deleted if the concepts and items that they reference are referenced in other restrictions because, in other case, these concepts and items will be unreachable.

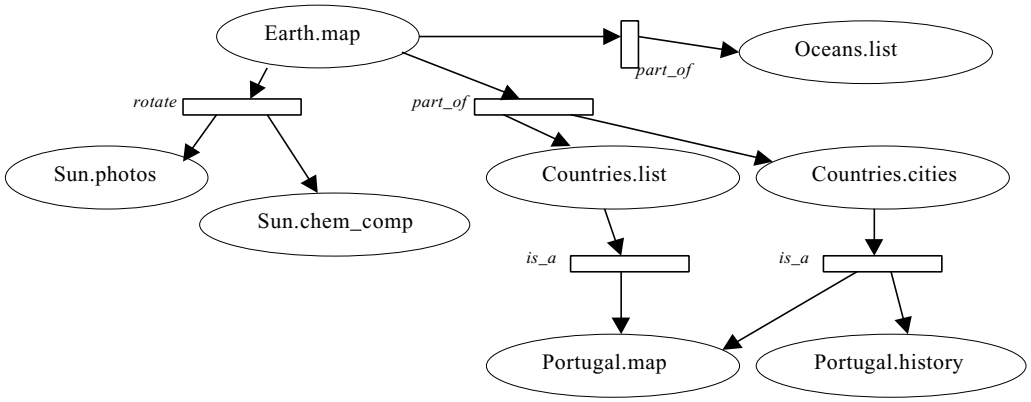


Figure 7. Petri Net of the Navigation System

Example:

The meta-restriction of the evolution operation *delRest* (deleting a restriction) is: to get the concept and the item of the head of the restriction rule by means of another navigation restriction is possible, or another navigation restriction which includes a reference to that concept and item in its body exists.

delRest(c.i, nav_rest) ← ◊ *existRest(c.i, nav_rest1) or* ◊ *(existRest(c1.i1, nav_rest2) and ref(nav_rest2, c.i)*
$\forall c \in C$, $\forall i \in II$,

nav_rest is the restriction for navigating to the item *i* of the concept *c: c.i* ← *nav_prec*

If *c.i* is instantiated with *Portugal.map***,** the meta-restriction holds, then the navigation restriction can be deleted. Navigation restriction:

c.Portugal.map ← ◊ *c.Countries.list and a.part_of*

can be deleted because there are another restriction which allows to reach that item:

c.Portugal.map ← ◊ *c.Countries.cities and a.is_a*

As navigation restrictions have changed, Petri net must be modified to deleting the transition *is_a,,* and their arcs, which link the places *Countries.list* and *Portugal.map*.

7. References

[1] Bieber, M.; Vitali, F. 1997. Toward Support for Hypermedia on the World Wide Web. IEEE Computer, January 1997.

[2] García-Cabrera, L.; Parets-Llorca, J. (2000) A Cognitive Model for Adaptive Hypermedia Systems. The 1st International Conference on WISE, Workshop on World Wide Web Semantincs. Hong-Kong, China, June 2000, pp 29-33.

[3] Lin, C.; Chaudhury, A.; Whinston, A. B.; Marinescu, D. C. "Logical Inference of Horn Clauses in Petri Net Models". IEEE Transactions on Knowledge and Data Engineering, vol 5,3. pp: 416-425. 1993.

[4] Nanard, J., Nanard, M. Using Structured Types to Incorporate Knowledge in Hypertext. Hypertext'91 Proc., ACM Press: 329-343.

[5] Nürnberg, P.J., Leggett, J.J., Schneider, E.R. As We Should Have Thought, Hypertext'97 Proc., ACM Press: 96-101.

[6] Paderewski, P; Parets-Llorca, J.; Anaya, A; Rodriguez, M.J; Sanchez, G; Torres, J; Hurtado, M.V. (1999) A software development tool for evolutionary prototyping of information systems". IEEE CSCC'99. Computers and Computational Engineering in Control. Electric and Computer. Engineering Series. World Scientific and Engineering Society Press, pp: 347-352.

[7] Parets, J; Anaya, A; Rodríguez, M.J.; Paderewski, P. (1994) A Representation of Software Systems Evolution Based on the Theory of the General System. Computer Aided Systems Theory, EUROCAST'93. Lecture Notes in Computer Science vol.763. Springer-Verlag. pp: 96-109.

[8] Rodriguez-Fortiz, M.J, Parets Llorca, J. (2000) Using Predicate Temporal Logic and Coloured Petri Nets to specifying integrity restrictions in the structural evolution of temporal active systems. ISPSE 2000. International Symposium on Principles of Software Evolution. Kanazawa, Japan, November 2000, pp 81-85.

[9] Rodriguez-Fortiz, M.J. Software Evolution: A Formalisation Based in Predicate Temporal Logic and Coloured Petri Nets. (in Spanish). Ph Thesis. October 2000.

[10] Schnase, J.L., Leggett, J.J, Hicks, D.L., Szabo, R.L. Semantic Data Modelling of Hypermedia Associations, ACM Trans. Information Systems, 11(1):27-50, January 1993.

[11] Stotts, P., Furuta, R., Ruiz-Cabarrus, C. Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking, ACM TOIS, 16(1): 1-30, January 1998.

[12] Uschold, M. Ontologies: Principles, Methods and Applications. Knowledge Engineering Review, 11(2), June 1996.

[13] Wang, W., Rada, R. Structured Hypertext with Domain Semantics, ACM Trans. Information Systems, 16(4), October 1998.

Software Development Contracts

Claudia Pons Gabriel Baum

LIFIA – Universidad Nacional de La Plata
50 esq.115. 1er Piso
CP 1900 Buenos Aires, Argentina
email: cpons@info.unlp.edu.ar

Abstract

While the notion of formal contract regulating the behavior of software agents is accepted, the concept of contract regulating the activities of software developers is quite vague. In general there is not documented contract establishing obligations and benefits of members of the development team. However, a disciplined software development methodology should encourage the use of formal contracts between developers.

We propose to apply the notion of formal contract to the object-oriented software development process itself. That is to say, the software development process can be seen as involving a number of agents (the development team and the software artifacts) carrying out actions with the goal of building a software system that meets the user requirements. In this way, contracts can be used to reason about correctness of the development process, and comparing the capabilities of various groupings of agents (coalitions) in order to accomplish a particular contract.

Keywords: object-oriented software development process, process modeling, formal methods, refinement calculus, contract.

1. Introduction

Object-oriented software development process (e.g. The Unified Process [Jacobson et al., 1999], Catalysis [D´Souza and Wills, 1998], Fusion [Coleman et al. 1994]) is a set of activities needed to transform user's requirements into a software system. A software development process typically consists of a set of software development artifacts together with a graph of tasks and activities. Software artifacts are the products resulting from software development, for example, a use case model, a class model or source code. Tasks are small behavioral units that usually results in a software artifact. Examples of tasks are construction of a use case model, construction of a class model and writing code. Activities (or workflows) are units that are larger than a task. Activities generally include several tasks and software artifacts. Examples of activities are requirements, analysis, design and implementation.

Modern software development processes are iterative and incremental, they repeat over a series of iterations making up the life cycle of a system. Each iteration takes place over time and it consists of one pass through the requirements, analysis, design, implementation and test activities, building a number of different artifacts. All these artifacts are not independent. They are related to each other, they are semantically overlapping and together represent the system as a whole. Elements in one artifact have trace dependencies to other artifacts. For instance, a use case (in the use-case model) can be traced to a collaboration (in the design model) representing its realization.

On the other hand, due to the incremental nature of the process, each iteration results in an increment of artifacts built in the former iteration. An increment is not necessarily additive. Generally in the early phases of the life cycle, a superficial artifact is replaced with a more detailed or sophisticated one, but in later phases increments are typically additive, i.e. a model is enriched with new features, while previous features are preserved.

Figure 1 lists the classical activities – requirements, analysis, design, implementation and test – in the vertical axis and the iteration in the horizontal axis , showing the following kinds of relations:

-horizontal relations between artifacts belonging to the same activity in different iterations (a use case is extended by another use case)

-vertical relations between artifacts belonging to the same iteration in different activities (e.g. an analysis model is realized by a design model).

Traditional specifications of development process typically consist of quite informal descriptions of a set of software development artifacts together with a graph of tasks and activities. But, the software development process should be formally defined since the lack of accuracy in its definition can cause problems, for example:

- Inconsistency among the different artifacts: if the relation existing among the different sub-models is not accurately specified, it is not possible to analyze whether its integration is consistent or not.

- Evolution conflicts: when a artifact is modified, unexpected behavior may occur in other artifacts that depend on it.

- Confusion regarding the order in which tasks should be carried out by developers.

- It is not possible to reason about the correctness of the development process.

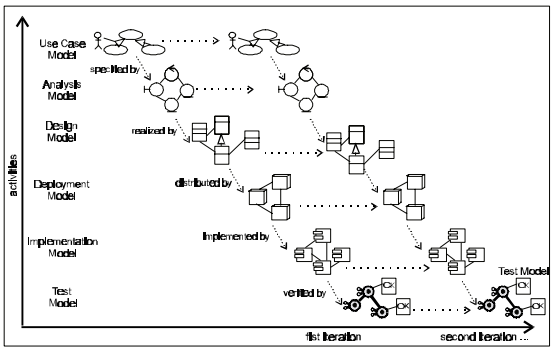


Figure 1. dimensions in the development process

We propose to apply the well-known mathematical concept of contract to the description of software development processes in order to introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their activities. Furthermore, development contracts are organized in a modular and hierarchical way leading to a better understanding of the whole software development process.

2. The notion of software contract

A computation can generally be seen as involving a number of agents (objects) carrying out actions according to a document (specification, program) that has been laid out in advance. This document represents a contract between the agents involved. The notion of contract regulating the behavior of a software system has been already introduced by several authors [Helm et. al 90, Meyer 91, Meyer 97, Back and von Wright, 98; Andrade and Fiadeiro 99]. A contract imposes mutual obligations and benefits. It protects both sides (the client and the contractor):

- It protects the client by specifying how much should be done: the client is entitled to receive a certain result.

- It protects the contractor by specifying how little is acceptable: the contractor must not be liable for failing to carry out tasks outside of the specified scope.

As example consider the contract in figure 2, in which a *subject* object, containing some data, and a collection of *view* objects, which represent the data graphically, cooperate so that at all times each *view* always reflects the current value of the *subject*. This contract defines the behavioral composition of subject and views participants. The contract specifies the following aspects: firstly, it identifies type obligations, where the participant must support certain external interface, and causal obligations, where the participant must perform an ordered sequence of actions and make certain conditions true in response to these messages. Secondly, the contract defines invariants that participants cooperate to maintain.

contract SubjectView

Participants: Subject, View

Subject supports [
data: Data
setData(val:Data) ~ Ddata {data=val};notify()
notify() ~ ⟨∀v:v∈ views: v.update()⟩
attachView(v:View) ~ {v∈ views}
detachView(v:View) ~ {v∉ views}]
Views:Set(View) where each View supports [
update() ~ {view *reflects* subject.data}
setSubject(s:Subject)~ {subject=s)}]
invariant
subject.setData(val) ~
⟨∀v∈ views: v *reflects* subject.data⟩

end contract

Figure 2: contract SubjectView [Helm et al, 90]

3. Software contracts as mathematical entities

We take the view of contracts as proposed by [Back and von Wright, 98] and [Back et al., 99]. The world that a contract talks about is described as a state σ. The state space Σ is the set of all possible states σ. The state is observed as a collection of attributes x_1, x_2, ...,x_n, each of which can be observed and changed independently of the others. Attributes are partitioned into objects

An agent changes the state by applying a function f to the present state σ, yielding a new state f.σ. A function f:Σ→ Σ that maps states to states is called state transformer. An example of state transformer is the assignment x:=exp, that updates the value of attribute x to the value of the expression exp.

A boolean function p: Σ→Bool is called a state predicate. A state relation R:Σ→ Σ→ Bool relates a state σ to a state σ′ whenever R.σ.σ′holds.

Assume that there is a fixed collection A of agents. Let ***a***, ***b***, ***c*** denote individual agents. We describe contracts using the notation for contract statements [Back and von Wright, 98]. The syntax for these is as follows:

S ::= ⟨f⟩ | if p then S_1 else S_2 fi |S_1 ; S_2| $assert_a$ p | R_a | $choice_a$ S_1 U S_2 | while p do S_1 od

Here a stands for an agent while f stands for a state transformer, p for a state predicate, and R for a state relation, both expressed using higher-order logic. Intuitively, a contract statement is executed as follows:

The functional update <f> changes the state according to the state transformer f, i.e., if the initial state is σ_0 then the final state is f. σ_0 . An assignment statement is a

special kind of update where the state transformer is expressed as an assignment. For example, the assignment statement <x:=x+y> requires the agent to set the value of attribute x to the sum of the values of attributes x and y. The name skip is used for the identity update <id>, where id.$\sigma = \sigma$ for all states σ.

In the conditional composition if p then S_1 else S_2 fi, S_1 is carried out if p holds in the initial state, and S_2 otherwise.

In the sequential composition S_1 ; S_2 , statement S_1 is first carried out, followed by S_2.

An assertion $assert_a$ p , for example, $assert_a$ (x+y=0) expresses that the sum of (the values of) x and y in the state must be zero. If the assertion holds at the indicated place when the agent a carries out the contract, then the state is unchanged, and the rest of the contract is carried out. If, on the other hand, the assertion does not hold, then the agent has breached the contract.

The relational update and choice both introduce non-determinism into the language of contracts. Both are indexed by an agent which is responsible for deciding how the non-determinism is resolved.

The relational update R_a requires the agent ***a*** to choose a final state σ' so that R.σ.σ' is satisfied, where σ is the initial state. In practice, the relation is expressed as a relational assignment. For example, $update_a$ {x := x´| x´ <x} expresses that the agent ***a*** is required to decrease the value of the program variable x. If it is impossible for the agent to satisfy this, then the agent has breached the contract.

The statement $choice_a$ S_1 U S_2 allows agent ***a*** to choose which is to be carried out, S_1 or S_2 .

Finally, recursive contract statements are allowed. A recursive contract is defined using an equation of the form X = S. where S may contain occurrences of the contract variable X. With this definition, the contract X is intuitively interpreted as the contract statement S, but with each occurrence of statement variable X in S treated as a recursive invocation of the whole contract S. Also it is permitted the syntax (rec X•S) for the contract X defined by the equation X=S. An important special case of recursion is the while-loop which is defined in the usual way: while p do S od =(rec X•if p then S ; X else skip fi)

3.1 Predicate transformer semantics (Weakest preconditions)

In order to analyze a contract it is necessary to express the precise meaning of each statement , i.e. we need the semantics of contract statements. The semantics is given within the refinement calculus using the weakest precondition predicate transformer [Back and von Wright, 98].

A predicate transformer is a function that maps predicates to predicates. Predicate transformers are ordered by pointwise extension of the ordering on predicates, so $F \subseteq F'$ for predicate transformers holds if and only if $F.q \subseteq F'.q$ for all predicates q. The predicate transformers form a complete lattice with this ordering, and $\cup$ and $\cap$ are the operators of this lattice.

Different agents are unlikely to have the same goals, and the way one agent makes its choices need not be suitable for another agent. From the point of view of a specific agent or a group of agents, it is therefore interesting to know what outcomes are possible regardless of how the other agents resolve their choices.

Consider the situation where the initial state σ is given and a group of agents A agree that their common goal is to use contract S to reach a final state in some set q of desired final states. It is also acceptable that the coalition is released from the contract, because some other agent breaches the contract. This means that the agents should strive to make their choices in such a way that the scenario starting from σ ends in a configuration σ', where either σ' is an element in q, or some other agent has breached the contract.

Assume that S is a contract statement and A a coalition, i.e., a set of agents. We want the predicate transformer wp_A.S to map postcondition q to the set of all initial states σ from which the agents in A jointly have a winning strategy to reach the goal q. Thus, wp_A.S.q is the weakest precondition that guarantees that the agents in A can cooperate to achieve postcondition q. This means that a contract S for a coalition A is mathematically seen as an element (denoted by wp_A.S) of the domain $P\Sigma \rightarrow P\Sigma$

These definitions are consistent with Dijkstra original semantics for the language of guarded commands [Dijkstra, 76] and with later extensions to it, corresponding to non-deterministic assignments, choices, and miracles.

The definition of the weakest precondition semantics is as follows (see[Back and von Wright, 98] for a more detailed explanation):

$wp_A.\langle f \rangle.q = (\lambda\sigma.q.(f.\sigma))$

wp_A.(if p then S_1 else S_2 fi).q =
$(p \cap wp_A.S_1.q) \cup (\neg p \cap wp_A.S_2.q)$

$wp_A.(S_1;S_2).q = wp_A.S_1.(wp_A.S_2.q)$

$wp_A.(assert_a\ p).q = \lambda\sigma.(p.\sigma \wedge q.\sigma)$, if $a \in A$
$\lambda\sigma.(\neg p.\sigma \vee q.\sigma)$, if $a \notin A$

$wp_A.R_a.q = \lambda\sigma.\exists\sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma'$, if $a \in A$
$\lambda\sigma.\forall\sigma' \bullet R.\sigma.\sigma' \rightarrow q.\sigma'$, if $a \notin A$

$wp_A.(choice_a\ S_1\ U\ S_2).q = wp_A.S_1.q \cup wp_A.S_2.q$, if $a \in A$
$wp_A.S_1.q \cap wp_A.S_2.q$, if $a \notin A$

4. The notion of software development contract

The notion of formal contract described in section 3, can be applied to the software development process itself. That is to say, the software development process can be seen as involving a number of agents (the development team and the software artifacts) who carry out actions with

the goal of building a software system that meets the user requirements.

While the notion of formal contract regulating the behavior of software agents is accepted, the concept of contract regulating the activities of software developers is quite vague. In general there is not documented contract establishing obligations and benefits of members of the development team. As we remarked in section 1, in the best of the cases the development process is specified by either graph of tasks or object-oriented diagrams in a semi-formal style, while in most of the cases activities are carried out on demand, with little previous planning.

However, a disciplined software development methodology should encourage the existence of formal contracts between developers, so that contracts can be used to reason about correctness of the development process, and comparing the capabilities of various groupings of agents (coalitions) in order to accomplish a particular contract.

Assume you are planning a work to be performed by a development team in order to adapt the model of a system to new requirements (e.g. during the n+1 iteration of the development process). This work can be expressed as a combination (in sequence or in parallel) of sub-works, each of them to be performed by a member of the development team. It is necessary to make sure that sub-works will be performed as required. This is only possible if the agreement is spelled out precisely in a contract document. This idea is based on the metaphor: software development is a sequence of documented contract decisions.

A remarkable difference between software contracts and development contracts is the kind of object constituting a state. While in software contracts, objects in the state represent object in a system, such as a bank account or a book, in software development contracts, objects in the state are development artifacts, such as a class diagram or a use case. But this difference is just conceptual, from the mathematical point of view we can reason about development contracts in the standard way, as if they were software contracts.

There are different levels of granularity in which development contracts can be defined. On one hand we have contracts regulating primitive evolution, such as adding a single class in a Class diagram, while on the other hand we have contracts defining complex evolution, such as the realization of a use case in the analysis phase by a collaboration diagram in the design phase, or the reorganization of a complete class hierarchy. Complex evolution are not atomic tasks, instead they are made up with primitive evolutions. So, we start specifying atomic contracts (contracts explaining primitive evolution) which will be the building blocks for non-atomic contracts (i.e. regulations for complex evolution).

4.1 Primitive development-contracts

In order to specify primitive development-contracts we may associate a precondition and a postcondition with each primitive evolution operation on models.

In order to make contracts more understandable and extensible, we use the object-oriented approach to specify them. The object oriented approach deals with the complexity of description of software development process better than the traditional approach. Examples of this are the framework for describing UML compatible development processes defined in [Hruby 99] and the metamodel defined by the OMG Process Working Group [OMG 98], among others. In the object-oriented approach, software artifacts produced during the development process are considered objects with methods and attributes. Evolution during the software development process is represented as collaborations between software artifacts and users of the method.

We use the following object oriented syntax for specifying classes of artifacts:

Specification of ClassName	
Superclasses list of direct superclasses	
Attributes	
	list of attributes and associations.
Derived Attributes	
	list of attributes and associations whose values can be calculated from other attributes or associations.
Predicates	
	list of boolean functions
Invariants	
	list of predicates that should be true in all states.
Operations	
	list of method declarations
End specification of ClassName	

Where a method declaration has a name m, a precondition p and an effect S (the body of the method). When a method is called there is an agent ***a*** responsible for the call. The method invocation is then interpreted as ´$assert_a$ p ; S´, i.e. the agent is responsible for verifying the preconditions of the method. If agent ***a*** has invoked the method in a state that does not satisfy the precondition, then ***a*** has breached the contract.

At the present the Unified Modeling Language [UML, 2000] is considered the standard modeling language for object oriented software development process. As example, we present the evolution contracts of some UML artifacts. Lets consider a part of the UML metamodel describing Class, Feature, Package and Generalization artifacts. The contract for some primitive operations on these artifacts can be specified as follows (parts of the specification are omitted due to space limitations):

Specification of GeneralizableElement	
Superclasses ModelElement	
Attributes	
	generalizations: Set of Generalization specializations: Set of Generalization

	isAbstract: Bool
Derived Attributes	
	[1] c.parents returns the set of direct parents of c. parents: Set of GeneralizableElement c.parents=c.generalizations.collect(parent) [2] c.children returns the set of direct child of c. children: Set of GeneralizableElement c.children = c.specializations.collect(child)
Predicates	
	IsA : GeneralizableElement x GeneralizableElement →Bool IsA(c,c1) ↔ c=c1 ∨ c1∈ c.allParents
Invariants ∀ c_1,c_2 : GeneralizableElement	
	[1] Circular inheritance is not allowed. $IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$
End specification of GeneralizableElement	

Specification of Classifier	
Suplerclasses GeneralizableElement, NameSpace	
Attributes	
	features: Seq of Feature associationEnds: Set of AssociationEnd
Derived Attributes	
	[1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features. allFeatures : Set of Feature c.allFeatures = c.features ∪ ($\cup_{ci\in c.parents}$ ci.allFeatures) [2] The operation allAssociationEnds results in a Set containing all AssociationEnds of the Classifier itself and all its inherited associationEnds. allAssociationEnds: Set of AssociationEnd c.allAssociationEnds= c.associationEnds ∪ ($\cup_{ci\in c.parents}$ ci.allAssociationEnds) [3] The operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the classifier. .....
Invariants ∀c:Classifier	
	[1] No Attributes may have the same name within a Classifier ∀f,g∈ c.attributes (f.name=g.name →f=g) [2] No Operations may have the same signature in a Classifier. ∀f,g∈ c.operations ((hasSameSignature(f,g) →f=g)
Operations	
	proc c.addFeature (f:Feature) ***Precondition*** [1] The class exists (it is stored in some package) c.package≠null [1] the new Feature does not belong to c f∉ c.attributes [2]No Features may have the same name within a Classifier ∀g∈ attributes(c) f.name≠ g.name [3] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd. ∀e∈ c.oppositeAssociationEnds f.name≠e.name [4] The connected type should be included in the Package of the Classifier. f.type∈ (c.package).allContents ***Effect*** [1] the feature is added to the list of features c.features:=c.features∪{f} ; f.owner:=c
End specification of Class	

Specification of Package	
Superclasses NameSpace, GeneralizableElement	
Attributes	
	importedElements: Set of ModelElement ownedElements: Set of ModelElement
Derived Attributes	
	[1] The operation contents results in a set containing all ModelElements owned or imported by the Package. contents : Set of ModelElement p.contents = p.ownedElements ∪ p.importedElements
Invariants ∀p: Package	
	[1] in a Package the Classifier names are unique ∀c_1,c_2: Classifier ((c_1∈p.contents ∧ c_2∈ p.contents ∧ c_1.name = c_2.name) → c_1 =c_2)
Operations	
	proc p.addGeneralization (g:Generalization) ***Precondition*** [1] the generalization is not in the package g∉ p.allContents [2] all elements connected by the new relationship must be included in the Package. g.parent∈p.allContents ∧ g.child∈p.allContents [5] Circular inheritance. *IsA*(g.parent, g.child) → g.parent = g.child [6] multiple inheritance. ∀c:Classifier (*IsA*(g.child,c) → ∀f,g:Feature((f∈ ((g.parent).allFeatures) ∧ g∈c.allFeatures ∧ f.name=g.name) → f=g)) ***Effect*** [1] the generalization is inserted into the package p.ownedElement::= p.ownedElement ∪ {g}; g.package:=p [2] The new generalization is linked to the generalizable elements g.parent.specializations := g.parent.specializations ∪ {g}; g.child.generalizations := g.child.generalizations ∪ {g} .....
End specification of Package	

4.2 Complex development-contracts

On top of primitive contracts it is possible to define complex contracts, specifying non-atomic forms of evolution through the software development process. Then, by using the wp predicate transformer we can verify whether a set of agents (i.e. software developers) can achieve their goal or not. We can analyze whether a developer (or team of developers) can apply a group of modifications on a model or not by means of a contract designed in terms of a set of primitive operations conforming the group.

Developers will successfully carry out the modifications if some preconditions hold. We can determine the weakest preconditions to achieve a goal by computing:

wp_A . C . Q

where C is the contract, A is the set of software developers (agents) and Q is the goal.

If computing the wp we obtain a predicate different from false, then we proved that with the contract the developers can achieve their goal under certain pre-conditions.

Example 1: a collaborative work

Lets consider a collaborative work, in which three software developers have to modify a class diagram. One of the agents will detect and delete all the features that could be lifted to a superclass (i.e. features that appear repeated in all of the subclasses of a given class). The second agent has the responsibility of lifting the feature (i.e. to add the deleted feature in the superclass). As a consequence of the lifting process, some classes may become empty (i.e. without proper features). Finally the third agent will detect and delete empty classes. Figure 3 illustrates the collaborative process described above.

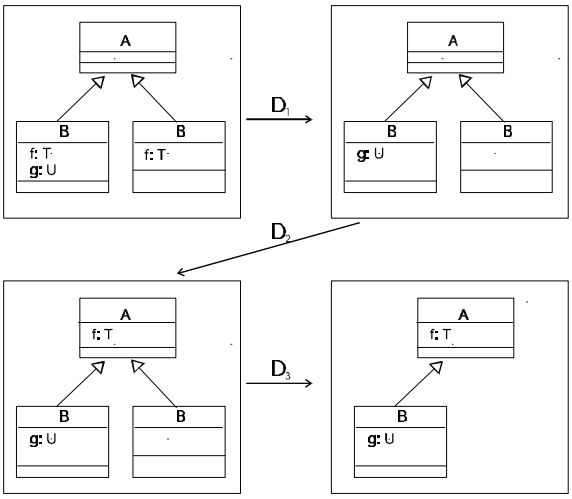


Figure 3: the collaborative refactoring task

We are interested in calculating the weakest precondition for agents D_1 , D_2 and D_3 to reach the goal Q by using the contract R. That is to say:

$wp_{\{D1,D2,D3\}}$. R. Q

Where:

Def 1: the contract

R ≡ CONTRACT refactoring
agents D, D_1, D_2, D_3
var p:Package, c:Class, f:Feature
proc liftingRepeatedFeature:
$update_{D1}$ c:=s | ∃f:Feature • (∀c´∈s.subclasses • f∈c'.features) ;
$update_{D1}$ f:=f´ | ∀c´∈ c.subclasses • f´∈ c'.features ;
while (∃c´∈c.subclasses • f∈c'.features)
do $update_{D1}$ c´:=c´´ | c´´∈ c.subclasses ∧ f∈c''.features ;
c´.deleteFeature(f)$_{D1}$;
od;
c.addFeature(f)$_{D2}$;
end proc.

proc deletingEmptyClass:
$update_{D3}$ c:=c´ | c'.features≠∅;
p.deleteClass(c)$_{D3}$;
end proc.

begin
while (¬Q)
do $choice_D$ liftingRepeatedFeature U deletingEmptyClass
od;
end.

Def 2: the postcondition

Q ≡ $q_1 \wedge q_2$
where:
q_1 ≡ ∀c:Class • ¬∃f:Feature • (∀c´∈c.subclasses • f∈c'.features)
q_2 ≡ ∀c:Class • c.features≠∅

Q specifies the expected effect of the refactoring process as the combination of two facts: q_1 says that there are no repeated features while q_2 specifies that the model does not contain any empty class.

Example 2: Using contracts to reasoning about evolution conflicts

Arbitrary modifications that do not cause problems when they are applied exclusively, may rise conflicts when they are integrated (i.e. they are applied together). For example if both evolutions - deleting a class and adding a feature to the class- are applied sequentially a conflict may occur because it is not possible to add a feature to a missing class.

C ≡ CONTRACT conflict
agent D1, D2

var p:Package, c:Class, f:Feature;
begin
$p.delClass(c)_{D1}$; $c.addFeature(f)_{D2}$
end.

We can prove that $wp_{\{D1,D2\}}$. C. Q is false. Where Q is any predicate. It is impossible for agents D1 and D2 to carry out the contract.

Example 3: checking consistency between artifacts

Lets consider a collaborative work in which two agents D1 and D2 need to add a generalization relationship respectively, preserving the well formedness property of the model.

In particular, it is possible to find out which is the weakest precondition to achieve the goal of introducing two generalization relationships without breaking the non-circularity principle of inheritance hierarchies by computing:

$wp_{\{D1,D2\}}$. C . Q

Where C is the contract between agents and Q is a predicate that specifies absence of circularity in the hierarchies and that the new relationships were established.

We will calculate the weakest precondition for agents D_1 and D_2 to reach the goal Q by using the contract C, That is to say:

$wp_{\{D1,D2\}}$. C. Q = P

where:

- C ≡ CONTRACT circular
agents D1, D2
var p:Package, r,g:Generalization;
begin
$p.addGeneralization(r)_{D1}$; $p.addGeneralization(g)_{D2}$
end.

- $Q \equiv q \wedge q'$

Where q specifies the effect of the evolution (generalizations were added in the package) and q´ specifies a well-formedness rule (there is no circular inheritance).

$q \equiv (r \in p.ownedElements \wedge g \in p.ownedElements)$

$q' \equiv \forall c_1,c_2$:GeneralizableElement. ($IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$)

Finally, the expected weakest precondition is as follows:

- $P \equiv P_1 \wedge P_2 \wedge H$

Where P_1 an P_2 specify preconditions for applying the first and the second evolutions respectively (as if they were applied in isolation). And H specifies a special requirement to avoid circular inheritance in case both evolution actions are applied together. $P_1 \equiv r \notin p.allContents \wedge r.parent \in p.allContents \wedge r.child \in p.allContents \wedge$

$IsA(r.parent, r.child) \rightarrow r.parent = r.child$

$P_2 \equiv g \notin p.allContents \wedge g.parent \in p.allContents \wedge g.child \in p.allContents \wedge$

$IsA(g.parent, g.child) \rightarrow g.parent = g.child$

$H \equiv \neg (IsA(g.parent, r.child) \wedge IsA(r.parent, g.child))$

The complete derivation can be read in [Pons and Baum, 2001] Figure 4 illustrate a conflictive case, in which the expected weakest pre-condition does not hold.

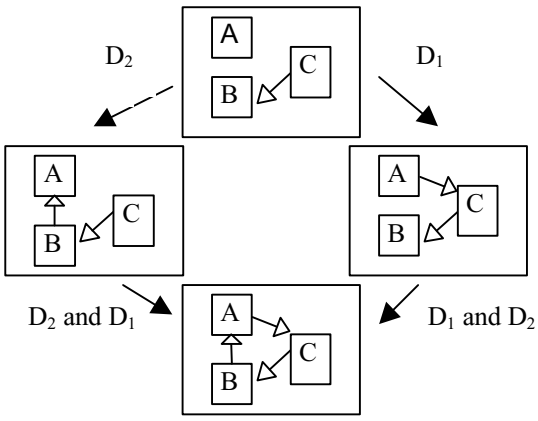


Figure 4: evolution conflict

5. Conclusion and related work

Software development process is a collaborative process. As a consequence it is necessary to formally specify benefits and obligations of partners involved in the process in order to avoid misunderstandings and conflicts.

We apply the well-known mathematical concept of contract to the specification of software development processes in order to introduce precision of specification, avoiding ambiguities and inconsistencies, and enabling developers to reason about the correctness of their joint activities.

Contracts provide a formalization of software artifacts and their relationships. Also contracts clearly establish pre and post conditions for each software development activity. The goal of the proposed formalism is to provide foundations for tools that assist software engineers during the development process.

In general there is not documented contract establishing obligations and benefits of members of the development team, i.e. software development processes are specified in a semi-formal style. For example the specification of the standard graphical modeling notation UML [UML, 2000] and the Unified Process [Jacobson et al., 99] is semi-formal, i.e. certain parts of it are specified with well-defined languages while other parts are described informally in natural language. There are an important number of theoretical works giving a precise

description of core concepts of the UML and providing rules for analyzing their properties; see, for instance [Back et al. 99; Breu et al., 1997; Evans et al., 1999; Kim and Carrington, 1999; Övergaard 1999; Pons et al. 1999, Pons et al 2000], while less effort has been dedicated to the formalization of UML compatible software development processes.

The mechanism of development contracts introduced in this paper, is related to the mechanism of reuse contracts [Steyaert et al. 96, Lucas 97]. A reuse contract describes a set of interacting participants. Reuse contracts can only be adapted by means of reuse operators that record both the protocol between developers and users of a reusable component and the relationship between different versions of one component that has evolved. Similarly, in [Mens et al. 2000] the authors translate the idea of reuse contracts in order to cope with reuse and evolution of UML models.

The originality of development contracts resides in the fact that software developers are incorporated into the formalism as agents (or coalition of agents) who make decisions and have responsibilities. Given a specific goal that a coalition of agents is requested to achieve , we can use traditional correctness reasoning to show that the goal can in fact be achieved by the coalition, regardless of how the remaining agents act. The wp formalism allows us to analyze a single contract from the point of view of different coalitions and compare the results. For example, it is possible to study whether a given coalition *A* would gain anything by permitting an outside agent *b* to join *A*.

Finally, sine the construction of formal development contracts is a hard task, it is important to consider evolution and reuse of contracts themselves. As contracts are written in an object-oriented style, it is possible to define a new contract by specializing an existing one. This feature does not solve the complexity problem completely, but it facilitates the task of creation and evolution of contracts.

References

Andrade,L and Fiadeiro,J.L, Interconecting objects via Contracts. Proceedings of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer Verlag. (1999).

Back, R and von Wright, J., Refinement Calculus: A Systematic Introduction, Graduate texts in Computer Science, Springer Verlag, 1998.

Back, R. Petre L. and Porres Paltor I., Analysing UML Use Cases as Contract. .Procs of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer. (1999).

Coleman, D..Arnols, P Bodoff,S, Dollin, C, Gilchrist,H, Hayes,F, Jeremaes,P. Object Oriented Development: The Fusion Method. Prentice-Hall 1994.

Dijkstra, E, A Discipline of Programming. Prentice Hall International, 1976.

D´Souza D. and Wills, A. Objects, Components and Frameworks with UML: the Catalysis approach, Addison Wesley, 1998.

Breu,R., Hinkel,U., Hofmann,C., Klein,C., Paech,B., Rumpe,B. and Thurner,V., Towards a formalization of the unified modeling language. ECOOP'97 procs., Lecture Notes in Computer Science vol.1241, Springer, (1997).

Evans,A., France,R., Lano,K. and Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, H,Kilov editor, , Kluwer Academic Publishers, (1999).

Helm,R. Holland,I and Gangopadhyay,D. Contracts: specifying behavioral compositions in object-oriented systems, Proc. OOPSLA'90. ACM Press. Oct 1990.

Hruby, Pavel, Framework for describing UML compatible development processes. in <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. R.France and B.Rumpe editors, Proceedings of the UML´99 conference, Lecture Notes in Computer Science 1723, Springer Verlag. (1999).

Jacobson, I..Booch, G Rumbaugh, J., The Unified Software Development Process, Addison Wesley. ISBN 0-201-57169-2 (1999)

Kim, S. and Carrington,D., Formalizing the UML Class Diagrams using Object-Z, In Proc. <<UML>>´99 - The Second International Conference on the Unified Modeling Language, Lecture Notes in Computer Sciencie 1723, (1999).

Lucas, Carine "Documenting Reuse and evolution with reuse contracts", PhD Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, September 1997.

Mens,T., Lucas,C. and D'Hondt, T.. Automating support for software evolution in UML. Automated Software Engineering Journal 7:1, Kluwer Academic Publishers, February 2000.

Meyer, B. Advances in object oriented software engineering. Chapter 1 "Design by contract". Prentice Hall, 1992.

Meyer,B. Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997.

OMG White Paper on Analysis and Design Process Engineering, Process Working Group, Analysis and Design Platform Task Force, OMG Document. July 1998.

Övergaard, G., A formal approach to collaborations in the UML, In Proc. <<UML>>´99 - The Second International Conference on the Unified Modeling Language. R.France and B.Rumpe editors, Lecture Notes in Computer Science 1723, Springer. (1999).

Pons,C., Baum,G., Felder,M., Foundations of Object-oriented modeling notations in a dynamic logic framework, Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, 1999.

Pons, C., Giandini, R. and Baum, G. Specifying Relationships between models through the software development process Tenth International Workshop on Software Specification and Design (IWSSD), California, IEEE Computer Society Press. November 2000.

Pons,C and Baum G. Software Development Contracts, extended version.In www-lifia.info.unlp.edu.ar/~cpons

Steyaert, P..Lucas, C.Mens K and D'Hondt, T. Reuse Contracts: Managing the evolution of reusable assets. In proceedings of OOPSLA'96, New York, Oct 1996.

UML, The Unified Modeling Language Specification – Version 1.3,. UML Specification, revised by the OMG, http://www.omg.org, March, 2000.

Thoughts on the Role of Formalisms in Studying Software Evolution

International Special Session on Formal Foundations of Software Evolution, co-located with CSMR 2001
13 March 2001, Lisbon

Meir M Lehman Juan F Ramil Goel Kahen
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
tel +44 20 7594 8216; fax +44 20 7594 8215
{mml,ramil,gk}@doc.ic.ac.uk

Summary

This paper presents a *system dynamics* model of a long-term software evolution process as an example of process *behavioural formalism* and shows how the model permits assessment of the impact of various policies on evolutionary attributes. The model provides a context and framework within which at least three crucial software management tasks, resource allocation, release planning, and process performance monitoring can be tackled. It is part of and exemplifies the methods for software process modelling being developed and applied in the FEAST, *F*eedback, *E*volution *A*nd *S*oftware *T*echnology, projects.

1 Introduction

The term *software evolution* relates to the activity and phenomenon of software change [leh85]. It includes two aspects that reflect, respectively, the complementary concerns of the *how* and the *what/why* [leh00b] of software evolution. Interest in the former is concerned with *methods*, *tools* and *techniques* to change functional, performance and other characteristics of the software in a controlled, reliable, fast and cost effective manner. This is the more widespread view and is exemplified by the contributions to a series of meetings on Principles of Software Evolution [ispse98,00]. Interest in the *what/why*, on the other hand, focuses on *understanding* the software evolution *phenomenon*, its underlying causes and drivers, common patterns of evolutionary behaviour, and the characteristics of that behaviour. This line of investigation, the focus of the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) studies in the Department of Computing at Imperial College [feast] and their antecedents, has also been pursued by a small number of other groups world-wide [e.g. kem99,coo00,gdf00,raj00].

Both views, the *how* and the *what/why*, must be pursued if mastery of the software evolution phenomenon is to be achieved in a world increasing dependent on computers and software. The following are examples of the type of questions whose answer is pursued under the latter view:

- why does software evolution occur?
- why is it inevitable?
- what are key attributes of the evolution process?
- what is their impact on the software process and its products?
- what are the practical implications of the above on the planning control and management of software system evolution?

One of the present authors (mml) has been actively involved in studies of software evolution for more than 30 years [leh69,85,feast]. This work has resulted in a set of laws of software evolution [leh74,78,80a,b,85,feast], the *SPE* program classification scheme [leh80b], a principle of software uncertainty [leh89,90] and, most recently, a FEAST hypothesis [leh94feast]. The results of the recent work within the FEAST projects are scattered in some 40 papers published since 1996 [feast]. A full listing is available from the project web site http://www.doc.ic.ac.uk/~mml/feast.

2 Towards a Formal Theory of Software Evolution

It is the view of the present authors that formalisms can play as important a role in the study of the *what* and the *why* of software evolution as they do in the *how* view, even though they serve different purposes. In the *how* mode, they are primarily intended to be used as representations of different models of the *application*; that is, specifications, programs, the operational, evolution domains and even of entities such as the evolution process, system architectures and relationships such as abstraction and satisfaction [mai00]. And all these models must permit continual representation of the subject as it evolves. The power of appropriate formalisms in this area is clear.

3 Behavioural Formalisms

One of the roles of formalisms under the *what/why* view can be to facilitate precise reasoning about the behaviour of the evolution process, and its product. Managers and process designers could frequently benefit from reasoned exploration of behavioural issues but lack the reasoning tools to do so. Of equal relevance is the potential role of formalisms in guiding the direction and likelihood of future

changes in process, product or domain attributes or the direction and likelihood of future changes in *needs*.

Formalisms to facilitate such reasoning have emerged, for example, from the work in process modelling languages over the last 15 years or so [ost87,97,pot97]. The emphasis of that work has been primarily on process *description* and *prescription.* Formalisms have also been applied by the workflow community [e.g., wir00]. Combining both concepts, models such as *process programs* [ost87,97] indicate the steps that constitute a process, workflow controls, conditions to activate sub-processes and so on. Within this view, fine-grained characteristics and properties such as absence of deadlock, were also of interest.

The present authors believe that reasoning about process *behaviour* and about properties such as the *economic feasibility* of a process or about its quality and other performance, however measured, is at least as relevant as is reasoning about process *description* and *prescription.*

The introduction of formalism to the study of process *behaviour* raises many issues[1]. Some of these have previously been analysed, for example, by those investigating the use of mathematics in sociology [col64]. This brings with it the question whether process behaviour is predominantly *indeterministic* (as defined by Chapman [cha96]) and therefore not, since it involves humans at all levels, in general amenable to mathematical formalisation. The same question arises in the study of the software process. If this view were to prevail, the use of formalisms to study process behaviour would be a futile exercise. Some software process behaviour has, however, been captured in empirical generalisations (e.g. laws of software evolution [leh74,78,80a,b,85,feast]) as has, for example, software process effort estimation in COCOMO [boe00] These are, by themselves, sufficient to demonstrate that there is a role for formalism in the study of process behaviour. Other evidence also derived from empirical studies [e.g. abd91,leh98] supports this conclusion; has demonstrated that mathematical formalisms such as differential equations for example, have their uses in other such studies.

One of the outcomes of the FEAST projects has been the realisation that one may extend the use of formalisms to achieve rigorous representations of behavioural invariants and empirical generalisations such as the laws of software evolution, on the one hand, and rules and guidelines [leh00a] for project management, on the other [leh00c]. If this can be successfully achieved one will be able to provide a formal rationale for what is termed *good practice.* Even more importantly, one will be able to provide a formal theory of software evolution as the foundations for a unified and coherent framework for software engineering. The development of such a theory is the theme of a recent project proposal [leh00c,d].

4 Software Process Simulation Modelling

The argument in favour of behavioural formalisms accords with a recent call for software engineering research to abandon the flatland of purely *technical* issues and to proceed to incorporate other dimensions such as cost and value [boe00]. One possible way to achieve this and to proceed to a disciplined study of process behaviour is by means of simulation languages and tools, and models derived therefrom. One example of this approach is provided by the work of the process simulation community [kel99,prosim00][2]. Another example is provided by the FEAST projects with its models reflecting aspects of long-term evolution management [feast]. That work involved development of *system dynamics* models [for61,abd91,coy96]. The tool used was *Vensim*® [ven95].

As briefly discussed in section 7, those models provide an example of the use of formalisms, in this case system dynamics, for the study software evolution from the behavioural point of view. The work is illustrated here by a model intended for use in long-term planning and management of software evolution processes. The outputs of this model all relate to the evaluation of effort allocation policies. However, alignment of the present model to actual industrial processes, its calibration against them and determination of its domain and extent of validity [for80] remain to be done. If successfully accomplished, the result will be a model that can be used within the processes it reflects for their further planning, management and improvement.

Incidentally, this application draws attention to an issue considered in other disciplines and specifically addressed by Heisenberg's Principle of Uncertainty. Using a model of system evolution to plan implementation of that evolution will influence resultant process behaviour, is indeed intended to do so. Thus, it may serve as a self-fulfilling prophecy, confirming (and perpetuating) the validity of the model, even though objectively it does not accurately reflect the phenomenon. This observation appears to point to a fundamental principle relating to the evolution process. It cannot be pursued here other

[1] See [mcg97] for a justification of software process behavioural formalisms from a different but complementary perspective.

[2] Formalisms such as Petri-Nets [e.g. kus97] or state charts and the *STATEMATE*® system [e.g. har90] have also been used in process behavioural modelling. We do not here discuss under what circumstances one formalism is more appropriate than another in this application or whether a combination of formalisms can offer an advantage [ram98].

than to observe that it is related to the observation that software operating in and with a real-world domain incorporates a model of itself [leh85].

In any event, what can be said is that the system dynamic models referred above incorporate behavioural formalisms of software evolution. Hence they are relevant, and hopefully, of interest to FFSE.

5 An Example: Change and Complexity in Evolving Software

Software evolution may be described as the achievement of disciplined software change. It is driven, *inter alia*, by the need to maintain user satisfaction within a changing application and usage domain. Changes are inevitably in the application domain, user familiarity, needs and domain properties. They result from user learning, familiarisation and other developments within an environment in which market forces, human interest and ambition, technology, the influence of factors and agents exogenous to the application and system also play a role. Evolution entails adaptation of existing properties, functionality in particular, and the addition of new capability. The latter implies system functional growth over time and releases. The ultimate goal is to at least maintain and, generally, to increase stakeholder satisfaction.

In the above context, one underlying fact of life must be accepted. As a consequence of the superposition of change upon change upon change, the *complexity* of software systems tends to increase as they evolve [leh74]. Such increase brings with it, pressure for a decline in the attainable *functional growth rate* [e.g. leh98]. Managers can either ignore this decline and face the inevitable consequences of eventual system stagnation. Alternatively they can take cognisance of the complexity growth and divert effort to *control* it and any other forces causing the decline in growth rate. Given an awareness that growth trends that constrain system evolution develop, they may well accept the need to direct effort to activities that might otherwise have been overlooked or neglected. However, if the need is not recognised or not accepted such anti-regressive activities will tend to be neglected. This, despite the fact that, unless controlled, as the system evolves, growing complexity will force down system maintenance, adaptation and extension productivity and system quality will deteriorate. This is a fact that cannot be permitted to materialise when control and mastery of system evolution is vital in a society increasingly reliant on *inventories* of ageing software.

6 Complexity Control: Anti-regressive Work

Growing complexity is reflected by increased size, more *interdependent* functionality, a larger number of integrated components, more control mechanisms, a higher level of reciprocal interdependency. It is reflected in and evidenced by greater inter-element connectivity and more complex (sic) interfaces. In this context, the achievement of a minimum level of complexity management and control activity is essential to maintain the rate of system evolution at the desired or required level.

Motivated by Baumol's classification [bau67] of activity into *progressive* and *anti-progressive* types, Lehman suggested [leh74] a further category, *anti-regressive*. Activities that, by addition or modification of functionality for example, enhance system value were termed *progressive*. Effort such as complexity control or reduction, on the other hand, does not, from the short-term point of view, contribute to the perceived value, as reflected, for example, by system functional power or performance. What it achieves is to prevent system decline. If this trend is not controlled, the cost and fault proneness of system evolution will grow; will ultimately constrain system evolution and, in a continually changing world, reduce its value or even render it valueless. This class of activity was termed *anti-regressive*. All effort that compensates for *ageing* effects is included in this class. Such work consumes effort without *immediate* visible stakeholder return. What it achieves is to facilitate continued evolution, more easily, more quickly, more reliably and with less effort. It preserves the opportunities for future growth in value.

7 A Model and its Use[3]

The *system dynamics* [for61,80,abd91,coy96,] model presented here has been inspired by the laws of software evolution [leh74,85,feast], fieldwork with FEAST/2 collaborators and a study of how others approached the development of models of the software process.

Originally inspired in the context of mathematical system theory, *system dynamics* (SD) [for61,coy96] and tools such as *Vensim*® [ven95] that implement and support it, was developed to study the behaviour over time (dynamics) of industrial and managerial systems. Its vocabulary, involving terms such as *levels* (or *stocks*), and *flow* (or *rate*) variables, was inspired by hydraulic systems that appeared to offer intuitive appeal. Guidelines for reinterpretation in other domains may be found in [e.g. for61,coy96].

SD's mathematical formalism is that of differential equations. An SD model is essentially a set of non-linear first-order differential equations:

$$d\mathbf{x}(t)/dt = f(\mathbf{x}(t),\mathbf{p})$$

where t represents the real-time variable, $\mathbf{x}(t)$ is a vector of *levels,* $\mathbf{p}$ a vector of parameters and f() is a non-linear vector-valued function. It is particularly powerful for the representation and simulation of

[3] For a more detailed description of the model see [kah00].

systems involving feedback loops and mechanisms and in that context makes heavy use of numerical methods for the integration of differential equations. In the context of systems dynamics the latter are derived from system visualisations as represented in the system dynamics formalism.

At first sight the underlying formal mathematical models would seem inappropriate in the software engineering context. As illustrated by the example that follows the results obtained so far in FEAST [feast], provides a degree of evidence that the approach is useful. It suggests that as a multi-loop, multi-level, multi-agent feedback system [leh94], the long term, global, behaviour of the global software process is primarily determined by its feedback nature, and by *implied* equations as defined by the visualisations. The model, and language used to represent it, constitute a formalism.

The semantics and syntax of system dynamic models and the procedures to build and validate them have been described in many references [e.g. coy96,ven95]. Two different representations are generally used: *influence* and *level-rate* diagrams.

The structure of the behavioural relationships within a software evolution process can be sketched using *influence diagrams* [coy96]. Influences between any two attributes can be either balancing as for negative feedback or reinforcing as for positive feedback. An influence diagram presents the attributes of interest (in pictures and/or text). Arrows represent influences. A "+" character close to the arrow indicates a positive influence, such as "...the higher the variable at the arrow's origin, the higher the attribute at the arrow's end...". A "-" character indicates the opposite influence. This represents a simple, but effective, view of expected relationships.

The influence diagram in figure 1 constitutes a simplified view of the model to be discussed and the influences it encompasses. In the figure, arrows with solid shafts indicate relationships that are definitively positive or negative. Arrows with dashed shafts indicate influences that, under some circumstances may be positive, under others, negative.

Fig. 2 is a *level-rate diagram* representing the full model as developed using the *Vensim*® tool [ven95].

The variables in the boxes represent *levels* or *stocks*. The variables on the valve icons represent *flows* or *rates*. The variables in circles are auxiliary variables. The remaining variables are model parameters. Arrows with double lines represent flows of information or material that are conserved throughout the execution of the model. Clouds represent either sources or sinks of information or material. The Appendix presents the model of figure 2 in the *Vensim*® tool's language [ven65].

This relatively simple model represents the process at a high-level of abstraction, enabling the *global* nature and influences on the process [leh94] to be more easily understood. It is intended to provide a tool for use in the context of planning and management of software evolution. It relates specifically to demonstrating the influence of the progressive to anti-regressive effort ratio on the long term growth rate using the model as in figure 3 as an executing process simulation. A detailed discussion of the plots is not appropriate here, and the plots are presented as results typical of what one would expect when studying process dynamic behaviour.

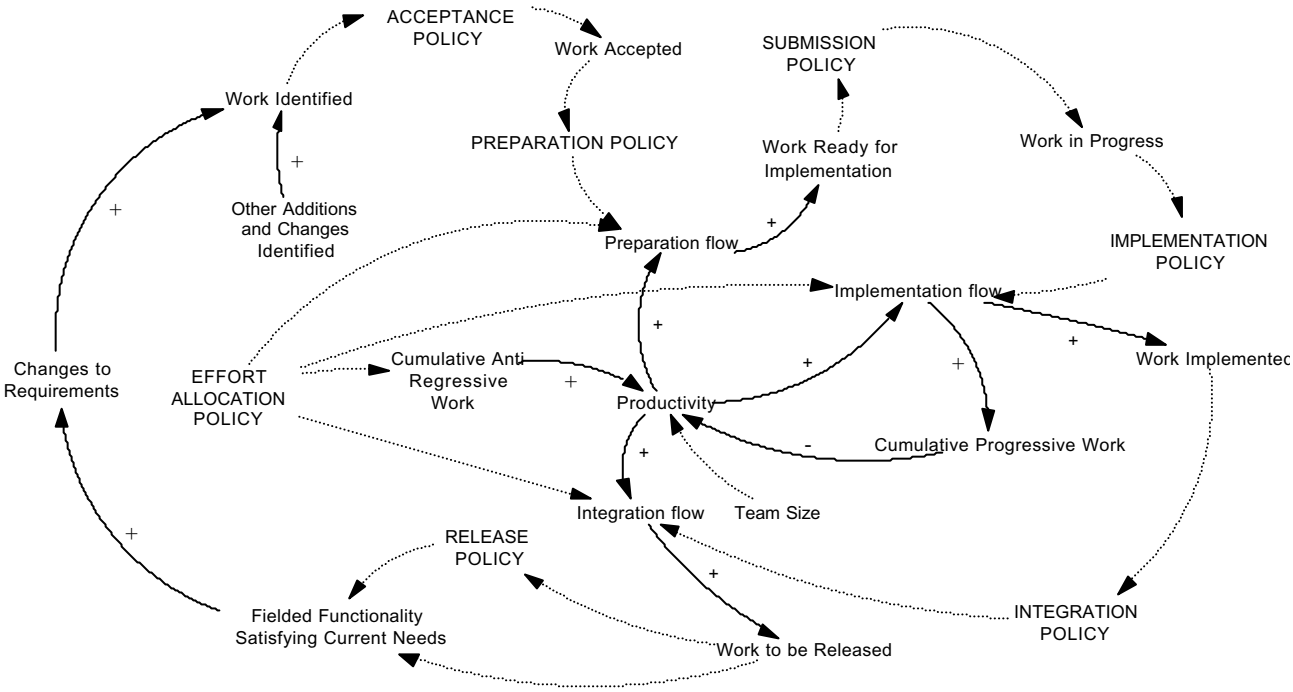


Figure 1: Influence diagram of an ideal software evolution process

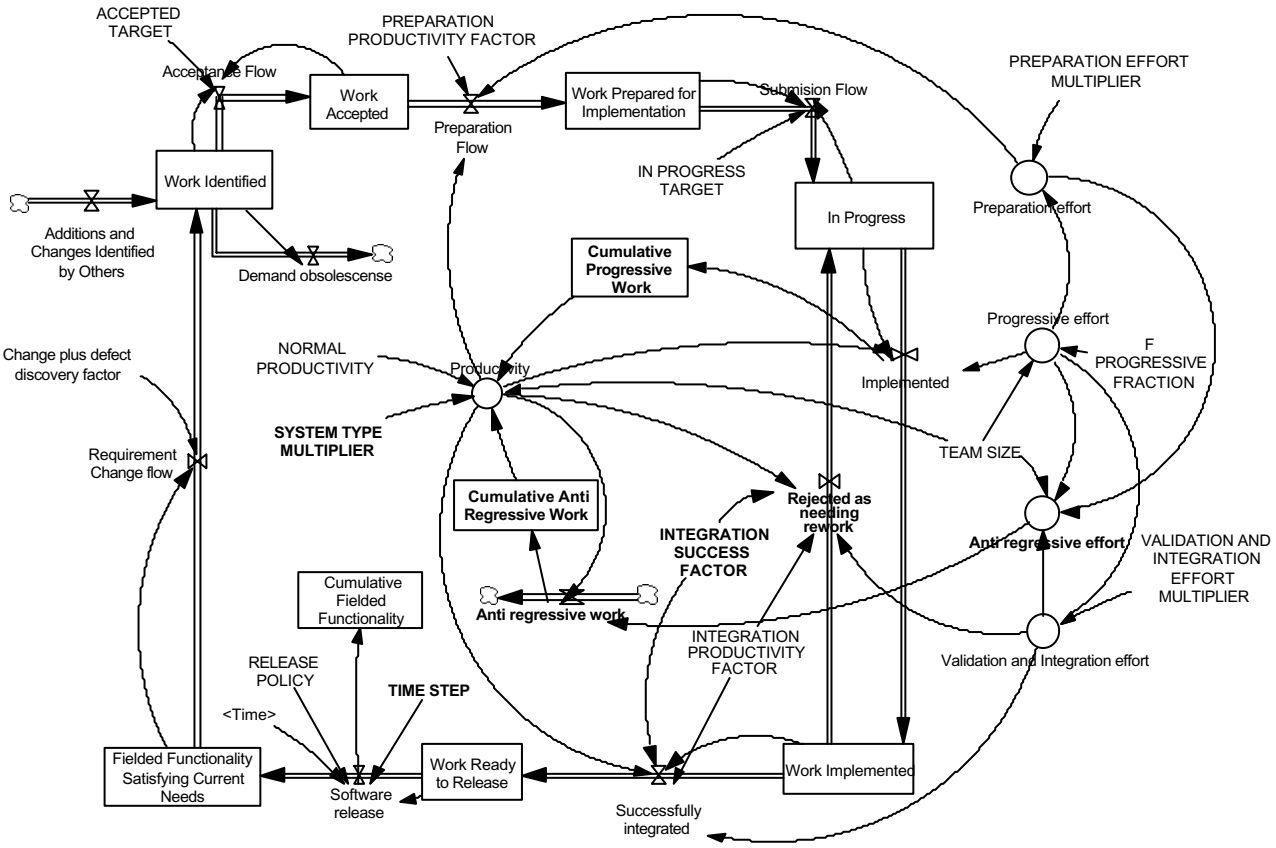


Figure 2: Detailed view of the system dynamics model
(no meaning is here to be attached to arrow line thickness)

The plots in figure 3 represent the effects of three different policies that address in particular the level of effort assigned to anti regressive work. Resource available is kept constant throughout. Three policies, AR60, AR40, AR0, are compared. They correspond to the application of 60, 40 and 0 percent of the available effort to anti regressive work. This, in turn, corresponds to 20, 30 and 50 percent, respectively, of the effort available to progressive work as reflected by the variable *F Progressive Fraction.* Remaining resources are shared by the two other activities, *Preparation* and *Validation and Integration.* Simulation results (see fig. 3) show that AR40 leads to higher *Cumulative Fielded Functionality*, than either AR60 or AR0. The accompanying behaviour of other model variables is also presented in the plots. The model offers the basis for other policy analyses relevant, in this instance, to *release planning* [e.g. leh00a]. Moreover, the variables in this model provide a set of attributes that are more generally useful in monitoring and planning evolution process performance.

This workshop, with its focus on formal foundations of software evolution is not the appropriate occasion to enlarge further on the model or to discuss what else may be learned from it with regards to the software process and its products. The brief discussion presented and the principles underlying its development are simply intended to demonstrate the relevance and application of formal methods *in the wider sense*. In this instance the discussion has focussed on the study of the *what*/*why* of software evolution and their potential as tools for the planning and management of long-term evolution processes. Another is provided by the proposal to develop a formal theory of software evolution. The middle ground between purely prescriptive (normative) and behavioural process models remains unexplored. *Semi-normative* theories [col64] may prove to be a useful path to follow for further study of this topic.

8 Final Remarks

This paper suggests that formalisms may not only be relevant in the context of methods and tools to evolve software, that is, the realm of the *how* to achieve software evolution through software change, but also within the investigation of the *what* and *why* of the evolution process.

Our thesis has been that such formalisms, together with models implemented using them, may help in planning and management of long-term evolution. The latter if undertaken, would aim at achieving the above in a reliable, timely and cost-effective way. Its achievement, of course, involves many unsolved challenges. Continuing change and increasing system

complexity phenomena, the focus of the simulation model presented, is, however, only one of many influences determining behavioural attributes of long-term software evolution processes and products.

More generally, simulation models developed according to some rigorous discipline may be considered as a formalisation of the software process that provides means to analyse and reason about its behaviour. Other formalisms may be useful for reasoning about and justifying *good practice.* The latter will, we believe, be derivable as theorems from a theory of software evolution to be developed in a project, currently awaiting funding decision [leh00c,d]. That development will be seeded and driven by the behavioural invariants and empirical generalisations observed over the years in the FEAST [feast] and similar studies.

9 Acknowledgements

Financial support from the UK EPSRC, grant GR/M44101 (FEAST/2), is gratefully acknowledged.

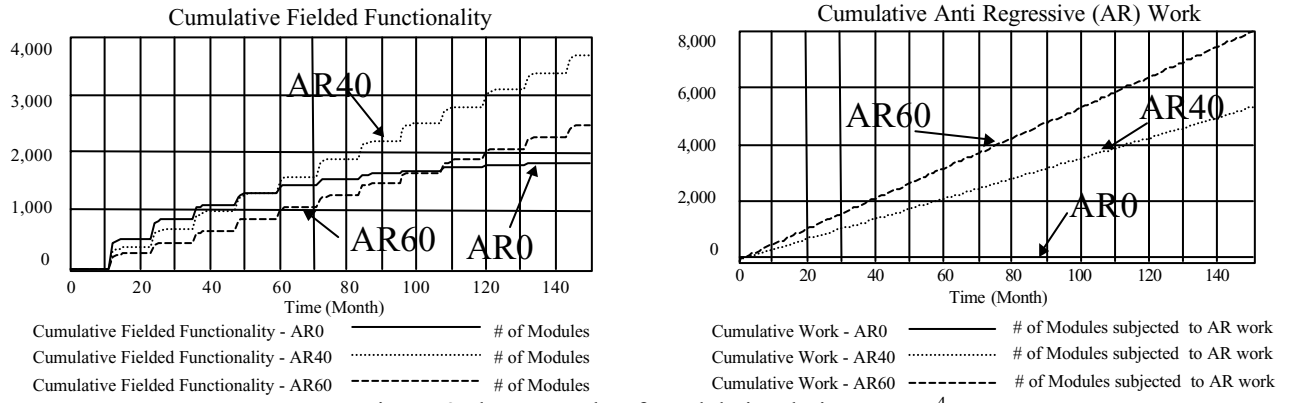


Figure 3a,b: Example of model simulation output[4]

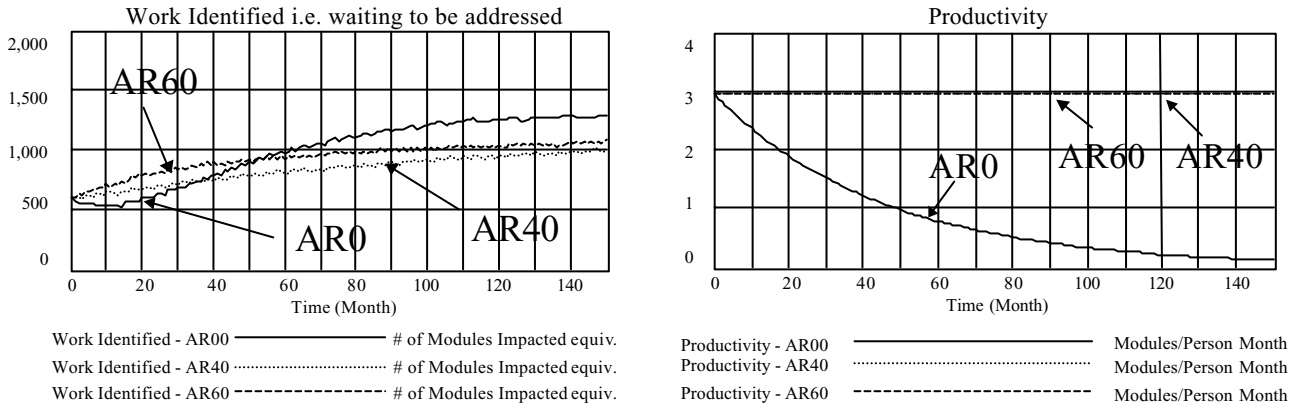


Figure 3c,d: Examples of model simulation output

10 References

- An * indicates that the reference has been reprinted in [leh85]. ** indicates that is available from links at http://www.doc.ic.ac.uk/~mml/feast

[abd91] Abdel-Hamid T and Madnick S, Software Project Dynamics - An Integrated Approach, Prentice-Hall, Englewood Cliffs, NJ.

[bau67] Baumol WJ, Macro-Economics of Unbalanced Growth - The Anatomy of Urban Cities, Am. Econ. Review, Jun 1967, pp. 415 - 426

[boe00] Boehm BW and Sullivan KJ, Software Economics: A Roadmap, in Finkelstein A (ed.), The Future of Softw. Eng., ICSE 22, pp. 321 - 343

[cha96] Chatfield C, The Analysis of Time Series - An Introduction, 5th Ed., Chapman & Hall, London, 1996

[col64] Coleman JS, Introduction to Mathematical Sociology, Collier-Macmillan Limited, London, 1964, 554 pp.

[coo00] Cook S *et al*, Software Evolution and Software Evolvability, working paper, U. of Reading, Aug. 2000

[coy96] Coyle RG, System Dynamics Modelling - A Practical Approach, Chapman & Hall London, 413 p

[feast] Feedback, Evolution And Software Technology, Project Web Site http://www.doc.ic.ac.uk/~mml/feast

[for61] Forrester JW, Industrial Dynamics, Cambridge, Mass.: MIT Press, 1961

[for80] Forrester JW and Senge P, Tests for Building Confidence in System Dynamics Models, In System Dynamics, Legasto AA Jr., Forrester JW and Lyneis JM (eds.), TIMS Studies in the Management Sciences, v. 14. North Holland, New York, 1980, pp. 209 - 228

[gdf00] Godfrey MW and Qiang T, Evolution in Open Source Software: A Case Study, Proc. ICSM 2000, 11-14 Oct. 2000, San Jose, CA, pp. 131-142

[har90] Harel D *et al*, STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. on Softw. Eng., v. 16, n. 4, Apr. 1990, pp. 403 - 414

[4] Module counts (e.g., # of files) exemplify one possible unit. *# of Modules subjected to Anti regressive work* may be too simplistic. In a practical case, other units must be used.

[kah00] Kahen G, Lehman MM and Ramil JF, System Dynamic Modelling for the Management of Software Evolution Processes, Research Report 2000/16, Dept. of Comp., Imp. Col., Nov. 2000

[kel99] Kellner MI, Madachy RJ and Raffo DM, Software Process Simulation Modelling: Why? What? How?, J. of Syst. and Software, v. 46, n. 2/3, April 1999, pp 91 - 106

[kem99] Kemerer C and Slaughter S, An Empirical Approach to Studying Software Evolution', IEEE Trans. on Softw. Engineering, v. 25, n. 4, July/Aug. 1999, pp. 493 - 509

[kus97] Kusumoto S *et al,* A New Software Project Simulator Based on Generalized Stochastic Petri-net, Proc. ICSE 19, Boston, May 17 - 23, 1997, pp. 293 - 302

[ispse98] ISPSE 98, International Workshop on the Principles of Softw. Evolution, 20-21 April 1998, co-located with ICSE 98, Kyoto, Japan

[ispse00] ISPSE 2000, International Symposium on Principles of Software Evolution, Kanazawa, Japan, Nov 1-2, 2000

[leh69]* Lehman MM, The Programming Process, IBM Res. Rep. RC 2722, IBM Res. Centre, Yorktown Heights, NY 10594, Sept. 1969

[leh74]* *id*, Programs, Cities, Students, Limits to Growth?, Inaugural Lecture, in Imp. Col. of Sc. and Techn. Inaug. Lect. Seri., v. 9, 1970 - 74, pp. 211 - 229. Also in Programming Methodology, Gries D. (ed.), Springer Verlag, 1978, pp. 42 – 62

[leh77]* *id*, Human Thought and Action as an Ingredient of System Behaviour. In Duncan R & Weston Smith M (eds.), Encyclopedia of Ignorance, Pergamon Press, Oxford, 1977

[leh78]* *id*, Laws of Program Evolution - Rules and Tools for Program Management'. Proc. Infotech State of the Art Conf., Why Software Projects Fail, - April 1978, 11/1-11/25

[leh80a]* *id*, On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle, J. of Sys. and Softw., 1980, 1, (3), pp. 213-221

[leh80b]* *id*, Programs, Life Cycles and Laws of Software Evolution, Proc. IEEE Spec. Iss. on Softw. Eng., 68, (9), Sept. 1980, pp. 1060-1076

[leh85] Lehman MM & Belady LA, Software Evolution - Principles of Software Change, Acad. Press, London, 1985

[leh89] Lehman MM, Uncertainty in Computer Application and its Control Through the Engineering of Software, J. of Softw. Maint.: Res. Pract., v. 1, n. 1, Sept. 1989, pp. 3-27

[leh90] Lehman MM, Uncertainty in Computer Application, Tech. Let., CACM, v. 33, n. 5, May 1990, pp. 584-586

[leh94] Lehman MM, Feedback in the Software Process, Keynote Address, CSR Eleventh Annual Wrksh. on Softw. Ev. - Models and Metrics. Dublin, 7-9th Sep. 1994. Also in Info. and Softw. Tech., spec. iss. on Softw. Maint., v. 38, n. 11, 1996, Elsevier, 1996, pp. 681 - 686

[leh98] Lehman MM, Perry DE and Ramil JF, On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution, Proc. Metrics'98, Bethesda, Maryland, 20-21 Nov. 1998, pp. 84 - 88

[leh00a]** Lehman MM, Rules and Tools for Software Evolution Planning and Management, pos. paper, FEAST 2000 Workshop, Imp. Col., 10 - 12 Jul. 2000, http://www.doc.ic.ac.uk/~mml/f2000

[leh00b]** Lehman MM, Ramil JF and Kahen G, Evolution as a Noun and Evolution as a Verb, SOCE 2000 Workshop on Software and Organisation Co-evolution, Imp. Col., London, 12-13 Jul. 2000

[leh00c] Lehman MM and Ramil JF, Towards a Theory of Software Evolution - And Its Practical Impact, invited talk, ISPSE 2000, Intl. Symp. on the Principles of Softw. Evolution, Kanazawa, Japan, Nov. 1-2, 2000

[leh00d] Lehman MM, TheSE - An Approach to a Theory of Software Evolution, proj. prop., DoC, Imp. Col., Dec. 2000

[mai00] Maibaum TSE, Mathematical Foundations of Software Engineering: a Roadmap, in A. Finkelstein (ed.), The Future of Software Engineering , ICSE 2000, June 4-11 Limerick, Ireland, pp. 161 - 172

[mcg97] McGrath GM, A Process Modelling Framework: Capturing Key Aspects of Organisational Behaviour, Proc. Australian Softw. Eng. Conf., 1997, pp. 118 - 126

[ost87] Osterweil L, Software Processes Are Software Too, Proc. 9th Int. Conf. on SEng., 1987, pp 2 - 12

[ost97] Osterweil L, Software Processes Are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9, Proc. 19th Int. Conf. on Software Engineering, May 17-23, 1987, Boston, MA, pp. 540 - 548

[pod97] Podoroznhy RM and Osterweil LJ, The Criticality of Modeling Formalisms in Software Design Method Comparison - Experience Report, ICSE'97, May 17-23 1997, Boston MA, pp. 303-313

[prosim00] Prosim 2000, Workshop on Software Process Simulation and Modelling, 12-14 July, 2000, Imp. Col., London, http://www.prosim.org

[raj00] Rajlich VT and Bennett KH, A Staged Model for the Software Life Cycle, Computer, Jul.2000,pp.66-71

[ram98]** Ramil JF and Lehman MM, Fuzzy Dynamics in Software Project Simulation and Support, Proc. 6th European Workshop on Softw. Process Technology (EWSPT-6), 16-18 Sept. 1998, Weybridge, UK, LNCS 1487, Springer-Verlag, pp. 122-126

[ven95] Vensim - Ventana Simulation Environment, Reference Manual, Version 1.62, Belmont, MA., 1995

[wir00] Wirtz G, Using a Visual Software Engineering Language for Specifying and Analysing Workflows, IEEE International Symp.on Visual Languages 2000, pp. 97 - 98

Appendix - The Model in Vensim [ven95]

```
Acceptance Flow =
IF THEN ELSE((Work Accepted<ACCEPTED TARGET)
:AND:(Work Identified>0),300,0)
    ~
    ~    |
ACCEPTED TARGET = 100
    ~
    ~    |
Additions and Changes Identified by Others = (RANDOM
POISSON(60))
    ~ Changes/Month
    ~    |
Anti regressive effort =IF THEN ELSE (
(TEAM SIZE-Preparation effort
-Progressive effort-Validation and Integration effort)>0,
(TEAM SIZE-Preparation effort-Progressive effort
-Validation and Integration effort),0)
    ~
    ~    |
Anti regressive work= Anti regressive effort * Productivity
    ~
    ~    |
Change plus defect discovery factor = 1/120
    ~
    ~    |
Cumulative Anti Regressive Work =INTEG(Anti regressive
work,0)
    ~
    ~    |
Cumulative Fielded Functionality = INTEG(Software release,0)
    ~
    ~    |
Cumulative Progressive Work  = INTEG(Implemented,0)
    ~
    ~    |
Demand obsolescense = Work Identified * 0.05
    ~ Changes/Month
```

```
    ~     |
F PROGRESSIVE FRACTION = 0.3
    ~
    ~     |
Fielded Functionality Satisfying Current Needs =
INTEG(Software release-Requirement Change flow,150)
    ~  [0,?]
    ~     |
Implemented =
IF THEN ELSE(In Progress > 0,
Progressive effort*Productivity,0)
    ~
    ~     |
In Progress =  INTEG(Submision Flow-Implemented+Rejected
as needing rework,100)
    ~  [0,?]
    ~     |
IN PROGRESS TARGET  =100
    ~
    ~     |
INTEGRATION PRODUCTIVITY FACTOR = 2
    ~
    ~     |
INTEGRATION SUCCESS FACTOR = 0.95
    ~
    ~     |
NORMAL PRODUCTIVITY = 2
    ~ Modules/Person Month
    ~     |
Preparation effort =Progressive effort *PREPARATION
EFFORT MULTIPLIER
    ~
    ~     |
PREPARATION EFFORT MULTIPLIER = 0.5
    ~
    ~     |
Preparation Flow = Productivity*Preparation effort*
PREPARATION PRODUCTIVITY FACTOR
    ~
    ~     |
PREPARATION PRODUCTIVITY FACTOR = 2
    ~
    ~     |
Productivity =NORMAL PRODUCTIVITY*
(  (TEAM SIZE^0.2) - (  (1/1800) *TEAM SIZE^2)   ) *
(1-MAX(0,SYSTEM TYPE MULTIPLIER*
(Cumulative Progressive Work - Cumulative Anti Regressive
Work) ))
    ~ Modules/Person Month
    ~     |
Progressive effort = F PROGRESSIVE FRACTION*
TEAM SIZE
    ~
    ~     |
Rejected as needing rework  =
Productivity*Validation and Integration effort*
(1-INTEGRATION SUCCESS FACTOR)*
INTEGRATION PRODUCTIVITY FACTOR
    ~
    ~     |
RELEASE POLICY  ([(0,0)-(100,10)],(0,0),
(11,0),(12,1),(14,1),(15,0),
(23,0),(24,1),(26,1),(27,0),
(35,0),(36,1),(38,1),(39,0),
(47,0),(48,1),(50,1),(51,0),
(59,0),(60,1),(62,1),(63,0),
(71,0),(72,1),(74,1),(75,0),
(83,0),(84,1),(86,1),(87,0),
(95,0),(96,1),(98,1),(99,0),
(107,0),(108,1),(110,1),(111,0),
(119,0),(120,1),(122,1),(123,0),
(131,0),(132,1),(134,1),(135,0),
(143,0),(144,1),(146,1),(147,0))
    ~
    ~     |
Requirement Change flow =Fielded Functionality Satisfying
Current Needs*
Change plus defect discovery factor
    ~ Changes/Month
    ~     |
Software release =MAX(0,(Work Ready to Release/TIME
STEP)*
LOOKUP EXTRAPOLATE(RELEASE POLICY, Time))
    ~
    ~     |
Submision Flow = IF THEN ELSE((In Progress<IN
PROGRESS TARGET)
:AND:(Work Prepared for Implementation > 0),300,0)
    ~
    ~     |
Successfully integrated = IF THEN ELSE
(Work Implemented > 0,
Validation and Integration effort*Productivity*
INTEGRATION SUCCESS FACTOR * INTEGRATION
PRODUCTIVITY FACTOR,0)
    ~
    ~     |
SYSTEM TYPE MULTIPLIER =0.0005
    ~
    ~     |
TEAM SIZE = 30
    ~
    ~     |
TIME STEP = 0.125
    ~
    ~     |
Validation and Integration effort = Progressive effort *
VALIDATION AND INTEGRATION EFFORT
MULTIPLIER
    ~
    ~     |
VALIDATION AND INTEGRATION EFFORT
MULTIPLIER =  0.5
    ~
    ~     |
Work Accepted = INTEG(Acceptance Flow-Preparation
Flow,100)
    ~
    ~     |
Work Identified= INTEG(Additions and Changes Identified by
Others +Requirement Change flow-Demand obsolescense-
Acceptance Flow,600)
    ~ Changes
    ~     |
Work Implemented = INTEG(Implemented-Successfully
integrated-Rejected as needing rework,200)
    ~ Changes
    ~     |
Work Prepared for Implementation = INTEG(Preparation Flow-
Submision Flow,100)
    ~
    ~     |
Work Ready to Release =  INTEG(Successfully integrated-
Software release,0)
    ~  [0,?]
    ~     |
**************************
    .Control
**************************~
Simulation Control Paramaters
    |
FINAL TIME  = 100
    ~ Month
    ~ The final time for the simulation.
    |
INITIAL TIME  = 0
    ~ Month
    ~ The initial time for the simulation.
    |
SAVEPER  = 1
    ~ Month
    ~ The frequency with which output is stored.
    |
```

Software Evolution based on Formalized Abstraction Hierarchy

Timo Aaltonen
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere, Finland
Timo.Aaltonen@tut.fi

Tommi Mikkonen
Nokia Mobile Phones
P.O.Box 68
FIN-33721 Tampere, Finland
Tommi.Mikkonen@nokia.com

Abstract

Software evolution is about visions and abstractions. The success in finding the right visions, i.e., directions of future evolution, and abstractions, i.e., concepts by which the system is understood, provide a good starting point for the evolution of a software system. In contrast, a failure makes the system practically unevolvable. Unfortunately, there is no universally accepted set of visions or abstractions that could be applied in all systems. Instead, it is up to the developers to find and document them in particular domains. Then, criteria are needed for determining the quality of interconnected abstractions and visions. This can be achieved by modeling the abstractions incorporated in the system as a hierarchy, where abstraction levels exceeding that of implementation facilities are used. The hierarchy can then be used for examining new visions and requirements that emerge over time as well as for supporting associated modifications. This paper introduces an approach where formalism is used for deriving the hierarchy, and provides an example on the evolution of abstractions.

1. Introduction

While software evolution can be considered as a force of nature, its sophisticated management is a necessity for the maintenance of complex systems. In engineering, this leads to a situation where discussions can be raised, to what extent systems built in an evolutionary fashion are different from systems developed from scratch. The answer to this question, however, lies outside the scope of actual systems themselves. Instead, we need to look at models of the systems, the abstractions needed for comprehending and developing them, and visions and expectations that we have on their future.

Based on artifacts without direct physical qualities, software as such is immensely flexible by its nature. Therefore, it is possible to define any system using any kind of partitioning into modules[1]. However, as pointed out in a classical paper by Parnas[16], some modules are more favorable than others. The design of modules, their enforcement in implementations, and the underlying rationale for selecting the modules provide a basis for arguing about qualities of different implementations. This is emphasized in practices like software architecture reviews [3], where the views of different stakeholders form the basis of evaluation. In creating the views, it is possible to use different models to highlight the important aspects, like modifiability of some parts of the system or implementability of some future visions.

In the technical sense, there is practically only one way to emphasize anything in software: To design a module dedicated to that particular issue. This, however, is not always an optimal solution, as components are also subjected to other concerns. They should also be units of compilation, reflect available effective implementation technologies, and, due to the introduction of recent practices, like design patterns [6], reuse acknowledged good design decisions. In addition to the above technical hinges, human capabilities also play a big role in decomposition. People need to be assigned the responsibility of the development and maintenance of components. Tackling all the above issues simultaneously with one architecture requires delicate architecting and excellent engineering at best, and is impossible at worst.

The biggest enemy of software evolution is increasing complexity. In coping with complexity, the most effective weapon is abstraction. In an ideal world, abstractions would always be incorporated in individual modules, and, furthermore, obey available implementation interfaces. In reality, however, abstractions related to conceptual properties often extend from one module to another. This is evident in design patterns [6], and in the use of centralized state of a distributed system [2], for instance.

The clarity of concepts in an implementation architecture also enables the determination of whether a modification af-

[1] For the purposes of this paper, we will treat packages, components, processes, etc. as modules without any exact definition.

fects the system in a fundamental fashion, or is a minor update that leads to minimal redesign. Therefore, mastering and maintaining the abstractions and their relations to the modules of the system is a key issue for preserving the clarity of concepts. This often means denial of the temptation to extend abstractions or related software modules with some application-specific additions. The temptation is increased by the fact that the actual part where the change belongs may not be clear in a legacy system. Then, it is easiest to implement the new function in the scope of familiar code instead of studying all possible alternatives. Furthermore, in the short run, a straightforward implementation can be much more effective than a laborious process of identifying all the alternatives and selecting the most justified one. However, giving in to the temptation leads to difficulties in the long run.

Conventional software engineering approaches provide little support for determining whether a change is a minor one, or such that major reengineering of key abstractions is required. Based on the above discussion, such suppport is a necessity for introducing a robust framework for software evolution. The rest of this paper addresses this issue as follows. Section 2 discusses the notion of an abstraction hierarchy, which aims at relating the different abstractions needed for comprehending the system. The section also introduces the notion of an abstraction hierarchy that can be used for evaluating different design decision. Section 3 discusses maintenance based on abstraction hierarchy. Section 4 provides a discussion on abstractions incorporated in a mobile switch and their relation to actual implementation. Finally, Section 5 concludes the paper.

2. Towards an Abstraction Hierarchy

Software engineering abstractions are two-fold. Some of the abstractions are such that they directly reflect available implementation facilities, whereas some others exceed limitations of direct implementation concerns. We will refer to these categories as *primitive* and *non-primitive* abstractions, respectively.

Primitive abstractions are straightforward to describe. They are what we think about when considering software. They represent conventional components or software modules that can be compiled into executables with available tools, or run with interpreters or virtual machines. However, straightforward use of primitive abstractions has been found to harden rather than simplify rigorous reasoning [13]. Therefore, while needed for effective implementations, the role of primitive abstractions is not to ease reasoning about the system as a whole.

Non-primitive abstractions, in contrast, are difficult to describe in terms of conventionally used software artifacts. They represent cross-cutting concerns that cannot be located in one module [9], provide a design step that has been acknowledged as universally favorable [6], or model collective state distributed in multiple implementation components [8], for instance. The special role of such abstractions has also been pointed out in [4], where patterns are advocated as something that extend over objects and tie them together. As these examples make obvious, there are several levels of non-primitive abstractions already in the approaches that are already available. For instance, aspect-oriented programming relies on implementation level sequences of program code, whereas design patterns are intended to be used as design guidelines.

Based on the above discussion, completed systems potentially include several levels of non-primitive abstractions. Therefore, formalizations of such systems require semantically sound and practically manageable representation of collaborative properties [11]. The DISCO method [7, 5] enables addressing of such abstractions without being tied to individual implementation techniques. DISCO is a formal method, whose semantics are in the temporal logic of actions [12], a state-based formalism. In addition to well-defined semantics, the DISCO method introduces stepwise specification capabilities as a methodological guideline. Each step forms a *layer* in the complete system, where state variables as well as actions modifying the values of the layer's variables are given. For the purposes of this paper, a simple layer can be given, for instance, as follows:

```
layer L = {
  class C = {b : boolean};
  action A(c1, c2: C): c1.b ≠ c2.b →
                       c1.b' = c2.b ∧
                       c2.b' = c1.b;
} -- layer L
```

Layer L introduces class C, which has one attribute b of type boolean. Moreover, the layer has one action: A, in which two objects of the class C can participate. The action can be executed for such objects, which have different values in their attributes b. In the body of the action the participating objects swap their values of attribute b.

Layers can also refer to contents of other layers by importing them. The following example depicts this:

```
layer LL = {
  import L;
  class C = L.C + {i : integer};
  invariant I = ∀ c: C :: ∃ i: integer :: i < c.i;
  action A(c1,c2:C) refines L.A(c1, c2) →
                            c1.i' = c2.i ∧
                            c2.i' = c1.i;
} -- layer LL
```

The capabilities of the DISCO method can be used in a fashion where abstractions are mapped to their implementations with invariants that uniquely determine the values of more abstract variables. The scheme can then be used so that abstract versions of specifications refer to abstract concepts. Then, these concepts can be refined towards an implementation by introducing lower-level abstractions, and

by proving the associated invariant. For more details regarding the refinement scheme incorporated in the formalism, the user is referred to [10].

When the above procedure is applied in a recursive fashion, a hierarchy of abstractions is obtained [14]. Each level of the hierarchy describes the system with its own concepts. These concepts can be mapped to more concrete ones when advancing towards implementation, or traced back to higher-level concepts where more abstract descriptions of the system can be found. The top level of the hierarchy is the most abstract description of the system where everything is possible. In this paper, we will refer to this specification as ϵ. The lowest level refers to actual code modules.

By establishing an abstraction hierarchy, it becomes possible to measure the relative complexity of the implementation with respect to its abstract specification. For the purpose of software evolution, this is a key concept to manage the direction the implementation is heading. The divergence of actual code modules from the abstractions included in the hierarchy provides evidence on potential future problems for future evolution.

A primitive abstraction hierarchy where all abstractions follow intermodule interfaces is a layered architecture. For instance, a file is an abstract concept that often has a layered implementation. We, however, allow abstractions as an auxiliary concept that can be used to support software evolution and the creation of related visions.

3. Maintenance based on Abstraction Hierarchy

When an abstraction hierarchy has been established, it provides a reference for any new features of the system. When a new requirement emerges, it can be related with the abstractions already incorporated in the system in terms of the hierarchy. Further, based on the level of abstraction in the hierarchy, the relative cost for implementing the new requirement can be justified due to the following. When a change is required at a very abstract level, it is likely that many implementation modules require changes, because the cross-cutting of the abstraction is large. On the other hand, if a change is related to a low-level abstraction only, it is likely that required modifications can be handled locally within the scope of that particular abstraction. In fact, at the level of primitive abstractions, interfaces can remain unchanged provided that the design of the abstraction has been appropriate. Obviously, based on the information obtained from the hierarchy, the designer can analyse different implementation alternatives, and their related effect in coding, testing, and integration.

For more details on the management of evolution, consider the following. Whenever a new requirement is identified, it is associated with a certain abstraction in the hierarchy by analysing the effects of the change. The lowest-level abstraction that will remain unchanged will be referred to *stable root*. This abstraction, all the abstractions above this level, and abstractions that are independent of stable root remain unchanged. In contrast, abstractions that are needed for deriving stable root into more concrete form potentially need to be reengineered. In order to identify the needed changes, the layers below stable root specification need to be analysed with respect to the new requirements. Then, the lower-level abstractions are modified to support the higher-level upgrades recursively. In reality, new layers are often required, or at least provide a justifiable way to specify the newly emerged properties.

In addition to the use of stable root as an indicator for changes in the specification level, verification and validation effort can be focused. As we know that only abstractions below stable root are modified, it is enough to re-test abstractions below the root. In reality, however, it is often desirable to run e.g. old test cases as a regression test to validate the preservation of unchanged abstractions. Still, even this case is made easier because we know that the test outcomes should remain unchanged, resulting in straightforward automatic analysis of test results.

Ideally (and also usually in practice) the top levels of the abstraction hierarchy experience little or no evolution. In contrast, towards the lower levels of abstraction, more and more changes occur. This reflects the intuitive assumption that maintenance is not risking the fundamental concepts of the system, but extends implementation with new details thus enhancing the system.

Based on the above discussion, the abstraction hierarchy supports separation of implementation details and high-level abstractions reflecting fundamental concepts. This is crucial for software evolution. Without such separation, it is difficult to justify the decisions taken to manage evolution except as a reflection of resulting implementation architecture. Then, evolution is effectively code manipulation with little possibilities for fact-based management of main concepts.

4. Example: Abstractions in a Mobile Switch

As an example we give an abstraction hierarchy for a mobile switch, and show how new properties could be attached to the specification. The switch routes calls[2] from callers to callees. In some cases a call is first routed to one subscriber and then forwarded to some other. The example is a simplified version of more comprehensive work carried out in DISCO project, where selected parts of an existing mobile switch were modeled.

[2]We do not give exact meaning to the notion *call*, which is perhaps the most intuitive starting point in modeling a switches. However, starting with call leads to difficulties, as pointed out by Zave in [17].

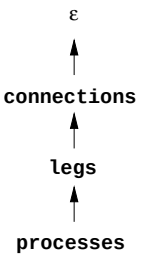


Figure 1. Abstraction hierarchy.

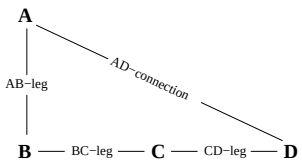


Figure 2. Abstractions connection and leg.

4.1. Deriving an Abstraction Hierarchy

The abstraction hierarchy derived in this subsection are *connection*, *leg* and *process*. The hierarchy is depicted in Figure 1. Abstractions are discussed in detail in the following.

The highest abstraction in the hierarchy is *connection*. Informally a caller has connection to the subscriber (callee) to whom a call has been routed. For example, if subscriber A calls B then after successful routing AB-connection is created. If the call is then forwarded to some other subscriber (C), AB-connection is replaced by AC-connection. And if it is again routed to D, AC-connection is replaced with AD-connection.

Formally connections are DISCO objects (introduced in layer `connections`) which have state machines with three states: `unborn`, `active` and `terminated`. Two variables (`from` and `to`) referring to subscribers are embedded in state `active`:

```
class Connection = {
  state = (unborn,
           active(from, to : reference Subscriber),
           terminated)}
```

In this layer three actions are introduced for changing the states of connections: `connect`, `redirect` and `disconnect`. Only action `redirect` is given as an example:

```
action redirect(to: reference Subscriber;
                c: Connection):
  c.state = active →
  c.active.to' = to;
```

Connections are implemented with *leg*s, which form chains from subscriber to subscriber. In our earlier example where A's call was first routed to B and then to C and finally to D there are three legs: AB- BC- and CD-leg, which all together implement an AD-connection (see Figure 2).

Formally legs are DISCO objects given in layer `legs`, which imports the layer `connections`.

```
class Leg = {
  state = (unborn,
           active(a, b : reference Subscriber;
                  next, prev: reference Leg)
           terminated)}
```

Variables `a` and `b` are references to the subscribers related by the `leg`; `next` and `prev` are used to form linked lists of legs.

In addition to plain `Legs`, the layer introduces relation `isPartOf` between legs and connections, which states that there is an arbitrary number of legs for each connection, and either no or one connection for each leg.

```
relation isPartOf(Leg, Connection) is
  0..*:0..1;
```

Invariant `legChainImplConnection`[3] relates connections and legs. Intuitively it states that if there is an active connection between two subscribers, then a chain of active legs (implementing the connection) exists between the two subscribers (as is the case in figure 2).

```
invariant legChainImplConnection =
  ∀ c: Connection | c.state.active ::
    ∃ first, last: Leg |
      isFirstLegInChain(first) ∧
      isLastLegInChain(last) ∧
      areMembersOfTheSameLeg(first, last) ::
        first.state.active.a = c.state.active.from ∧
        (first, c) ∈ isPartOf ∧
        last.state.active.b = c.state.active.to ∧
        (last, c) ∈ isPartOf;
```

The layer has five actions: `startLeg`, `addLeg`, `startTearingDown` and two actions for tearing down a chain of legs. Action `addLeg` is given below as an example. The action is a refinement of the action `redirect` in the layer `connections`. It states that if there is a chain of legs ending in subscriber `sa`, then a new leg can be set from `sa` to any subscriber `sb` (and `connection c`, which is partly implemented by the legs `lPrev` and `lNext`, is atomically redirected to `sb`).

```
action addLeg(sa, sb: Subscriber;
              c: Connection;
              lPrev, lNext: Leg)
refines connections.redirect(sb, c) ∧
  lPrev.state.active.b = sa ∧
  isLastLegInChain(lPrev) ∧
  (lPrev, c) ∈ isPartOf ∧
  lNext.state.unborn →
  lNext.state' = active(a'=sa, b'=sb, next'=null) ||
  lPrev.state.active.next' = lNext ||
  isPartOf' = isPartOf + {(l, c)};
end;
```

[3] Moreover, the layer has three more invariants stating that there is one `Connection` for each `active Leg`, and there exists at least one `Leg` for each `active Connection`, and that consecutive `Legs` are implementing the same `Connection`. These are omitted here for brevity.

The next step in the abstraction hierarchy is this layer `processes`, where legs are implemented with processes. The layer is omitted here.

4.2. Evolution

Having the three-level abstraction hierarchy described above enables us to measure how big is the cross-cutting of our visions of changes to the system. If, for example, the change is such that a connection is the stable root, we can conclude that the change is relatively large (or our original understanding of the system was poor). On the other hand, if the change affects only the process level (connection and leg remain unchanged) then it is minor upgrade. In the following, we give some examples on how to manage software evolution with the abstraction hierarchy established above.

Difficult Modification: Eavesdropping

An example of a difficult evolutionary step is adding eavesdropping to the mobile switch. In some countries the government requires that there must be a possibility for legal authorities to listen calls of suspicious customers. In our abstraction hierarchy the stable root is an empty specification ϵ above `connection` abstraction. In other words, all layers require modification.

The modifications are as follows. In layer `connections`, we must add reference to possible eavesdropper to the state `active`:

```
class Connection = {
  state = (unborn,
           active(from, to: reference Subscriber;
                  eavesdropper: reference Subscriber),
           terminated)}
```

After this we must investigate and possibly reengineer the actions handling connections. For example, in action `redirect` we must take care that attribute `eavesdropper` is updated properly[4]. If the callee to whom the call is rerouted is suspicious then the eavesdropper starts to listen the call, else the eavesdropping status remains as it was before the action:

```
action redirect(to: reference Subscriber;
                c: Connection):
  c.state = active →
  c.active.to' = to ∧
  c.active.eavesdropper' = if isSuspicious(to) then
                             theEavesdropper
                           else
                             c.active.eavesdropper;
```

Changes to layer `legs` are similar. We must add new attribute (`eavesdropper`) to class `Leg` and reengineer actions referring to leg objects. Moreover, invariant `legChainImplConnection` must be revisited. Changes to layer `Processes` are omitted here for brevity.

[4]For simplicity we have only one legal authority carrying out eavesdropping.

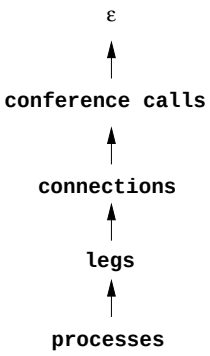


Figure 3. Upgraded abstraction hierarchy.

Related verification and validation activities also require major attention. In fact, all test cases should be rerun as such (regression tests) and in a fashion where eavesdropping is active. In reality, this degree of testing for eavesdropping only is of course unrealistic.

In order to add eavesdropping to the specification we made changes to existing abstractions. This is not the case always, it is also possible to add totally new abstractions for the system. For instance, conference call would be a totally new abstraction for our example. A normal call could then be derived from conference calls by limiting the number of participants to two. The upgraded hierarchy is illustrated in Figure 3. Obviously, the changes related to this upgrade as well as related validation and verification effort can be estimated to be considerable.

Simple Modification: Knocking

Example of a minor cross-cutting is *knocking*. If subscriber `a` is speaking on a phone with `b` and she is called by a third subscriber `c` then `a` hears a voice of knocking in her phone and can answer that call. In this case the stable root is the layer `Legs` because only the layer `Processes` is changed.

Obviously, verification and validation effort implied by this modification is also minimal.

5. Conclusions

We have presented an approach to handling a hierarchy of non-primitive abstractions to ease software evolution. The main contribution of the paper is in showing that such hierarchies can be rigorous. Moreover, we have outlined an example of using abstraction hierarchies in a mobile switch, and showed how this makes software evolution more manageable. The example was a simplified version of a more comprehensive case study carried out during DISCO project.

A similar approach has already been introduced in [15], although in an informal setting. In that context, the relation between higher-level abstractions and their implementations is handled with links of a browser tool and the underlying data base. This practical example also supports our claim that lower levels of abstraction evolve more than higher abstractions. While the demonstration in that context provides justification on industrial applicability of this approach. The introduction of the related formalism in this paper is an obvious improvement in the theoretical sense. In practice, this also results in the option to use the tools associated with the formalism [1].

In real life software engineering, the approach requires more work in short turn. We must investigate the effect of evolution to the specification, and reflect the changes to implementation level via the abstraction hierarchy. However, more comprehensive understanding of changes, and related documentation in the specification, compensates this in the long run.

Acknowledgements

This work is partly funded by the Academy of Finland (No. 100005).

References

[1] T. Aaltonen, M. Katara, and R. Pitkänen. DisCo Toolset – The New Generation. *Journal of Universal Computer Science*, Vol 7:1, 3–18, Springer-Verlag, 2001.

[2] M. Awad, and J. Ziegler. A practical approach to object-oriented state modeling. *Software - Practice and Experience*, 27(3):311–328, March 1997.

[3] G. Abowd, L. Bass, P. Clements, R.Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Software Engineering Institute, Technical Report CMU/SEI-96-TR-025, January 1997.

[4] J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, 14(1):36-42, January 1997.

[5] DisCo homepage. `http://disco.cs.tut.fi/`.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns*. Addison Wesley, Reading, MA, 1995.

[7] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71, 1990.

[8] P. Kellomäki and T. Mikkonen. Modeling distributed state as an abstract object. In *Distributed and Parallel Embedded Systems, proceedings of the IFIP WG 10.3 / WG 10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, pages 223–230. Kluwer Academic Publishers, 1999.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (Eds. M. Aksit and S. Matsuoka)*, pages 220–242, Springer-Verlag LNCS 1241, 1997.

[10] R. Kurki-Suonio and T. Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, 1998.

[11] R. Kurki-Suonio and T. Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modeling and Knowledge Bases X*, pages 1–11, IOS Press, 1999.

[12] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] L. Lamport. Composition: A way to make proofs harder. Digital Systems Research Center, Technical Note 1997-030a, December 1997.

[14] T. Mikkonen and H.-M. Järvinen. Specifying for releases. *International Workshop on Principles of Software Evolution*, pages 118–122. April 20–21, Kyoto, Japan, 1998.

[15] T. Mikkonen, E. Lähde, M. Siiskonen, and J. Niemi. Managing software evolution with the service concept. *Proceedings of the International Symposium on Principles of Software Evolution* (Eds. Takyo Katayama, Tetsuo Tamai, and Naoki Yonezaki), pages 43–47, Kanazawa, Japan, November 1–2, 2000.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15:2*, December 1972.

[17] P. Zave. 'Calls considered harmful' and other observations: A tutorial on telephony. *Services and Visualization: Towards User-Friendly Design* (Eds. T. Margaria, B. Steffen, R. Rückert and J. Posegga), pages 8–27, Springer-Verlag LNCS 1385, 1998.

Coordination Contracts, Evolution and Tools

L.Andrade, J.Gouveia, G.Koutsoukos
Oblog Software S.A.
Alameda Antonio Sergio 7-1A
2795 Linda-a-Velha, Portugal
{landrade, jgouveia, gkoutsoukos}@oblog.pt

Jose Luiz Fiadeiro
Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

Abstract

In this paper, we propose the adoption of coordination contracts – a modeling primitive grounded on a formal semantics based on the Category Theory – as a basis for a discipline that effectively supports software evolution. We give an overview of coordination contracts, present examples of how they can support evolution, and describe a development environment through which they can be used in practice.

1. Introduction

No one can admittedly deny the fact that today we are witnessing tremendous technological advances in a highly competitive business environment. To the question whether technology is forming business or vice-versa, organisations are replying by integrating their business and IT strategies, thus using technology to do business. As a result, there is an increasing pressure for building software systems that are able to cope with new requirements imposed by both technological advances and different business rules. Even worse, as a result of e-economics, systems often have to be able to accommodate changes in run-time, even performed directly by customers. As a consequence, organisations are seeking answers on how to conceive and develop systems that are adaptive to change.

For better or for worse, organisations are looking for solutions to this problem in the context of object-oriented development techniques such as the UML, and component-based frameworks such as COM and CORBA. However, as explained in [1,2] experience has shown that the benefits that object-oriented techniques have brought to software *construction* cannot be extended directly to software *evolution*. Moreover, disciplines that, in theory, support software evolution, in practice often fail to provide a means for their implementation and, unfortunately, end up being buried in the literature. As a consequence, it is not surprising that existing tools that intend to offer support for evolution are far from ideal.

In this paper, we propose the adoption of the coordination contract modelling primitive presented in [1,2,4], grounded on a formal semantics based on Category Theory [2,4], as a basis for a discipline that can lead to software systems that are adaptive to change. We briefly discuss coordination contracts and we present examples from banking and telecommunications on how contracts can support software evolution. Moreover, we present and discuss the scope, applicability and impact on the development life-cycle of an environment that we have been building for allowing coordination contracts to be effectively used as a technology.

2. Coordination Contracts

As discussed in [1,2] the rationale for the definition of coordination contracts is the realization that, in highly volatile domains, one can distinguish between two different kinds of entities as far as the evolution of the application domain is concerned. On the one hand, there are classes of objects that correspond to business entities that are relatively stable in the sense that they capture core concepts or "invariants" of the domain. On the other hand, we need objects that have to keep changing in order for the system to reflect the dynamics of the application

domain. These require a layer of coordination to be established over the functionalities of the stable entities so that the behavior that is required from the system can emerge, at each state, from the interconnections that this layer of coordination puts in place.

These coordination aspects need to be made available explicitly in the system models so that they can be changed, as a result of modifications occurring at the level of requirements, without affecting the basic objects that compose the system. The purpose of contracts is to provide mechanisms for that layer of coordination to be modeled and implemented in a compositional way.

In general terms, a coordination contract is a connection that is established between a group of objects where rules and constraints are superposed on the behavior of the participants, thus enforcing a specific form of interaction. From a static point of view, a contract defines what in the UML is known as an *association class*. However, the way interaction is established between the partners is more powerful than what can be achieved within the UML and similar OO languages because it relies on the mechanism of *superposition* as developed for parallel and distributed system design [5,6,8]. When a call is made from a client object to a supplier object, the contract "intercepts" the call and *superposes* whatever forms of behavior it prescribes. In order to provide the required levels of pluggability, neither the client, nor any other object in the system, needs to know what kind of coordination is being superposed. To enable that, a contract design pattern, presented in [3,7], allows coordination contracts to be superposed on given objects in a system to coordinate their behavior without having to modify the way the objects are implemented.

Coordination Contracts are currently supported by a specification language called Oblog, but the underlying technology is independent of the language. In Oblog notation, a coordination contract is defined as follows:

contract class <name>
participants <list of partners>
constraints <the invariant the partners should satisfy>
attributes
operations
coordination <interaction with partners>
end class

The classes of objects that are related by the contract are identified under *participants*. A contract may also specify constraints that represent invariants defining in which conditions instances from the participating classes may be related by the contract, attributes and operations private to the contract, and the prescription of the coordination effects that will be superposed on the partners. Each interaction under a coordination rule is of the form:

<name> **when** <trigger>
with <condition>
do <set of actions>

The name of the interaction is used for establishing an overall coordination among the various interactions and the contract's own actions. The condition under "when" establishes the trigger of the interaction. The trigger can be a condition on the state of the participants, a request for a particular service, or a signal received by one of the participants. Several conditions can be placed in the "when" clause using the keyword "AND". If one of such conditions is not satisfied, the contract is considered as being "inactive" and, as a result, either the original code of the trigger or another contract is executed. This mechanism provides the ability for controlling which of the contracts imposed on a component will be responsible for coordinating it.

The "do" clause identifies the reactions to be performed, usually in terms of actions of the partners and some of the contracts own actions. When the trigger corresponds to an operation, three types of actions may be superposed on the execution of the operation:

1. **before**: to be performed before the operation
2. **replace**: to be performed instead of the operation
3. **after**: to be performed after the operation

In the case in which an object participates in multiple contracts with the same trigger, the sequence of execution for the before, replace and after clauses is shown in Figure 1. It should be noted that the semantics of contracts allow for only one "replace" clause to be executed, thus preventing the undesirable situation of having two alternative actions for the same trigger. Furthermore, any such replacement action must adhere to whatever specification clauses apply to the operation (e.g. contracts in the sense of Meyer [9] specifying pre- and post-conditions). This ensures that the functionality of the original operation, as advertised through its specification, is preserved.

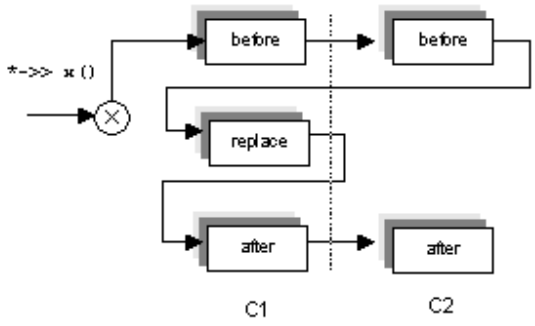


Figure 1. Execution of multiple contracts

The actions that are executed as part of the "do" clause are called the synchronization set associated with the trigger. The semantics of contracts require that this set be executed atomically, guarded by the conjunction of the guards of the individual actions together with the conditions included in the "with" clause. Therefore, the "with" clause puts further constraints on the execution of the actions involved in the interaction. If any condition under the "with" clause is not satisfied, an exception is thrown as a result and none of the actions in the synchronization set is executed.

For a detailed description of coordination contracts and their formal semantics, the reader is urged to consult [2,4]. In what follows we present an example from banking to motivate the scope and solutions coordination contracts can offer. Consider a world of bank accounts in which clients can, as usual, make withdrawals. The object class *account* is usually specified with an attribute *balance* and a method *withdrawal* with parameter *amount*. In a typical implementation one can assign the guard *Balance>=amount* restricting this method to occur in states in which the amount to be withdrawn can be covered by the balance. However, as explained in [1] assigning this guard to *withdrawal* can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like account. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type. As discussed in [1] inheritance is not a good way of changing the guard in order to model these different situations. Firstly, inheritance views objects as white boxes in the sense that adaptations like changes to guards are performed on the internal structure of the objects, which from the evolution point of view of is not desirable. Secondly, from the business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In our example, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard that applies to withdrawals. The reason the guard will end up applied to *withdrawal*, and the specialisation to *account,* is that, in the traditional clientship mode of interaction, the code is placed on the supplier class. Therefore, it makes more sense for business requirements of this sort to be modeled explicitly outside the classes that model the basic business entities, because they represent aspects of the domain that are subject to frequent changes (evolution). Our proposal is that guards like the one discussed above should be modeled as *coordination contracts* that can be established between clients and accounts. For instance, consider the following contract that allows for using the functionality of *withdrawal* to relax the situations in which an account may be overdrawn.

```
contract   VIP package
participants x : Account; y : Customer;
constants      CONST_VIP_BALANCE: Integer;
attributes     Credit : Integer;
constraints    ?owns(x,y)=TRUE;
   x.AverageBalance() >= CONST_VIP_BALANCE;
coordination
vp:   when     y.calls(x.withdrawal(z))
      with     x.Balance() + Credit() > z;
      do       x.withdrawal(z)
end contract
```

To further illustrate how contracts can be applied to support the evolution of requirements we present a second example from a telecommunications transaction processing system. Consider the following specification of an account from a telephone service provider. The main purpose of the class is, simply, to charge the account whenever a phone-call finishes. The other operations of the class are, also, self-explanatory.

```
class Account
attributes
  object
  tel_number: Integer;
  balance: Integer:=0;
  charge_rate: Integer;
operations
  class
  *Create(client: Customer);
  object
  ?Balance(): Integer; // function, returns balance
  Charge(call_time: Integer);
body
methods
```

```
Charge
  is {
  set balance:= Balance()+call_time*charge_rate;
  }end
end class
```

A second class specification can be defined with the purpose of modelling the phone calls that each client makes. The operations specified here are used for illustrative purposes. Therefore, they are limited to the one that calculates the duration of a call and the one that determines the end of a call.

```
class Call
declarations
attributes
  object
  caller_number:Integer;
operations
  class
  *Create(client: Customer);
  object
  FinishCall();
  ?CalculateCallTime():Integer;
  body
methods
FinishCall
  is {
  // body of finish call detects end of call
  }end
CalculateCallTime
  is {
  // body-calculates the duration of the call
  }end
end class
```

In order to achieve the charging of the Account as soon as the phone-call ends we have to consider two possible scenarios, both related to the implementation of the two components. Either the Account and Call components are completely independent and are not aware of the existence of each other, in which case a third component is needed that becomes responsible for detecting the end of the phone-call, calculate the duration and perform the charge (Figure 2a), Or, Call is responsible for calling the Charge() method, for instance inside the FinishCall() method (Figure 2b). It should be clear that the latter would be a "weak" implementation. Indeed, it is hardly the role of a component that models phone-calls to charge an Account. However, such implementations often exist in real life applications. We argue that in the first scenario the best choice is to have a contract as the third component and that, in both cases, contracts provide a very effective way to evolve the system without modifying the existing components.

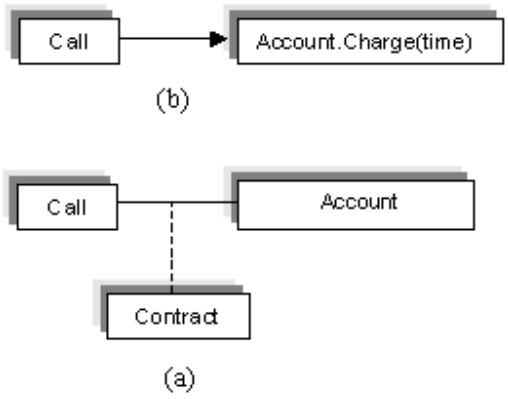


Figure 2. (a) Components Independent, (b) Components dependent

As far as the first scenario is concerned, the following simple contract, *Traditional Charging,* has the role of the third component and provides the required functionality while offering the advantage that the mechanism (contract) that controls the usage of the given objects is modelled as a first-class entity and, hence, can be evolved independently of the other two.

```
contract class Traditional Charging
participants x : Account; y : Call;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
      local time: Integer:= y.CalculateCallTime();
      x.Charge(time);
end class
```

Consider now the situation in which the telephone provider wants to have two types of customers and charge them according to different rules. For instance, it could charge important customers only after the call exceeds a specific number of seconds, whereas not important customers are charged for the whole duration of their phone-call. If Account and Call are independent, i.e the first scenario described above is in place, the solution is simply to add to the system a contract like the one below.

```
contract class VIP_Charging
participants x : Account; y : Call;
attributes free_call_limit:Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
      local time: Integer:= y.CalculateCallTime();
      if (time> free_call_limit){
      x.Charge(time - free_call_limit);
```

```
    }
end class
```

The functionality of both the previous contracts is straightforward. They coordinate the charging of the Account according to the type of customer and the business rules the network operator defined. Notice that if a "*->>" is specified in the coordination part of the contract, any call to the service triggers the rule and that the keyword **local** just defines a local variable. If a future business requirement determines different behaviour for the components, a new contract, like the VIP_Charging contract above, can be added to the system in a "plug and play" mode in order to achieve the required behaviour.

Consider now the second scenario in which the two components, Account and Call, are aware of the existence of each other and that, in fact, an instance of a Call has to invoke the Charge() method in order to perform the charging of the customer's account (possibly as soon as the call ends). In this scenario, evolving the system to comply with the new requirement of having different charging schemes for different kinds of customers is not possible without modifying the components. For instance, consider the case in which inside FinishCall () there is a statement of the form Account.Charge (CalculateCallTime). Clearly, it is not possible to change the charging mechanism without changing the source code of either FinishCall() or Charge(). However, a contract like the one below can achieve the required functionality without having to modify the implementation of Call and Account.

```
contract class VIP_Charging_2
participants x : Account; y : Call;
attributes free_call_limit:Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when y->> x.Charge(time);
  replace
     if (time > free_call_limit){
     local newtime: Integer:= time-free_call_limit;
     x.Charge(newtime);
     }
// implied "else" is void i.e. if time<free_call_time
// nothing is executed (it does not charge)
end class
```

A third scenario of evolution is the one in which we have different charging schemes related to the charge rate. For instance, a VIP Customer, can be charged with a charge_rate_1 when the duration of the call is within a time range [0-time_limit] and with a charge_rate_2 when the duration of the call exceeds time_limit. Again the following contract where the charging rates are decided inside the contract can offer a very effective and flexible solution.

```
contract class VIP_Charging_3
participants x : Account; y : Call;
attributes charge_rate_1, charge_rate2, time_limit :
Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
     local time: Integer:= y.CalculateCallTime();
     if (time <= time_limit){
  .  charge_rate:= charge_rate1;
     }
     else if(time > time_limit){
     x.charge_rate:= charge_rate2;
     }
     x.Charge(time);
end class
```

It should be noted that there are a large number of additional examples from different application domains that show how contracts can externalize the interactions between objects making them explicit in the conceptual model and support the compositional evolution of systems. Due to space limitations we will neither present such examples, nor the contracts formal semantics and the design pattern that puts them in practice. The reader can consult [1,2,3,4,7], for more details. In what follows we discuss how coordination contracts can be related to software tools.

3. Coordination Contracts and Tools

Coordination contracts can be related to tools in two ways: Firstly, in terms of tools that aim to apply coordination contracts as an intermediate stage in the development of larger systems and, secondly, in terms of using contract-based development for building software tools. The latter case draws from the realization that software tools are often themselves complex systems that are under constant evolution to cope with different requirements. As a result, techniques, such as coordination contracts, that aim to support software evolution in general, are also directly applicable to such tools. Therefore, it makes more sense to further discuss the former case only.

The former case is concerned with building tools to put in practice the concept of coordination contract. The implementation of such tools normally involves the following stages:

a. adopting the concepts of contracts to re-engineer components, in terms of making their functionality independent of their interconnection, thus

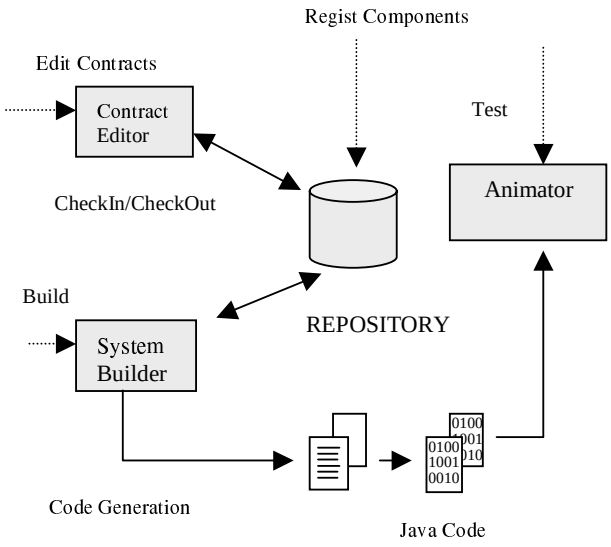


Figure 3. Coordination Contracts Tool Architecture

enabling externalization of volatile elements (business rules).

b. having *design/implementation techniques* that implement the contract concept in a way that satisfies the necessary requirements to achieve dynamic system evolution. The main technique is using a *Design Pattern* like the one presented in [3,7].

c. using a *specification tool* that will manage contract development and automated implementation of the pattern, by code generation/adaptation.

d. using a *configuration tool* that will "deploy", activating and deactivating, contracts.

Assuming implementation of the first stage, in other words, assuming having components of a suitable form either generated by a tool or coded by hand, we focus on the coordination layer of such tools. The activities of the development process that are supported by such tools are the following:

- *Registration*: components are registered in the tool.
- *Edition*: Contracts are defined connecting some registered components. Coordination rules and constraints are defined on those contracts.
- *Deployment*: the code necessary to implement the coordinated components and the contract semantics in the final system is produced by *generating some* parts according to the contract design pattern and *adaptation* of the given component part.
- *Animation*: some facilities are provided allowing testing/prototyping of contract semantics.

The logical architectural components, namely the Editor, the Repository, the Builder and the Animator that support the previous activities, are presented in Figure 3.

In this context, coordination contract tools may be applied to different levels in system development depending on several factors, such as the characteristics of the components, the way components are built, the development phase where the contract concept is going to be used, among others. We illustrate this diversity with two possible scenarios of using coordination contract tools:

Model Coordination: Coordination is used at the Analysis or Design phases. Components are model classes (e.g. UML classes) and coordination contracts make a Coordination Model on top of the Analysis/Design Model. The deployment activity must take into account the way final coded components are obtained from model components and provide the necessary integration.

Construction Coordination: Coordination is used in the Implementation phase. Components are the final coded components of the basic building blocks of the system and coordination contracts are defined directly over implementation classes. It is suitable to be applied on the evolution of an existing system.

We realize, however, that the type of components to coordinate may define the context and capabilities in which such a tool is used and that the specific language and technical environment may impose constraints on the coordination features that can be used, since techniques to achieve the implementation of its semantics may not be available. We intend to further discuss and provide solutions for such issues in the future. However, to this end, we strongly believe that coordination contracts, its formal semantics that admits an implementation via design patterns and a contracts' development environment form a very strong basis for Software Engineers and developers to meet the challenge of designing and developing systems (and tools) that are better structured, consist of reusable parts, and are adaptive to change.

4. References

[1] L.F Andrade and J.Fiadeiro, “Evolution by Contract”, position paper presented at the *ECOOP'00 Workshop on Object-Oriented Architectural Evolution*.

[2] L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", *in UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.

[3] L.F.Andrade, J.L.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger, "Patterns for Coordination", in *Coordination Languages and Models*, G.Catalin-Roman and A.Porto (eds), LNCS 1906, Springer-Verlag 2000, 317-322.

[4] L.Andrade and J.Fiadeiro, “Coordination: the evolutionary dimension”, in *Proc. TOOLS Europe 2001,* Prentice-Hall, in print.

[5] K.Chandy and J.Misra, Parallel Program Design - A Foundation, Addison-Wesley 1988.

[6] N.Francez and I.Forman, Interacting Processes, Addison-Wesley 1996.

[7] J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro, “Tool Support for Coordination-Based Software Evolution”, *in Proc. TOOLS Europe 2001,* Prentice-Hall, in print.

[8] S.Katz, "A Superimposition Control Construct for Distributed Systems", *in ACM TOPLAS* 15, 1993 337-356

[9] B.Meyer, "Applying Design by Contract", *in IEEE Computer*, Oct.1992, 40-51.

Graph Transformation as a Meta Language for Dynamic Modeling and Model Evolution*

Reiko Heckel and Gregor Engels
Dept. of Math. and Comp. Science, University of Paderborn
D-33095 Paderborn, Germany

http://www.upb.de/cs/ag-engels

E-mail: reiko|engels@upb.de

Abstract

A major concern of software evolution is to achieve and maintain consistency between both different (sub)systems and different layers of the same system. Based on a conceptual model of distributed systems which distinguishes the three layers of objects, software, and hardware components *and, orthogonally, a* type *and an* instance level, *we discuss solutions to several consistency problems. We classify the state changes a system experiences during its lifetime as the system's* dynamics *(if the changes happen at the instance level) and the* evolution *of the system (if the type or schema level is affected). An approach based on graph transformation and meta modeling is used to formalize these concepts.*

1. Consistency Problems in Software Evolution

One of the major forces driving software evolution today is the integration of applications over the internet. E-commerce or e-business applications, for example, combine services of different enterprises to yield one integrated product. Thereby, boundaries between different data formats, computational platforms, and administrative domains have to be bridged, in particular, if the applications have been developed under different authorities using different process models, methodologies, and tools. A major concern of software evolution is, therefore, to achieve and maintain the consistency between different (sub)systems. This problem of *horizontal consistency* occurs at two levels, the *application logic* and the *software architecture* level, and it concerns both *static and dynamic* aspects of a software system.

*Research supported by the ESPRIT Working Group APPLIGRAPH and the TMR network GETGRATS.

We distinguish between *dynamics* and *evolution* depending on whether or not the type-level is involved: Creating a new object or installing a new instance of a component are dynamic changes as long as their corresponding classes are present. Updating a class or downloading a new type of component are examples of evolution.

A second issue, which is classical in distributed systems, is *deployment* or, more generally, the *vertical consistency* of the application logic with the network architecture. However, internet applications are not like distributed applications in a local-area network, which provides a reliable infrastructure with mostly static connections and a simple (or transparent) structure. The internet is, instead, unreliable, highly dynamic, and hierarchically structured. That means, new nodes and connections are frequently added, nodes may be only temporarily connected (e.g., via private phone lines), or they may be temporarily unconnected (due to failures or overload), while others move around between subnets (e.g., as mobile computing devices). Thus, also *vertical consistency* has a static and a dynamic aspect.

In the next section we shall review some solutions to horizontal and vertical consistency problems. Then, we sketch a framework based on meta modeling with graphs and graph transformation which supports an integration of these individual solutions.

2. Consistency Through Modeling

One of the main approaches to support consistency in both dimensions is to build a model which represents an abstraction from both implementation details and irrelevant aspects of the real world. Models play a central role in forward, reverse, and re-engineering, and they serve as communication medium in distributed development teams. For this purpose, a modeling language is required which is intuitively understandable by customers, software engineers,

and programmers and which provides a formal semantics based on an underlying conceptual model which is agreed upon by the users of the language.

Various modeling languages have been proposed to solve one or more of the consistency problems stated above. Next we shall discuss some of them, pointing out the use of graph transformation wherever appropriate.

Horizontal consistency of application logic. In order to achieve static consistency, a model is required which defines the shared knowledge of the subsystems about the concepts and structure of the application domain. The classical example for this use of models is the conceptual model of a data base underlying a distributed information system, where the data of several client applications is stored in an integrated way. The connection between the conceptual data model and the clients local models is established by the notion of *view*.

In order to achieve dynamic consistency, object-oriented development processes like the Unified Process [10] extend the static *object model* by a description of the overall *workflows* or *business processes* that are or shall be supported by the integrated application. A notion of view taking into account this dynamic aspect is proposed, e.g., in [6], where the operations on objects are described by graph transformation rules. A view induces a projection of this globally specified behavior to the client applications thus fixing their individual role and function in the overall process.

Horizontal consistency of architecture. When integrating previously independent applications, problems arise from incompatible protocols or data formats which have to be adapted and translated into each other, respectively. One way of avoiding this is the definition of *architectural styles* specifying kits of components and connectors which can be combined freely while ensuring interoperability. The definition of architectural styles is supported by architectural description languages like WRIGHT [21] or DARWIN [14]. These are specialized modeling languages which allow to describe the behavior of components and connectors by means of process calculi or abstract programming languages. Object-oriented approaches like UML/RT use statechart diagrams for this purpose.

The situation becomes more complex when the architecture changes dynamically. In this case, one has to ensure that, e.g., changes in different subsystems do not violate referential integrity (e.g., when one local application relocates a component that is used by another local application), and that they do not disturb or interrupt running protocols [13]. Here, graph transformation techniques have proven useful because they provide a very general way of specifying dynamic change of architectures, which can be perceived as graphs of components and connectors [23, 9, 24].

Vertical consistency. Software architectures provide the link between the application logic and the physical network architecture: Objects and classes are clustered into components which, in turn, are the units of deployment. Components with precisely defined interfaces support the flexibility of this mapping by making the dependencies between components explicit. On a technical level, this is supported by standards like CORBA, DCOM, and ENTERPRISE JAVA BEANS [19, 17, 22] which provide the necessary interface definitions and services for platform independent distributed applications. Still, this indispensable infrastructure does not ensure the consistency of the software architecture with that of the underlying network, in particular, if the latter is changing dynamically. Again, a model is required which specifies both network changes and resulting reconfigurations at the software architecture level.

Relatively little support is provided for this problem so far. Although object-oriented methodologies allow to specify both the deployment of components at network nodes and the clustering of objects into components, they do not provide satisfactory means to speak about reconfiguration of hardware and software. Architectural description languages, if they provide dynamic features, usually restrict to the level of software architectures. Based on these observations, in [5] we have proposed a semantic framework for distributed systems based on hierarchical graph transformation which supports an integrated specification of dynamic change and evolution at the three layers of objects, software components, and hardware components.

3. Dimensions in Modeling Dynamic Change and Evolution

In the previous section, we have sketched graph transformation solutions to individual consistency problems. Although many of them build upon a similar formal basis they differ significantly w.r.t. their interpretation of graphs and transformations. In the approaches sketched above, graphs represent object structures, software, or hardware architectures and transformation rules model operations on objects or architectural reconfigurations. Other approaches use graphs for modeling class and entity-relationship diagrams or actual programs, and transformation rules for describing schema evolution and refactoring [1, 15, 11, 16, 12]. In this section, we shall identify, by means of a small example, some general concepts and dimensions in modeling dynamic change and evolution which shall allow us to put the cited approaches in a common perspective.

Type and instance level. A wide-spread feature is the distinction between graphs at the *type level* (like data base schemata, class diagrams, or architectural styles) and at the *instance level* (like data base states, object structures, and

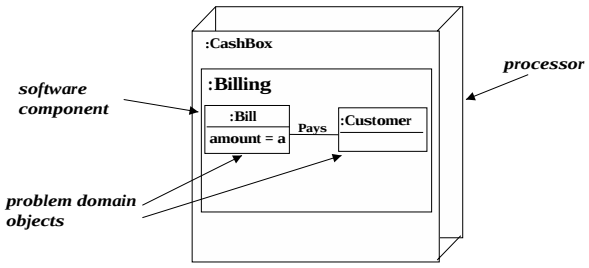


Figure 1. Hierarchical state (instance level).

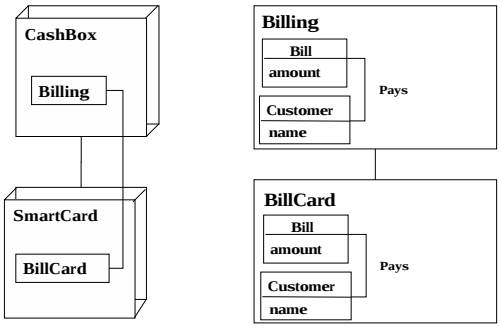


Figure 2. Hierarchical state (type level).

concrete architectures). Mostly, a type-level graph is considered as a static description of a class of instance-level graphs which represent, e.g., the states of the system.

Hierarchical structure. Within both type and instance level, a hierarchical structure of three layers can be identified. At the instance level, we have objects at the first layer residing in software components at the second layer which, in turn, are deployed at hardware components (called *nodes*) forming the third layer.

The allocation of software components at hardware nodes and of objects at software components can be described in the notation of UML deployment and component diagrams. A sample is shown in Fig. 1 where a CashBox node hosts a Billing component responsible for issuing Bill objects and storing them together with Customer objects until they are paid. A similar hierarchy exists at the type level, as exemplified in Fig. 2 where two separate diagrams are used to describe the potential for deploying, e.g., a BillCard component at a SmartCard node, and the ability of components to store certain types of objects.

Dynamic change. So far, we have only dealt with the structural aspects of our model. In order to capture distributed and mobile applications with dynamic reconfiguration we have to specify operations for transforming this structure.

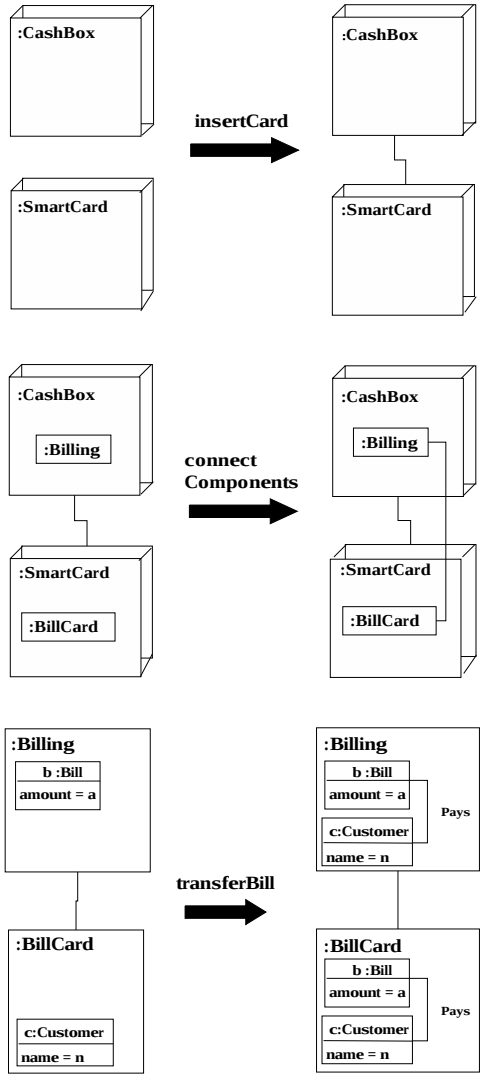


Figure 3. Dynamic change (instance level)

Dynamic change at the instance level is specified by transformation rules as shown in Fig. 3. A rule consists of two instance-level diagrams, the left- and the right-hand side, where the former specifies the situation before the operation and the latter the situation afterwards. The rules in Fig. 3 describe the scenario of downloading a Bill object originally residing in the Billing component of a CashBox to the BillCard component of a SmartCard: When the card is inserted into the cash box's card reader, a hardware connection is established. This triggers the connection of the Billing with the BillCard component. Then, the bill is stored in the BillCard component and the customer's identity is recorded by the Billing component of the cash box.

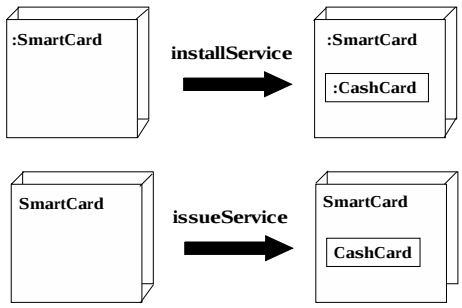


Figure 4. Evolution (type and instance level)

Notice, that we have described dynamic changes within all three layers of our hierarchy and all but the first transformation are concerned with two layers at the same time. Thus, not only the system's states are hierarchical, but also the operations have to take care of the hierarchical structure, as they are potentially not restricted to a single layer.

Evolution. The last point to be discussed in this section is the conceptual difference between dynamic change at the instance level and the evolution of the system by changes at the type level. As systems have to be adapted to new requirements, not only their configuration may change, but also it may be necessary to introduce new types of hardware or to deploy new software components containing objects of classes which have not been present before.

In our application scenario, a new component CashCard is provided, which has to be downloaded on the card in order to provide the additional service of using the card directly for paying bills at a cash box. The operation installing a new component instance on an individual card is specified by the upper rule in Fig. 4. However, if the component is newly developed, it has to be added to the type-level as well, in order to enable the change at the instance level. The corresponding rule is shown in the bottom of Fig. 4.

We conclude our discussion of the different dimensions in modeling systems with dynamic change and evolution by a rough classification of the cited approaches. According to our terminology [13, 6, 23, 9, 24] deal with dynamic change rather than evolution because changes are restricted to the instance level. Instead [1, 15, 11, 16] consider also changes at the type level. Moreover, [6, 1, 15, 11, 16] work at the object layer while [13, 9, 24] consider the layer of software architecture and [23] covers both software architecture and objects. In the next section, we will formalize these conceptual dimensions in a meta model.

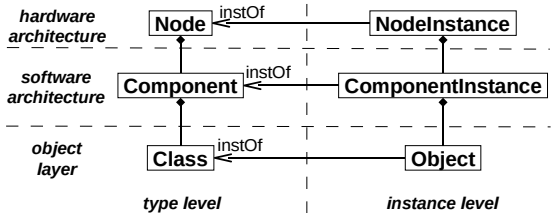


Figure 5. Meta model for hierarchical states (fragment)

4. A Meta Model for Dynamic Change and Evolution

Following the algebraic approach to graph transformation [3], the relation between diagrams at the type level and diagrams at the instance level is formalized by the concept of *type* and *instance graphs* [2]. An instance graph is equipped with a structure-preserving mapping (i.e., a homomorphism) towards a type graph which is fixed for the entire model. However, as noted above, evolution is concerned with changes at the type or schema level. Therefore, in order to represent rule-based model evolution within the framework of static typing, the type level of the model (given, e.g. by the diagrams in Fig. 2) has to be represented as part of the instance graphs. The actual type graph, instead, represents a *meta model* of the language which specifies, for example, the relation between classes and objects in the model. A well-known example of this approach is the UML meta model [18] which specifies syntactic dependencies between the UML diagrams based on their abstract syntax.

A meta model integrating the features discussed above would look like the fragment in Figure 5. According to this meta model, a model for a distributed software system is a three-layered hierarchical graph where objects at the lowest layer are clustered by components at the second layer, which themselves are located at nodes of a computer network at the third layer. The vertical integration of these layers is represented by *aggregation edges* with solid diamonds at the top. Orthogonal to these three layers, the instance level and the type level are distinguished. The association of instances to their types is modeled by horizontal instOf links. (Structural links, like associations or connectors, are omitted for simplicity.)

An instance graph over the (meta) type graph in Figure 5 is shown in Fig. 6. It represents the abstract syntax of the hierarchical state jointly given in Fig. 1 and 2. Notice, that this instance graph contains both representations of classes (components, nodes) and objects (component instances, node instances), i.e., it integrates type- and instance-level diagrams of the model. The type graph in

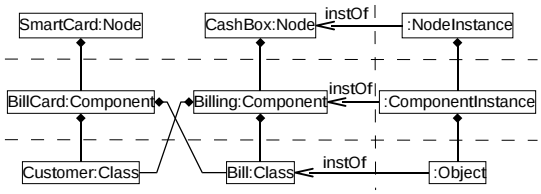


Figure 6. Abstract syntax of state in Fig. 1/2

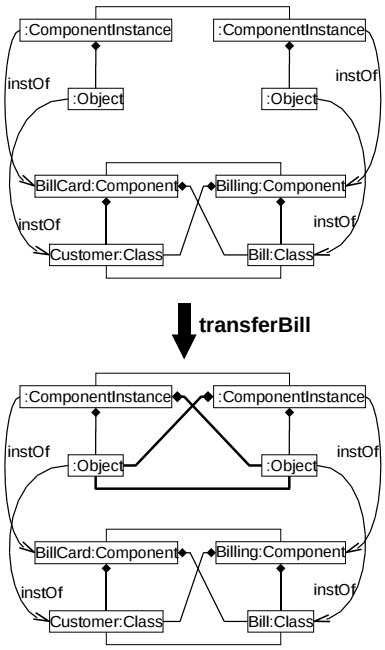


Figure 7. Abstract syntax of transferBill

Fig. 6, instead, specifies the structure of static diagrams of the entire language rather than an aspect of an individual model.

With this encoding, transformation rules can change both the instance and the type level of a model. A graph transformation rule conforming to the meta type graph of Fig. 6 is shown in Fig. 7. It represents the abstract syntax of the rule transferBill in Fig. 3. Notice the sharing of Bill and Customer between the Billing and the BillCard components, which becomes evident in this presentation. The meta model presentations of the two rules in Fig. 4 are shown in Fig. 8. Also here, we do not change the formal meta type graph but "implement" model evolution through the representation of type information at the instance level.

In the framework described so far, the evolution of a system is limited to structural modifications at the type level. If the evolution shall include other aspects of the model, the meta model has to be be extended by representations of these aspects. For example, evolution of transformation

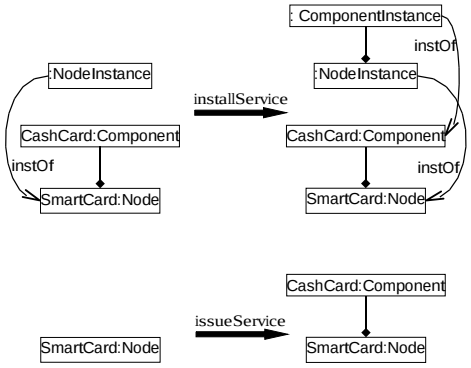


Figure 8. Abstract syntax of installService

rules (specifying dynamic change) is considered in [15]. A meta model for an object-oriented graph transformation approach that captures dynamic change is developed in [8].

5. Conclusion

In this paper, we have identified three kinds of consistency problems and their individual solutions through modeling. Then, we have sketched a framework based on graph transformation and meta modeling which could be the conceptual and formal basis for integrating these approaches. Finally, we shall briefly discuss some of the benefits this idea and its potential application for developing tools.

First, a general advantage of meta modeling (e.g., over the use of logic, set theory or category theory) is that syntax and semantics definitions based on meta models are much easier to communicate to the "average software engineer". Still, using a formal meta modeling approach as suggested above, they can be as precise as mathematical definitions. Second, since meta modeling uses the same concepts that are also used for modeling and implementing software systems, we can reuse techniques and tools. For example, a meta model for the dynamic semantics of statechart diagrams [4] provides a model for a statechart interpreter. Graph transformation tools like PROGRES [20] or FUJABA [7], which may execute such models, can be used as meta case tools to generate interpreters from graphical semantics definitions.

References

[1] I. Claßen and M. Löwe. Scheme evolution in object–oriented models. In M. Wirsing, editor, *ICSE–17 Workshop on Formal Methods Application in Software Engineering Practice*, Seattle, Washington, April 1995.

[2] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[3] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997. Preprint available as Tech. Rep. 96/17, Univ. of Pisa, `http://www.di.unipi.it/TR/TRengl.html`.

[4] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*, pages 323–337. Springer-Verlag, 2000.

[5] G. Engels and R. Heckel. Graph transformation as unifying formal framework for system modeling and model evolution. In *Proc. ICALP 2000*, LNCS, Geneva, Switzerland, July 2000. Springer-Verlag.

[6] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engeneering*, 7(4):457–477, 1997.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.

[8] R. Heckel and A. Zündorf. How to specify a graph transformation approach: A meta model for FUJABA. In H. Ehrig and J. Padberg, editors, *Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS 2001, Genova, Italy*, 2001. To appear.

[9] D. Hirsch and U. Montanari. Consistent transformations for software architecture styles of distributed systems. In G. Stefanescu, editor, *Workshop on Distributed Systems*, volume 28 of *Electronic Notes in TCS*, 1999.

[10] G. B. Ivar Jacobson and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[11] J. Jahnke and A. Zündorf. Using graph grammars for building the VARLET database reverse engineering environment. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.

[12] W. Kahl. Software evolution via hierarchical hypergraphs with flexible coverage. In T. Mens and M. Wermelinger, editors, *International Special Session on Formal Foundations of Software Evolution, Lisboa, Portugal*, March 2001. Co-located with the European Conference on Software Maintainance and Reengineering (CSMR 2001).

[13] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change manegement. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[14] J. Kramer and J. Magee. Distributed software architectures. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 633–634. Springer-Verlag, May 1997.

[15] M. Löwe. Evolution patterns. Postdoctoral thesis, Technical University of Berlin. Tech. Report 98-4, Dept. of Comp. Sci, 1997.

[16] T. Mens. Transformational software evolution by assertions. In T. Mens and M. Wermelinger, editors, *International Special Session on Formal Foundations of Software Evolution, Lisboa, Portugal*, March 2001. Co-located with the European Conference on Software Maintainance and Reengineering (CSMR 2001).

[17] Microsoft Corp. Distributed component object model protocol – DCOM, V 1.0, 1998. `http://www.microsoft.com/com/tech/dcom.asp`.

[18] Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`.

[19] Object Management Group. The common object request broker: Architecture and specification, v. 3.0, 2000. `http://www.omg.org`.

[20] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 487–550. World Scientific, 1999.

[21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[22] Sun Microsystems Inc. Enterprise Java Beans specification. `http:://java.sun.com/products/ejb`, 2000.

[23] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In H. Ehrig, G. Engels, H.-J. Kreowsky, and G. Rozenberg, editors, *Proceedings TAGT'98*, volume 1764 of *LNCS*. Springer-Verlag, 2000.

[24] M. Wermelinger and J. Fiadero. A graph transformation approach to software architecture reconfiguration. In H. Ehrig and G. Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000), Berlin, Germany*, March 2000. `http://tfs.cs.tu-berlin.de/gratra2000/`.

Change Impact Analysis for Architectural Evolution *

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0214, Japan
Email: zhao@cs.fit.ac.jp

Abstract

Chang impact analysis is useful in software maintnenance and evolution. Many techniques have been proposed to support change impact analysis at the code level of software systems, but little effort has been made for change impact analysis at the architectural level. In this paper, we present an approach to support change impact analysis of software architectures based on architectural slicing technique. The main feature of our approach is to assess the effect of changes in a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

1 Introduction

Software change is an essential operation for software evolution. The change is a process that either introduces new requirements into an existing system, or modifies the system if the requirement were not correctly implemented, or moves the system into a new operation environment. The mini-cycle of change as described in [15] is composed of the several phases: request for change, planning phase which consists of program comprehension and change impact analysis, change implementation including restructuring for change and change propagation, verification and validation, and redocumentation. Among these phases, in this paper we will focus our attentions on the issue of planing phase, in particularly, change impact analysis.

Change impact analysis is the task that through which the programmers can assess the extent of the change, i.e., the software component that will impact the change, or be impacted by the change. Change impact analysis provide techniques to address the problem by identifying the likely ripple-effect of software changes and using this information to re-engineer the software system design.

Most work on software change impact analysis focused on code level of software systems which are derived solely from source code of a program [3, 6, 7], and the study of architectural-level change impact analysis has received little attention. However, as software systems become large and complex, it is necessary to perform architectural-level impact analysis because it allows you to capture the information of change effect of the a system's architecture earlier in the system life cycle so you can perform software evolution actions earlier [10].

However, the study of architectural-level impact analysis has received little attention in comparison with code-level impact analysis. One important reason is while the code level for software systems is now well understood, the architectural level is currently understood mostly at the level of intuition, anecdote, and folklore [12]. Existing representations that a system architect uses to represent the architecture of a software system are usually informal and *ad hoc*, and therefore can not capture enough useful information of the system's architecture. Moreover, with such an informal and *ad hoc* manner, it is difficult to develop analysis tools to automatically support the change impact analysis at the architectural level of software systems. In order to develop architectural-level change impact analysis tool to support architectural evolution during software design, formal modeling of software architectures is strongly required.

Recently, as the size and complexity of software systems increases, the design and specification of the overall software architecture of a system is receiving increasingly attention. The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction [12]. Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall application rather than the implementation details of any specific source module. In order to support formal representation and reasoning of software architecture, a number of ADLs such as WRIGHT [1], Rapide [8], and UniCon [11] have been proposed. By using an ADL, a system architect can formally represent various general attributes of a software system's architecture. This provides a promising solution to develop techniques to support change impact analysis for software architectures because formal language support for software architecture provides a useful platform on which automated support tools for architectural-level impact analysis can be developed.

In this paper, we present an approach for change impact analysis of software architectures based on *architectural slicing* technique. The main feature of our ap-

*This work is partly supported by The Ministry of Education, Science, Sports and Culture of Japan under Grand-in-Aid for Encouragement for Young Scientists (No.11780241) and by a grant from the Computer Science Laboratory of Fukuoka Institute of Technology.

proach is to assess the effect of changes in a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

Traditional program slicing, originally introduced by Weiser [14], is a decomposition technique which extracts program elements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*.

In contrast to traditional program slicing, architectural slicing is designed to operate on a formal architectural specification of a software system, rather than the source code of a conventional program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a conventional program. Our purpose for development of architectural slicing is to support architectural-level impact analysis, maintenance, reengineering, and reverse engineering of large-scale software systems.

Applying slicing technique to change impact analysis of software architectures promises benefit for software architecture understanding and maintenance. When a maintainer wants to modify a component in a software architecture in order to satisfy new design requirements, the maintainer must first investigate which components will affect the modified component and which components will be affected by the modified component. This process is usually called *impact analysis*. By slicing a software architecture, the maintainer can extract the parts of a software architecture containing those components that might affect, or be affected by, the modified component. The slicing tool which provides such information can assist the maintainer greatly.

The primary idea of architectural slicing has been presented in [16, 17, 18], and this article can be regarded as an outgrowth of applying architectural slicing to support impact analysis of software architectures.

The rest of the paper is organized as follows. Section 2 briefly introduces how to represent a software architecture using WRIGHT: an architectural description language. Section 3 shows a motivation example. Section 4 describes some notions about architectural slicing. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 Software Architectural Specification in WRIGHT

We assume that readers are familiar with the basic concepts of software architecture and architectural description language, and in this paper, we use WRIGHT architectural description language [1] as our target language for formally representing software architectures. The selection of WRIGHT is based on that it supports to represent not only the architectural structure but also the architectural behavior of a software architecture.

Below, we use a simple WRIGHT architectural specification taken from [9] as a sample to briefly introduce how to use WRIGHT to represent a software architecture.

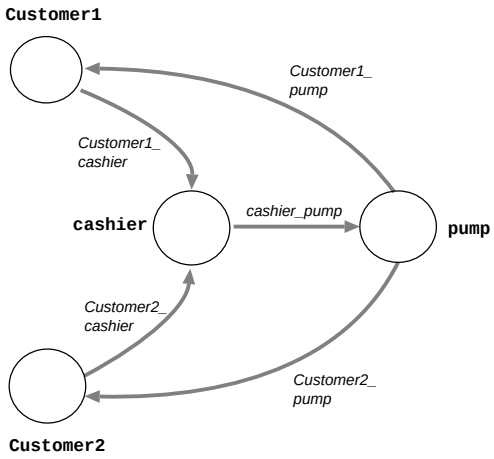


Figure 1: The architecture of the Gas Station system.

The specification is showed in Figure 2 which models the system architecture of a Gas Station system [4].

2.1 Representing Architectural Structure

WRIGHT uses a *configuration* to describe architectural structure as graph of components and connectors.

Components are computation units in the system. In WRIGHT, each component has an *interface* defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment.

Connectors are patterns of interaction between components. In WRIGHT, each connector has an *interface* defined by a set of *roles*. Each role defines a participant of the interaction represented by the connector.

A WRIGHT architectural specification of a system is defined by a set of component and connector type definitions, a set of instantiations of specific objects of these types, and a set of *attachments*. Attachments specify which components are linked to which connectors.

For example, in Figure 2 there are three component type definitions, `Customer`, `Cashier` and `Pump`, and three connector type definitions, `Customer_Cashier`, `Customer_Pump` and `Cashier_Pump`. The configuration is composed of a set of instances and a set of attachments to specify the architectural structure of the system.

2.2 Representing Architectural Behavior

WRIGHT models architectural behavior according to the significant events that take place in the computation of components, and the interactions between components as described by the connectors. The notation for specifying event-based behavior is adapted from CSP [5]. Each CSP process defines an alphabet of events and the permitted patterns of events that the process may exhibit. These processes synchronize on common events (i.e., interact) when composed in parallel. WRIGHT uses such process descriptions to describe the behavior of ports, roles, computations and glues.

A *computation* specification specifies a component's behavior: the way in which it accepts certain events on certain *ports* and produces new events on those or other

```
Configuration GasStation
    Component Customer
        Port Pay = pay!x → Pay
        Port Gas = take → pump?x → Gas
        Computation = Pay.pay!x → Gas.take → Gas.pump?x → Computation
    Component Cashier
        Port Customer1 = pay?x → Customer1
        Port Customer2 = pay?x → Customer2
        Port Topump = pump!x → Topump
        Computation = Customer1.pay?x → Topump.pump!x → Computation
            [] Customer2.pay?x → Topump.pump!x → Computation
    Component Pump
        Port Oil1 = take → pump!x → Oil1
        Port Oil2 = take → pump!x → Oil2
        Port Fromcashier = pump?x → Fromcashier
        Computation = Fromcashier.pump?x →
            (Oil1.take → Oil1.pump!x → Computation)
            [] (Oil2.take → Oil2.pump!x → Computation)
    Connector Customer_Cashier
        Role Givemoney = pay!x → Givemoney
        Role Getmoney = pay?x → Getmoney
        Glue = Givemoney.pay?x → Getmoney.pay!x → Glue
    Connector Customer_Pump
        Role Getoil = take → pump?x → Getoil
        Role Giveoil = take → pump!x → Giveoil
        Glue = Getoil.take → Giveoil.take → Giveoil.pump?x → Getoil.pump!x → Glue
    Connector Cashier_Pump
        Role Tell = pump!x → Tell
        Role Know = pump?x → Know
        Glue = Tell.pump?x → Know.pump!x → Glue
    Instances
        Customer1: Customer
        Customer2: Customer
        cashier: Cashier
        pump: Pump
        Customer1_cashier: Customer_Cashier
        Customer2_cashier: Customer_Cashier
        Customer1_pump: Customer_Pump
        Customer2_pump: Customer_Pump
        cashier_pump: Cashier_Pump
    Attachments
        Customer1.Pay as Customer1_cashier.Givemoney
        Customer1.Gas as Customer1_pump.Getoil
        Customer2.Pay as Customer2_cashier.Givemoney
        Customer2.Gas as Customer2_pump.Getoil
        casier.Customer1 as Customer1_cashier.Getmoney
        casier.Customer2 as Customer2_cashier.Getmoney
        cashier.Topump as cashier_pump.Tell
        pump.Fromcashier as cashier_pump.Know
        pump.Oil1 as Customer1_pump.Giveoil
        pump.Oil2 as Customer2_pump.Giveoil
End GasStation.
```

Figure 2: An architectural specification in WRIGHT.

ports. Moreover, WRIGHT uses an overbar to distinguish initiated events from observed events *. For example, the `Customer` initiates `Pay` action (i.e., `pay!x`) while the `Cashier` observes it (i.e., `pay?x`).

A *port* specification specifies the local protocol with which the component interacts with its environment through that port.

A *role* specification specifies the protocol that must be satisfied by any port that is attached to that role. Generally, a port need no have the same behavior as the role that it fills, but may choose to use only a subset of the connector capabilities. For example, the `Customer` role `Gas` and the `Customer_Pump` port `Getoil` are identical.

*In this paper, we use an underbar to represent an initiated event instead of an overbar that used in the original WRIGHT language definition [1].

A *glue* specification specifies how the roles of a connector interact with each other. For example, a `Cashier_Pump` tell (Tell.pump?x) must be transmitted to the `Cashier_Pump` know (`Know.pump!x`).

As a result, based on formal WRIGHT architectural specifications, we can infer which ports of a component are input ports and which are output ports. Also, we can infer which roles are input roles and which are output roles. Moreover, the direction in which the information transfers between ports and/or roles can also be inferred based on the formal specification. Such kinds of information can be used to construct the architectural flow graph of a software architecture for computing an architectural slice efficiently.

In this paper we assume that a software architecture be represented by a formal architectural specification which contains three basic types of design enti-

ties, namely, *components* whose interfaces are defined by a set of elements called *ports*, *connectors* whose interfaces are defined by a set of elements called *roles* and the *configuration* whose topology is declared by a set of elements called *instances* and *attachments*. Moreover, each component has a special element called *computation* and each connector has a special element called *glue* as we described above. In the rest of the paper, we assume that an architectural specification P be denoted by (C_m, C_n, c_g) where C_m is the set of components in P, C_n is the set of connectors in P, and c_g is the configuration of P.

3 Motivation Example

We present a simple example to explain our approach on change impact analysis for software architectures via architectural slicing.

Consider the Gas Station system whose architectural representation is shown in Figure 1, and WRIGHT specification is shown in Figure 2. Suppose a maintainer needs to modify the component `cashier` in the architectural specification in order to satisfy some new design requirements. The first thing the maintainer has to do is to investigate which components and connectors interact with component `cashier` through its ports `Customer1`, `Customer2`, and `Topump`. A common way is to manually check the source code of the specification to find such information. However, it is very time-consuming and error-prone even for a small size specification because there may be complex dependence relations between components in the specification. If the maintainer has an architectural slicer at hand, the work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, an architectural slicer is invoked, which takes as input: (1) a complete architectural specification of the system, and (2) a set of ports of the component `cashier`, i.e., `Customer1`, `Customer2` and `Topump` (this is an *architectural slicing criterion*). The slicer then computes a backward and forward architectural slice respectively with respect to the criterion and outputs them to the maintainer. A backward architectural slice is a partial specification of the original one which includes those components and connectors that might affect the component `cashier` through the ports in the criterion, and a forward architectural slice is a partial specification of the original one which includes those components and connectors that might be affected by the component `cashier` through the ports in the criterion. The other parts of the specification that might not affect or be affected by the component `cashier` will be removed, i.e., sliced away from the original specification. The maintainer can thus examine only the contents included in a slice to investigate the impact of modification.

4 Architectural Slicing

In this paper we assume that a software architecture be represented by a formal architectural specification which contains three basic types of design entities, namely, *components* whose interfaces are defined by a set of elements called *ports*, *connectors* whose interfaces are defined by a set of elements called *roles* and the *configuration* whose topology is declared by a set of elements called *instances* and *attachments*. Moreover, each component has a special element called *computation* and each connector has a special element called *glue* as we described above. In the rest of the paper, we assume that an architectural specification P be denoted by (C_m, C_n, c_g) where C_m is the set of components in P, C_n is the set of connectors in P, and c_g is the configuration of P.

Intuitively, an *architectural slice* may be viewed as a subset of the behavior of a software architecture, similar to the original notion of the traditional static slice. However, while a traditional slice intends to isolate the behavior of a specified set of program variables, an architectural slice intends to isolate the behavior of a specified set of a component or connector's elements. Given an architectural specification $P = (C_m, C_n, c_g)$, our goal is to compute an architectural slice $S_p = (C'_m, C'_n, c'_g)$ which consists of those components and connectors of P that preserve partially the semantics of P. we can give some notions of architectural slicing as follows.

In a WRIGHT architectural specification, for example, a component's interface is defined to be a set of ports which identify the form of the component interacting with its environment, and a connector's interface is defined to be a set of roles which identify the form of the connector interacting with its environment. To understand how a component interacts with other components and connectors to making changes, a maintainer must examine each port of the component of interest. Moreover, it has been frequently emphasized that connectors are as important as components for architectural design, and a maintainer may also want to modify a connector during the maintenance. To satisfy these requirements, we can define a slicing criterion for a WRIGHT architectural specification as a set of ports of a component or a set of roles of a connector of interest.

Let $P = (C_m, C_n, c_g)$ be an architectural specification. A *slicing criterion* for P is a pair (c, E) such that: (1) $c \in C_m$ and E is a set of elements of c, or (2) $c \in C_n$ and E is a set of elements of c.

Note that the selection of a slicing criterion depends on users' interests on what they want to examine. If they are interested in examining a component in an architectural specification, they may use slicing criterion 1. If they are interested in examining a connector, they may use slicing criterion 2. Moreover, the determination of the set E also depends on users' interests on what they want to examine. If they want to examine a component, then E may be the set of ports or just a subset of ports of the component. If they want to examine a connector, then E may be the set of roles or just a subset of roles of the connector.

Let $P = (C_m, C_n, c_g)$ be an architectural specification. A *backward architectural slice* S_{bp} of P on a given slicing criterion (c, E) is a set of those reduced components, connectors, and configuration that might directly or indirectly affect the behavior of c through elements in E. A *forward architectural slice* S_{fp} of P on a given slicing criterion (c, E) is a set of those reduced components, connectors, and configuration that might be directly or

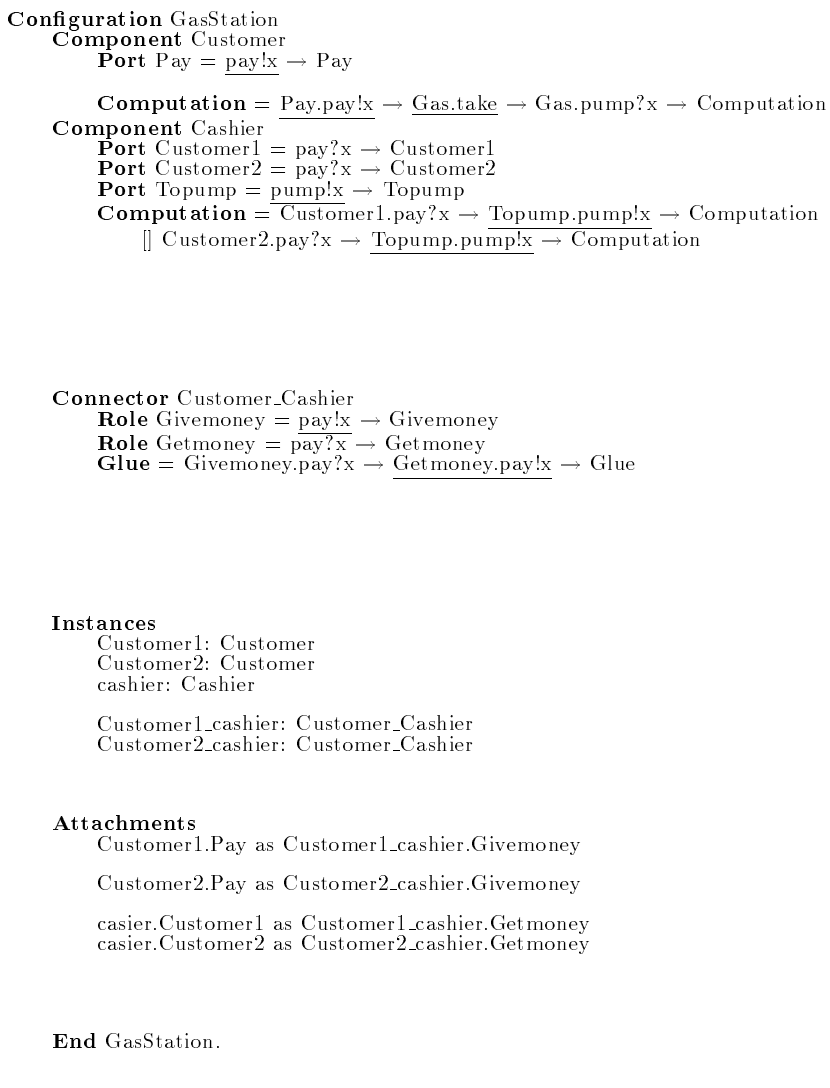


Figure 3: A backward slice of the architectural specification in Figure 2.

indirectly affected by the behavior of c through elements in E.

The view of an architectural slice defined above contains enough information for a maintainer to facilitate the modification.

The slicing notions defined here give us only a general view of an architectural slice, and do not tell us how to compute it. In [17, 18] we presented a two-phase algorithm to compute a slice of an architectural specification based on its information flow graph. Our algorithm contains two phases: (1) Computing a slice S_g over the information flow graph of an architectural specification, and (2) Constructing an architectural slice S_p from S_g.

Figure 3 shows a backward slice of the WRIGHT specification in Figure 2 with respect to the slicing criterion (`cashier`, E) such that E={`Customer1`, `Customer2`, `Topump`} is a set of ports of component `cashier`.

5 Related Work

Many researches have been done to support change impact analysis of software systems at the code level. Bohner and Arnold [2] recently edited a book which is a collection of many papers and articles related to change impact analysis of software systems at the code level. However, in comparison with code-level change impact analysis, the study of architectural-level change impact analysis of software systems has received little attention. To the best of our knowledge, the only work that is similar with ours is that presented by Stafford, Richardson and Wolf [13], who introduced a software architecture dependence analysis technique, called *chaining* to support software architecture development such as debugging and testing. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependences that can be followed during analysis. However, their technique is mainly focused on handling Rapide architectural description language in which connectors are not explicitly modeled.

6 Concluding Remarks

In this paper, we presented an approach for change impact analysis of software architectures based on *architectural slicing* technique. The main feature of our approach is to assess the effect of changes of a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

In architectural description languages, in addition to provide both a conceptual framework and a concrete syntax for characterizing software architectures, they also provide tools for parsing, displaying, compiling, analyzing, or simulating architectural specifications written in their associated language. However, existing language environments provide no tools to support architectural-level change impact analysis from an engineering viewpoint. We believe that such a tool should be provided by any ADL as an essential means to support software architecture development and evolution.

To demonstrate the usefulness of our impact analysis approach, we plan to implement an impact analysis tool for WRIGHT architectural descriptions to support architectural-level understanding and evolution.

References

[1] R. Allen, "A Formal Approach to Software Architecture," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1997.

[2] S. A. Bohner and R. S. Arnold, "Software Change Impact Analysis," IEEE Computer Society Press, 1996.

[3] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[4] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol.2, No.2, pp.47-57, 1985.

[5] C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985.

[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[7] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," *Proceedings of the International Conference on Software Maintenance*, 1993.

[8] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann, "Specification Analysis of System Architecture Using Rapide," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.336-355, April 1995.

[9] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J.Osterweil, "Applying Static Analysis to Software Architectures," *Proceedings of the Sixth European Software Engineering Conference*, LNCS, Vol.1301, pp.77-93, Springer-Verlag, 1997.

[10] H. D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, pp.17-25, March 1990.

[11] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.314-335, April 1995.

[12] M. Shaw and D. Garlan, "Software Architecture: Perspective on an Emerging Discipline," Prentice Hall, 1996.

[13] J. A. Stafford and A. L. Wolf, "Architecture-level Dependence Analysis for Software Systems," Technical Report CU-CS-913-00, Department of Computer Science, University of Colorado, December 2000.

[14] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," PhD thesis, University of Michigan, Ann Arbor, 1979.

[15] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple Effect Analysis of Software Maintenance," *Proc. of the COMPSAC'78*, pp.60-65, IEEE Computer Society Press, 1978.

[16] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.

[17] J. Zhao, " Software Architecture Slicing," *Proceedings of JSSST'97*, pp.49-52, September 1997

[18] J. Zhao, " Applying Slicing Technique to Software Architectures," *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, Monterey, USA, August 1998.

Understanding Software Evolution using a Flexible Query Engine

Michele Lanza, Stéphane Ducasse, Lukas Steiger
Software Composition Group, University of Berne
Neubrückstrasse 12, CH – 3012 Berne, Switzerland
{lanza,ducasse,steiger}@iam.unibe.ch — http://www.iam.unibe.ch/∼scg/

Submitted to the Formal Foundations of Software Evolution Workshop of CSMR 2001

Abstract

One of the main problems which arises in the field of software evolution is the sheer amount of information to be dealt with. Compared to reverse engineering where the main goal is the main understanding of one single system. In the field of software evolution this information is multiplied by the number of versions of the system one wants to understand. To counter this problem we have come up with a flexible query engine which can perform queries on the different versions of a system. In this paper we give an outlook on our current work in the field of software evolution and focus particularly on the concepts behind the query engine we have built.

Keywords: *Reverse Engineering, Evolution, Moose, Object-Oriented Programming*

1 Introduction

Understanding software systems that have evolved over several versions is difficult because of two main obstacles:

- The changes on a system during its development are often not or badly documented for several reasons. We believe one of the main forces is the weak enforcement of change documentation policies in companies: the people who perform the changes know what they are doing, so what's the point of documenting it?
- The original design document is not updated according to the performed changes, which leads to a rapid decay in the original design coherence.
- The amount of information is multiplied by the number of versions of the subject system: coping with such amounts of information is difficult and time-consuming.

Software Evolution is confronted with the difficulty of recovering such changes through the analysis of two or more versions of the same system. The main problem here is the amount of useless "noise" (i.e. false positives) which is returned.

To counter this problem we have come up with the idea of a flexible query engine similar to those used for professional databases. In a query language like SQL it is farily easy to define a query which can retrieve a certain set of data out of a possibly huge collection of data. Moreover it is also possible to further refine the query by adding more criteria.

This paper is structured as follows: in the next section we present the concepts and prerequisites of our query engine. We then show how the queries are made. Then we shortly present the tool which was realized using those concepts, and present some results obtained using the query engine on several case studies. In the final section of the paper we discuss the current and future work that we plan to do in this domain.

2 The Concepts and Prerequisites of the Query Engine

2.1 The Concept

The whole concept of such a query engine is based on the Composite Pattern[8]: The intent is to compose objects (in our case queries) into tree structures to represent part-whole hierarchies. A composite lets clients treat individual objects (queries) and compositions of objects (composed queries) uniformly.

A composed query can thus be seen as a hierarchy of queries and subqueries glued together by binary logical operators, i.e. AND and OR. A query can of course also be negated by assigning a unary NOT operator to the query. A

name can be assigned to a query, through which it can be included by reference in other queries.

2.2 The Prerequisites

A query engine like ours has some prerequisites which must be fulfilled. The following prerequisites must hold:

- **A Collection of Data.** The primary prerequisite for such a query engine is a collection of data which behaves like a database on which queries can be performed. In our case we have our reengineering environment Moose[7] that we have developed during the FAMOOS ESPRIT project[4]. Note that Moose keeps all entities in memory, instead of using a file based approach like a database. Although we know that a database is more scalable we have not encountered size problems until now.

  The Moose reengineering environment is an implementation of the language independent FAMIX metamodel[3]. At this time the following languages can be represented in our metamodel: Smalltalk, Java, C++ and COBOL.

  We parse the source code (directly in the case of Smalltalk and using parsers in the case of the other languages) and end up with a collection of entities which are an internal representation of software artifacts. In the context of evolution it is important that we can have several metamodels (e.g. several versions of the same software) parallel to each other at the same time in memory.

- **A Query Language.** Although we could have used Smalltalk as the query language, we have decided to build a textual query language which can be expressed at a graphical user interface level. The benefits of this are that non-Smalltalkers can also make use of it and a bigger ease of expression.

- **A Metrics Framework.** Most of the queries we perform are based on metric properties of the entities. For that purpose we have implemented a large framework of metrics (at this time more than 50), which is better explained in [9].

3 A Taxonomy of Queries

In this section we explain what kinds of queries we can build and how they can be composed into more complex ones. Note that the notation we use in this paper does not reflect the actual notation we use, which is much more verbose. For the sake of simplicity and readability we have decided on this easy-to-understand notation.

In this section we will show how with our query engine we can compose step by step a query which in the end will return the following result:

3.1 Basic Queries and Composite Queries

A basic query checks whether a certain condition holds or not, i.e. it iterates over one or several metamodels and returns entities which match the query. We now present how basic queries can be combined to compose a refined query which returns specific results. We distinguish four kinds of basic queries, i.e.

1. **Type Query**

   A type query returns all entities which belong to a certain type. The example below returns all classes of a system.

   ```
   ClassesQuery :=
   [Type(x) = CLASS]
   ```

2. **Name Query**

   This is a simple name matching query including wildcards. The example below returns all classes whose name contains the string “Abstract”.

   ```
   AbstractClassesQuery :=
   [ClassesQuery] AND
   [Name(x) = '*Abstract*']
   ```

3. **Property Query**

   In our metamodel we can annotate properties on an entity. Examples of such properties include whether a class is abstract, whether a method is an accessor (i.e. get/set), whether an attribute is private, etc. A property query tries to match a property which always returns a boolean value. The example below returns all classes which contain the substring “Abstract” in their name but in fact are not abstract.

   ```
   FalseAbstractClassesQuery :=
   [AbstractClassesQuery] AND
   [Property.Abstract(x) = FALSE]
   ```

4. **Metric Query**

   Moose provides an extensive set of metrics for the entities, including most of the metrics mentioned in [1] and [10]. In the case of such a query we either match the exact value or check on whether a metric value of an entity is above or below a certain threshold. The example below returns the false abstract classes in the system which implement more than 30 methods.

```
LargeFalseAbstrClassesQuery :=
[ FalseAbstractClassesQuery ]
AND
[ NOM(x) > 30 ]
```

3.2 Software Evolution Queries

A query can be composed of other (sub)queries. Those can be combined using binary logical operators, i.e. AND and OR like we have seen above.

In the case of Software Evolution Queries, we build queries which return results from different versions of the software and combine those results using logical unary (NOT) and binary (AND,OR) operators.

Suppose we have three versions of the software *Foo*. We call the versions *Foo1, Foo2, Foo3*. If we consider only *Foo1* and *Foo2*. We want to find all find all classes which from one version to next increased their number of methods by more than 20 (e.g. the class grew rapidly).

For that purpose we build the query

```
GrowQuery :=
[(NOM(x.new) - NOM(x.old)) > 20]
```

Here x represents the classes present in the new and the old version of the software and NOM is the value of the metric "Number of Methods" for x. This will return the results for (*Foo1, Foo2*). We can apply the same query to (*Foo2, Foo3*).

The combination of these through a logical AND operator will return the classes which grew constantly by at least 20 methods over the whole time frame we are considering. The combination of these through a logical OR operator will return the classes which grew at an arbitrary point in time.

3.3 Defining the Environment of a Query

Sometimes it is necessary to define a subquery on a query. We call this subquery the *environment* of the query. As an example, we want to find out all classes who grew by addition of methods and whose subclasses (at least one of them) shrank by removal of methods. Our guess is that in such a case the step in between performed by the developers was to push up the functionality of the subclasses into the superclass which grew. The criteria are in this case:

```
PushUpCandidates :=
[(NOM(x.new) - NOM(x.old)) > 0]
AND
[((NOM(subclasses(x.new) -
    NOM(subclasses(x.old)) < 0]
```

3.4 The Renamed Entity Tracking Problem

One of the major problems which must be dealt with, is that although conceptually two different versions of the same software entity have a "becomes" relationship, in our metamodel those are two different objects. To establish the connection between them, the obvious way is to go over the naming: if two entities have the same unique name, they are two versions of the same software artifact. However, what happens if an entity has been renamed?

We have found two simple and effective solutions to this problem which cover almost all cases:

1. Using the metrics. We compare the metric measurements of the "new" entities (i.e. those which have appeared for the first time in a certain version of the software) with those of the previous ones and see if there is a match. This solution is straight forward but not very effective.

2. In the case of classes or higher level software constructs like packages, etc. we go over the entities contained in them. As an example, in the case of a renamed class we check if we have a match regarding the methods: if the name of certain methods stays the same, but the unique name (i.e. including their class name) changes we can be nearly sure that we have a renamed class.

These two approaches work well enough for us, although in both cases there are false positives. However, the only bullet-proof way to track the renamed entities would be to have a versioning software which tracks all entities including the renamed ones.

4 The Implementation of the Query Engine

We have implemented the concept of the query engine in a tool called MooseFinder.

We have seen that the ease and flexibility of the query composition mechanism is very important: Often a query which works (i.e. returns useful results) in one context must be changed for another context.

For that purpose MooseFinder supports an easy and graphical way to compose queries including drag and drop support. This is necessary to enable the user to quickly adapt complex query structures to new contexts.

The window shown in Figure 1 is the main interface of MooseFinder. Here we can select the queries and run them.

The Query Composition Window shown in Figure 2 enables the user to build the basic queries and compose them into composite ones.

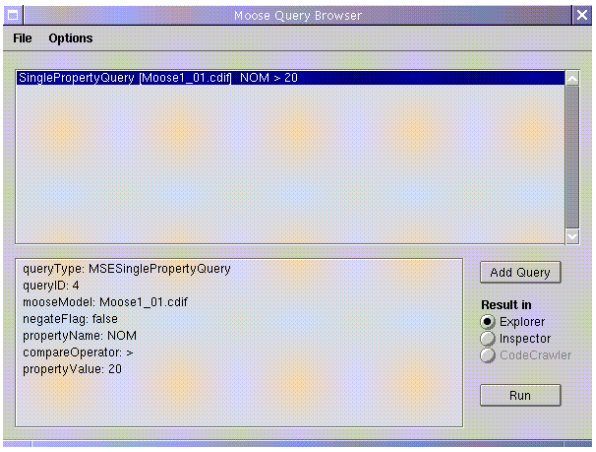


Figure 1. The Main Window of MooseFinder.

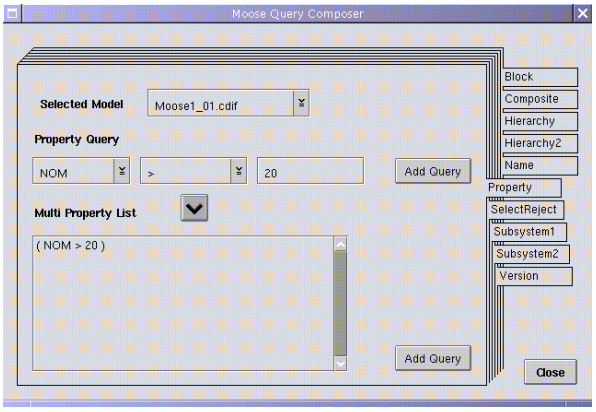


Figure 2. The Query Composition Window of MooseFinder.

5 Applying the Approach

The result of the approach we are working on, is to obtain a set of queries which return meaningful results in the field of software evolution. For that purpose we have set up a number of large and very large case studies we want to work on.

This work is still under way but we have already identified some useful queries. We list here what we can detect with each query:

- Introduction of a class on top of a large hierarchy
- Subclasses that become the sibling of their superclasses, i.e. that have been pushed up one hierarchy level
- Classes where methods and/or attributes have been pushed up into their superclass
- Classes that have rapidly grown/shrunk from one version to the next
- Classes which have been merged
- Entities which have been added to/removed from the software at a certain point
- Classes which have been renamed

6 Conclusions and Future Work

The preliminary results obtained using this approach have already shown that it is indeed useful and can return meaningful results. However, we have encountered the following problems:

- The usefulness of the approach is tied to the flexibility and power of the query language. This is on one hand the query language per se, on the other hand the user interface with which we can compose the queries.
- This approach goes into the direction of data mining and data reverse engineering. One of the main problems in those fields is the representation of the results. For the time being we still use textual representations, although we can easily interface with visualization software.
- The more general and less specific a query is, the more results it will return. On the other hand a very specific query can return an empty set of results. The fine-tuning of the queries requires a considerable deal of expertise on side of the user and flexibility on side of the query engine.

Our future work in this context includes the publication of a paper with the major results obtained with this approach applied on several large and very large case studies.

Furthermore we will extend the query engine and its query language to render it as flexible and powerful as possible.

We also plan to use the software visualization tool CodeCrawler [9, 2, 5] in this context.

References

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[2] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[3] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, Aug. 1999.

[4] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Berne, Oct. 1999. See http://www.iam.unibe.ch/~famoos/handbook.

[5] S. Ducasse and M. Lanza. Towards a reverse engineering methodology for object-oriented systems. *Technique et Science Informatique*, 2001. To appear in Techniques et Sciences Informatiques, Edition Speciale Reutilisation.

[6] S. Ducasse, M. Lanza, and L. Steiger. A query-based approach to support software evolution. In *ECOOP'2000 International Workshop of Architecture Evolution*, 2000.

[7] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[9] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, Oct. 1999.

[10] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

A Graph Transformation Approach to Architectural Run-Time Reconfiguration

Michel Wermelinger
Departamento de Informática
Fac. de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica, Portugal
mw@di.fct.unl.pt

Antónia Lopes and José Luiz Fiadeiro
Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
mal@di.fc.ul.pt jose@fiadeiro.org

Abstract

The ability of reconfiguring software architectures in order to adapt them to new requirements or a changing environment has been of growing interest. We propose a uniform algebraic approach that improves on previous formal work in the area due to the following characteristics. First, components are written in a high-level program design language with the usual notion of state. Second, the approach deals with typical problems such as guaranteeing that new components are introduced in the correct state (possibly transferred from the old components they replace) and that the resulting architecture conforms to certain structural constraints. Third, reconfigurations and computations are explicitly related by keeping them separate. This is because the approach provides a semantics to a given architecture through the algebraic construction of an equivalent program, whose computations can be mirrored at the architectural level.

1 Introduction

1.1 Motivation

One of the topics which is raising increased interest in the Software Architecture (SA) community is the ability to specify how a SA evolves over time, in particular at run-time, in order to adapt to new requirements or new environments, to failures, and to mobility. There are several issues at stake, among them:

modification time and source Architectures may change before execution, or at run-time (called dynamic reconfiguration). Run-time changes may be triggered by the current state or topology of the system (called programmed reconfiguration [6]) or may be requested unexpectedly by the user (called ad-hoc reconfiguration [6]).

modification operations The four fundamental operations are addition and removal of components and connections. Although their names vary, those operators are provided by most reconfiguration languages (like [6, 15, 1]). In programmed reconfiguration, the changes to perform are given with the initial architecture, but they may be executed when the architecture has already changed. Therefore it is necessary to query at run-time the state of the components and the topology of the architecture.

modification constraints Often changes must preserve several kinds of properties: structural (e.g., the architecture has a ring structure), functional, and behavioural (e.g., real-time constraints).

system state The new system must be in a consistent state.

1.2 Related Work

There is a growing body of work on architectural reconfiguration, some of it related to specific Architecture Description Languages (ADL), and some of formal, ADL-independent nature. Most of the proposals exhibit one of the following drawbacks.

- Arbitrary reconfigurations are not possible: Darwin [13] only allows component replication; ACME [18] only allows optional components and connections; Wright [1] requires the number of distinct configurations to be known in advance; [11] use

context-free reconfiguration rules, which does not permit to create a new connection between exiting components, for example.

- The languages to represent computations are very simple and at a low level: rewriting of labels [11], process calculi [16, 2, 1], term rewriting [20, 8], graph rewriting [19]. They do not capture some of the abstractions used by programmers and often lead to cumbersome specifications.

- The combination of reconfiguration and computation, needed for run-time change, leads to additional formal constructs: [11] uses constraint solving, [16, 1, 2] define new semantics or language constructs for the process calculi, [8] must dynamically change the rewriting strategies, [19] imposes many constraints on the form of graph rewrite rules because they are used to express computation, communication, and reconfiguration. This often results in a proposal that is not very uniform, or has complex semantics, or does not make the relationship between reconfiguration and computation very clear.

1.3 Approach

To overcome these disadvantages, we have proposed an algebraic framework [22] using categorical diagrams to represent architectures, the double-pushout graph transformation approach[1] [5] to describe reconfigurations, and a program design language with explicit state to describe computations.

In this paper we refine our approach, introducing the notions of productive reconfiguration step and architectural style. To accommodate the latter, we have made the underlying mathematical definitions (not shown in this extended abstract) more uniform, based on the category of typed graphs [4], a generalisation of labelled graphs. Moreover, we cope with ad-hoc reconfiguration.

The running example is an airport luggage distribution system. One or more carts move continuously in the same direction on a N-units long circular track. A cart advances one unit at each step. Carts must not bump into each other. This is achieved by changing the movement interactions between carts, depending on their location. Reconfigurations may be due not only to mobility but also to component upgrade: a cart may be replaced by one with a built-in lap counter.

[1] To make the paper self-contained, the appendix contains an informal summary of the needed mathematical definitions.

2 CommUnity

2.1 Programs

CommUnity [7] is a parallel program design language based on Unity [3] and IP [9]. A program consists of a set of typed input and output variables, a boolean expression to be satisfied by the initial values of the output variables, and a set of actions, each of the form *name*: *guard* $\rightarrow$ *assignment(s)*. Action names act as *rendez-vous* points for program synchronisation (see Section 3). The empty set of assignments is denoted by `skip`. At each step, one of the actions is selected and, if its guard—a boolean expression over the variables—is true, its assignments are executed simultaneously. The values of the input variables are given by the environment and may change at each step. Input variables may not be assigned to by the program.

The next program describes the behaviour of a cart.

```
prog Cart
out    l : int
init   0 ≤ l < N
do     move: true → l := (l + 1) mod N
```

Henceforth we abbreviate "(l + 1) mod N" as "l $+_N$ 1" and omit the action guards when they are "true".

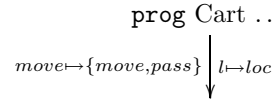
To take program state into account, we introduce a fixed set of typed variables, called logical variables. For the rest of the paper, it is the set $\{i, j : \text{int}, n : \text{nat}\}$. A program instance is then defined as a program together with a valuation function that assigns to each output variable a term (over logical variables) of the same type. No valuation is assigned to input variables because those are not under control of the program. Notice also that the valuation may return an arbitrary term, not just a ground term. Although in the running system the value of each variable is given by a ground term, we need variables to be able to write reconfiguration rules whose left-hand sides match components with possibly infinite distinct combinations of values for their variables. We represent program instances in tabular form (see below).

2.2 Superposition

A morphism from a program P to a program P' states that P is a component of the system P' and, as shown in [7], captures the notion of program superposition [3, 9]. Mathematically speaking, the morphism maps each variable of P into a variable of P' of the same type—such that output variables of the component P are mapped to output variables of the system

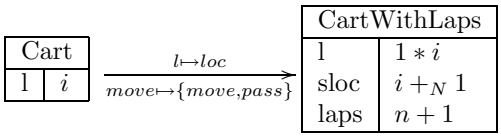
P'—and it maps each action name a of P into a (possible empty) set of action names $\{a'_1, \ldots, a'_n\}$ of P' [21]. Those actions correspond to the different possible behaviours of a within the system P'. Thus each action a'_i must preserve the functionality of a, possibly adding more things.

The next diagram shows in which way program "Cart" can be superposed with a counter that checks how often the cart passes by its start position. Notice how the second program strengthens the initialisation condition and it divides action "move" in two sub-cases.



`prog`	CartWithLaps
`out`	loc, sloc, laps : int
`init`	$0 \leq \text{loc} < N \wedge \text{sloc} = \text{loc} \wedge \text{laps} = 0$
`do`	pass: $\text{loc} +_N 1 = \text{sloc}$
	$\rightarrow \text{loc} := \text{loc} +_N 1 \parallel \text{laps} := \text{laps} + 1$
[]	move: $\text{loc} +_N 1 \neq \text{sloc} \rightarrow \text{loc} := \text{loc} +_N 1$

A morphism between program instances is simply a superposition morphism that preserves the state. To be more precise, if an output variable of P is mapped to an output variable of P', their valuations must be the same for any substitution of the logical variables. An example is



where the instance on the right represents a cart that has completed at least one lap and will complete another one in the next step.

3 Architectures

3.1 Configurations

Interactions between programs are established through action synchronisation and memory sharing. This is achieved by relating the relevant action and variable names of the interacting programs.

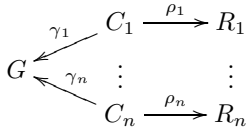
The categorical framework imposes the locality of names. To state that variable (or action) a_1 of program P_1 is the same as variable (resp. action) a_2 of P_2 one needs a third, "mediating" program C—the channel—containing just a variable (resp. action) a and two morphisms $\sigma_i : C \rightarrow P_i$ that map a to a_i. A channel has no computations of its own. Therefore it has no output variables (hence no assignments nor initialisation condition) and all actions have true guards. We abbreviate a channel as $\langle I \mid A \rangle$, where I is the set of input variables and A is the set of action names.

Problems arise if two synchronised actions update a shared variable in distinct ways. As actions only change the values of output variables, it is sufficient to impose that output variables are not shared, neither directly through a single channel nor indirectly through a sequence of channels. We call such diagrams configurations. This restriction forces interactions between programs to be synchronous communication of values (from output to input variables), a very general mode of interaction that is suitable for the modular development of reusable components, as needed for architectural design.

It can be proved that every finite configuration has a colimit, which returns the minimal program that simulates the execution of the overall system. Briefly put, the colimit is obtained by taking the disjoint union of the variables (modulo shared variables), the cartesian product of actions (modulo synchronized ones)—to denote parallel execution of non-synchronised actions—, and the conjunction of the initialisation conditions. Actions are synchronized by taking the conjunction of the guards and the parallel composition of assignments. An example is provided in the next section.A configuration instance is a configuration whose nodes are program instances. Since output variables are not shared, they have no conflicting valuations. Therefore every configuration instance has a colimit, given by the colimit of the underlying configuration together with the union of the valuations of the program instances.

3.2 Connectors

SA has put forward the notion of connector to encapsulate the interactions between components. An n-ary connector consists of n roles R_i and one glue G stating the interaction between the roles. These act as "formal parameters", restricting which components may be linked together through the connector. We represent a connector by a diagram of the form



where the channels indicate which variables and actions of the roles are used in the interaction specification, i.e., the glue. An n-ary connector can be applied to components $P_1, \ldots, P_n$ when morphisms $\iota_i : R_i \rightarrow P_i$ exist. This corresponds to the intuition that the "actual ar-

guments" (i.e., the components) must instantiate the "formal parameters" (i.e., the roles).

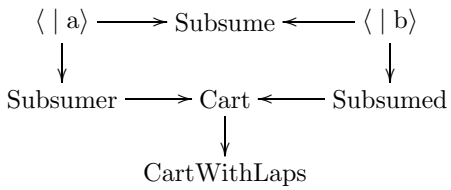
An architecture (instance) is then a configuration (instance) where all components interact through connectors, and all roles are instantiated. Hence any architecture has a semantics given by its colimit, which returns the minimal program that simulates the execution of the overall system.

To avoid a cart c_1 colliding with the cart c_2 right in front of it we only need to make sure that if c_1 moves, so must c_2, but the opposite is not necessary. We say action a subsumes action b if b executes whenever a does. This can be seen as a partial synchronisation mechanism: a is synchronised with b, but b can still execute freely. The diagram in Figure 1 shows the application of the generic action subsumption connector to two carts and the resulting colimit. Notice that although the two roles are isomorphic, the binary connector is not symmetric because the channel morphisms and the glue treat the two actions differently: "b" may be executed alone at any time, while "a" must co-occur with "b".

3.3 Style

In general, a role may be instantiated by different components, and it may be even the case that the same component can instantiate the same role in different ways (e.g., if 'Cart' had other actions). But normally only a few of all the possibilities are meaningful to the application at hand. The allowed ways to apply connectors to components can be described by typed graphs. This leads to a declarative notion of architecture style: it consists of a set of components, a set of connectors, and a diagram T in the category of programs and superposition morphisms using only those connectors and components. Every architecture written by the user must then come equipped with a morphism to T proving that it obeys the restrictions imposed by T. As for an architecture instance, it is well-typed if the underlying architecture, obtained by forgetting the valuations, is. We believe that this approach to architectural styles, besides being simple to use, is also sufficient in many occasions, namely when only the kinds of interactions between the given components have to be restrained. Abstract architectural patterns (e.g., pipe-filter, layer) cannot be described with our approach.

For our example, the set of components is 'Cart' and 'CartWithLaps', the set of connectors is just the action subsumption connector shown before, and the architecture type T (with morphisms as shown in previous diagrams) is



stating that the connector may be applied to carts only, which in turn may be refined with a lap counter.

Notice that a style T, by showing all possible morphisms that may occur in an architecture, also restricts the visibility of variables, stating which output variables are to be shared (and how) and which are private to each program.

It is important to notice that T is not necessarily a configuration: since it shows in a single diagram all morphisms that may occur in architectures, it may happen that output variables are shared in T.

4 Dynamic Reconfiguration

Basically, we represent dynamic reconfiguration as a rewriting process over graphs with nodes labelled by program instances and arcs labelled by instance morphisms. In essence, a reconfiguration rule is a graph production, and a reconfiguration step is a direct derivation. This ensures that the state of components and connectors that are not affected by a rule does not change, because node labels (which include the variables' valuations) are preserved, thus keeping reconfiguration and computation separate. However, we must make slight adaptations of the basic graph transformation framework to our setting.

First, in the double-pushout approach, there is no restriction on the obtained graphs, but in reconfiguration we must check that the result is indeed an architecture, otherwise the rule (with the given match) is not applicable. Without this restriction, it would be possible for a rule to introduce a connector that would lead to sharing of output variables, for example.

Second, it should not be possible to apply the same rule in the same way (i.e., to the same program instances) more than once because that would lead to infinite reconfiguration sequences. To this end we restrict the allowed reconfiguration sequences by considering only productive direct derivations $G \overset{p,m}{\Longrightarrow} H$: there are no graph morphisms $lr : L \to R$ and $x : R \to G$ such that $lr; x = m$. The existence of lr shows that production p does not delete any nodes or arcs. The remaining conditions check that the match is being applied to a part of G that corresponds to the right-hand side R

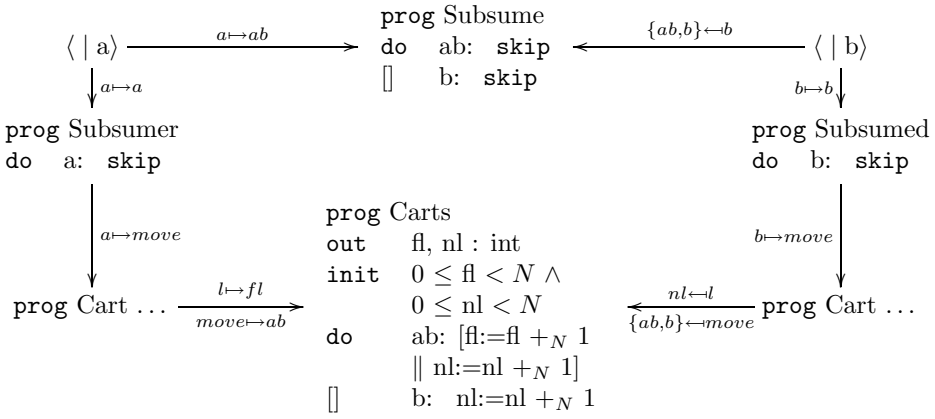


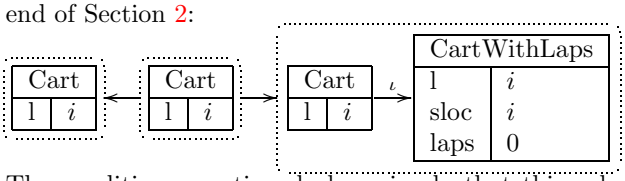
Figure 1. An applied action subsumption connector and its colimit

and therefore can have been generated by a previous application of this production. Our definition is a particular case of productions with application conditions [10]: a derivation is productive if p is applicable to G using the negative application condition lr.

Third, dynamic reconfiguration rules must be conditional, because they depend on the current state. Thus they are of the form $L \xleftarrow{l} K \xrightarrow{r} R$ **if** B, with B a proposition over the logical variables occurring in L. Moreover, a rule can only be applied if every new component added by the rule is in a precisely determined state that satisfies the initialisation condition, in order to be able to perform computations right away. For that purpose, we require that the logical variables occurring in R also occur in L. The definition of reconfiguration step must be changed accordingly. At any point in time the current system is given by an architecture instance whose valuations return ground terms. Therefore the notion of matching must also involve a compatible substitution of the logical variables occurring in the rule by ground terms. If we apply the substitution to the whole rule, we obtain a rule without logical variables that can be directly applied to the current architecture using the normal definition of derivation as a double pushout over labelled graphs. However, the notion of state introduces two constraints. First, the substitution must obviously satisfy the application condition B. Second, the derivation must make sure that the state of each program instance added by the right-hand side satisfies the respective initialisation condition.

Returning to our example, to avoid collisions we give in Figure 2 a rule that applies the action subsumption connector to two carts that are less than 3 units apart, where the graph morphisms l and r are obvious. The opposite rule (with the negated condition) is necessary to remove the connector when no longer needed.

As a second example, if we want to add a counter to a cart, no matter which connectors it is currently linked to, we just unconditionally superpose the 'CartWithLaps' program on it, with ι the morphism shown at the end of Section 2:



The conditions mentioned above imply that this rule can only be applied with a substitution that satisfies $0 \le i \le N$. This example illustrates how to describe the transfer of state from old to new components. In this case it is just a copy of value i, but in general the right-hand side may contain arbitrarily complex terms that calculate the new values from the old ones.

If there is an architectural style T, then the three architecture instances in a reconfiguration rule must be typed by T. It can be proved that the graph obtained through direct derivation is also well-typed.

To coordinate computations and reconfigurations, the run-time infrastructure executes the following sequence:

1. allow the user to change the style and the set of reconfiguration rules;
2. find a maximal sequence of reconfiguration steps starting with the current architecture instance A, obtaining A';
3. compute the colimit S of A';

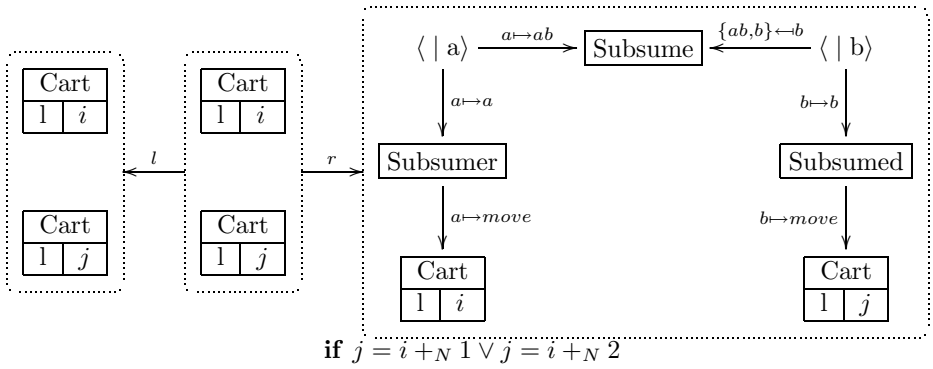


Figure 2. Introduction of the action subsumption connector

4. if none of the S's actions can be executed, stop, otherwise update the values of S's variables according to the chosen action;

5. propagate through the colimit morphisms the changes back to the variables of the program instances of A', call the new diagram A, and go to step 1.

The first step caters for ad-hoc reconfiguration. In our example, it allows to add the CartWithLaps program to the style and to add the last rule shown. Step 5 keeps the state of the program instances in the architectural diagram consistent with the state of the colimit, and ensures that at each point in time the correct conditional rules are applied. As [14, 11] we adopt a two-phase approach: computations (step 4) are interleaved with reconfiguration sequences (step 2). In this way, the specification of the components is simpler, because it is guaranteed that the necessary interconnections are in place as soon as required by the state of the components.

5 Concluding Remarks

We have refined our algebraic foundation for dynamic software architecture reconfiguration. Our approach has several advantages over previous work [11, 16, 1, 2, 8, 20]:

- context-dependent rewriting allows arbitrary reconfigurations;
- computations (on a program) and reconfigurations (on an architecture) are explicitly related through a colimit operation, because we do not rewrite just graphs, but diagrams in a category of programs with superposition;
- the maintenance of state consistency during reconfiguration—how to transfer state, in which state reconfigurations are possible, what is the state of new components—is straightforward to specify, due to the use of a program design language that is more natural than terms, process calculi, or graphs, leading to easy to read rules.

The algebraic graph transformation approach combines well with our categorical framework for architectural design and has several advantages: it enforces that component state is only changed by computations, not by reconfiguration steps; the application conditions of the double-pushout approach enforce that components are not removed while linked to connectors, thus not leaving "dangling" roles (not shown in this abstract); the negative application conditions can be used to avoid useless changes to the architecture; typed graphs provide, besides a uniform mathematical basis, a declarative and simple notion of style—sufficient to describe certain structural modification constraints—that can be automatically maintained during reconfiguration.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 21–37. Springer-Verlag, 1998.

[2] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Kluwer Academic Publishers, 1999.

[3] K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamentae Informatica*, 26(3–4):241–266, 1996.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical Report TR-96-17, University of Pisa, Mar. 1996.

[6] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[7] J. L. Fiadeiro and T. Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[8] J. L. Fiadeiro, M. Wermelinger, and J. Meseguer. Semantics of transient connectors in rewriting logic. Position Paper for the First IFIP Working International Conference on Software Architecture, Feb. 1999.

[9] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[10] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4), 1996.

[11] D. Hirsch, P. Inverardi, and U. Montanari. Modelling software architectures and styles with graph grammars and constraint solving. In *Software Architecture*, pages 127–143. Kluwer Academic Publishers, 1999.

[12] M. Löwe. Algebraic approach to graph transformation based on single pushout derivations. Technical Report 90/5, Technische Universität Berlin, Fachbereich 13, Informatik, 1990.

[13] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[14] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2), Feb. 1998.

[15] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.

[16] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[17] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[18] R. T. Monroe, D. Garlan, and D. Wile. *Acme Straw-Manual*, Nov. 1997.

[19] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, 1998.

[20] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings—Software*, 145(5):130–136, Oct. 1998.

[21] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[22] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *Software Engineering—ESEC/FSE'99*, volume 1687 of *LNCS*, pages 393–409. Springer-Verlag, 1999.

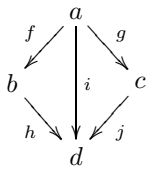
A Mathematical Definitions

A.1 Category Theory

Category Theory [17] is the mathematical discipline that studies, in a general and abstract way, relationships between arbitrary entities. A category is a collection of objects together with a collection of morphisms between pairs of objects. A morphism f with source object a and target object b is written $f : a \to b$ or $a \xrightarrow{f} b$. Morphisms come equipped with a composition operator ";" such that if $f : a \to b$ and $g : b \to c$ then $f;g : a \to c$. Composition is associative and has identities id_a for every object a.

Diagrams are directed graphs—where nodes denote objects and arcs represent morphisms—and can be used to represent "complex" objects as configurations of smaller ones. For categories that are well behaved, each configuration denotes an object that can be retrieved through an operation on the diagram called colimit. Informally, the colimit of a diagram returns the "minimal" object such that there is a morphism from every object in the diagram to it (i.e., the colimit contains the objects in the diagram as components) and the addition of these morphisms to the original configuration results in a commutative diagram (i.e., interconnections, as established by the morphisms of the configuration diagram, are enforced).

Pushouts are colimits of diagrams of the form $b \xleftarrow{f} a \xrightarrow{g} c$. By definition of colimit, the pushout returns an object d such that the diagram



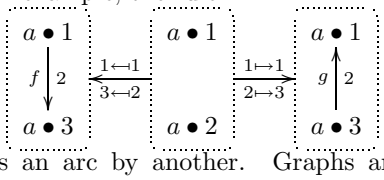
exists and commutes (i.e., $f;h = i = g;j$). Furthermore, for any other pushout candidate d', there is a unique morphism $k : d \to d'$. This ensures that d, being a component of any other object in the same conditions, is minimal. Object c is called the pushout complement of diagram $a \xrightarrow{f} b \xrightarrow{h} d$.

A.2 Graph Transformation

The algebraic approach to graph transformation [5] was introduced over 20 years ago in order to generalize

grammars from strings to graphs. Hence it was necessary to adapt string concatenation to graphs. The approach is algebraic because the gluing of graphs is done by a pushout in an appropriate category. There are two main variants, the double-pushout approach [5] and the single-pushout approach [12]. We only use the former. It is based on a category whose objects are labelled graphs and whose morphisms $f : a \to b$ are total maps (from a's nodes and arcs to those of b) that preserve the labels and the structure of a.

A graph transformation rule, called graph production, is simply a diagram of the form $L \xleftarrow{l} K \xrightarrow{r} R$ where L is the left-hand side graph, R the right-hand side graph, K the interface graph and l and r are injective graph morphisms. The rule states how graph L is transformed into R, where K is the common subgraph, i.e., those nodes and arcs that are not deleted by the rule. As an example, the rule



substitutes an arc by another. Graphs are written within dotted boxes to improve readability. Nodes and arcs are numbered uniquely within each graph to show the mapping done by the morphisms.

A production p can be applied to a graph G if the left-hand side can be matched to G, i.e., if there is a graph morphism $m : L \to G$. A direct derivation from G to H using p and m exists if the diagram

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

can be constructed, where each square is a pushout. Intuitively, first the pushout complement D is obtained by deleting from G all nodes and arcs that appear in L but not in K. Then H is obtained by adding to D all nodes and arcs that appear in R but not in K. The fact that l and r are injective guarantees that H is unique. An example derivation using the previously given production is Figure 3.

A direct derivation is only possible if the match m obeys two conditions. First, if the production removes a node $n \in L$, then each arc incident to $m(n) \in G$ must be image of some arc attached to n. Second, if the production removes one node (or arc) and maintains another one, then m may not map them to the same node (or arc) in G.

Two examples in which the match violates these conditions are represented by the following diagrams, where $\emptyset$ is the empty graph.

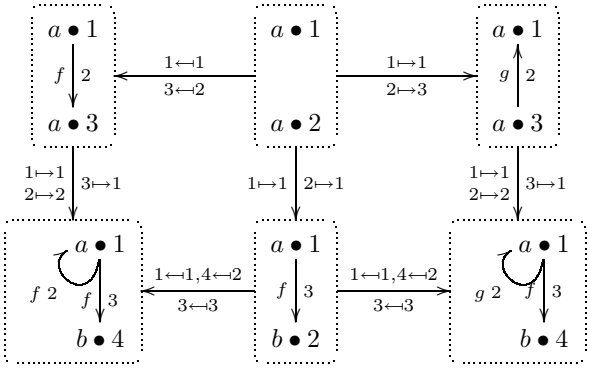
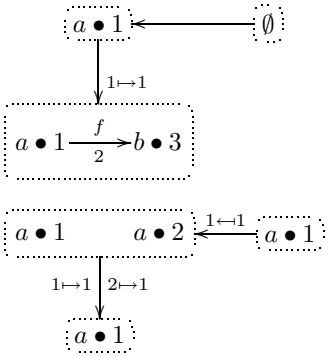


Figure 3. Applying a graph production



Both conditions are quite intuitive. The first one prevents dangling arcs, the second one avoids contradictory situations. Both allow an unambiguous prediction of removals. A node of G will be removed only if its context (i.e., adjacent arcs and nodes) are *completely* matched by the left-hand side of some production. The advantage is that the production specifier can control exactly in which contexts a node is to be deleted. This means it is not possible to remove a node no matter what other nodes are linked to it.

Transformational Software Evolution by Assertions

Dr. Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
Tom.Mens@vub.ac.be

ABSTRACT

This paper explores the use of software transformations as a formal foundation for software evolution. More precisely, we express software transformations in terms of assertions (preconditions, postconditions and invariants) on top of the formalism of graph rewriting. This allows us to tackle scalability issues in a straightforward way. Useful applications include: detecting syntactic merge conflicts, removing redundancy in a transformation sequence, factoring out common subsequences, etc.

1. INTRODUCTION

Software evolution is one of the most important problems in software engineering, because of its inherent complexity and because of the lack of a solid formal foundation. In an attempt to provide such a foundation, this paper elaborates on the paradigm of *transformational software evolution*. In this paradigm, evolution is achieved by means of explicit software transformations that can be manipulated directly. This gives rise to a wide range of interesting ways to improve support for evolution.

One area of interest lies in support for merging parallel evolutions of the same software [3, 9]. Software merging is needed when separate lines of software development are carried out in parallel and have to be merged at regular intervals. Because this is a complex time-consuming process, automated support tools are essential. Unfortunately, most existing merge tools either lack flexibility or expressive power. To counter this problem, we need to establish the formal foundations of software merging first. For this purpose, graph rewriting appears to be a promising lightweight formalism [11].

Software transformations are also useful to provide support for refactoring application frameworks in a behaviour-preserving way. Refactorings improve the design or structure of object-oriented frameworks, making them more robust towards evolution [13, 14, 16].

For merging as well as refactoring, there is a need to express evolution transformations in a scalable way. Indeed, in practice, the software that is being developed as well as the software transformations that are applied to it can be quite large.

A promising formal approach which has not yet been thoroughly explored is the use of *assertions* for expressing software transformations. In [16], pre- and postconditions were used to express refactoring transformations. In [11], pre- and postconditions were attached to software transformations to detect merge conflicts. This paper performs a more thorough investigation, and shows how assertions allow us to express software transformations in a uniform and scalable way.

2. CONDITIONAL GRAPH REWRITING

We represent software artifacts (whether it be analysis, architecture, design or implementation artifacts) in a uniform way as graphs [10]. This enables us to use the powerful formalism of *conditional graph rewriting* [4, 5, 6, 11] for representing evolution transformations.

2.1 Graphs and Graph Rewriting

Graphs provide a simple yet expressive formalism for representing software. *Nodes* in a graph can represent any kind of software entity (classes, modules, objects, methods, variables, statements, etc...), while *edges* express dependencies between these entities (data-flow, control-flow, containment relationships, etc...). Each node and edge has a *label* and a *type* attached to it.

Definition. Let *NodeID* be the set of node identifiers, *EdgeID* the set of edge identifiers, *Label* the set of node and edge labels, and *Type* the set of node and edge types. A **graph** G is a tuple $(V, E, source, target, label, type)$ consisting of a node set $V \subseteq NodeID$ and an edge set $E \subseteq EdgeID$ with $V \cap E = \varnothing$; functions $source: E \rightarrow V$ and $target: E \rightarrow V$; and functions $label: V \cup E \rightarrow Label$ and $type: V \cup E \rightarrow Type$.

For example, in graph R depicted in Figure 1, $V=\{a,c\}$, $E=\{f\}$, $label(a)$=area, $type(a)$=***operation***, $label(f)$=*uses*, $type(f)$=***uses***, $source(f)=a$ and $target(f)=c$. We distinguish types from labels by writing types in boldface.

Since graphs represent software artifacts, evolution of these artifacts can be expressed by *graph rewriting*. Because we will manipulate graph rewritings explicitly, they should be decoupled from the actual graphs to which they are being applied. This is achieved by introducing the notion of a ***graph production*** $P: L \rightarrow R$ that transforms a source graph L into a target graph R. In order to apply this production to an initial graph G, a match $m: L \rightarrow G$ is needed to specify which part of the initial graph G is

---

* Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium) (F.W.O.-Vlaanderen)

[3] Primitive productions *Relabel* and *Retype* can be used for nodes as well as edges. We often use the notation *RelabelN* and *RetypeN* (resp. *RelabelE* and *RetypeE*) to stress that we are changing the label or type of a node (resp. edge).

being transformed. Together, *P* and *m* uniquely define a ***graph rewriting*** $G \Rightarrow_{P,m} H$. This graph rewriting also induces a co-match *m*: R→H* that specifies the embedding of *R* in the result graph *H*.

As an example, consider the graph rewriting of Figure 1. The match *m: L→G* maps node *a* of *L* on node *2* of *G*. The co-match *m*: R→H* additionally maps node *c* of *R* on node *3* of *H*, and edge *f* of *R* on edge *f* of *H*.

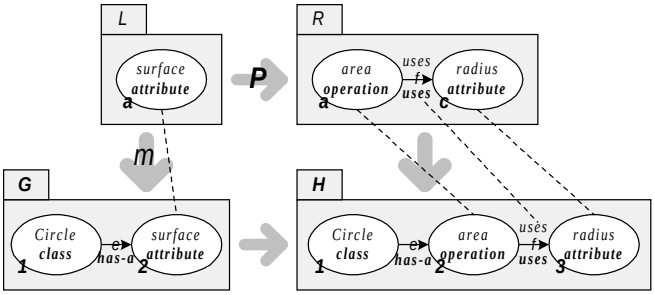


Figure 1: An example of a graph rewriting

2.2 Assertions

Assertions are well established in the software community as a formal way to specify the behaviour of programs [7, 12]. Three kinds of assertions are distinguised. *Preconditions* must be satisfied for a certain operation to be applicable. *Postconditions* are guaranteed to be true after the operation has been applied. *Invariants* are assumptions that remain unaltered by the operation.

Another distinction is made between *positive* assertions, that indicate the presence of a certain property, and *negative* assertions that indicate its absence. Table 1 presents the positive assertions that can be expressed in our graph formalism, together with the notation used throughout this paper. Negative assertions are precisely the opposite: they express the absence of some entity in a graph, and are denoted by a minus sign. E.g., *–source(E,N)* expresses that edge *E* does not have node *N* as its source.

Table 1: Positive assertions

Positive assertion	Notation
A node or edge with identifier *Id* should be present	*+Id*
Edge *E* should have node *N* as its source	*+source(E,N)*
Edge *E* should have node *N* as its target	*+target(E,N)*
A node or edge *Id* should have label *L*	*+label(Id,L)*
A node or edge *Id* should have type ***T***	*+type(Id,**T**)*

We also want to express more general constraints like: "node *N* does *not* have *any* outgoing edges" or "node *N* is the target of *at least one* edge". The former constraint is expressed as *-source(*,N)*, and the latter as *+target(*,N)*. All positive wildcard assertions used in this paper are enumerated in Table 2. Negative wildcard assertions are merely the negation of their positive equivalents. For example, *-source(*,N)* is the negation of $\exists E \in EdgeID: source(E) = N$, i.e., $\forall E \in EdgeID: source(E) \neq N$

Table 2: Positive wildcard assertions

Positive assertion	Notation
$\exists E \in EdgeID: source(E) = N$	*+source(*,N)*
$\exists E \in EdgeID: target(E) = N$	*+target(*,N)*
$\exists N \in NodeID: source(E) = N$	*+source(E,*)*
$\exists N \in NodeID: target(E) = N$	*+target(E,*)*
$\exists L \in Label: label(Id) = L$	*+label(Id,*)*
$\exists \mathbf{T} \in Type: type(Id) = \mathbf{T}$	*+type(Id,*****)*

Some assertions automatically imply other assertions. For example, the absence of a node implies the absence of any label or type for this node, as well as the absence of any incoming or outgoing edges for this node. These implicit assertions are called *derived assertions* and are mentioned in Table 3. Whenever we specify a set of assertions *S*, we assume that *all derived assertions are also included in this set, even if they are not specified explicitly*.

Table 3: Derived assertions

Assertion	Derived Assertions
-N	*-label(N,*), -type(N,*****), -source(*,N), -target(*,N)*
-E	*-label(E,*), -type(E,*****), -source(E,*), -target(E,*)*
+source(E,N)	*+E, +N*
+target(E,N)	*+E, +N*
+label(Id,L)	*+Id*
*+type(Id,**T**)*	*+Id*

2.3 Conditional Graph Productions

The main distinction between our approach and the "common" use of assertions [7, 12, 15] is that we do not use assertions to attach behavioural constraints to programs. Instead, we use assertions to represent evolution transformations (as in [11, 16]). In other words, we attach assertions to graph productions rather than to graphs themselves.

Each assertion can be used either as precondition, postcondition or invariant of a graph production *P*. The sets of all these assertions are denoted by ***Pre(P)***, ***Post(P)*** and ***Inv(P)*** respectively. We also use the shorthand notations ***Before(P)*** = $Pre(P) \cup Inv(P)$ and ***After(P)*** = $Post(P) \cup Inv(P)$.

Given a graph rewriting $G \Rightarrow_{P,m} H$, one can easily write an algorithm that calculates the minimal set of assertions that determines the production *P*. For example, in Figure 1 we can identify the following minimal assertions:

*Pre(P) = {-c, -f, +label(a,surface), +type(a,**attribute**)}*

Inv(P) = {+a, -source(,c)}*

*Post(P) = {+label(a,area), +type(a,**operation**), +c,*
*+label(c,radius), +type(c,**attribute**), +f, +source(f,a),*
*+target(f,c), +label(f,uses), +type(f,**uses**)}*

If necessary, extra assertions can be added to these sets in order to restrict the applicability of production *P* to a smaller set of initial graphs. For example, if we would impose the extra invariant -

target(,a)*, *P* would not be applicable anymore to the graph *G* of Figure 1.

Following the notation of Perry [15], the assertions for production *P* are depicted as ellipses in Figure 2, while *P* is represented as a grey rectangle. Preconditions appear on the upper horizontal side of the rectangle, postconditions on the lower horizontal side, and invariants on the vertical sides. For positive assertions, the + sign is omitted in the figures. When they are needed, derived assertions are depicted by dashed ellipses. Finally, we abbreviated the last five postconditions of *P* to *(f,a,c,uses,**uses**)*.

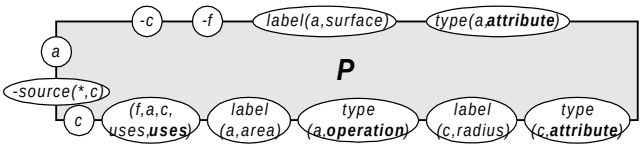


Figure 2: Graphical notation of a conditional production

[11] expressed every possible graph transformation in terms of a number of primitive productions that are sufficient to express any kind of change to a graph. For example, *AddEdge(f,a,c,uses,**uses**)* adds an edge *f* from *a* to *c* with label *uses* and type ***uses***. Table 5 shows all primitive productions and their corresponding assertions.[3]

Table 5: Primitive graph productions

Graph Production	Pre	Inv	Post
AddNode (N,L,T)	*-N*	*-source(*,N)* *-target(*,N)*	*+N* *+label(N,L)* *+type(N,**T**)*
AddEdge (E,N_s,N_t,L,T)	*-E*	*+N_s* *+N_t*	*+E* *+label(E,L)* *+type(E,**T**)* *+source(E,N_s)* *+target(E,N_t)*
DropNode (N)	*+N*	*-source(*,N)* *-target(*,N)*	*-N*
DropEdge (E,N_s,N_t)	*+E* *+source(E,N_s)* *+target(E,N_t)*	*+N_s* *+N_t*	*-E*
Relabel (Id,L_1,L_2)	*+label(Id,L_1)*	*+Id*	*+label(Id,L_2)*
Retype (Id,T_1,T_2)	*+type(Id,T_1)*	*+Id*	*+type(Id,T_2)*

3. PRODUCTION SEQUENCES

3.1 Well-formedness

A *production sequence* is a sequence of graph productions that can be applied successively. It is well-formed if the assertions imposed by a production in the sequence do not contradict assertions imposed by earlier productions.

Definition. A production sequence $P_1; P_2; ..; P_n$ is **well-formed** if $\forall A_k \in Before(P_k)$ with $k \in \{2..n\}$: if ($\exists A_i \in After(P_i)$ with $i<k$ such that A_i contradicts A_k) then ($\exists A_j \in After(P_j)$ with $i<j<k$ such that $A_j = A_k$). Otherwise, the production sequence is **ill-formed**.

Table 6 mentions all possible *contradicting assertions.* For example, the sequence $P_1; P_2$ = *AddNode(a,surface,**attribute**);* *AddNode(a,area,**attribute**)* is ill-formed because $+a \in After(P_1)$ contradicts $-a \in Before(P_2)$. The sequence $P_1; P_2; P_3$ = *AddNode(a,l_1,t_1); RelabelN(a,l_1,l_2); RelabelN(a,l_2,l_3)* is well-formed because the contradiction between $+label(a,l_1) \in After(P_1)$ and $+label(a,l_2) \in Before(P_3)$ is absorbed by $+label(a,l_2) \in After(P_2)$.

Table 6: Contradicting assertions

Assertion	Contradicts	where
+A	*-A*	*+A is some arbitrary positive assertion*
+source(E,N_1)	*+source(E,N_2)*	$N_1 \neq N_2$
+target(E,N_1)	*+target(E,N_2)*	$N_1 \neq N_2$
+label(Id,L_1)	*+label(Id,L_2)*	$L_1 \neq L_2$
+type(Id,T_1)	*+type(Id,T_2)*	$T_1 \neq T_2$

3.2 Detecting Syntactic Merge Conflicts

Ill-formed production sequences can be used to detect syntactic merge conflicts. These typically occur when different software developers are making changes to the same software in parallel, and these changes need to be merged.

Using the formalism of conditional graph rewriting, software merging can be formalised [11] by the notion of *parallel independence* [5]. Intuitively, two graph rewritings are parallel independent if they can be sequentialised in any order without changing the end result. Unfortunately, this definition does not specify what to do when two graph rewritings *cannot* be merged (read: sequentialised). If this is the case, we say that they give rise to a *syntactic conflict*. For example, suppose that graph *G* contains a node, and production P_1 removes this node while P_2 independently adds an edge originating from this node. This yields a syntactic conflict since trying to merge both parallel evolutions would lead to an edge without a source.

Definition. Two graph rewritings $G \Rightarrow_{P1,m1} H_1$ and $G \Rightarrow_{P2,m2} H_2$ lead to a **syntactic conflict** if the production sequence $P_1; P_2$ (or $P_2; P_1$) is ill-formed.

By comparing the different kinds of assertions that hold for P_1 and P_2, we can easily determine when a syntactic conflict occurs. It suffices to find a contradicting assertion between $After(P_1)$ and $Before(P_2)$, using Table 6. For example, for the primitive productions of Table 5 we identify the following syntactic conflicts:

- **Prohibited node removal** if $-v \in After(P_1)$ and $+v \in Before(P_2)$. This is for example the case if P_1 = *DropNode(v)* and P_2 = *AddEdge(e,v,w,l,t).* One cannot add an edge with a certain source node if this node has been removed before. **Prohibited edge removal** is defined similarly.
- **Dangling source** if $+source(e,v) \in After(P_1)$ and $-source(e,v) \in Before(P_2)$. This is for example the case if P_1 = *AddEdge(e,v,w,l,t)* and P_2 = *DropNode(v)*. One cannot remove a node that still has outgoing edges. **Dangling target** is defined similarly.
- **Prohibited node introduction** if $-v \in Before(P_2)$ and $+v \in After(P_1)$. **Prohibited edge introduction** is defined similarly.

- **Prohibited relabeling** if $+label(id,l_1) \in After(P_1)$ and $+label(id,l_2) \in Before(P_2)$. **Prohibited retyping** is defined similarly.

For approaches that can detect *semantic conflicts* rather than syntactic conflicts, we refer to [1, 2, 8].

3.3 Dependencies

Between the productions in a sequence we can determine *dependencies* based on which assertions are satisfied by assertions of productions earlier in the sequence. These dependencies will be used to address scalability issues in section 4.

Definition. Let $P_1; P_2; ..; P_n$ be a well-formed production sequence and $i<j$. An assertion $A_j \in Before(P_j)$ **is satisfied by** an assertion $A_i \in After(P_i)$ if $A_j = A_i$.

We can distinguish four satisfaction dependencies:

- $A_i \in Post(P_i)$ and $A_j \in Pre(P_j)$: P_j modifies (or removes) an entity that was already modified (or introduced) by P_i. For example, P_j = *DropEdge(e,b,c)* depends on P_i = *AddEdge(e,b,c,uses,**uses**)* because P_j removes the edge e that was introduced by P_i. This is detected by $+e \in Post(P_i) \cap Pre(P_j)$
- $A_i \in Post(P_i)$ and $A_j \in Inv(P_j)$: P_j relies on an entity that is modified by P_i. For example, P_j = *AddEdge(e,b,c,uses,**uses**)* depends on P_i = *AddNode(c,radius,**attribute**)* because $+c \in Post(P_i) \cap Inv(P_j)$
- $A_i \in Inv(P_i)$ and $A_j \in Pre(P_j)$: P_j modifies an entity that was relied on by P_i. For example, P_j = *DropNode(b)* depends on P_i = *DropEdge(e,b,c)*
- $A_i \in Inv(P_i)$ and $A_j \in Inv(P_j)$: P_j relies on the same entity as P_i. For example, P_j = *RetypeN(a,**attribute**,**operation**)* depends on P_i = *RelabelN(a,surface,area)*

The first three satisfaction dependencies are **strong dependencies** because changing the order of P_i and P_j yields an ill-formed production sequence. For example, we cannot add an edge between two nodes if one of these nodes is not yet present. Graphically, strong dependencies are represented by a solid line from A_j to A_i.

The fourth dependency is a **weak dependency**, because P_i and P_j can still be commuted without affecting the end result. For example, it is irrelevant whether we first relabel a node and then retype it or vice versa. Weak dependencies are represented by a dotted line from A_j to A_i.

Figure 4 shows all satisfaction dependencies in a sequence of three primitive productions. There is a strong dependency from the invariant $+b$ of the second production to the postcondition $+b$ of the first production, and from the precondition *type(b,attribute)* of the second production to the postcondition *type(b,attribute)* of the first. Finally, there is a weak dependency from the invariant $+b$ of the third production to the same invariant of the second production.

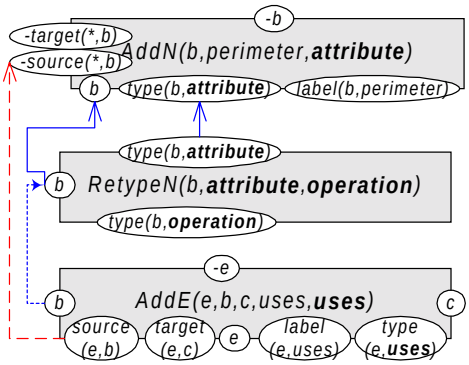


Figure 4: An illustration of satisfaction dependencies

Figure 4 also shows another kind of dependency from the postcondition $+source(e,b)$ of the last production to the invariant $-source(*,b)$ of the first. In general, some assertions of earlier productions can become captured by a postcondition of a later production, meaning that the earlier assertion can be ignored.

Definition. Let $P_1; P_2; ..; P_n$ be a well-formed production sequence and $i<j$. An assertion $A_j \in Post(P_j)$ **captures** an assertion $A_i \in After(P_i)$ if A_j contradicts A_i.

A capture is also a **strong dependency** in the sense that it prevents P_i and P_j from being commuted. Graphically, such a dependency is represented by a dashed line from postcondition A_j to postcondition (or invariant) A_i. This is illustrated in Figure 4 between $+source(e,b)$ and $-source(*,b)$.

The following complex production sequence illustrates all the dependencies introduced before:

*RelabelN(a,surface,area); AddNode(b,perimeter,**attribute**); RetypeN(a,**attribute**,**operation**); RetypeN(b,**attribute**,**operation**); AddNode(c,radius,**attribute**); AddEdge(e,b,c,uses,**uses**); AddEdge(f,a,c,uses,**uses**); DropEdge(e,b,c); DropNode(b)*

Figure 7 displays the assertions of each production in the sequence, together with all dependencies between them. Each assertion is the source of at most one dependency, that always points to the closest preceding assertion on which it depends.

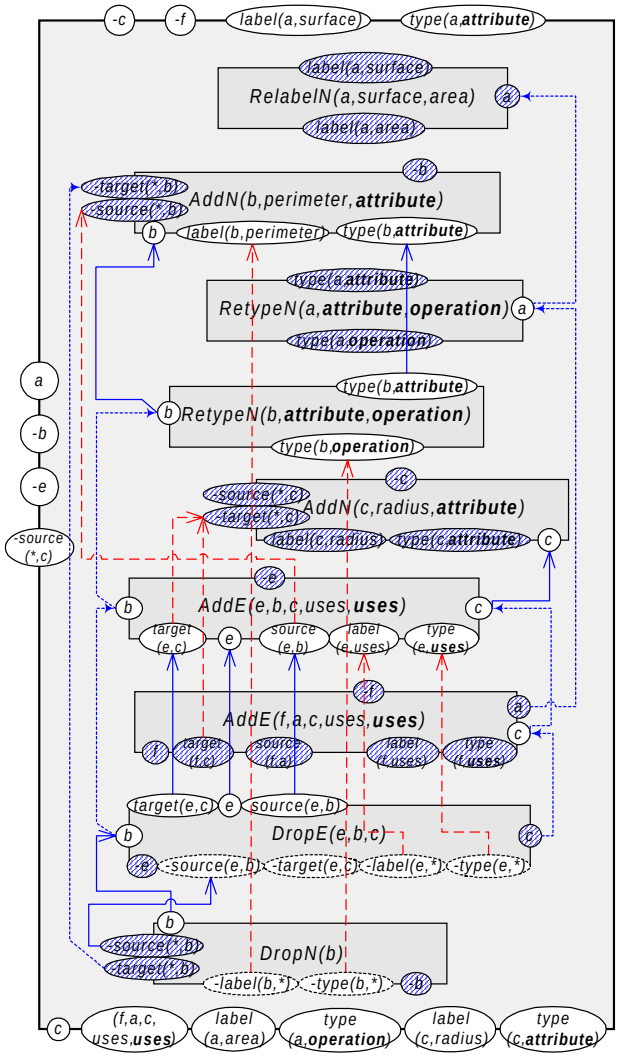


Figure 7: Dependencies in a production sequence

4. COMBINING GRAPH PRODUCTIONS

This section illustrates some important ways in which dependencies between assertions can address scalability issues when using large evolution sequences.

4.1 Composite Graph Production

A first way to address scalability is by treating complex sequences in exactly the same way as primitive productions. For example, the production sequence of Figure 7 can be considered as an atomic production *P*, as long as we are able to determine all of its assertions from the assertions of its constituent productions and the dependencies between them. The assertions of the so-called **composite production** *P* are calculated as follows:

(1) Identify all preconditions *Pre* and invariants *InvPre* that have no *outgoing* dependencies. Omit all derived assertions.

(2) Identify all postconditions *Post* and invariants *InvPost* that have no *incoming* dependencies. Omit all derived assertions.

(3) Calculate the assertions of the composite production *P*:

$Inv(P) = (InvPre \cap InvPost) \cup (Pre \cap Post)$

$Pre(P) = (InvPre \setminus InvPost) \cup (Pre \setminus Post)$

$Post(P) = (InvPost \setminus InvPre) \cup (Post \setminus Pre)$

In Figure 7, all the assertions in the sets *Pre*, *InvPre*, *Post* and *InvPost* of steps (1) and (2) are represented as shaded ellipses.

The actual preconditions, postconditions and invariants of the composite production *P* are shown as ellipses on the surrounding rectangle of Figure 7. For example, $Pre(P) = \{-target(*,c)\} \cup \{-c, -f, label(a,surface), type(a,\textbf{attribute})\}$, but the assertion *-target(*,c)* is omitted since it can be derived from *-c*.

4.2 Simplifying pairs of productions

Another way to address the scalability is by reducing a production sequence $P_1; P_2; ...; P_n$ by simplifying or eliminating pairs of successive[5] productions $P_i; P_{i+1}$. This is particularly relevant if we rely on a predefined set of productions (as in Table 5). Two kinds of simplifications can be distinguished. A pair of successive productions can be *absorbed* into a single predefined production, or the pair is *redundant* when the constituent productions cancel each other's effect. In the latter case, the pair can be removed without changing the overall behaviour of the graph rewriting. For both situations, a definition and concrete example is presented below.

Definition. A sequence of two graph productions $P_1; P_2$ is **absorbing** if there is a predefined graph production *P* such that $Pre(P) = Pre(P_1; P_2)$, $Post(P) = Post(P_1; P_2)$, and $Inv(P) = Inv(P_1; P_2)$

Figure 8 illustrates an absorbing production pair. Node addition *AddNode(b,perimeter,**attribute**)* followed by node retyping *RetypeN(b,**attribute**,**operation**)* is absorbed into a single node addition *AddNode(b,perimeter,**operation**)*.

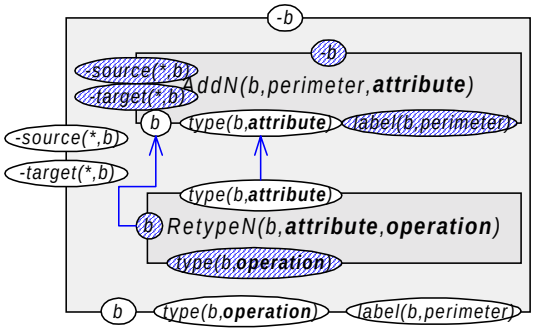


Figure 8: An absorbing production pair

Definition. A sequence of two graph productions $P_1; P_2$ is **redundant** if $Pre(P_1; P_2) = \emptyset$ and $Post(P_1; P_2) = \emptyset$.

With redundant pairs of productions, only the invariant set can be nonempty. Figure 9 illustrates a redundant production pair $P_1; P_2$. A node *b* is added and removed again. The resulting composite

[5] In section 4.4 we discuss the more complex case where redundant or absorbing productions do not directly follow one another in the sequence.

production has an empty set of pre- and postconditions, while $Inv(P_1; P_2) = \{-b\}$.[6] Also note the capture dependencies originating from *-type(b,*)* and *-label(b,*)*.

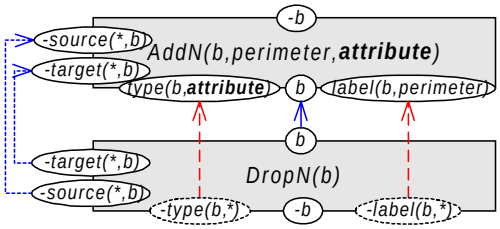


Figure 9: A redundant production pair

4.3 Reordering

If two successive productions in a sequence do not have a strong dependency between them, their order can be changed. When doing this, we need to modify all involved dependencies accordingly. This is illustrated in Figure 11 where we changed the order of the last two productions in the sequence of Figure 4. This was possible because there is only a weak dependency between the two productions that are being commuted. The reordered production sequence has the same overall effect as the original one because the assertions of the corresponding composite production are identical in both cases.

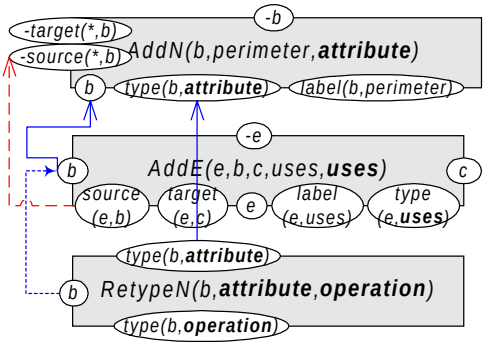


Figure 11: Reordering primitive productions in the sequence of Figure 4

4.4 Removing Redundancy

Reordering can be used to remove redundant and absorbing production pairs in a given sequence, even if the involved productions do not directly follow one another. In this way we can make the production sequence shorter, thus reducing the amount of memory required to store a production sequence (compression); improving the efficiency of algorithms that manipulate production sequences; making the production sequence easier to understand; etc…

Instead of giving the details of the redundancy removal algorithm, we illustrate how it works by means of a nontrivial example. Removing redundancy in the production sequence of Figure 7 yields the production sequence of Figure 12, containing only 4 instead of the original 9 primitive productions:

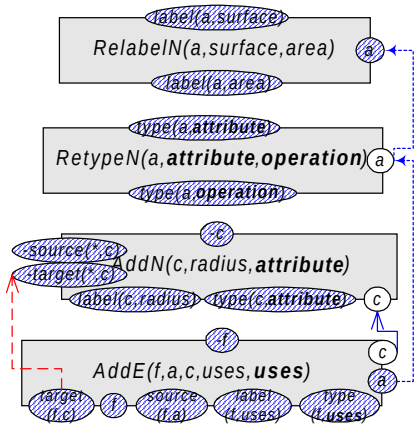


Figure 12: Final result after redundancy removal

This result is achieved by applying the following steps, starting from the production sequence of Figure 7:

1. Reorder of *RetypeN(a,**attribute**,**operation**)* and its immediate successor *RetypeN(b,**attribute**,**operation**)*, making *RetypeN(b,**attribute**,**operation**)* the immediate successor of *AddNode(b,perimeter,**attribute**)*.

2. Transform the absorbing subsequence *AddNode(b,perimeter,**attribute**); RetypeN(b,**attribute**,**operation**)* into a single production *AddNode(b,perimeter,**operation**)*.

3. Reorder of *AddEdge(f,a,c,uses,**uses**)* and its immediate successor *DropEdge(e,b,c)*, making *DropEdge(e,b,c)* the immediate successor of *AddEdge(e,b,c,uses,**uses**)*.

4. Transform the redundant subsequence *AddEdge(e,b,c,uses,**uses**); DropEdge(e,b,c)* into a single trivial production that only consists of invariants: *{-e,+b,+c}*.

5. Remove this trivial production, and redirect the dependencies accordingly.

6. Move the production *DropNode(b)* to directly behind *AddNode(b,perimeter,**operation**)*. This does not require redirection of any dependencies, since *DropNode(b)* only depends on *AddNode(b,perimeter,**operation**)*.

7. Transform the redundant subsequence *AddNode(b,perimeter,**operation**); DropNode(b)* into a single trivial production that only consists of invariants: *{-b}*.

8. Remove this trivial production. This concludes the redundancy removal, since no absorbing or redundant production pairs remain.

4.5 Refactoring Common Subsequences

In the context of team development, tool support is essential, especially when making parallel evolutions or customisations of the same software artifact. We can identify similarities between these changes by factoring out all commonalities between the parallel transformations. This is not only useful for reducing code duplication, but also during software merging to reduce the number of merge conflicts.

[6] The assertions *-source(*,b)*, *-target(*,b)*, *-type(b,*)* and *-label(b,*)* can be ignored as they are derived assertions of *-b*.

Figure 17: Factoring out commonalities in parallel evolutions

Schematically, the idea is represented in Figure 17. If we have two parallel productions *P* and *Q* that are applied to the same initial graph *G*, we can compare their assertions, and construct a new production *C* that contains only the common assertions, while the variable ones are specified in two other productions V_P and V_Q.

4.6 Undo Mechanism

In an industrial-strength software development environment, it should be possible to make changes undone selectively, even if these changes are part of a complex sequence. Suppose we want to undo only one production in a sequence. We cannot simply remove the production and reapply the resulting shorter sequence, because later productions in the sequence may still depend on the removed one. Therefore, we additionally need to remove all later productions that strongly depend on the removed production (either directly or indirectly).

For example, in order to undo *AddNode(b,perimeter,**attribute**)* in the sequence of Figure 7, we also need to undo all its strongly dependent productions *RetypeN(b,**attribute**,**operation**)*, *AddEdge(e,b,c,uses,**uses**)*, *DropEdge(e,b,c)* and *DropNode(b)*.

4.7 Parallelising Independent Subsequences

A final use of dependencies has already been discussed by Roberts [16]. In order to apply large production sequences in a more efficient way, they can be split up in parallel subsequences that can be applied independently from one another. This allows us to parallelise the process of applying complex transformations to a graph. It also makes large evolution transformations more manageable by splitting them up in smaller independent chunks that are more understandable.

For example, the production sequence of Figure 12 can be parallelised into the following independent subsequences:

*RelabelN(a,surface,area); RetypeN(a,**attribute**,**operation**)* and

*AddNode(c,radius,**attribute**); AddEdge(f,a,c,uses,**uses**)*

5. RELATED WORK

Perry was one of the first to use assertions for dealing with certain aspects of software evolution. In [15] he describes a *semantic interconnection model* that uses assertions to annotate software artifacts. This model is used to detect the effects of changes by recursively determining the assertions that are affected by the change. In our approach, we do not use assertions for expressing the behaviour of software artifacts themselves, but to express semantic dependencies between the evolution transformations instead.

If we focus on formal support for merging parallel evolutions, our work is closely related to [9]. Lippe and van Oosterom propose an *operation-based merge technique* that uses software transformations (called operations) to represent evolution, and detects and resolves merge conflicts using the information contained in these transformations. Dependency information between transformations is used to address the issue of scalability, but assertions are not used to identify the dependencies.

The research in this paper is a logical consequence of the work on *reuse contracts* [17]. Mens [10, 11] provides a formalism for reuse contracts that uses pre- and postconditions to express graph transformations and relies on formal properties of conditional graph rewriting [4, 5, 6].

The research of Roberts [16] is also closely related. Pre- and postconditions are used to express refactoring transformations (which are usually behaviour-preserving), and some scalability issues are addressed as well.

6. CONCLUSION

Typed graphs, combined with graph transformations that are based solely on assertions (i.e., preconditions, postconditions and invariants) provide a general formalism for software evolution. Assertions make it easy to detect syntactic merge conflicts between parallel evolution transformations, and allow us to define composite graph transformations in an intuitive and straightforward way. Dependencies between the assertions allow us to address several scalability issues, such as changing the order in a transformation sequence, removing redundant transformations in a sequence, and extracting a common subsequence from two (or more) given transformation sequences.

The approach seems very promising, but still needs to be validated in a large-scale case study. Also, the underlying formalism can be extended in many ways: a notion of subtypes could be introduced; more complex assertions could be defined; the productions could be made more generic; etc…

7. REFERENCES

1 V. Berzins, *Software Merge: Semantics of Combining Changes to Programs*, ACM Trans. Programming Languages and Systems, Vol. 16, No.6, 1994, pp. 1875-1903.

2 D. Binkley, S. Horwitz, and T. Reps, *Program Integration for Languages with Procedure Calls*, ACM Trans. Softw. Eng. and Methodology, Vol. 4, No. 1, 1995, pp. 3-35.

3 M. S. Feather. *Detecting Interference when Merging Specification Evolutions*. Proc. Int. Workshop Softw. specification and design, pp. 169-176, ACM Press, 1989.

4 A. Habel, R. Heckel, G. Taentzer. *Graph Grammars with Negative Application Conditions*. Fundamenta Informaticae, Special Issue on Graph Transformations, 26(3,4): 287-313, IOS Press, June 1996.

5 R. Heckel. *Algebraic Graph Transformations with Application Conditions*. Dissertation, Technische Universität Berlin, 1995.

6 R. Heckel, A. Wagner. *Ensuring Consistency of Conditional Graph Grammars: A Constructive Approach*. Lecture Notes in Theoretical Computer Science 1 (1995), Elsevier Science, 1995.

7 C.A.R. Hoare. *An axiomatic approach to computer programming*. Comm. ACM 12(10): 576-580, 583. ACM Press, October 1969.

8 D. Jackson, D.A. Ladd. *Semantic Diff: A Tool for Summarizing the Effects of Modifications*, Int. Conf. Softw. Maintenance, IEEE Press, 1994.

9 E. Lippe, N. van Oosterom. *Operation-based Merging*. Proc. Fifth ACM SIGSOFT Symp. Softw. Development

Environments. ACM SIGSOFT Softw. Eng. Notes, 17(5): 78-87, ACM Press, 1992.

10 T. Mens. *A formal foundation for object-oriented software evolution.* PhD Dissertation, Vrije Universiteit Brussel, September 1999.

11 T. Mens. *Conditional graph rewriting as a domain-independent formalism for software evolution*. Proc. Int. Agtive '99 Conference, LNCS 1779: 127-143, Springer-Verlag, 2000.

12 B. Meyer. *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

13 W.F. Opdyke. *Refactoring object-oriented frameworks*, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Technical Report UIUC-DCS-R-92-1759, 1992.

14 W.F. Opdyke, R.E. Johnson. *Creating abstract superclasses by refactoring*, Proc. ACM Computer Science Conference, pp. 66-73, ACM Press, 1993.

15 D.E. Perry. *Software Interconnection Models*. Proc. Int. Conf. Softw. Eng., IEEE Press, 1987.

16 D. Roberts. *Practical Analysis for Refactoring*. PhD Dissertation, University of Illinois at Urbana-Champaign, 1999.

17 P. Steyaert, C. Lucas, K. Mens, T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. Proc. OOPSLA '96, SIGPLAN Notices 31(10): 268-286, ACM Press, 1996.

Transformation of Binary relations into Associations and Nested Classes

Jamal Said, Eric Steegmans
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven
Belgium
Tel: +32-(0) 16-32 76 56
Fax: +32-(0) 16-32 79 96
E-mail: jamal.said@cs.kuleuven.ac.be

Abstract
In object-oriented paradigm, as the complexity of the software system increases, it's cost to develop and to maintain goes exponentially. This complexity emerges from the continuous evolution in the software systems to cope with changing requirements. Throughout our study we found that maintaining traceability between the evolved software processes (e.g. analysis, design) in parallel with examining the ultimate software quality factors needed is an efficient way to cope with this crucial problem. To maintain traceability requires keeping the line between the analysis and the design phase crisp and distinct. This line can be defined by performing an active transformation of the elements (i.e. classes and relations) of the conceptual model to produce optimum design model. This transformation requires the structure and the semantics of the predefined elements to be kept consistent with their equivalent ones in the design model. The transformation process ends up with an optimum design model, thereby reducing complexity and finally reducing the cost.
In this paper we will show two transformations for a simple conceptual model consisting of three inter-related classes having a binary relation. Each of these transformations satisfies particular software quality factor(s), from which the software engineer can choose the one that matches the system intended functional requirements. The added value of this approach is that less manual optimization is required and high maintenance is achieved.

1. Introduction

In the mid-nineties the idea of design patterns started to attract considerable attention in the area of object-oriented software development. Design patterns [1] are architectural ideas applicable across abroad range of application domains; each pattern enables the software engineer a solution to a certain design issue. In fact, the patterns developed in the past few years are only incremental additions to the software professional's bag of standard tricks [2]. To put it more precisely, the underlying representation of a design pattern and of its application, and the binding between these two levels is not exactly defined and thus can be interpreted in different ways [7]. Other researchers [3] have followed the qualitative design trends, which lead to designs that exhibit a desirable quality and forms a movement from bad design model to good design model. These two approaches (i.e. design patterns and the qualitative heuristics) have common basis since both strive to reuse general knowledge rather than domain-specific code. Although these two approaches show interest to software engineers, they lack the ability to keep traceability and maintainability between the analysis and design models.
The analysis phase usually ends up with the conceptual model, in which the external world with corresponding classes and objects is represented. In the traditional approach where we have a chasm between analysis and design, the major input to the design process is the Software Requirements Specification *document*. Because incompatible and non-integrated notations are used from analysis to architectural design, a lot of rework is required, discovering the same ambiguities again, maybe committing the same errors again, and (hopefully) correcting the same

errors again. This paper proposes an approach where the architectural design model, doesn't start off with an empty design, but it starts off with a design model, which is a copy of the analysis model in the design notation. This model represents a complete description of the way the system could work, covering all functional requirements. It does not represent a solution that meets all the other requirements. It is then an approach to transform analysis model into a description of the way we want the system to work. This approach can be worked out by considering the elements of the conceptual model as a collection of simple conceptual model's fragments and based on object-oriented concepts and the software quality factors, these fragments are transformed into design model. The transition from the conceptual model to the design model is often an iterative process; thus it is crucial to be able to develop a framework that performs a reliable and convenient transition between the two models.

Currently, software developers based on the conceptual model try to accomplish some actions manually, which in most cases leads to a big distinction between experienced and inexperienced developers and increases the cost of the software system due to maintenance. Given the fact that software engineering is aiming at building robust and reliable software systems, an approach that supports modeling and provides insights into understanding the software requirements and the software design is crucial. This approach should not restrict the software engineer to a particular phase of the software life cycle but it maintains link between the early phases (analysis and design).

Without necessarily inhibiting choices of the design, taking a copy of the analysis model as an initial design model is likely to enable smother transition from requirement modeling to design. It also prevents unnecessary and non-justified differences between the analysis and design model. It guarantees a better traceability between the analysis model and final design model. It also makes design choices more explicit, as these are highlighted as justified changes between the analysis and the design model.

2. Binary Relations at the Level of Analysis

The early stages of object-oriented analysis is mainly concerned with specification of the objects that are relevant to the application being developed, then comes the refinement step in which the relationships among those corresponding objects are examined in parallel with the study of the events by means of which these relationships are manipulated.

In our view relationships are considered as characteristics of the involved objects. Consequently, relationships of the same sort are grouped in a class. As an example, relationships between persons and companies, expressing that companies employ persons, first of all lead to the introduction of a class of employments. As a result a relation is said to refine objects of a given class, a refinement expressing that these objects cannot exist without being related to objects of the classes participating in the given relation [8, 9]. In our example, a relation will be introduced refining the objects of the class of employments, in order to express that no employment can exist without being related to a person on the one hand, and to a company on the other hand. Such kind of relations is called binary relations which involve two participating classes and one refined class. For example in banking application both classes persons and banks as illustrated in Figure1 are known as the participating classes and the class of accounts as the refined class.

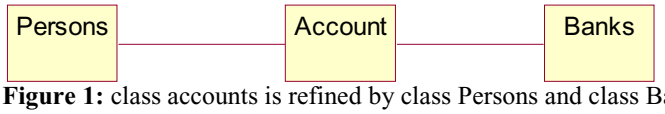


Figure 1: class accounts is refined by class Persons and class Banks.

As mentioned before, any specified relation between objects is complemented with a specification of operations for manipulating those involved objects.

For classes refined by a binary relation, at least a constructor, destructor and two queries must be introduced. The constructor will initialize the binding of the new refined object with the given

objects of the two participating classes; the inspectors will return the objects of the two participating classes involved in the refined relation. Furthermore, the refined class may introduce mutators for changing the binding of refined objects to some other objects of the participating classes. For example constructing a new account requires specifying the Person that will hold this account and the grantor (bank) that will grant this account. Furthermore, a destructor for closing the given account, a query (e.g. getBank, getPerson) for retrieving the owner (getPerson) and the grantor (getBank) of this Account, and a mutator (e.g. transferTo) to transfer accounts from one person to the other is required. Besides the constraint of mutability, constraint of multiplicity is also important at the early stages of the analysis. For classes refined by a binary relation, the multiplicity specifies how many objects of the one participating class can be associated at most with the same object of the other participating class through objects of the refined class. The resulted structural and behavioral aspect of the pattern shown in Fig. 1 is illustrated in Figure 2 below.

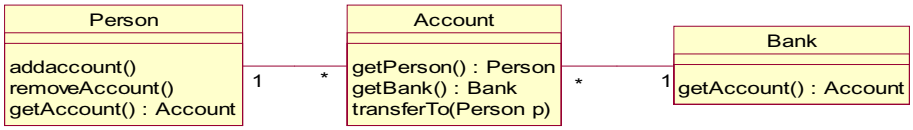


Figure 2: Structural and behavioral aspects of three classes involved in a binary relation.

3. Transformation of binary relations

During analysis the software engineer focuses on the issue of specifying the needed objects for the system to meet its requirements and lining these objects with appropriate relationships to construct a meaningful and complete conceptual model. In other words, the software developer is only interested in which objects are needed not how these objects should be implemented, the later is the subject matter of the design phase which will give the description of the involved objects and relationships between them. The description of the classes and their relations are prime items of the design model.

This paper presents a transformational approach to object-oriented design. Basically, a design model is obtained by transforming fragments, as they can be observed in conceptual models. Because a single fragment can be designed in many different ways, the designer chooses the most appropriate one, based on quality factors for the ultimate system being developed.

This paper discusses transformations for a simple conceptual model defining the refinement of a class by means of binary relation. For a pattern consisting of binary relation (Fig 3), there exist different alternatives to transform it to design elements. In this paper we will focus on the association and nesting transformations.

3.1 Association Transformation

The binary relation involves two participating classes and a refined class can be design in terms of an association between the refined class and the participating classes. Associations represent relationships between instances of classes (e.g. a person holds accounts in Banks; a bank grants accounts to person From the conceptual perspective, associations represent conceptual relationships between classes. In Figure 3, the diagram indicates that an

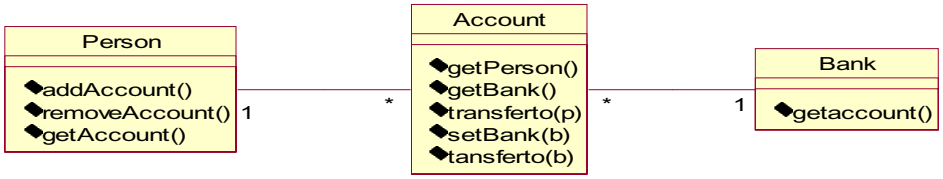


Figure 3: Class diagram with association relationships.

account has to reference one person and must be granted by one bank. As far as the multiplicity is concerned, which is an indication of how many objects may participate in the given relationship. In Figure 3, the * between person and accounts indicates that a Person may have many accounts associated with it; 1 indicates that an account related to only one person. The multiplicity between accounts and Bank indicate that a Bank grants many accounts and an account has to be granted by only one Bank.

- Within the specification perspective, associations represent responsibilities. Figure 3, implies that there are one or more methods (i.e. getAccount) associated with Person that will tell us know what accounts a given Person is holding. Similarly, there are methods (getPerson, getBank) within Account that will let us know which Person holds this account and which Bank grants a given account
- The given structure explained above would be transformed into design taking into consideration both structural and behavioral aspects defined at the level of analysis.
- Because of the property of existential dependency-accounts cannot be created without being attached to a Person on one hand and to a Bank on the other hand-- the construction of objects of the refined class (Account) must initialize references to objects of the participating classes (Person and Bank). Objects of the participating classes (Person, Bank), on the other hand, can exist without being involved in associations with objects of the refined class., Consequently, the constructor at the level of the participating class initializes a new object without any association to objects of the refined class.
- The destructor for the refined class objects as it is specified at the level of analysis is transformed into the method removeAccount. Notice that the reference to the destroyed object is removed from the participating object.
- The existential dependency should also be considered when destroying the participating objects. Before any participating object is destroyed one must check whether this object is holding references to a refined object or not. If so all these refined objects must be destroyed beforehand. For example, when a Person is removed from a Bank, it means his account will also be.
- The inspectors defined at the level of the participating classes are transformed into the method getAccount applicable to objects at the level of design. Notice that the method returns an array in which references to all the Account's objects are stored.
- Similarly the inspectors defined at the level of the refined class is transformed into the method getPerson and getBank applicable to Account objects. This method and will return the Person and the Bank attached to this account.

The mutator defined at the level of analysis is transformed into the method transferAccount. This method transfers this Account to the specified Person and Bank.

Below we will show some methods with their specification implemented in Java. Notice the specification's notation used here is widely used in the literature [11].

```
import java.util.*;
/**
 * A class of person.
 */
public class Person {
  /**
   *  Initialize a new Person with no Account nor bank objects attached to It.
   *  @post   No Bank object and account-objects are attached to the new person.
   *        | new.getAccounts().size()= 0
   */
  public Person()
}
//Definition of the refined class Account
import java.util.*;
```

```
/**
 * A class for dealing with accounts attached to a Person and a Bank
 * @invar  An account must all times be attached to a Person and a Bank.
 * @invar  The Person and the Bank to which this account is
 *         attached, must reference back to that account.
 *       | getPerson().hasAccount(this)
 *        | getBank().hasAccount(this)
 */
public class Account {
  /**
   * Initialize a new account attached to the <person> and the <bank>.
   * @param  <person>
   *         The Person to which the new account will be attached.
   * @pre    <person> must be effective
   *        | person <> null
   * @post   The new account is attached to <person> and vice versa.
   *        | (new.getPerson() = person )
   *          and (((Person)((new person).getAccounts()).contains(this)) = true )
   * @param  <bank>
   *         The Bank to which the new account will be attached.
   * @pre    <bank> must be effective
   *        |bank <> null
   * @post   The new account is attached to <bank> and vice versa
   *        | (new.getBank() = bank )
   *        |(((Bank)((new bank).getAccounts()).contains(this)) = true)
   */
  public Account( Person person, Bank bank)
  /**
   * Transfer the new account to specified person
   * @param  <person>
   *         The specified person to become participant to this account
   * @pre    The specified person must be effective
   *       | person <> null
   * @post   The specified person is associated with this account
   *       | new.getPerson() = person
   * @post   This Account is no longer referenced by the person
   *         to which it was associated before.
   *       | for each i in 0..(this.getPerson()).getAccounts().size() - 1:
   *      (this.getPerson()).getAccounts.elementAt(i) != this
   *          and (this.getPerson()).getAccounts.size()
   *           =(this(this.getPerson())).getAccounts.size() –1
   */
  public void transferAccount(Person person)
}
//definition of class Bank
/**
 *  Definition of participating class Bank
 */
public class Bank {
  /**
   * Initialize a new bank with no accounts attached to it
   * @post No accounts attached to the new bank
   *       | new.getNbAccounts() = 0
   */
  public Bank()
  }
```

Implementation1: Implementing association transformation.

With the association transformation the software engineer selects for the quality factors flexibility, and re-usability over efficiency and simplicity.

- Limiting each of the involved classes to a specific area of interest (i.e. cohesion) highlights flexibility. Furthermore, flexibility is stressed by allowing future modifications to the software system. As far as coupling is concerned this transformation strives to have high coupling by allowing the components to cooperate via message passing.
- This transformation is considered to be highly reusable since most of the structural and behavioral aspects of the classes specified at the level of analysis are transformed at the level of design with limited loss of information.
- As far as the efficiency is concerned this type of transformation is not the most efficient one in terms of time and space since the memory requirement is high. Part of the objects of the involved classes needs a separate location in memory, which in turn affects the performance of the software system . The creation of new objects of the classes and the message passing between them requires the execution to take more time than if they were integrated in one class.
- Simplicity is not supported by this transformation since it requires message passing between objects of the classes involved. The message passing might lead to inconsistencies, if bi-directional associations are not designed and implemented with great care.

3.2 Nesting Transformation

Nesting transformation occurs when one class is fully defined inside the other the concept which known in Java as inner classes. Inner classes are powerful abstraction mechanism [5] that facilitate much more convenient and manageable software than it would be when using only top-level classes. They are remarkable as they allow to group classes and control the visibility of one within the other.

Classes with binary relation can be transformed by defining one class inside the other. For example, Figure 4, shows the participating class Bank having association with class Account which is nested inside the participating class Person. Class Person serves as the outer class through which the refined class Account (inner class) can be accessed. Notice also that the outer class is responsible for creating and the Account objects.

The account objects are created by applying the method openAccount to Person objects. This method when applied to person object will also initialize a bank object with the created Account object due to the existential dependency. Notice that since the creation of the accounts depends on the person objects then the accounts will automatically store implicit references to person objects. Therefore, an object of the refined class is directly associated with the object of the outer class; objects that created them. As a result the inner class object has direct access to the instance variables of the enclosing class object. Notice that the compiler does the implicit reference to the outer class objects itself. Concerning mutation Accounts cannot be transformed at the level of design since the refined objects are nested in person objects, which are designated, immutable.

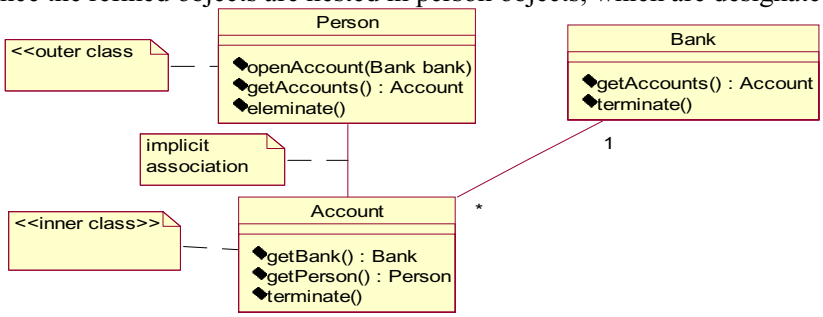


Figure 4: Account class nested in Person class and has an association with class Bank.

```
/**
 * The participating class Person.
 */
 public class Person {
   /**
    *  Initializes a new Person with no Account nor bank objects
    *  attached to it.
    *  @post   No Bank object and account-objects are attached to
    *          the new person.
    *        | new.getAccounts().size()= 0
    */
   public Person()
   }
   /**
    * Definition of the inner class Account.
    */
   public class Account {
      /**
       * initialize a new Account
       * @post  Bank object must also be initiated
       *       | this.getBank() == bank
       */
      Account(Bank bank)
      /**
       * Terminate this account
       * @post This account is terminated and detached from its participating
       *       objects.
       *       |!((new getPerson()).getAccounts().contains(this)
       *       |!((new getBank()).getAccounts().contains(this)
       */
      public void terminate ()
    }//end of inner class
  /**
   * Creation of the account object
   */
  public void openAccount(Bank bank)
  }//end of outer class
/**
 *  Definition of participating class Bank
 *  Class bank has an association relationship with class Account
 */
public class Bank {
  /**
   * Initialize a new bank with no accounts attached to it
   * @post No accounts attached to the new bank
   *       | new.getNbAccounts() = 0
   */
  public Bank()
}
```

Implementation 2: Implementing nesting transformation.

The added value to the object oriented software design by nesting transformation is that it increases modularity as will as simplicity over efficiency.

- Modularity is the term that covers reusability and extendibility. Nesting transformation helps in making these two classes easy to change. In association when one of the two associated classes is expected to change we must take the navigability under consideration whether the involved class is bi-directional or unidirectional, whereas, in nesting we know already that the inner class objects have implicit references to their outer ones.

Concerning the reusability, nested transformation is highly reusable particularly for applications where accessibility constraints are important

- This transformation is considered to be simple since it decreases the number of classes developed at the package level. Which make the model easier to understand and maintain, also it limits the number of message passing between the associated classes
- As far as efficiency is concerned it helps in time efficiency because both inner and outer classes are stored in one file which makes message passing requires less time than if they were stored in two separate files. However this transformation doesn't help so much in space efficiency since both the classes are stored in different places in memory

4 conclusion

In this paper we have shown that designing convenient and transparent software system can be handled easily by keeping the line between the design and the analysis definite and distinct. This line can be defined by performing an active transformation of the conceptual model's elements and relations (i.e. fragments) to produce a design model that perform the system intended functionalities. We have seen that this technique offers the user to select among different transformations the one that meets the design goals. As a result, this new technique doesn't require the software engineer to optimize the design model, which is lacking in the current methodologies. Furthermore, this technique establishes a strict correspondence between conceptual models at the level of analysis and design models at he level of design, which results in high maintenance throughout the software system.

In this paper we have discussed binary relations and their transformations, and in the future work this will be extended to cover the classes involved in generalization specialization and statics.

5. References

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object Oriented Software*. Addison- Wesley.

[2] Bertrand Meyer 1997. *Object Oriented Software Construction*. Prentice Hall.

[3] Arthur J. Riel (1996). *Object Oriented Design Heuristics*. Addison Wesley.

[4] Rumbaugh, J. Jacobson, I. & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[5] Martin Fowler with Kendall Scott (1998). UML Distilled : Applying the standard object modeling language. Addison Wesley.

[6]Charles Richter (1999). *Software Engineering Series. Designing Flexible Object –Oriented System with UML*. Macmillan Technical.

[7] Gerson Sunye, Alain le Guennec, and Jean-marc Lezequuel. *Design Patterns application in UML. ECOOP' 2000 – Object oriented Programming 14th European Conference, Sophia Antipolis and Cannes, France , , volume 1850 of lecture notes in Computer Science, pages 44 –62*. Springer – NY, June 2000.

[8] Van Baelen, S., Lewi, J., and Steegmans, E., *Constraints in Object-Oriented Analysis and Design*, *Technology of Object-Oriented Languages and Systems TOOLS 13*, eds. Magnusson, B., Meyer, B., Nerson, J.-M., and Perrot, J.-F., Prentice-Hall, Hertsfordshire, UK, 1994, pp. 185-199.

[9] Van Baelen, S., Lewi, J., Steegmans, E., and Swennen, B., *Constraints in Object-Oriented Analysis, Object Technologies for Advanced Software*, LectureNotes in Computer Science 742, eds. Nishio, S., and Yonezawa, A., Springer-Verlag, Berlin, D, 1993, pp. 393-407.

[10] Said J., Steegmans E., Transformation of Unary Relations: Proceeding ICSSEA 2000, 13th International Conference on Software & System Engineering and their Applications , Paris – France (2000).

[11] Bruegge B., Dutoit A.H., Object-Oriented Software Engineering. Prentice-Hall, Inc. 2000, pp. 239.