Modernizing Legacy Systems through Runtime Models

Ricardo Pérez-Castillo[1], Barbara Weber[2], Ignacio García-Rodríguez de Guzmán[1] and Mario Piattini[1]
[1] Alarcos Research Group, University of Castilla-La Mancha
Paseo de la Universidad, 4 13071, Ciudad Real, Spain
{ricardo.pdelcastillo, ignacio.grodriguez, mario.piattini}@uclm.es
[2] University of Innsbruck
Technikerstraße 21a, 6020 Innsbruck, Austria
barbara.weber@uibk.ac.at

Resumen

Software modernization advocates reengineering processes for legacy information systems taking model-driven development principles into account. Modernization projects consider different legacy software artifacts as knowledge sources like, for example, source code, databases, user interfaces. In addition, the knowledge necessary to modernize a respective legacy system is extracted by analyzing the legacy artifacts in a static way. Unfortunately, there is a large amount of knowledge that is only known during system execution. Thus, this paper suggests a technique based on dynamic analysis of source code to obtain runtime models representing the system execution events. The technique is contextualized within MARBLE, a modernization framework to obtain business processes form legacy information systems. Firstly, the technique obtains a runtime model that represents events related to the execution of the business activities of the business processes supported by the legacy system. Secondly, a model transformation is proposed to obtain a higher-level model from the runtime model, which is represented according to an extended event metamodel of the Knowledge-Discovery Metamodel standard. As a consequence, the runtime model can be integrated and used in any modernization scenario in a standardized manner.

1. Introduction

Most companies have existing information systems which can be considered as *legacy systems,* because the code in these systems was written long time ago and in the meantime is technologically obsolete. Legacy Information Systems (LISs) are information systems that significantly resist modification and evolution to meet new and constantly changing business requirements [19]. The continuous evolution implies that the maintainability of LISs eventually diminishes below acceptable limits requiring the LISs to be modernized [15]. Modernization means that the LIS is re-implemented using another, better platform or an enhanced design, while the business knowledge of the system is preserved [11]. When a LIS evolves over time it embeds much business knowledge. The preservation of this knowledge during system modernization is a very important challenge to be addressed for two main reasons: (i) the embedded knowledge is not present in any other artifact, and (ii) it must be considered to align the new improved system with the current business processes of the organization [9].

In the past, reengineering was the main tool for addressing the evolutionary maintenance of legacy systems preserving the business knowledge [1]. However, reengineering is usually carried out in an *ad hoc* manner, thus it fails when it is applied to large and complex LISs [23]. Since reengineering lacks formalization and standardization it is very difficult to automate reengineering for those large systems. Nowadays, the typical reengineering concept has shifted to so-called Architecture-Driven Modernization (ADM) [16] as a solution to the formalization and standardization problems. ADM advocates carrying out reengineering processes following the MDA (Model-Driven Architecture) standard [12], i.e., it treats all the legacy software artifacts as models and establishes model transformations between the different MDA abstraction levels. In addition, ADM defines the standard KDM (*Knowledge Discovery Metamodel*), which has also been recognized as standard ISO 19506 [10]. KDM provides a common repository structure that makes it possible to exchange information about existing software artifacts involved in LISs. KDM can be compared with the *Unified Modeling Language* (UML) standard; while UML is used to generate new code in a *top-down* manner, ADM-based processes involving KDM start from the legacy source code and build a higher level model in a *bottom-up* manner [13].

There are several works in literature that address the problem of preserving the embedded business knowledge in software modernization processes. *Zou et al* developed a MDA-based framework for extracting business processes through static analysis of source code based on a set of heuristic rules [26]. *Pérez-Castillo et al.* [20] propose MARBLE, an ADM framework that uses the KDM standard to obtain business processes from legacy source code. Source code, however, is not the only legacy artifact considered to recover business knowledge. *Di Francescomarino et al*, for example, consider graphical user interfaces of Web applications to recover business processes [5]. *Paradauskas et al.* [19], in turn, present a framework to recover business knowledge through the inspection of the data stored in databases. *Ghose et al* [7] propose a set of text-based queries in source code and documentation for extracting business knowledge. In addition, *Cai et al.* [2] propose an approach that combines the requirement reacquisition aided by system users with static analysis. All these works are mainly based on a static view of LISs, and runtime models are often ignored to preserve the business knowledge. However, since there exists much valuable information that is only known during system execution, there is a big potential for exploiting runtime knowledge as well.

This paper presents, within MARBLE, a reverse engineering technique based on dynamic analysis (combined with static analysis) of source code to obtain runtime models. Firstly, the static analysis syntactically analyzes the source code and injects pieces of source code in a non-invasive way in specific parts of the system. Secondly, the dynamic analysis of the modified source code makes it possible to write events during system execution. The events are written according to the MXML (Mining XML) metamodel [8], an XML format used in the process mining field to obtain event logs. The proposed technique is further supported by specific information provided by business experts and system analysts who know the system.

The runtime models represent the events related to the underlying business processes that occur during system execution, thus these models provide another valuable source of knowledge to understand what is actually going on in a LIS from a dynamic perspective [24]. To facilitate the use of runtime models for modernizing LISs, this paper provides a model transformation between levels L1 and L2 of the MARBLE framework to represent the runtime models according to the KDM standard. Due to the fact that KDM is considered as the common modernization format for representing any legacy artifacts, these models can be used for any modernization framework.

The remainder of this paper is organized as follows. Section 2 briefly introduces MARBLE, the modernization framework for which the technique is proposed. Section 3 presents the reverse engineering technique to obtain runtime models, and Section 4 shows the model transformation defined to represent runtime models in KDM. Finally, Section 5 provides conclusions and discusses future work.

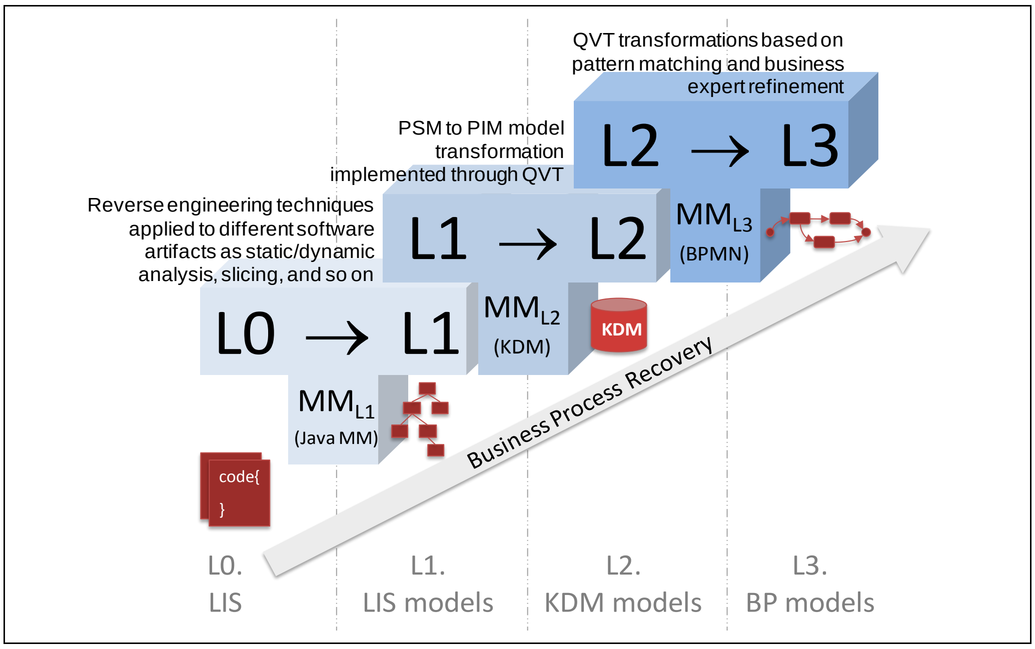
2. MARBLE

MARBLE (*Modernization Approach for Recovering Business processes from LEgacy systems*) [20] is an ADM-based framework for recovering business processes from legacy systems and business knowledge preservation. MARBLE is organized into four abstraction levels (cf. Section 2.1) representing four different kinds of artifacts needed to obtain the embedded business processes. In addition, MARBLE specifies three model transformations (cf. Section 2.2) to complete the path obtaining each kind of model at a specific level from the previous one (see Figure 1).

2.1. Abstraction levels defined in MARBLE

MARBLE defines four abstraction levels related to four different kinds of models: L0 to L3. These models are progressively refined from legacy software artifacts in L0 until business process models are obtained in L3 (see Figure 1).

- **L0. Legacy information system.** This level represents the entire legacy system in the real world, i.e. a set of interrelated software artifacts like source code, user interfaces, databases, documentation.
- **L1. Software artifacts models.** This level contains a set of models representing one or more software artifacts of the LIS. L1 models can be seen as platform-specific models (PSM) because they represent different views or concerns of the system from a technological point of view at a lower abstraction level. As a consequence, L1 models must be represented according to specific metamodels (e.g. a hypothetic L1 level could be formed by a code model represented according to the Java metamodel and a database model depicted according to the SQL metamodel).
- **L2. KDM model.** This level integrates all knowledge of the L1 models in a single model, but at a higher abstraction level. This model represents the entire LIS from a platform-independent point of view at an intermediate abstraction level (PIM). This model is represented according to the metamodel of the KDM standard. KDM provides a common repository structure that makes it possible to exchange information about artifacts involved in the LIS.
- **L3. Business process model.** L3 is the top level and corresponds to a business process model that represents the recovered business processes. This model can be seen as computer-independent models (CIM), since it depicts a business view of the system from a computation independent viewpoint at the highest abstraction level. MARBLE uses the metamodel of the BPMN (Business Process Modeling and Notation) standard [18] for representing the business process model. BPMN offers a well-known graphical notation that is easily understood by both system analysts and business experts.



QVT transformations based on pattern matching and business expert refinement
L2 → L3
PSM to PIM model transformation implemented through QVT
L1 → L2
MM_L3 (BPMN)
Reverse engineering techniques applied to different software artifacts as static/dynamic analysis, slicing, and so on
L0 → L1
MM_L2 (KDM)
KDM
MM_L1 (Java MM)
code{ }
Business Process Recovery
L0. LIS
L1. LIS models
L2. KDM models
L3. BP models

Figure 1. Abstraction level organization and model transformation in MARBLE.

2.2. Model transformations in MARBLE

Besides different abstraction levels, MARBLE defines three model transformations between the four abstraction levels (see Figure 1).

- **L0-to-L1 transformation.** The first transformation takes the different software artifacts from the LIS (L0) and obtains a specific model for each artifact (L1). This transformation takes software artifacts into account depending on the specific business process recovery method based on MARBLE (since MARBLE defines a generic framework). So far, we consider a recovery method using MARBLE, which considers legacy source code as the unique software artifact, since it is the artifact that embeds most business knowledge [14]. In this case, the L0-to-L1 transformation consists of the static analysis of the source code files carried out by means of a syntactical parser, which generates a source code model according to the proper metamodel.
- **L1-to-L2 transformation.** The second transformation establishes the model transformation between the code model (the PSM model in L1) and the KDM code model (the PIM model in L2). The KDM metamodel is divided into different metamodel packages organized into four abstraction layers. Each package focuses on modeling a different concern of the LIS. Currently, the KDM model in L2 only considers the KDM packages *code* and *action* that conform to the *program element layer*. These packages are enough to represent all concepts of the code models of L1 at a higher abstraction level in L2. This model transformation is formalized by means of QVT (Query / Views / Transformations) [17].
- **L2-to-L3 transformation.** The third transformation aims to obtain a business process model in L3 from the KDM model in L2. This transformation consists of two steps: (i) a model transformation that obtains a set of preliminary business process models; and (ii) an optional manual intervention by business experts for refining the obtained business processes to improve them. So far, the model transformation of the first step uses a set of business patterns [21], which define what pieces of the source code (represented in the KDM model) are transformed into well-known structures of business processes. Then, the pattern matching following those patterns is implemented using QVT [22].

3. A Technique to Obtain Runtime models

Despite source code models are valuable models at level L1 of MARBLE, there are specific, relevant aspects of the source code (e.g. the accurate execution order of the pieces of source code, dead source code) which are lost if only static analysis is used. Thus, dynamic analysis can be used together with static analysis, additionally considering knowledge related to system execution, to obtain more meaningful business knowledge. For this reason, we propose a technique based on dynamic analysis to extract runtime models at level L1 of MARBLE.

The technique proposes the representation of runtime models as event logs derived from the system execution. Event logs are commonly used in the process mining field as the input for several mining algorithms to discover the business process of an organization [3]. Thereby, event logs save the list of business activities carried out in an organization according to their business processes. Usually, these event logs are obtained from Process-Aware Information Systems (PAIS) [6], i.e., process management systems (e.g. Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems). The nature of these systems (in particular their process-awareness) facilitates the registration of events throughout process execution.

The vast majority of LISs, however, are non process-aware systems that also support the business processes of organizations. Obtaining an event log of a non process-aware system and representing it in a model at level L1 of MARBLE implies five key challenges:

- **Challenge C1. Missing Process-Awareness**. Process definitions are implicitly described in legacy code. A traditional LIS consists of a control flow graph implicitly representing the business process it supports. Thus, it is not obvious which events (related to a specific business activity) should be recorded in the event log. To address this challenge, the technique considers the *"a callable unit / a* *business activity"* principle proposed by *Zou et al.* [25].

- **Challenge C2. Granularity**. While some of the callable units of LISs support the main business functionalities, many callable units are very small and do not directly support any business activity (e.g. setter/getter methods, printer methods, etc.).
- **Challenge C3. Discarding Technical Code**. Legacy source code not only contains business activities, but also technical aspects which have to be discarded when the runtime model is obtained.
- **Challenge C4. Process Scope**. Due to the fact that traditional LISs do not explicitly define processes, it has to be established when a process starts and ends. Unfortunately, this information is only known by business experts and system analysts.
- **Challenge C5. Process Instance Scope**. It is not obvious how business activities and the multiples instances of a process should be correlated. To solve this challenge the system analyst's knowledge is necessary.

Our technique for obtaining runtime models representing an event log is based on a static analysis of source code combined with a dynamic analysis. Firstly, the static analysis examines the legacy source code and modifies it by injecting code for writing specific events during its execution (cf. Section 3.1). After the static analysis has been conducted, the modified source code is dynamically analyzed at runtime by means of the injected sentences (cf. Section 3.2). Figure 2 gives an overview of the technique, the tasks carried out and their inputs/outputs.

3.1. Static analysis to modify the source code

The static analysis modifies the original source code in a non invasive way to enable the registration of events during system execution (see Figure 2). To address the previously introduced challenges, the static analysis is supported with information provided by business experts and system analysts. In Task 1, business experts establish the start and end business activities of the business processes to be discovered (Challenge C4). In parallel, system analysts examine in Task 2 the legacy source code and filter the *domain set* of the directories, files or specific callable units that support business activities. This information is used to reduce potential noise in the runtime model due to technical source code (Challenge C3). Task 3 consists of the mapping by system analysts between start/end business activities and the callable units supporting them (Challenge C4). In addition, system analysts establish through Task 4 the *correlation data* set for each callable unit which is uniquely identifying a process instance (Challenge C5). Each *correlation data* is mapped to one or more parameters of each callable unit by system analysts. Finally, Task 5 carries out the syntactic analysis of the source code. A parser analyzes and injects on the fly the sentences for writing the event long during system execution.

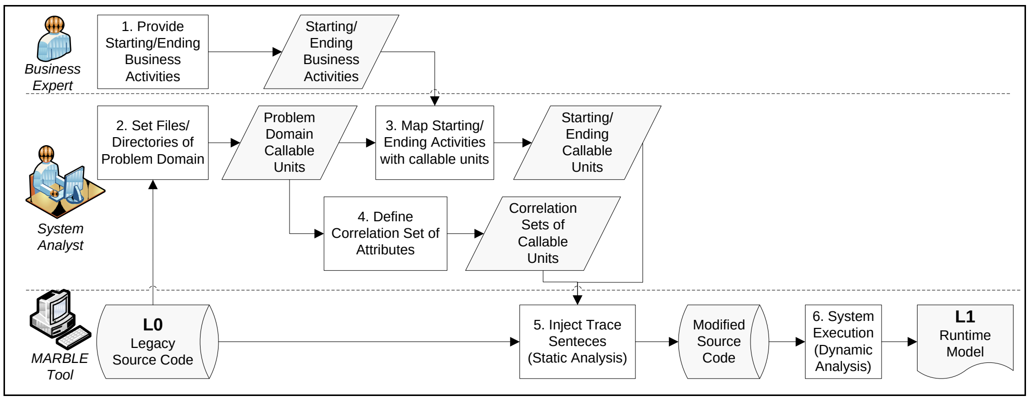

Business Expert
1. Provide Starting/Ending Business Activities
Starting/ Ending Business Activities
System Analyst
2. Set Files/ Directories of Problem Domain
Problem Domain Callable Units
3. Map Starting/ Ending Activities with callable units
Starting/ Ending Callable Units
4. Define Correlation Set of Attributes
Correlation Sets of Callable Units
MARBLE Tool
L0 Legacy Source Code
5. Inject Trace Senteces (Static Analysis)
Modified Source Code
6. System Execution (Dynamic Analysis)
L1 Runtime Model

Figure 2. The overall process carried out by means of the proposed technique.

Task 5 is automated following the algorithm presented in Figure 3. During the static analysis, the source code is broken down into callable units (Challenge C1), although the algorithm only modifies the units of the *domain set* selected by system analysts in Task 3 (Challenge C3). In addition, fine-grained callable units (e.g., *setter*, *getter*, *constructor*, *toString* and *equals* callable units) are automatically discarded (Challenge C2). After that, two sentences are injected at the beginning and the end of each filtered callable unit. The first sentence writes a start event related to the business activity mapped to the callable unit. This sentence is injected between the signature and the body of the callable unit. The second sentence writes an end event for the respective business activity and is injected at the end of the body. Both sentences have additional parameters like the *correlation data* defined for the unit and the information whether or not the unit represents a start or end activity. This additional information is used when the injected sentences invoke the *writeEvent* function at runtime, which writes the respective event into the runtime model (cf. Section 3.2).

```
injectTraces (CallableUnits, DomainCallableUnits,
StartingCallableUnits, EndingCallableUnits)
  ModifiedCallableUnits ← $\phi$
  c' ← null
  For (c $\in$ CallableUnits)
    If (c $\in$ DomainCallableUnits  and
      isFineGrainedUnit(c))
      If (c $\in$ StartingCallableUnits)
        position ← 'first'
      Else If (c $\in$ EndingCallableUnits)
        position ← 'last'
      Else
        position ← "intermediate"
      $sentence_1$ ← "writeEvent (c.name, 'start',
        position, c.correlationSet)"
      $sentence_2$ ← "writeEvent (c.name, 'complete',
        position, c.correlationSet)"
      c'.signature ← c.signature
      c'.body ← $sentence_1$ + c.body + $sentence_2$
      ModifiedCallableUnits ←
        ModifiedCallableUnits $\cup$ {c'}
    Else
      ModifiedCallableUnits ←
        ModifiedCallableUnits $\cup$ {c}
  Return ModifiedCallableUnits
```

Figure 3. Algorithm to inject trace sentences by means of static analysis.

3.2. Dynamic analysis to obtain runtime models

After having modified the source code through static analysis it is released to production. The new code makes it possible to obtain runtime models representing the event log of the LIS. These runtime models are represented in MARBLE according to a metamodel based on the MXML format [8], which is used in the process mining field.

Figure 4 shows the MXML metamodel, which provides the *WorkflowLog* metaclass to represent an event log as a set of instances of the *Process* metaclass. Each *Process* element contains several *ProcessInstances*, which have a sequence of *AuditTrailEntry* elements. Each *AuditTrailEntry* element represents an event and consists of four main elements: (i) the *WorkflowModelElement* that represents the executed activity; (ii) the *EventType* that represents if the activity is being executed (start) or was completed (complete); (iii) the *Originator* that provides the user who starts or completes the activity; and finally (iv) the *Timestamp* that records the date and time of the event. Moreover, all these elements can have a *Data* element including additional information endorsed into *Attribute* elements.

Dynamic analysis is automatically carried out during system execution. Thus, when the control flow of the LIS reaches an injected sentence, a new event is added to the event log. The events are written by means of the *writeEvent* function. Before adding the new event representing a business activity to the runtime model, it is necessary to find out the correct process and process instance where the event must be added. The adequate process and process instance are located by means of *Xpath* expressions [4]. If the process is null, then a new process is created. In addition, these expressions take the *correlation data* into account to establish the correct process instance. The attributes that contain the *correlation data* were already established during static analysis, however, their values are only known during system execution.

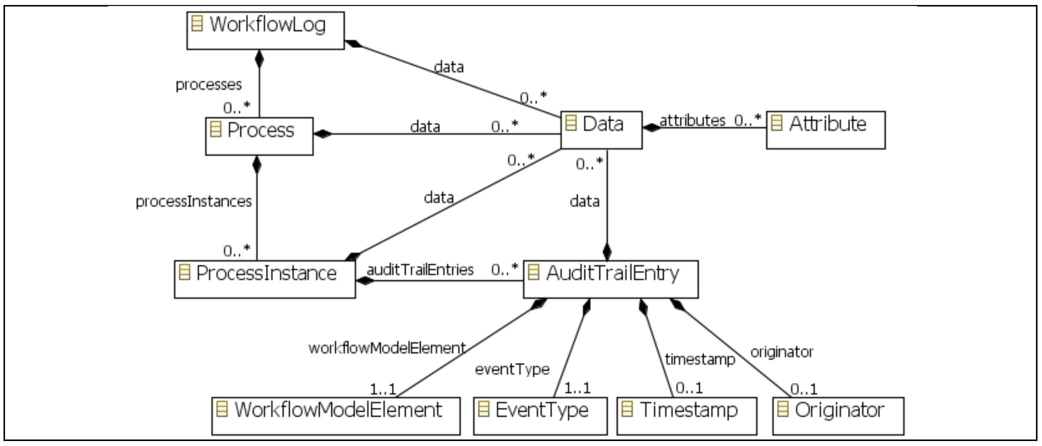


Figure 4. MXML Metamodel used to represent runtime models in MARBLE.

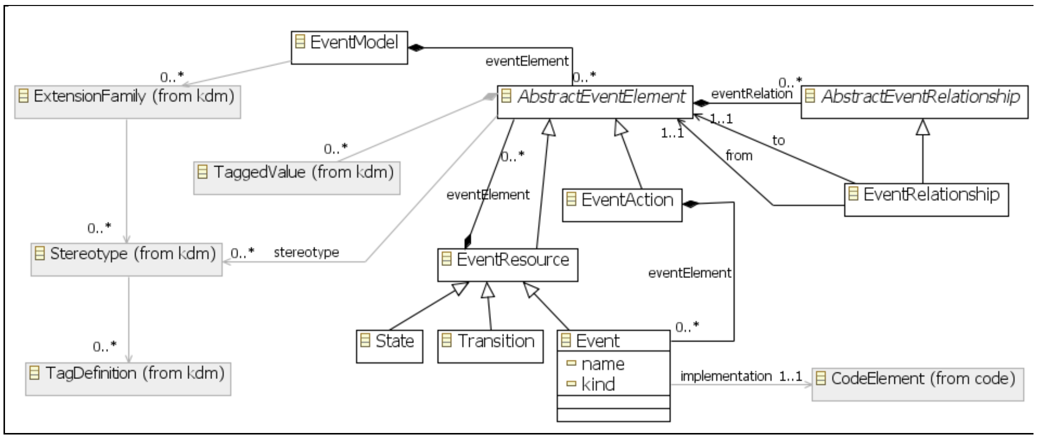


Figure 5. Event metamodel package in the KDM standard

Finally, when the *writeEvent* function has determined the correct process instance, it adds the event to that particular instance. The event, represented as an *AuditTrailEntry* element in the runtime model according to the MXML metamodel, is created using: (i) the name of the executed callable unit represents the *WorkflowModelElement*; (ii) the event type that is also a parameter of this function; (iii) the user of the system that executed the callable unit (or the user of the session if the system is a web application), which represents the *originator* element; and finally (iv) the system date and time when the callable unit was executed to represent the *timestamp* element.

4. Runtime Models in KDM

This paper also provides the model transformation between levels L1 and L2 of MARBLE, in order to represent the runtime models according to the KDM standard. As a consequence, this runtime model can be used in any software modernization context.

Specifically, the runtime models are represented using the *event* metamodel package of the *runtime resource layer* of the KDM metamodel [10]. Figure 5 shows the *event* metamodel package as well as other metaclasses of other KDM packages used in the runtime models. The *EventModel* metaclass represents the runtime model, which contains a set of *EventResource* and *EventAction* elements. An *EventResource* element can be specialized into a *State* element, a *Transition* element, an *Event* element, or it can even be a container of other *EventResources*. The *Event* element is used to model the *AuditTrailEntries* of the runtime model represented according to the MXML metamodel in L1. The feature *name* represents the *WorkflowModelElement*, and the feature *kind* represents (with a 'start' or 'complete' value) the EventType.

The transformation is formalized by means of QVT-Relations (the declarative language of the QVT standard). A relation transforms a MXML model in L1into an instance of the *EventModel* metaclass. This relation calls to the relation *'auditTrailEntry2Event'* that transforms each *AuditTrailEntry* in L1 into an *Event* in L2 (see Figure 6).

The *event* metamodel package of KDM makes it impossible to represent the process and process instance where the event belongs as well as the originator and timestamp of the event (see Figure 5). For this reason, the proposal uses the default extension mechanism of the KDM metamodel: the *extension families*. The *EventModel* includes an *ExtensionFamily* element (see Figure 6), which defines four *Stereotype* elements: *<process>*, *<processInstance>*, *<originator>* and *<timestamp>*. Each stereotype has a *TagDefinition* element that is used by stereotyped elements of the runtime model to put the specific value by means of a respective *TaggedValue* element.

Therefore, the problematic elements are represented in KDM as follows: (i) A process is represented as an *EventResource* element annotated with the <*process*> stereotype and containing a *TaggedValue* with the name of the process. (ii) A process instance is also represented as an *EventResource* element, which is nested within another *EventResource* that represents a process. This *EventResource* is annotated with the <*processInstance*> stereotype and contains a *TaggedValue* with the process instance identification. (iii) The originator is represented as a *TaggedValue* associated to an *Event* stereotyped as <*originator*>. (iv) The timestamp is also represented with a *TaggedValue* in an Event annotated with the <*timestamp*> stereotype.

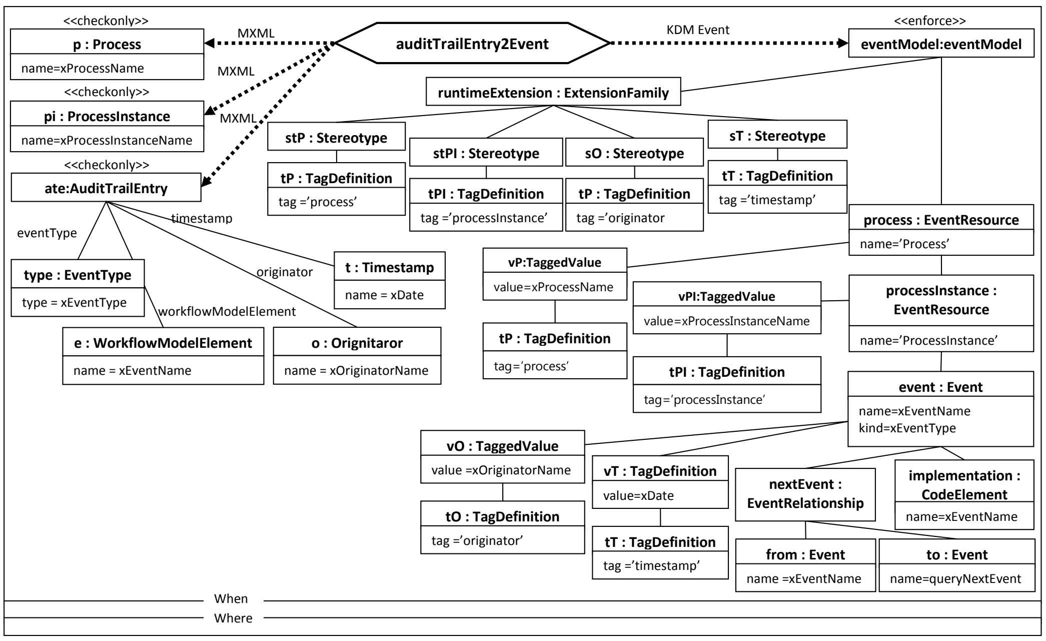


Figure 6. The QVT relation 'auditTrailEntry2Event' to transform runtime models in L1 to KDM models in L2.

Despite the fact that the order of the events can be derived from the timestamp information, the relation *'auditTrailEntry2Event'* (see Figure 6) establishes at level L2 the same sequence of events register in the model at level L1. This order is represented as an *EventRelationship* element within each *Event* representing a reference to the next *Event* element.

Finally, the relation *'auditTrailEntry2Event'* (see Figure 6) maps each *Event* to a *CodeElement* of the KDM code model. The *resource runtime layer* of the *event* package is above the *program element layer* (that contains the *code* and *action* metamodel packages), thus the elements of a runtime model can be mapped to the callable units represented in the KDM code model. As a consequence, the *feature location* is also improved throughout the modernization of a LIS.

5. Conclusions and Future Work

Software modernization projects typically take several software artifacts as source of knowledge into account like, for example, source code, databases, user interfaces. Thereby, modernization projects often recover knowledge from a static point of view. However, a dynamic approach allows modernization projects to extract more meaningful knowledge, which cannot be recovered analyzing artifacts in a static way only. For this reason, this paper proposes a technique, within a specific modernization framework (i.e., MARBLE), to obtain runtime models by means of dynamic analysis of source code.

Firstly, the proposed technique statically analyzes the legacy source code, and modifies it by injecting special sentences that make it possible to register execution events. Secondly, during system execution, the modified code is dynamically analyzed through the injected sentences and a runtime model is written at level L1of MARBLE. The obtained runtime models represent an event log according to the MXML metamodel, and depict a sequence of executed events related to business activities of the business processes embedded in the source code.

Moreover, the runtime model is transformed into a runtime model represented at level L2 of MARBLE according to the KDM metamodel. For this purpose, an extension of the *event* package of the KDM metamodel is proposed as well as a model transformations implemented by means of *QVT-Relation*. Due to the fact that the runtime model is represented following the KDM standard, the proposed technique can be used in other modernization frameworks based on ADM as MARBLE.

The future work will focus on the validation of the proposal by means of a case study involving a real-life LIS in a healthcare context. Another research direction in the future will be the use of the runtime models combined with other models of KDM as the *code* and *data* model in order to obtain more meaningful business process models at level L3 of MARBLE.

Acknowledgement

This work was supported by the FPU Spanish Program; by the R+D projects funded by JCCM: ALTAMIRA (PII2I09-0106-2463), INGENIO (PAC08-0154-9262) and PRALIN (PAC08-0121-1374); and the PEGASO/MAGO project (TIN2009-13718-C02-01) funded by MICINN and FEDER. In addition, this work was supported by the Quality Engineering group at the University of Innsbruck.

References

[1] Bianchi, A., D. Caivano, V. Marengo, and G. Visaggio, "Iterative Reengineering of Legacy Systems". IEEE Trans. Softw. Eng., 2003. 29(3): p. 225-241.

[2] Cai, Z., X. Yang, and W. Wang. "Business Process Recovery for System Maintenance - An Empirical Approach". in 25 th International Conference on Software Maintenance (ICSM'09). 2009. Edmonton, Canada: IEEE CS p. 399-402.

[3] Castellanos, M., K.A.d. Medeiros, J. Mendling, B. Weber, and A.J.M.M. Weitjers, Business Process Intelligence, in Handbook of Research on Business Process Modeling, J. J. Cardoso and W.M.P. van der Aalst, Editors. 2009, Idea Group Inc. p. 456-480.

[4] Clark, J. and S. DeRose, XML Path Language (XPath). 1999, World Wide Web Consortium (W3C).

[5] Di Francescomarino, C., A. Marchetto, and P. Tonella. "Reverse Engineering of Business Processes exposed as Web Applications". in

13th European Conference on Software Maintenance and Reengineering (CSMR'09). 2009. Fraunhofer IESE, Kaiserslautern, Germany: IEEE Computer Society p. 139-148.

[6] Dumas, M., W. van der Aalst, and A. Ter Hofstede, Process-aware information systems: bridging people and software through process technology. 2005: John Wiley & Sons, Inc.

[7] Ghose, A., G. Koliadis, and A. Chueng. "Process Discovery from Model and Text Artefacts". in IEEE Congress on Services (Services'07). 2007 p. 167-174.

[8] Günther, C.W. and W.M.P. van der Aalst, "A Generic Import Framework for Process Event Logs". Business Process Intelligence Workshop (BPI'06), 2007. LNCS 4103: p. 81-92.

[9] Heuvel, W.-J.v.d., Aligning Modern Business Processes and Legacy Systems: A Component-Based Perspective (Cooperative Information Systems). 2006: The MIT Press.

[10] ISO/IEC, ISO/IEC DIS 19506. Knowledge Discovery Meta-model (KDM), v1.1 (Architecture-Driven Modernization). http://www.iso.org/iso/catalogue_detail.htm?csnumber=32625. 2009, ISO/IEC. p. 302.

[11] Khusidman, V. and W. Ulrich, Architecture-Driven Modernization: Transforming the Enterprise. DRAFT V.5. http://www.omg.org/docs/admtf/07-12-01.pdf. 2007, OMG. p. 7.

[12] Miller, J. and J. Mukerji, MDA Guide Version 1.0.1. www.omg.org/docs/omg/03-06-01.pdf 2003: OMG.

[13] Moyer, B. (2009) Software Archeology. Modernizing Old Systems. Embedded Technology Journal, http://adm.omg.org/docs/Software_Archeology_4-Mar-2009.pdf

[14] Müller, H.A., J.H. Jahnke, D.B. Smith, M.-A. Storey, S.R. Tilley, and K. Wong. "Reverse engineering: a roadmap". in Proceedings of the Conference on The Future of Software Engineering. 2000. Limerick, Ireland: ACM.

[15] Newcomb, P. "Architecture-Driven Modernization (ADM)". in Proceedings of the 12th Working Conference on Reverse Engineering. 2005: IEEE Computer Society.

[16] OMG. ADM Task Force by OMG. 2007 9/06/2009 [cited 2008 15/06/2009]; Available from: http://www.omg.org/.

[17] OMG, QVT. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. 2008, OMG.

[18] OMG, Business Process Model and Notation (BPMN) 2.0. 2009, Object Management Group. p. 496.

[19] Paradauskas, B. and A. Laurikaitis, "Business Knowledge Extraction from Legacy Information Systems". Journal of Information Technology and Control, 2006. 35(3): p. 214-221.

[20] Pérez-Castillo, R., I. García-Rodríguez de Guzmán, O. Ávila-García, and M. Piattini. "MARBLE: A Modernization Approach for Recovering Business Processes from Legacy Systems". in International Workshop on Reverse Engineering Models from Software Artifacts (REM'09). 2009. Lille, France: Simula Research Laboratory Reports p. 17-20.

[21] Pérez-Castillo, R., I. García-Rodríguez de Guzmán, O. Ávila-García, and M. Piattini. "Business Process Patterns for Software Archeology". in 25th Annual ACM Symposium on Applied Computing (SAC'10). 2010. Sierre, Switzerland: ACM p. 165-166.

[22] Pérez-Castillo, R., I. García-Rodríguez de Guzmán, and M. Piattini. "Implementing Business Process Recovery Patterns through QVT Transformations". in International Conference on Model Transformation (ICMT'10). 2010. Málaga, Spain: Springer-Verlag p. In Press.

[23] Sneed, H.M., Estimating the Costs of a Reengineering Project. Proceedings of the 12th Working Conference on Reverse Engineering. 2005: IEEE Computer Society.

[24] van der Aalst, W., H. Reijers, and A. Weijters, "Business Process Mining: An Industrial Application.". Information Systems, 2007. 32(5): p. 713-732.

[25] Zou, Y. and M. Hung. "An Approach for Extracting Workflows from E-Commerce Applications". in Proceedings of the Fourteenth International Conference on Program Comprehension. 2006: IEEE Computer Society p. 127-136.

[26] Zou, Y., T.C. Lau, K. Kontogiannis, T. Tong, and R. McKegney. "Model-Driven Business Process Recovery". in Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004). 2004: IEEE Computer Society p. 224-233.