International Journal of Software Engineering and Knowledge Engineering

TOWARDS AUTOMATIC ESTABLISHMENT OF MODEL DEPENDENCIES USING FORMAL CONCEPT ANALYSIS

IGOR IVKOVIC

Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L3G1 Canada
iivkovic@swen.uwaterloo.ca
http://swen.uwaterloo.ca/∼iivkovic

KOSTAS KONTOGIANNIS

Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L3G1 Canada
kostas@swen.uwaterloo.ca
http://swen.uwaterloo.ca/∼kostas

Software evolution is an iterative and incremental process that encompasses the modification and alteration of software models at different levels of abstraction. These modifications are usually performed independently, but the objects to which they are applied to, are in most cases mutually dependent. Inconsistencies and drift among related artifacts may be created if the effects of an alteration are not properly identified, recorded, and propagated in other dependent models. For large systems, it is possible that there is a considerable number of such model dependencies, for which manual extraction is not feasible. In this paper, we introduce an approach for automating the identification and encoding of dependence relations among software models and their elements. The proposed dependency extraction technique first uses association rules to map types between models at different levels of abstraction. Formal concept analysis is then used to identify clusters of model elements that pertain to similar or associated concepts. Model elements that cluster together are considered related by a dependency relation. The technique is used to synchronize business process specifications with the underlying J2EE source code models.

Keywords: Software evolution; model synchronization; formal concept analysis.

1. Introduction

Software evolves iteratively and incrementally from its inception to its retirement. The evolution includes change of artifacts at different levels of abstraction, from very abstract ones, such as business process specifications, to very concrete ones, such as source code. A change on a design model for instance, such as addition of a new class in the structural view, could directly affect architecture and implementation models and through ripple effects even the requirements specifications. If each change is not propagated to all affected models and if changes that are performed in parallel are not coordinated, consistency among artifacts is lost and semantic drift

is created. To propagate and coordinate transformations that are applied in one model, it is necessary to resolve differences in the other models. Furthermore, different software models are often used by different stakeholders and may pertain to various schemas and levels of technical detail. These models are based on languages that exhibit different capabilities and limitations with regard to their expressiveness and semantics. Therefore, mapping change from one modeling domain to another also requires resolving language dissimilarities.

In previous research, we have addressed the problem of model synchronization through a framework for incremental change propagation and translation titled **mSynTra** [8] [9] [21]. The framework is based on the view of evolution in the context of an iterative and incremental lifecycle such as the Rational Unified Process (RUP) [6]. The methodology follows the Model-Driven Software Evolution (MDSE) paradigm that is based on the general concepts of Model Driven Architecture (MDA) [11] [12]. To synchronize changes in mSynTra, using established model dependencies, a modification that is applied to one model is appropriately translated and applied to all other affected models. The process is through a collection of synchronization relations.

This paper represents an extension to the mSynTra framework, where we discuss an approach to systematically establish and maintain associations among models and their elements through the use of formal concept analysis (FCA) [4]. The basics of the approach were previously introduced [7]. In this paper we extend these findings by presenting a formalization of the algorithm for the identification of model dependencies; a demonstration of the prototype implementation; additional experimentation results for run-time, space, precision and recall performance; and finally, elaboration on the formalization of the approach that before was not presented in detail due to space limitations.

In our approach, each software model is viewed as a *context* in terms of its objects and attributes. Furthermore, Formal Concept Analysis (FCA) can be used as a formal method for identifying groups of objects that share common attributes and are hence considered dependent. Given that the models are possibly from different domains, we also introduce the notion of *attribute association rules* for creating mappings among heterogeneous attributes. These rules can be defined both at the domain model and, at the concrete model level. At the domain model level, the rules represent the mappings between compatible types and relations, while at the model level, they represent the mappings between objects, attributes, values, and annotations. It is also possible for one to define the association rules at the metamodel or metametamodel level if needed.

In the proposed approach, we consider that one or more models conform to a specific domain model that denotes relations and properties between model elements. Since models may pertain to software artifacts at different levels of abstraction, they may have a significant semantic gap when compared to each other. We propose that domain model information can be used to generate a sequence of transformations that can alter these models to a level where they can be compared and their depen-

dencies can be extracted. These domain model transformations can take the form of association rules or type mappings from one domain model to another.

To validate the approach and demonstrate its applicability in a practical scenario, we make use of a case study where the goal is to synchronize business process models (BPM) with the enacting Java and Enterprise Java Beans (EJB) [18] source code. The business models are represented as business workflows that include processes, tasks, decisions, data, *etc.* Due to significant semantic gap between the two domains, before dependencies can be established, it is necessary on one hand to augment the representation of the business workflows with implementation information, and, on the other hand, to abstract the code representation in terms of related business functionality. Once models, domain models and, metamodels are represented in a MOF-compliant form [13], they can then be represented as XML documents and consequently their transformations can be encoded as XSLT transforms [5].

The remaining paper content is structured as follows. Section 2 discusses the relation between this research and related findings previously published in the area of software reuse and hierarchical data management. Section 3 gives an introduction to the framework for establishment and maintenance of relations among models and their elements using FCA. Section 4 describes our prototype implementation and demonstrates the validity and applicability of the approach over a case study, while Section 5 gives the conclusions and future research directions.

2. Related Research

In the approaches that focus on software reuse, Spanoudakis and Constantopoulos [17] measure similarity through a distance metric in order to evaluate the reuse potential of software artifacts. Engels *et al.* [2] discuss the transformations between Unified Modeling Language (UML) Class Diagrams and UML Collaboration Diagrams [14] and Java source code. The approach considers the structural and behavioral mappings using transformation patterns. The patterns used are not trivial to extract and the pattern repository needs to be updated as new transformations are introduced. The approach in this paper uses formal concept analysis to establish the mappings at the level of model elements. For objects that belong to more than one concept, conflict resolution is performed using a similarity metric, represented for instance as a sum of weighted scores.

In the research area of hierarchical data management, an approach by Faid *et al.* [3] uses formal concept analysis to discover concepts and rules based on structured complex objects. Gianolli and Mylopoulos [16] perform semantic mapping of XML data stores using a common DTD schema. In this paper, the semantics of hierarchical data structures are mapped using intermediate models. However, the relations among individual model elements are also inferred based on the mappings of related attributes.

In the research published by Rich and Wills, subgraphs are used to recognize

program design [15]. In their approach, several categories of problems related to establishing model dependencies are recognized. These include non-contiguousness — adjacent elements from one flow may be separated in another related flow, implementation variation — the same design under differing contexts may be represented by different implementations, overlapping implementations — two or more implementations may overlap in implemented functionality, and unrecognizable implementations — no relevant semantic information can be extracted from an information flow. This view is extended through our research by considering challenges related to: n-ary relations — dependencies are not only one-to-one or one-to-many, but also many-to-many mappings of model elements, partial dependencies — only parts of model elements are related, and non-applicable dependencies — an element from one model is not directly mapped to any element in a related model. These challenges are addressed through the following:

- Non-contiguousness, overlapping implementations and partial dependencies are addressed through mappings of individual model elements instead of flow patterns.
- Implementation variation is addressed through mappings of interfaces and recognition of differing contexts for each mapping.
- N-ary relations are addressed through definition of model dependencies as tuples of model elements.
- Unrecognizable implementations and non-applicable dependencies are addressed through inclusion of user feedback.

3. A Framework for Establishing Model Dependencies using FCA

This section describes the framework for automatically establishing model dependencies using formal concept analysis (FCA), which was introduced in [7]. As a part of this view, models are viewed as collections of objects and attributes. Before FCA can be used, attribute associations need to be established. To establish relations between heterogeneous attributes, we make use of attribute association rules to map attributes and values between models that relate and conform to different domain models. The rules are established both at the domain model level and at the concrete model level. The differences in levels of expressiveness and semantics are augmented through generation of intermediate models. Once the rules are established, FCA is used for clustering objects based on shared attributes. A clustered group of objects constitute a concept and the objects are then said to be dependent.

More specifically, the steps in this process are:

1. Defining Metamodels and Domain Models
   The elements of domain models such as types, relations, and attributes, are extracted through domain analysis and represented using MOF syntax.

2. Generating Intermediate Models
   To bridge possible syntactic and semantic gaps between domain models, the more abstract ones are iteratively annotated and enhanced into more specific ones, while the more concrete ones are iteratively abstracted and refined into more abstract ones in an attempt to bridge the syntactic and semantic gap between the models that need to be associated.

3. Establishing Model Dependencies
   After semantic gaps are bridged, the association rules are defined based on compatible properties. Using FCA and the association rules, clusters of objects that share common attributes are identified, and represented as tuples of model dependencies.

4. Validating Established Dependencies
   The resulting tuples are validated by comparing them to previously confirmed results, or through feedback from developers and domain experts.

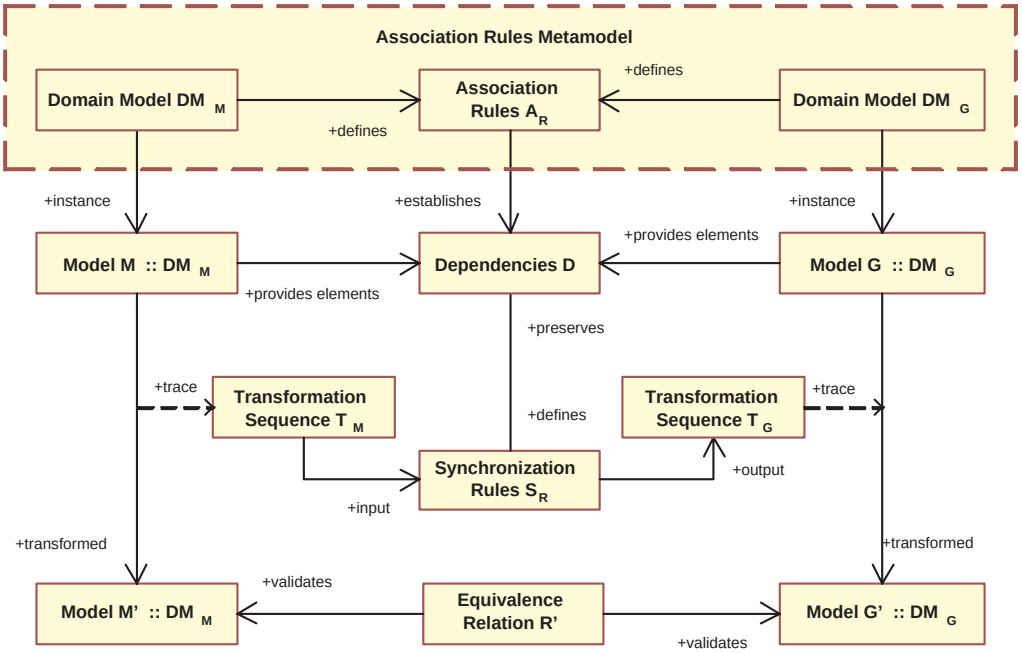


Fig. 1. **mSynTra** Model Synchronization Framework

The broader context of this approach, the mSynTra model synchronization framework, is illustrated in Figure 1. In this illustration, two models M and G,

which are instantiated from domain models DM_M and DM_G respectively, are in a consistent state. For DM_M and DM_G, the corresponding association rules A_R are established based on compatible domain types and relations. The rules from A_R are used to identify model dependency tuples D from elements of M and G. The consistent state between M and G is disrupted when M is transformed, through a sequence of transformations T_M, into M′. To identify elements of G that are affected by change to M, the tuples from D are used. For a tuple (m_i, g_j) in the dependency set D, if element m_i from model M is changed, then the element g_j from model G is identified as one of the elements that may need to altered to maintain the synchronization between M, G. Once all of the affected elements are found, they can be updated automatically if possible, or highlighted for manual updating.

In the proposed framework, systematic change of software models is attained through model transformations. These include insertion, deletion and modification of model attributes and model elements. The changes made at the model level are applied on properties and predicates defined at the domain model level. Each change performed on one of the models can be mapped or traced to its domain or even its corresponding metamodel. Hence, we can view individual model transformations in terms of their atomic elements (*i.e.*, as graph transformations) and recognize and interpret them as combinations of these basic elements.

To establish that two models are synchronized, a relation between the models needs to be defined. Such a synchronization relation can be a binary equivalence relation (*i.e.*, models are or are not synchronized) or expressed as a partial level or degree of synchronization. An example of a model dependency relation would be consistency between UML models that represent artifacts from different stages of design (*e.g.*, message sequence charts and collaboration diagrams) [1].

3.1. *Defining metamodels and domain models*

The models that need to be synchronized are based on schemas that we refer to as domain models. The elements of domain models represent, using MOF syntax, types, relations, and attributes that are specific to a particular domain. The domain models need to accurately represent their domain constituents in a format that provides for easy access and manipulation. Domain models based on MOF could be for instance represented in XML [23], where the domain model elements are used to define the corresponding DTD schema.

As an example, a domain model for business workflows, shown in Figure 2, is a schema represented in UML that denotes Processes, Tasks, Decisions, Data, *etc.* as UML classes, where process is a container class for itself and other classes.

A domain model can be instantiated to yield a concrete model that describes a specific process or a source code segment. Relationships between one or more domain models can be represented by the use of metamodels. In this context, models, domain models, and metamodels can be considered as typed, attributed, labeled, directed graphs. If the domain models are not available, as part of the requirements

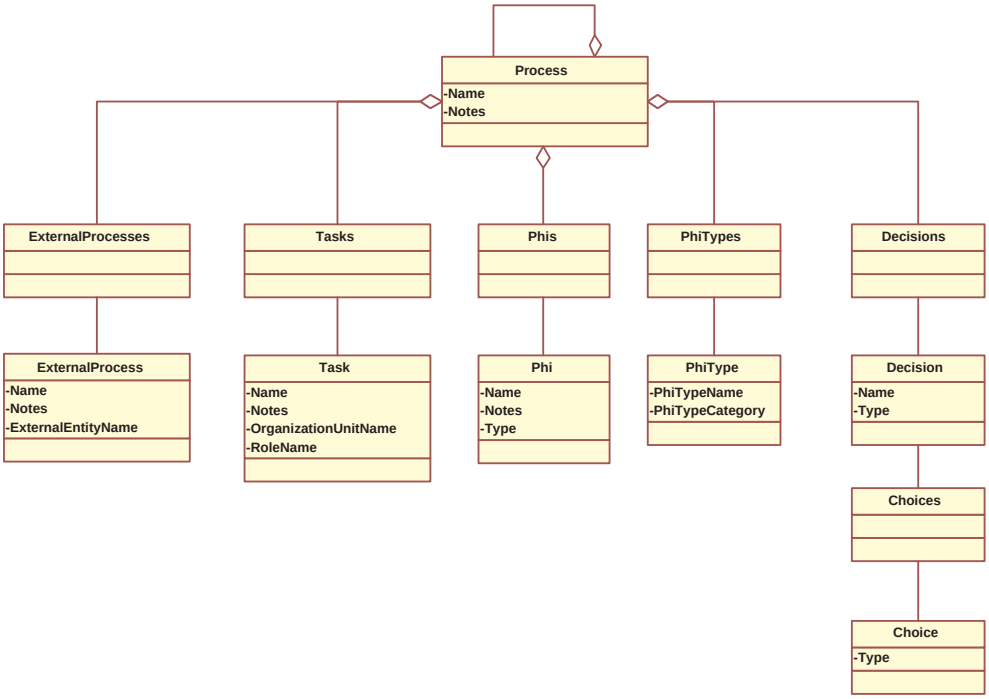


Fig. 2. Business Process Domain Model

or design documentation, they may need to be extracted using a technique such as Feature-Oriented Domain Analysis (FODA) [10].

3.2. *Generating intermediate models*

For two models that need to be synchronized, the relations between them can be established directly at the model level, as also at the domain model, or even meta-model level. To establish dependencies at the metamodel level, associations between MOF-compliant and non-MOF-compliant models need to be established first. Since we consider for our study only MOF-compliant models we therefore, do not discuss the problem of establishing relations between metamodels. Finally, to establish relations at the domain model level, types, relations, and attributes that are dependent need to be derived and recorded as association rules. To establish relations at the concrete model level, the rules established at one of the higher levels are used to identify tuples of model elements that are related.

Since the approach is focused on heterogeneous models at different levels of abstraction, establishing relations directly without refinement may be quite difficult. To overcome the differences in model expressiveness and semantics, we propose to generate more closely related intermediate models. These models would be less

syntactically and semantically diverse and therefore, establishing relations among them would be easier.

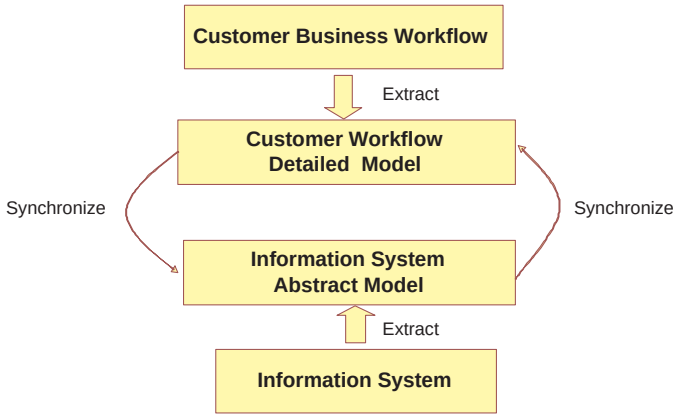


Fig. 3. Synchronizing Business Processes with Source Code

For example, in the problem of synchronizing business processes with source code, we annotate and enhance business process models with concrete data flow and control flow information, and at the same time, we abstract the source code to yield activity-like models. In Figure 3, this convergence is illustrated, where Customer Business Workflow models are annotated to yield Customer Workflow Detailed models, and Information System models are abstracted to yield Information System Abstract models.

We describe this process more formally as follows:

1. Let M and G be be instantiated from DM_M and DM_G respectively. Analyze DM_M and DM_G to establish compatible domain elements such as domain types, relations, events, *etc.*
2. Create convergent $\mathrm{DM}_M{}'$ and $\mathrm{DM}_G{}'$ from DM_M and DM_G respectively, so that $\mathrm{DM}_M{}'$ and $\mathrm{DM}_G{}'$ are more closely related and relations among them can now be established.
3. From M and G, generate M′ and G′ based on $\mathrm{DM}_M{}'$ and $\mathrm{DM}_G{}'$, respectively.

The sections below will discuss this process in more detail.

Figure 4 illustrates an example of an abstracted source code domain model. The source code elements such as classes and methods are represented through abstraction as ControllerCommand that are containers for other ControllerCommand and second-level abstractions such as JavaBeans for invocation of Java Beans, TaskCommands for external invocation, PseudoTaskCommands for clustered fragments of

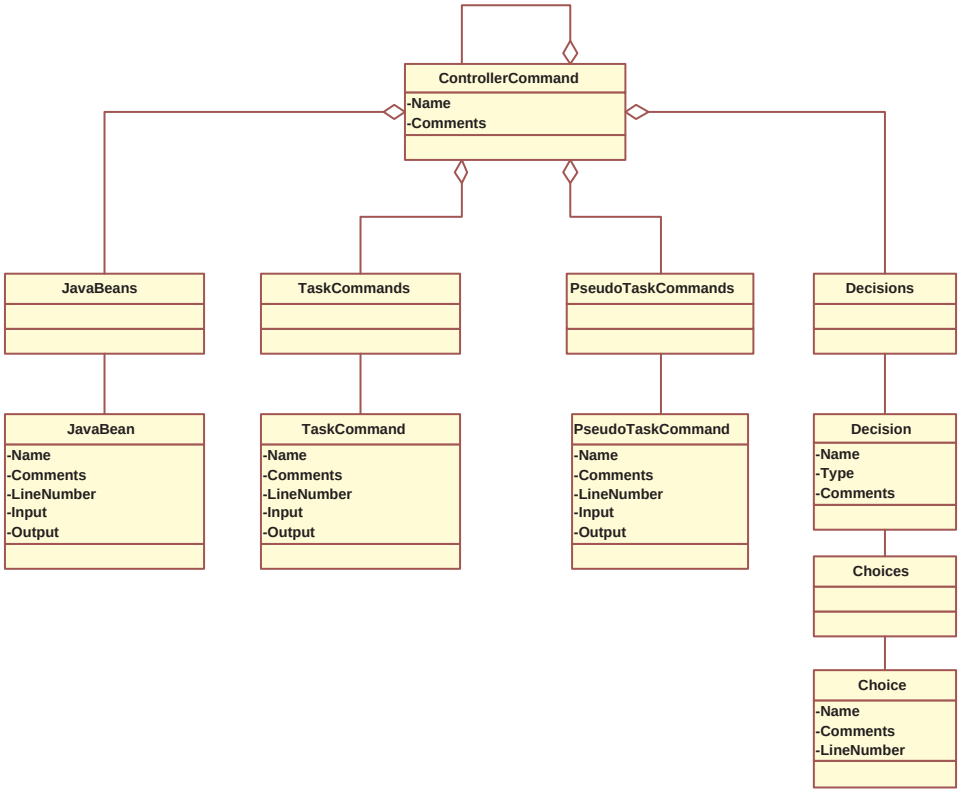


Fig. 4. Source Code Domain Model

code, and Decisions. The source code files are represented in XML, where this domain model is encoded as a DTD schema [24]. The mappings between source code and other models can thus be performed as mappings of XML Document Object Model (DOM) trees.

3.2.1. *Attribute association rules*

The domain models for a particular domain at a particular level of abstraction based on our assumption would have common properties such as consistent features maps, lexicons, ontologies, *etc.* The attribute association rules can thus be viewed as mappings between attributes of heterogeneous models based on the mappings of the domain-specific properties. The classification of the association rules is as follows, and the examples for each of the rules are based on the domain models for business workflows and source code (Figures 2 and 4 respectively).

- **Hierarchical Association Rules** — Models are parts of model hierarchies and

feature maps for the hierarchies are extracted. Based on the containment relations between models and features, two association types are recognized:

(a) direct, where models M and G are associated directly since they implement related features F_M and F_G, or
(b) indirect, where models M and G are associated indirectly since M contains a model M′ that implements a feature F_M that is related to a feature F_G that G implements. In this case, M′ and G are directly related.

To illustrate hierarchical association rules, let M be a process "Process order" and let M′ be process "Prepare inventory" that is contained within M and that implements a feature F_M "Inventory management". Also, let G be a source code class "InventoryAllocation" that implements a feature F_G "Inventory management", and let features F_M and F_G be related. From the relation between F_M and F_G, it follows directly that M′ and G are related, and indirectly that M and G are related since M contains M′.

- **Type-Based Association Rules** — Domain elements such as domain types, relations, and attributes defined in different domain models are associated based on compatible structural properties (*e.g.*, equivalent domain model, metamodel, or even metametamodel classification).
  To illustrate type-based association rules, let us consider the types Process and ControllerCommand of respective domain models. The compatible properties are that they are both root types of each model, that they contain themselves and other subtypes, and that they have related attributes (Name, Name) and (Notes, Comments). As a result, Process and ControllerCommand are associated.
- **Spatial Association Rules** — A flow of data in a model is represented as order attributes, such that object o_1 precedes object o_2. The associations are established based on the order attributes.
  To illustrate spatial association rules, let us consider a flow (m_1, m_2) for a model M and a flow (g_1, g_2) for a model G. The attributes of the first flow would be for m_1 "Preceded by null" and "Followed by m_2", and for m_2 "Preceded by m_1" and "Followed by null". The attributes for the second flow would be analogous, so m_1 and g_1 could be matched since they are both preceded by null, and m_2 and g_2 could be matched since they are both followed by null.
- **Text-Based Association Rules** — Informal information attributes (*i.e.*, comments, descriptions, variable names, types, *etc.*) are viewed as strings of text. The difference in syntax and semantics are resolved through:
  (a) thesaurus replacements — related synonyms are mapped,
  (b) stemming — each word is reduced to its root (*e.g.*, resource, resources, resourceful to resource),
  (c) abbreviation expansion — abbreviations that are recognized for a particular domain are expanded,
  (d) stop-word elimination — words with no semantic meaning, locally or globally, are eliminated,

(e) word-matrix matching — groups of attributes that share words with semantic meaning are recognized, and
(f) n-gram matching — strings are divided into substrings of size n and matched accordingly.

To illustrate text-based association rules, let m_a := "Verify the order line item is still old" and g_b := "Verify that this order item still meets the criteria for being stale" be related attributes for elements m ∈ M and g ∈ G. We perform text-based matching as follows:

(1) Eliminate stop words and replace synonyms to obtain m_a' := "verify order line item stale" and g_b' := "verify order item meets criteria stale"
(2) Use 3-gram matching and the match level of 0.6 on m_a'' := "ver eri rif ify ord rde der lin ine ite tem sta tal ale" and g_b'' := "ver eri rif ify ord rde der ite tem mee eet ets cri rit ite ter eri ria sta tal ale" to obtain average(3-gram-match(m_a'', g_b'''), 3-gram-match(g_b''', m_a''')) = 0.744 for the successful match.

The attribute association rules are formally represented as OCL descriptions [22]. The following illustrates in OCL the preceding 3-gram matching rule at the match level of 0.6:

```
M− >iterate( m : ModelElement |
    G− >iterate( g : ModelElement;
        result : Boolean = NGramMatching.ApplyRule( m− >a, g− >b, 3, 0.6)))
```

Unmatched objects

The unmatched objects may be recognized through association of their immediate neighbors based on additional or refined attributes and rules. For instance, in the following two flows (m_1, m_2, m_3, m_4) and (g_1, g_2, g_3, g_4), tuples (m_1, g_1), (m_1, g_2), (m_3, g_4) are found as model dependencies. It may be possible to ascertain dependencies for unmatched objects m_2, g_3, and m_4 by performing additional clustering on these objects and their matched neighbors with new or changed attributes and rules. If for instance m_2 is now found to be related to m_1 and g_1, a tuple (m_2, g_1) would be created.

Conflict resolution rules

For the objects that are a part of two or more clusters, it may be necessary for practical reasons to decide to which cluster they more strongly belong. This conflict resolution may be achieved by weighted scoring of individual rules, where some rules are assigned a higher value through initial experiments for a particular domain. The best match is selected by maximizing the weighted score. In case more than one cluster is found with the maximum score, all of the tuples from such clusters would be encoded as model dependencies.

More formally, let A_R be a set of applicable attribute association rules and let MG := M x G be a set of tuples of model elements. For a tuple (m_p, g_q) ∈ MG,

the weighted score $WS_{pq} := \sum w_i * ar_i(m_p, g_q)$ where ar_i : Boolean $\in A_R$ and w_i are domain-specific weight parameters. The maximum score for an element m_p is $WS_{pmax} := \max\{WS_{p0}, WS_{p1} \ldots\}$ and the top matches for an element m_p are tuples (m_p, g_i) for which the $WS_{pi} := WS_{pmax}$.

3.3. *Establishing model dependencies*

The complexity and the accuracy of individual mappings between models depends on the type and the amount of information that is available from each model. If we compare concurrent mapping of structural and temporal properties to mapping of only structural or only temportal properties, it holds that the number of eligible elements available would increase in the former case as would the complexity of the mapping. As a result, the accuracy of the mapping in the latter case may be higher.

This paper focuses on a set of specific domain model properties based on the corresponding structural, temporal, spatial, and behavioral attributes. As part of the approach, all domain models and all instantiated concrete models are viewed as directed, labelled, attributed graphs. All model properties are then viewed as labels and attribute-value pairs of individual nodes and edges. With regard to semantic homogeneity of considered domains, we assume that models belonging to a particular domain, such as database management, represented at a particular level of abstraction would be based on consistent features and types, and would also contain specific lexicons and ontologies. Based on this assumption, we create matches of models and model elements by associating related lexical and ontological concepts as part of the attribute association rules.

3.3.1. *Introduction to FCA*

Before using FCA to establish model dependencies, we first need to introduce related FCA definitions [4].

Definition 1. A *formal context* K := (O, A, I) contains two sets O and A, and a relation I between O and A. The elements of the first set O are called the objects, and the elements of the second set A are called the attributes of the context.

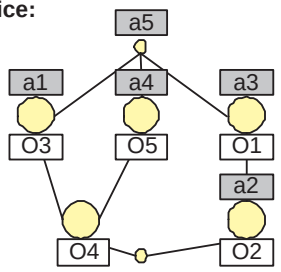
If we need to express that an object o from O is in a relation I with an attribute a from A, we would write this as oIa and read it as "the object o has the attribute a". The relation I in this case is called the incidence relation of the context M.

From this definition, we interpret domain models as contexts $DM := (O_{DM}, A_{DM}, I_{DM})$ that consist of a set of domain types O_{DM}, a set of domain attributes A_{DM}, and a set of domain relations I_{DM}. The models instantiated from respective domain models are also contexts $M := (O_M, A_M, I_M)$, but with model elements O_M, model attribute values A_M, and relations I_M, which are the mappings of types and attributes from domain model and the values in M. It follows then that a domain model DM_M is a metacontext for a model M that is instantiated from DM_M. Since

Context:

	a1	a2	a3	a4	a5
O1			x		x
O2		x	x		x
O3	x				x
O4	x			x	x
O5				x	x

Concept Lattice:



Association Rules:

1 < 5 > { } ==> a5;
2 < 0 > a3 a4 a5 ==> a1 a2;
3 < 1 > a2 a5 ==> a3;
4 < 0 > a1 a3 a5 ==> a2 a4;

Fig. 5. FCA Example

FCA dictates binary contexts, we provide a mapping of n-ary to binary relations through combinatorial scaling. For example, an object o and n possible values v_1, v_2... are represented as binary relations (o, $\emptyset$), (o, v_1), (o, (v_1, v_2)), ... (o, (v_1, v_2, ... v_n)).

For a set $O_1 \subseteq O$ of objects, let

$$A_1 := \{a \in A \mid oIa, \forall o \in O_1\} \tag{1}$$

be the set of attributes common to the objects in O_1. Also, for a set $A_2 \subseteq A$ of attributes, let

$$O_2 := \{o \in O \mid oIa, \forall a \in A_2\} \tag{2}$$

be the set of objects which have all the attributes in A_2.

Definition 2. A *formal concept* of the context (O, A, I) is a tuple (O_1, A_2) with $O_1 \subseteq O$, $A_2 \subseteq A$, $O_1 = O_2$ and $A_2 = A_1$, where O_1 is the extent and A_2 is the intent of the concept (O_1, A_2).

The relations between attributes in A are represented through a relation of format "$A_i \rightarrow A_j$", where A_i and A_j are subsets of A. This statement implies that

an object that has the attributes in A_i also has the attributes in A_j. To visualize these relations, a concept lattice can be built.

In Figure 5, a context of five objects $O := \{o_1, o_2, o_3, o_4, o_5\}$ and five attributes $A := \{a_1, a_2, a_3, a_4, a_5\}$ is represented as a matrix. Each cell in the matrix represents a relation between an object and an attribute, where a filled cell indicates that a particular object has a particular attribute (*e.g.*, $o_3 I a_1$ since the cell (3, 1) is filled). From the context, the attribute associations can be inferred as shown in the figure (*e.g.*, $\{\} \rightarrow a_5$ implies that all objects in the context exhibit attribute a_5). The concept lattice is built from the attribute associations, and from the concept lattice concepts can be read by reading the objects from the bottom and the attributes from the top (*e.g.*, object o_3 and attributes a_1 and a_5 are a concept, objects o_3, o_4, o_5 and attributes a_1, a_4, a_5 are a concept).

We refer to the rules for defining associations between attributes of different contexts as *attribute association rules*.

3.3.2. *Establishing model dependencies*

Before discussing the method for establishing dependencies, we first define model dependency as follows.

Definition 3. A *model dependency* between models M and G is a set of tuples (m_i, g_j) of elements obtained from models M and G such that m_i and g_j have associated attributes.

More formally, let M and G be two models at different levels of abstraction with corresponding domain models DM_M and DM_G. Let O_M and O_G be the objects and let A_M and A_G be the attributes for M and G. Also, let

$$A_{MG} := \{a_M, a_G \mid a_M \subseteq A_M, a_G \subseteq A_G, a_M \Rightarrow a_G\} \tag{3}$$

be related attributes from A_M and A_G. The attribute logic for attribute associations $a_M \Rightarrow a_G$ of attributes from different contexts is defined as a set A_R of attribute association rules.

For nonempty sets $O_M' \subseteq O_M$ and $O_G' \subseteq O_G$ of objects, let

$$A'_{MG} := \{a \in A_{MG} \mid o_1 I a, o_2 I a, \forall o_1 \in O'_M, \forall o_2 \in O'_G\} \tag{4}$$

be the set of attributes common to the objects in O_M' and O_G'. Also, for a nonempty set $A_{MG}' \subseteq A_{MG}$, let

$$B'_{MG} := \{o_1 \in O_M, o_2 \in O_G \mid o_1 I a, o_2 I a, \forall a \in A'_{MG}\} \tag{5}$$

be the set of objects which have all the attributes in A_{MG}'.

Definition 4. An *inter-context concept* of two contexts M and G is a set (O_M', O_G', A_{MG}') with $O_M' \subseteq O_M$, $O_G' \subseteq O_G$, $A_{MG}' \subseteq A_{MG}$, and $B_{MG} = (O_M' \cup O_G')$.

The algorithm for establishing model dependencies then includes the following steps.

Algorithm MD-FCA Establishing Model Dependencies Through FCA
Input:
1. Model M := (O_M, A_M, I_M)
2. Model G := (O_G, A_G, I_G)
3. Domain Model DM_M
4. Domain Model DM_G

Output:
1. Established Model Dependencies D

Steps:

Step 1. Identify a set A_{MG} of related attributes from A_M and A_G based on compatible properties such as matched types, associations, data flow, spatial properties, informal information, *etc.* of DM_M and DM_G.

Step 2. Given A_{MG}, derive a set A_R of corresponding association rules based on the matched properties of DM_M and DM_G. For example, for attributes matched based on spatial properties, use the spatial-matching rules.

Step 3. Iterate through elements of M and G to identify an initial set of inter-context concepts D that share the attributes from A_{MG} using the rules from A_R for association of concrete attributes.

Step 4. Exclude from D all of its irrelevant (*e.g.*, include only the objects from one context) and redundant (*e.g.*, equivalent association results based on different sets of attributes) elements.

Step 5. For those elements of M and G that cannot be matched using attribute associations, attempt matching using their already-matched neighbors.

Step 6. For those elements of M and G that are matched to more than one cluster, select the matchings using a corresponding conflict resolution rule. For example, each association rule could be assigned a weight and those elements with a maximum weighted score are selected as the top results.

Step 7. Return D as the set of established model dependencies.

To further clarify the algorithm, we demonstrate its usage on a simple scenario of two models, a workflow M and a source code class G, that were matched based on suitable properties. We focus on matching of their elements, so we apply the algorithm. For Step 1, we identify a set of compatible attributes A_{MG} := ((Process Name, Class Name), (Notes, Comments), (Subsystem, Package), (Previous Element in the Process Flow, Previous Element in the Information Flow), (Next Element in the Process Flow, Next Element in the Information Flow)) from the workflow and abstracted source code domain models. For Step 2, based on A_{MG} we identify

a set of association rules A_R that includes: text-based matching using a combination of techniques including stemming, abbreviation expansion, stop-word elimination, n-gram matching, *etc.* on attribute pairs (Process Name, Class Name) and (Notes, Comments); type-based and hierarchical matching by extracting workflow and source code hierarchies on attribute pair (Subsystem, Package), and type-based and spatial matching using position and data flow on attribute pairs (Previous Element in the Process Flow, Previous Element in the Information Flow) and (Next Element in the Process Flow, Next Element in the Information Flow). In Step 3, we iterate through elements of M and G to identify those that cluster together. For example, we matched the following two model elements:

- o_M := ((Process Name, "Find stale order line items"), (Notes, "Find order line items that are stale"), (Subsystem, "Order"), (Previous Element in the Process Flow, "Time to execute" (Task)), (Next Element in the Process Flow, "Are there more order items"(Decision))) and
- o_G := ((Class Name, "abOrderJDBCHelper.findStaleOrderItems"), (Comments, "Call the Order query to get the list of expired order items"), (Package, "Order"), (Previous Element in the Information Flow, "startUse" (Task)), (Next Element in the Information Flow, "hasMoreElements" (Decision))).

The matching of elements, if we consider text-based matching rules, is based on:

- n-gram distance between semantically relevant words using thesaurus replacements of (Process Name, "Find stale order line items") and (Class Name, "abOrderJDBCHelper.findStaleOrderItems"),
- n-gram distance and hierarchical (*i.e.*, Subsystem to Package) mapping of (Subsystem, "Order") and (Package, "Order"), and
- n-gram distance, type-matching (*i.e.*, Decision to Decision), and spatial matching of (Previous Element in the Process Flow, "Time to execute" (Task)) and (Previous Element in the Information Flow, "startUse" (Task)), and (Next Element in the Process Flow, "Are there more order items" (Decision)) and (Next Element in the Information Flow, "hasMoreElements" (Decision)).

3.4. *Validating established dependencies*

Once the dependency tuples are established, they have to be validated. If there exists a source of previously encoded dependencies, the validation may be automated. If not, developers and domain experts are to be queried for feedback, where the results are submitted for review. For unsatisfactory recall and precision levels, the process settings may need to be changed. For example:

- Existing attribute association rules may need to be excluded or new ones defined and added,
- Attribute associations at the domain model or metamodel level may need to be adjusted, and

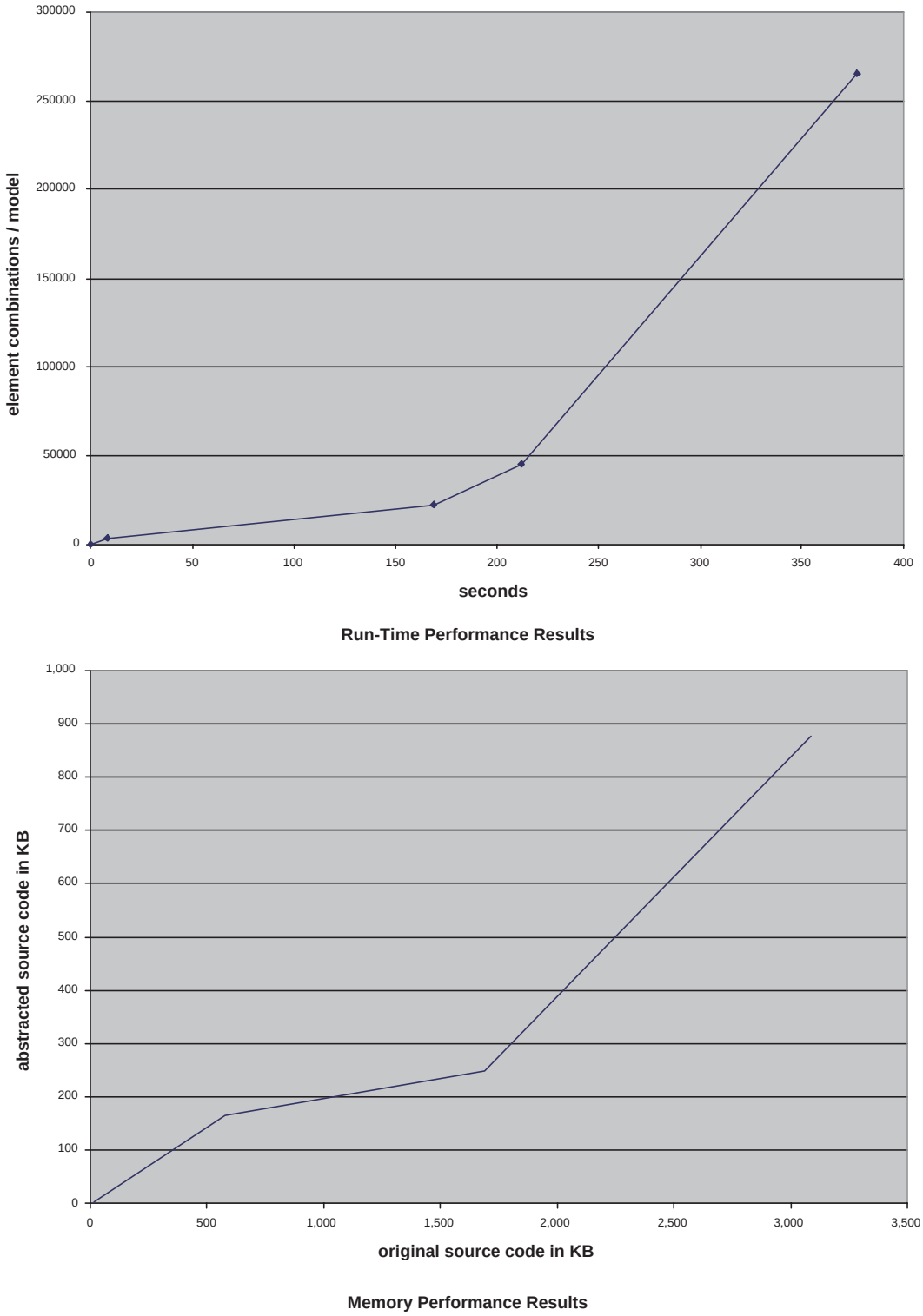


Fig. 6. Run-Time and Memory Performance Results

- Threshold levels, which can be used to eliminate results with scores lower than the threshold, may need to be increased (*i.e.*, higher precision but lower recall) or decreased (*i.e.*, higher recall but lower precision).

In its current form, the proposed framework is based on user assisted validation where dependencies are presented to the user for evaluation. A weighted score WS, as defined above, is used to rank the dependencies from the most plausible to the least plausible ones.

To analyze run-time and memory performance of the framework, we have performed a set of four subsystem queries, one for each of the four data sets from the case study and we have included the results as Figure 6. The run-time performance was measured in seconds versus number of element combinations per model that were parsed in the matching process, and the analysis of the results indicates a quadratic performance. The memory performance was measured in kilobytes of source code versus kilobytes of abstracted source code, and the analysis indicates approximately linear performance.

4. Case Study: Synchronizing Business Workflows and J2EE Source Code

We have implemented a prototype of our framework in Java, and we have integrated it with the previously developed mSynTra plug-in for the Eclipse development environment [20].

Figure 7 depicts the prototype as a Workflow Synchronization Plug-In (WSP) Perspective in Eclipse. The dependency view shows two model repositories, the BPMs on the left and the source code models on the right. Clicking on one of the model elements on the left will show the corresponding dependencies selected on the right, with the details of the match including positively matched rules, the weighted score, and other possible matches in the middle. Additional interface is provided for reparsing of model elements based on the changed criteria such selecting or deselecting particular model attributes, and increasing or decreasing the threshold level for the weighted score. Through these settings users can change the size of the domain of matching, and hence, iteratively improve the accuracy of the results.

The prototype is used in a case study of synchronizing business process models (BPM) represented as business workflows with the enacting Java 2 Enterprise Edition (J2EE) platform compliant source code [19]. Based on the domain analysis and discussion with different stakeholders, it was concluded that the business workflows are typically created independently of source code. It was also noted that the mappings between the related models are not consistently recorded, and as a result, they are incomplete and out of date. Thus, developers and architects of the system would be required to validate the extracted model dependencies.

The main target for the case study is to enable bidirectional change propagation, where changes from a workflow are propagated to underlying source code and changes from a source code file are propagated to related workflows. The first step

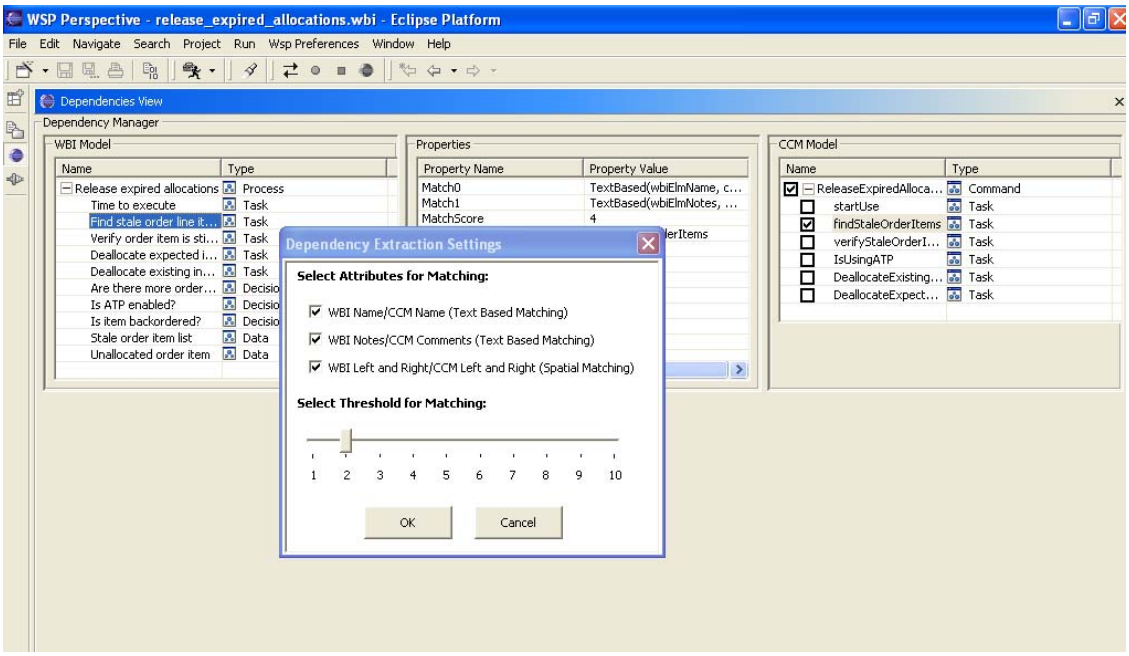


Fig. 7. Prototype Implementation in Eclipse

in this process is to establish relations among models and model elements, and the approach presented in this paper is used to fulfill the requirements of this goal.

In the ensuing text, we describe how each of the steps of the approach is applied. Namely, we discuss the process of recovery of intermediate models to bridge syntactic and semantic gaps between the workflows and the source code. The derived intermediate models are used as a basis for extraction of inter-context concepts, derived as clusters of objects that share related attributes. The extracted dependencies are validated based on the corresponding feedback, from which the precision and recall levels are also measured.

4.1. *Generating intermediate models*

In this step, we transform the business workflows by enriching and annotating their representations. We also transform the source code models through abstraction and annotation.

The workflow transformation includes:

1. Analyzing the workflow models and deriving their simplified representation in relation to source code artifacts. The workflow functionality that is not performed by a machine along with elements that have no source code representation are excluded. The data flow and control flow information is preserved. The call in-

formation for processes and run-time modules, which can be found in the source code, is also preserved.
2. Adding annotations as attributes to workflow elements for which additional information may be derived. The annotations may include attributes for context, role, hierarchical relation, implemented features, *etc.*
3. Extracting content from workflows automatically into XML, with the domain model schema as a DTD for the XML files.

The source code transformation includes:

1. Analyzing the source code models and deriving their simplified representations in relation to business process artifacts. The source code functionality that has no relation to business processes along with elements that have no workflow representation are excluded. The data flow and control flow information is preserved. The call information for task commands, pseudo task commands, and Java Beans is also preserved.
2. Adding annotations as attributes to source code elements for which additional information may be derived. The annotations may include attributes for parameter passing, informal text, hierarchical relation, implemented features, *etc.*
3. Extracting content from source code files automatically into XML, with the domain model schema as an XML DTD.

4.2. *Establishing model dependencies*

From the refined domain models, the attribute association rules may be created. For the BPM domain model, type-based and hierarchical associations are created. The types are mapped to the compatible source code types, while processes are recognized as parts of the overall hierarchy and relations are established based on directly or indirectly implemented features. For the source code domain model, hierarchical associations could not be established in addition to the type-based ones since no meaningful hierarchy of source-code models could be identified. For the concrete models, the compatible attributes are identified and spatial and text-based association rules are defined. The settings for each rule are adjusted through initial experimentation on a representative set of related workflow and source code models. Using FCA and the association rules, dependency tuples are identified and the results are stored in XML.

Figure 8 shows the mapping of the BPM and the source code attributes and properties. Different levels of mapping, from Level 0 to Level 3, indicate different steps in the matching process: at Level 0, hierarchical and type-based clustering of top-level elements (*i.e.*, processes and classes); at Level 1, clustering of top-level elements using specified attributes; at Level 2, clustering of model elements using specified attributes; and at Level 3, matching of unmatched elements using spatial information.

Figure 9 demonstrates a successful mapping between a workflow (at the top)

	comparing	Business Flow	Source Code
Level 0:	Objects	A set of workflow processes	A set of Java source code files
	Properties	Hierarchical and type-based associations	Type-based associations
Level 1:	Objects	Workflow process	A set of corresponding Java classes
	Attributes	Process name, process notes	Class name, top-level comments
Level 2:	Objects	Elements of the workflow process	Elements of the source code flow
	Attributes	Element name, type, description, data input and output, spatial info	Element name, type, comments, method params, spatial info
Level 3:	Objects	Unmatched elements of the workflow process	Unmatched elements of the source code flow
	Attributes	Spatial information	Spatial information

Fig. 8. Attribute and Property Mappings

and a corresponding source code model (at the bottom). As part of this figure, two spatial association rules are presented in OCL.

4.3. *Validating established dependencies*

Since the previously established model relations are not available, or are, at best, incomplete, the validation process is based on feedback from developers and architects. Several iterations are performed, each with the goal of improving precision and recall levels through adjustment of corresponding settings (*e.g.*, change of attribute associations, addition of new rules).

Based on the set of four queries, one for each of the four subsystems that were available to us, we have measured and analyzed the precision and recall levels for the top-level matches derived using our framework. Figure 10 shows the initial results of estimated precision versus estimated recall, where it is indicative that there is a decline in recall levels from approximately eighty to approximately sixty percent with the precision slightly increasing but remaining close to twenty six percent. The decline in recall levels may depend on several factors including:

- possible attrition of information when considering source code elements that have indirect or partial relations to their business workflow counterparts,
- nonconformity of certain related workflows and source code flows, and
- inconsistency and drift that may have occurred over time between related workflow and source code models.

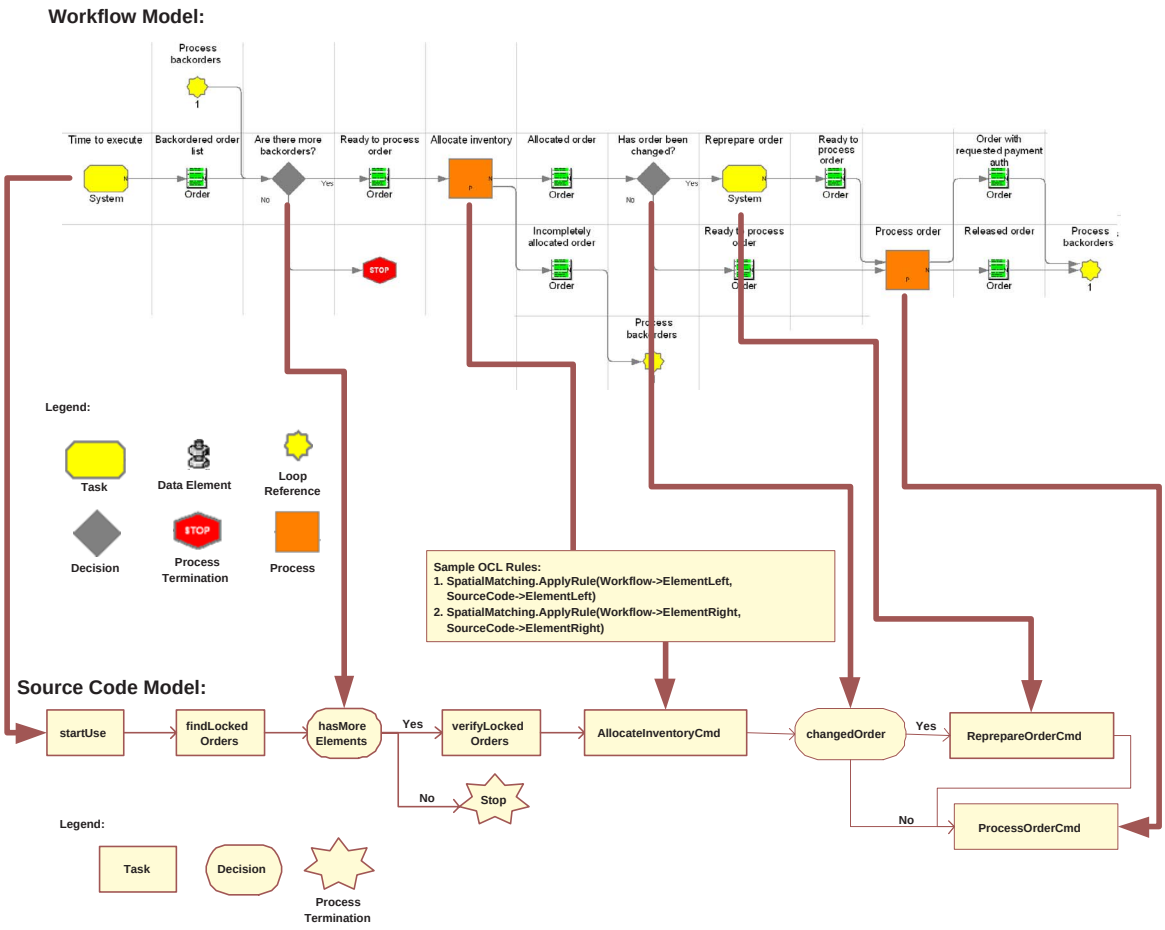


Fig. 9. Dependency Mapping Example

5. Conclusions and Future Research

This paper discussed a framework that makes use of Formal Concept Analysis (FCA) to systematically establish relations among models and their elements. The steps in the framework included (1) creation of domain models for chosen domains, (2) generation of intermediate models to bridge possible representational and abstraction gaps between domains, (3) establishing model dependencies based on attribute association rules using FCA, and (4) validation of established dependencies. For the third step, details and an OCL representation for attribute association rules is discussed along with an algorithm for establishing model dependencies. The approach was evaluated and its applicability demonstrated through a case study of synchronizing business workflows with its underlying J2EE source code.

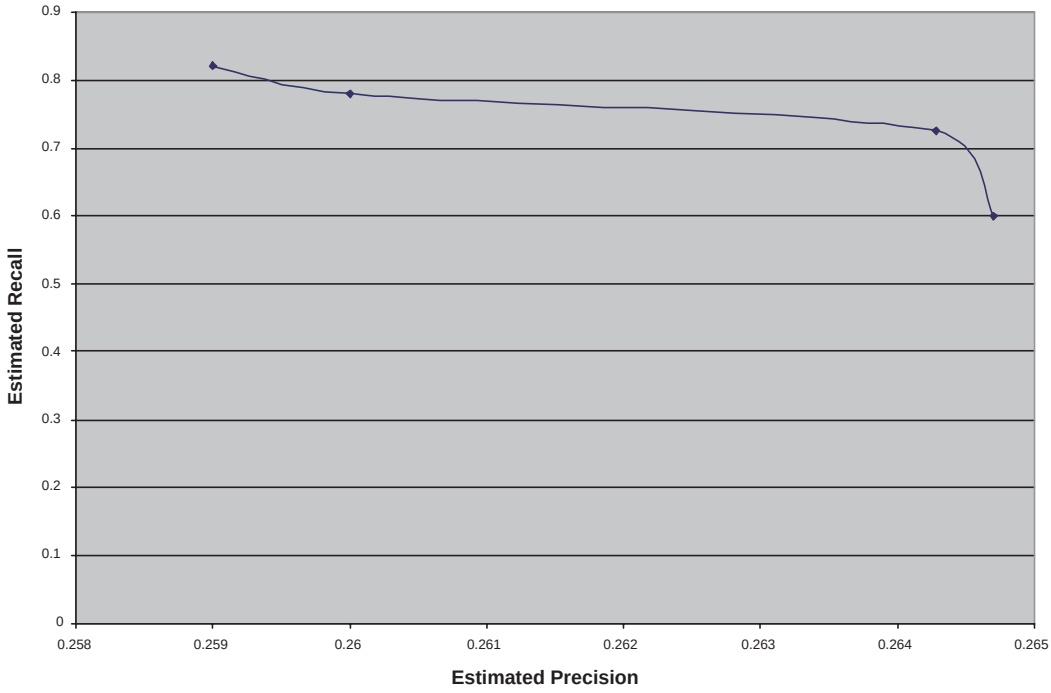


Fig. 10. Estimated Precision and Recall

In the next steps of our research, we aim to further extend the approach by applying it to additional case studies that exhibit different concerns. We also plan to investigate an approach where model synchronization can be viewed as a language translation problem, whereby models are treated as sentences in languages that are defined through grammar-based representation of their domain models. The synchronization would then become a problem of language translation, where algorithms from the theory of natural language processing (NLP) could also be considered as part of a possible solution.

6. Acknowledgements

This work is funded in part by and performed in collaboration with the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto. We would especially like to thank Tack Tong and Ross McKegney for their comments and suggestions.

References

1. A. F. Egyed. *Heterogeneous View Integration and Its Automation*. PhD thesis, University of Southern California, August 2000.
2. G. Engels, R. Huecking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proceedings of the Second International Conference on The Unified Modeling Language (UML)*, Fort Collins, CO, Oct 1999.
3. M. Faid, R. Missaoui, and R. Godin. Knowledge Discovery in Complex Objects. *Computational Intelligence*, 15(1), Jan 1999.
4. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.
5. G. K. Holman. What is XSLT? *O'Reilly's XML.com*, Aug 2000. http://www.xml.com/pub/a/2000/08/holman/.
6. IBM. Rational Unified Process (RUP). Online by IBM Corporation, 2004. http://www.ibm.com/software/awdtools/rup/.
7. I. Ivkovic and K. Kontogiannis. Using Formal Concept Analysis to Establish Model Dependencies. In *Proceedings of the IEEE International Conference on Information Technology Coding and Computing*, Las Vegas, NV, Apr 2005.
8. I. Ivkovic and K. Kontogiannis. Model Synchronization as a Problem of Maximizing Model Dependencies. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, BC, Oct 2004.
9. I. Ivkovic and K. Kontogiannis. Tracing Evolution Changes through Model Synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago, IL, Sep 2004.
10. L. Kean. Feature-Oriented Domain Analysis. *Software Technology Review*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.
11. A. Kleppe, J. Warmer, and W. Bast. *The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
12. OMG. Model Driven Architecture - A Technical Perspective. Object Management Group's (OMG's) Architecture Board ORMSC Document ORMSC/01-07-01, Object Management Group, Jul 2001.
13. OMG. Meta Object Facility (MOF) Specification Version 1.4. Technical report, Object Management Group (OMG), April 2002. http://www.omg.org/docs/formal/02-04-03.pdf.
14. OMG. Unified Modelling Language (UML) Specification. Technical report, Object Management Group, Mar 2003. http://www.omg.org/docs/formal/03-03-01.pdf.
15. C. Rich and L. M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, Jan 1990.
16. P. Rodriguez-Gianolli and J. Mylopoulos. A Semantic Approach to XML-Based Data Integration. In *Proceedings of the 20th International Conference on Conceptual Modeling*, 2001.
17. G. Spanoudakis and P. Constantopoulos. Measuring Similarity between Software Artifacts. In *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering (SEKE 94)*, Jurmala, Latvia, Jun 1994.
18. Sun Microsystems. Enterprise JavaBeans (EJB) Technology Specification. Online by Sun Microsystems Inc, Nov 2003. http://java.sun.com/products/ejb/docs.html.
19. Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE) Specification. Online by Sun Microsystems Inc, Nov 2003. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.
20. The Eclipse Foundation. Eclipse Platform Technical Overview. Online by the Eclipse Foundation, Jul 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

21. T. Tong, R. McKegney, T. Lau, K. Kontogiannis, I. Ivkovic, P. Liew, Y. Zou, Q. Zhang, and M. Hung. Model Synchronization for Efficient Software Application Maintenance. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC 2004)*, Bari, Italy, Jun 2004.
22. J. Warmer and A. Kleppe. *The Object Constraint Lanugage (OCL)*. Addison-Wesley, 1999.
23. W3C XML Core Working Group. Extensible Markup Language (XML) Specification. Technical report, W3C, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.pdf.
24. Y. Zou, T. Lau, K. Kontogiannis, T. Tong, and R. McKegney. Model Driven Business Process Recovery. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Amsterdam, The Netherlands, Nov 2004.