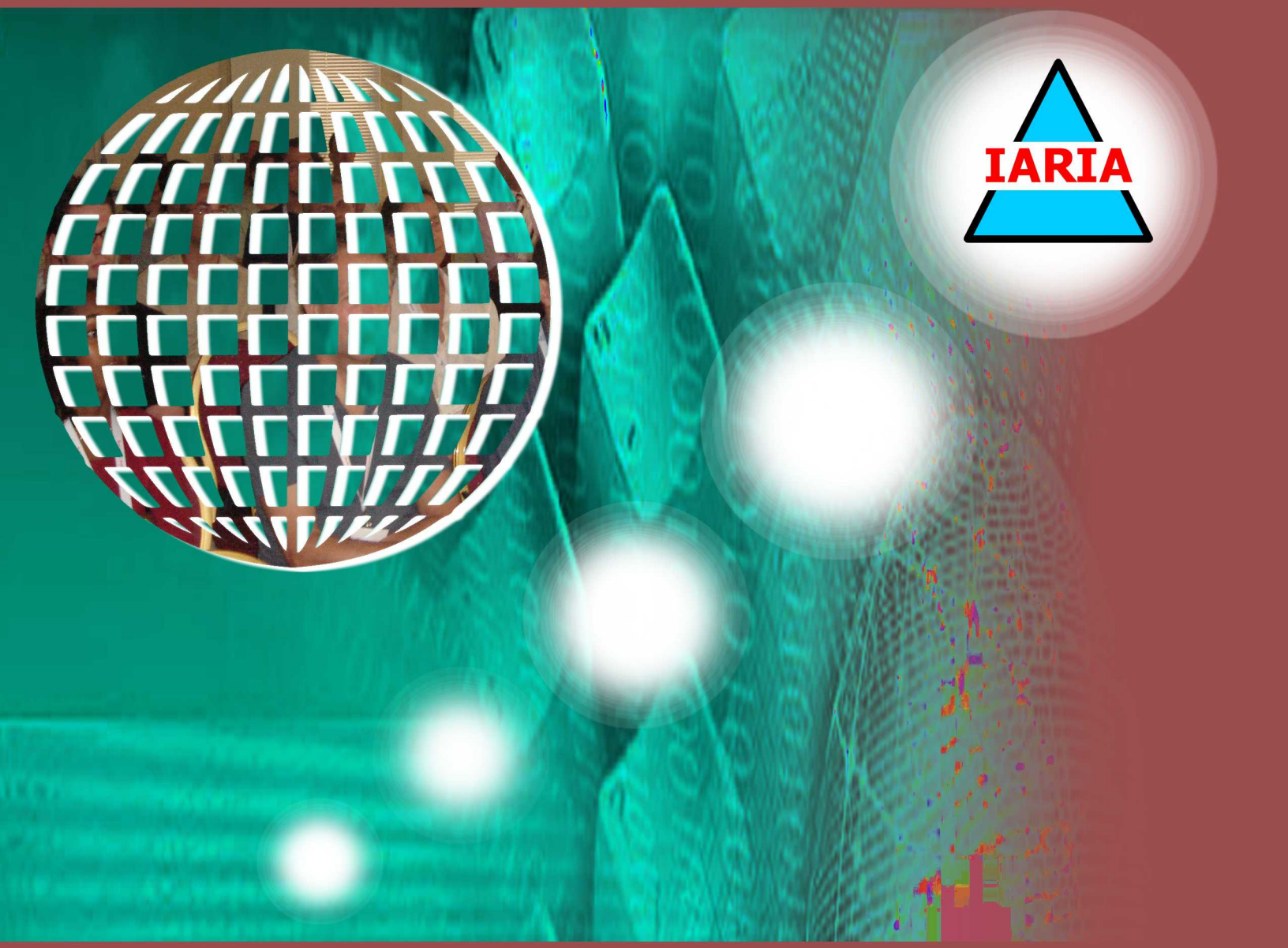
International Journal on Advances in Software



IARIA

2008 vol. 1 nr. 1

The *International Journal On Advances in Software* is Published by IARIA.
ISSN: 1942-2628
journals site: http://www.iariajournals.org
contact: petre@iaria.org

Sponsored by IARIA
www.iaria.org

International Journal On Advances in Software
Volume 1, Number 1, 2008

Editorial Board

First Issue Coordinators

Jaime Lloret, Universidad Politécnica de Valencia, Spain
Pascal Lorenz, Université de Haute Alsace, France
Petre Dini, Cisco Systems, Inc., USA / Concordia University, Canada

Software Engineering

- Marc Aiguier, Ecole Centrale Paris, France
- Sven Apel, University of Passau, Germany
- Kenneth Boness, University of Reading, UK
- Hongyu Pei Breivold, ABB Corporate Research, Sweden
- Georg Buchgeher, SCCH, Austria
- Dumitru Dan Burdescu, University of Craiova, Romania
- Angelo Gargantini, Universita di Bergamo, Italy
- Holger Giese, Hasso-Plattner-Institut-Potsdam, Germany
- Jon G. Hall, The Open University - Milton Keynes, UK
- Herman Hartmann, NXP Semiconductors- Eindhoven, The Netherlands
- Hermann Kaindl, TU-Wien, Austria
- Markus Kirchberg, Institute for Infocomm Research, A*STAR, Singapore
- Herwig Mannaert, University of Antwerp, Belgium
- Roy Oberhauser, Aalen University, Germany
- Flavio Oquendo, European University of Brittany - UBS/VALORIA, France
- Eric Pardede, La Trobe University, Australia
- Aljosa Pasic, ATOS Research/Spain, NESSI/Europe
- Robert J. Pooley, Heriot-Watt University, UK
- Osamu Takaki, Center for Service Research (CfSR)/National Institute of Advanced Industrial Science and Technology (AIST), Japan
- Michal Zemlicka, Charles University, Czech Republic

Advanced Information Processing Technologies

- Mirela Danubianu, "Stefan cel Mare" University of Suceava, Romania
- Michael Grottke, University of Erlangen-Nuremberg, Germany
- Josef Noll, UiO/UNIK, Sweden
- Olga Ormandjieva, Concordia University-Montreal, Canada
- Constantin Paleologu, University ‘Politehnica’ of Bucharest, Romania
- Liviu Panait, Google Inc., USA

- Kenji Saito, Keio University, Japan
- Ashok Sharma, Satyam Computer Services Ltd – Hyderabad, India
- Marcin Solarski, IBM-Software Labs, Germany

Advanced Computing

- Matthieu Geist, Supelec / ArcelorMittal, France
- Jameleddine Hassine, Cisco Systems, Inc., Canada
- Sascha Opletal, Universitat Stuttgart, Germany
- Flavio Oquendo, European University of Brittany - UBS/VALORIA, France
- Meikel Poess, Oracle, USA
- Kurt Rohloff, BBN Technologies, USA
- Said Tazi, LAAS-CNRS, Universite de Toulouse / Universite Toulouse1, France
- Simon Tsang, Telcordia Technologies, Inc. - Piscataway, USA

Geographic Information Systems

- Christophe Claramunt, Naval Academy Research Institute, France
- Dumitru Roman, Semantic Technology Institute Innsbruck, Austria
- Emmanuel Stefanakis, Harokopio University, Greece

Databases and Data

- Peter Baumann, Jacobs University Bremen / Rasdaman GmbH Bremen, Germany
- Qiming Chen, HP Labs – Palo Alto, USA
- Ela Hunt, University of Strathclyde - Glasgow, UK
- Claudia Roncancio INPG / ENSIMAG - Grenoble, France

Intensive Applications

- Fernando Boronat, Integrated Management Coastal Research Institute, Spain
- Chih-Cheng Hung, Southern Polytechnic State University, USA
- Jianhua Ma, Hosei University, Japan
- Milena Radenkovic, University of Nottingham, UK
- DJamel H. Sadok, Universidade Federal de Pernambuco, Brazil
- Marius Slavescu, IBM Toronto Lab, Canada
- Cristian Ungureanu, NEC Labs America - Princeton, USA

Testing and Validation

- Michael Browne, IBM, USA
- Cecilia Metra, DEIS-ARCES-University of Bologna, Italy
- Krzysztof Rogoz, Motorola, USA
- Sergio Soares, Federal University of Pernambuco, Brazil
- Alin Stefanescu, SAP Research, Germany
- Massimo Tivoli, Universita degli Studi dell'Aquila, Italy

Simulations

- Robert de Souza, The Logistics Institute - Asia Pacific, Singapore
- Ann Dunkin, Hewlett-Packard, USA
- Tejas R. Gandhi, Virtua Health-Marlton, USA
- Lars Moench, University of Hagen, Germany
- Michael J. North, Argonne National Laboratory, USA
- Michal Pioro, Warsaw University of Technology, Poland and Lund University, Sweden
- Edward Williams, PMC-Dearborn, USA

International Journal On Advances in Software
Volume 1, Number 1, 2008

Foreword

Finally, we did it! It was a long exercise to have this inaugural number of the journal featuring extended versions of selected papers from the IARIA conferences.

With this 2008, Vol. 1 No.1, we open a long series of hopefully interesting and useful articles on advanced topics covering both industrial tendencies and academic trends. The publication is by-invitation-only and implies a second round of reviews, following the first round of reviews during the paper selection for the conferences.

Starting with 2009, quarterly issues are scheduled, so the outstanding papers presented in IARIA conferences can be enhanced and presented to a large scientific community. Their content is freely distributed from the www.iariajournals.org and will be indefinitely hosted and accessible to everybody from anywhere, with no password, membership, or other restrictive access.

We are grateful to the members of the Editorial Board that will take full responsibility starting with the 2009, Vol 2, No1. We thank all volunteers that contributed to review and validate the contributions for the very first issue, while the Board was getting born. Starting with 2009 issues, the Editor-in Chief will take this editorial role and handle through the Editorial Board the process of publishing the best selected papers.

Some issues may cover specific areas across many IARIA conferences or dedicated to a particular conference. The target is to offer a chance that an extended version of outstanding papers to be published in the journal. Additional efforts are assumed from the authors, as invitation doesn't necessarily imply immediate acceptance.

This particular issue covers papers invited from those presented in 2007 and early 2008 conferences. The papers cover mechanisms, techniques and applications using agile technology, modular design, process-aware and workflow diagrams for life cycles. Particular experiments are reported on semantic data processing and security-critical applications.

We hope in a successful launching and expect your contributions via our events.

First Issue Coordinators,
Jaime Lloret, Universidad Politécnica de Valencia, Spain
Pascal Lorenz, Université de Haute Alsace, France
Petre Dini, Cisco Systems, Inc., USA / Concordia University, Canada

International Journal On Advances in Software
Volume 1, Number 1, 2008

CONTENTS

Goal sketching: An Agile Approach to Clarifying Requirements

Kenneth Boness
University of Reading, Berkshire, RG6 6AY UK
k.d.boness@reading.ac.uk

Rachel Harrison
Stratton Edge Consulting Ltd, GL7 2LS, UK
rachel.harrison@strattonedge.com

Kecheng Liu
University of Reading, Berkshire, RG6 6AY UK
k.liu@reading.ac.uk

Abstract

This paper describes a technique that can be used as part of a simple and practical agile method for requirements engineering. It is based on disciplined goal-responsibility modelling but eschews formality in favour of a set of practicality objectives. The technique can be used together with Agile Programming to develop software in internet time. We illustrate the technique and introduce lazy refinement, responsibility composition and context sketching. Goal sketching has been used in a number of real-world development projects, one of which is described here.

Keywords: goal-oriented requirements engineering, agile development, evolving systems.

1. Introduction

Our motivation for goal sketching is to help stakeholders who need to make project critical decisions in projects which develop emergent systems. The agility here concerns the manner of obtaining and maintaining the rationale of problem and solution requirements so as to be able guide projects. Hence goal sketching applies to, but is not limited to, software projects using agile development methodologies.

Decisions about investment and requirements priorities are the responsibility of stakeholders and can only be made rationally when supported by a coherent depiction of what is known about the requirements. It is well known that this is problematical: for example the importance of "creating realistic expectations in the minds of stakeholders" has been noted [1] and the observation that "..customers on agile projects are often asked to make critical, project-defining decisions, and very little of the methodology can help them make those calls." [2].

In contrast we suggest that (at least in principle) given enough time, information and skill, goal-responsibility refinement models can be constructed to represent the stakeholders' expectations for a system-to-be that will operate in an expected environment, in fulfilment of a contract. Such models can be produced using KAOS [3] and some use-case methodologies [4,5]. Each has a structured argument framework that allows the rationale to be verified and thus affords the possibility of formulating systematic evaluation of the adequacy and feasibility of the intended system. However the prerequisite criteria (time, information and skill) are not satisfied in the situations with which we are concerned. Hence our research question which we are investigating with an action research methodology is: can a lightweight adaptation of KAOS style goal-responsibility modelling meet the practical demands of the analysts and designers?

Of paramount importance is the clarity of the disciplined structure of goal-responsibility argumentation (with its quasi hierarchical depiction) as a possible basis for capturing what is known about the requirements and the agreed rationale for their satisfaction. Our methodology has 4 objectives:-

Table 1. Objectives of goal sketching

1.	To maintain a coherent depiction of the intention (the agreed-upon requirements and the rationale for their satisfaction) as it unfolds over time.
2.	To be simple enough to allow a project manager or analyst to achieve a first draft, at a resolution good enough to steer high level priority decisions, at the outset of the project.
3.	To keep the depiction understandable to business as well as technical stakeholders.
4.	To support formal rigour on a "just enough" and "when needed" basis [6].

The methodology we are developing is called goal sketching [7,8]. It is also the foundation for our work on appraising development projects [9] called goal-

responsibility appraisal of soft projects (GRASP). It embraces established practices evolved to cope with uncertainty such as spiral [10] and breadth before depth [11] techniques. Similarly 'just enough' approaches such as in [12] inform our approach to time-constrained development.

This paper proceeds as follows. In section 2 we introduce the concept of structurally complete goal-responsibility (G-R) models and their adaptation to our purpose. In section 3 we present the current state of our goal sketching methodology. Section 4 reinforces the description using a hypothetical exemplar and section 5 uses an industrial application to illustrate our efficacy in regard to the above four objectives.

2. Goal-Responsibility Models

An example of what we mean by goal-responsibility model is shown in Figure 1. Models like this are used in goal oriented requirements engineering (GORE) such as the KAOS and also (with provision for the representation of responsibilities [8]) in some use case techniques.

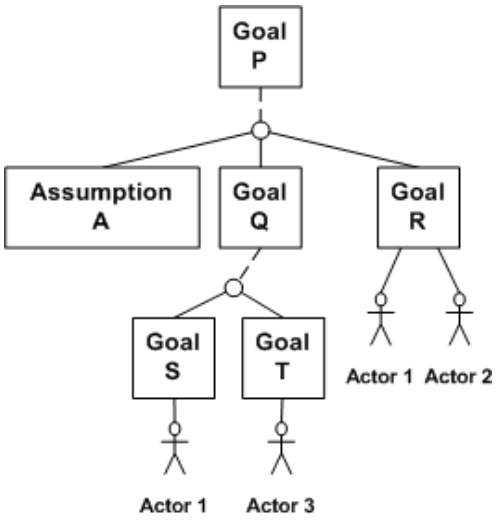


Figure 1. Goal-Responsibility model

Each box in Figure 1 is referred to as a 'goal oriented proposition' (GOP). In keeping with propositional logic each GOP must be defined in such a way that it may be refuted. The figure uses two types of proposition: assumption and goal. There is more to say about types but for now it is enough to note that a G-R graph can record explicit assumptions as well as goals. The aim when constructing a G-R graph is to capture the logic of the problem in hand moving by step-wise refinement from relatively abstract root propositions (e.g. goal P) to relatively concrete propositions that may be operationalized (e.g. goals S,T and R) or assumptions (which can only be trusted but not operationalized). Although the structure is hierarchical the analysis to create it is rarely top down; an analyst typically works with GOPs at all levels of abstraction. The aim (and skill) of the analyst (in goal sketching at least) is to organise the GOPs into a convincing rationale. In doing this it is usual that the analyst may discover gaps in the argument and then invents additional GOPs in order to complete it.

Each step of refinement is a satisficing argument where a proposition is refined into sub propositions such that the sub-propositions can be agreed to be collectively sufficient and individually pertinent to adequately satisfy the parent. We call this the *refinement argument criterion.*

Each argument step is deemed valid if by some defensible criteria (e.g. expert judgement and/or 'policy' such as in goal structuring notation (GSN) [13] or 'root definition' as in soft systems methodology (SSM) [18]).

The model is said to be *structurally complete* if (as in the figure) all objectives are ultimately satisfied by actors of the system-to-be. Thus P is satisfied by the combined actions and qualities of Actors 1, 2 and 3. It is important to note that in this type of representation the necessary behavior (and other qualities) that must be instantiated is described only at the leaves of the model; it is not distributed across the model. So if Q harbours required behavior to be explicitly represented in S and T then a further GOP should be added along with S and T [8].

When a G-R model is constructed in a formal logic (such as KAOS) there are calculi to verify the argumentation. Hence if the model is also structurally complete and all necessary root GOPs are included the model should amount to an adequate intention for the stakeholders. Further, if the responsibilities are individually and collectively operationalizable within the constraints of the project the intention can be said to be feasible.

This potential for systematically evaluating the adequacy and feasibility of an informally produced G-R model is a key intended benefit of our goal sketching technique; especially since for our purposes (with our assumption of a incomplete information) structural completeness is only possible if the analyst places assumptions and, or very low precision GOPs into the rationale. A G-R model constructed this way, out of necessity, is a rich resource to draw on to promote informed negotiation among the stakeholders.

3. Goal Sketching Technique

In this section we present the details of the technique so far developed through our programme of action research. We outline a methodology for using the technique and then proceed to particular details concerning the support of building refinement arguments.

3.1. Using the Technique

Our goal sketching technique starts with the creation of a goal graph which expresses the high level motivations behind the intention to develop the software. This is typically a coarse but structurally complete sketch of what is understood about the overall intention. In general there is often a vague long-term vision coupled with some short-term clarity. A series of staged developments are planned using the system graph as a guide. This compliments the practice of sprinting in Scrum [14], and the increments in an iterative and incremental development process [15]. Each stage is preceded by taking a portion of the system graph in its current state and refining it so that there are no remaining vague intentions. This is called the 'stage graph'. In the execution of any stage it is possible that the stage graph will be updated as a result of the usual agile practice of improving the quality of the work in hand. At the completion of each such stage its graph is used to update the system graph. Thus the true goal graph emerges by successive iterations and refactoring and so becomes the inventory, recording the associated rationale for posterity.

When preparing each stage the goals are refined only as far as necessary for the stage in hand (a technique called *lazy refinement)* relying on stories, use cases or activity sketches. (This does not preclude the use of formality as problem frames [16] or temporal logic etc may be used when necessary.)

We advocate using *pair sketching*, in which the goal graphs are sketched by two people working together (often the analyst with a stakeholder) to ensure that the refinement argument is sound, in a manner akin to pair programming. Once an acceptable goal graph has been produced it is incorporated into the system goal graph. The system graph may need to be re-factored for the next stage.

The goal graphs are exported to a database for subsequent analysis. From the database we can produce matrices to expose *composition* issues which may arise from cross-cutting concerns for analysts, designers, developers and testers.

3.2. Refinement Argument Supports

In our goal sketching the GOPs are written in natural language and must satisfy the *refinement argument criterion*. This is a very simple principle but in practice it can be very difficult to do. Errors that we have observed in students and would-be industrial practitioners, and ourselves, include:-

1. Mixing two or more problem contexts (e.g. mixing operation with construction of the system-to-be) in a confusing argument.
2. Expressing 'milestone' refinement patterns [3] as multi-level rather than single level refinements. This leads to an invalid though seemingly structurally complete G-R model.
3. Volatile functional refinement arguments that depend upon the current outlook of the analyst.

As mentioned above, *pair sketching helps* but we have found that it is very important to to be mindful of four aspects of a GOP, which we list Table 2 as support to the practitioner.

Table 2. Aspects of GOPs

1.	The type of proposition: e.g. assumption or objective.
2.	The proposition owner: e.g. a stakeholder role or a system.
3.	The problem context of the proposition in terms of where operationalization can be enacted; e.g. in the domain of the operating system-to-be or the domain of the development of the system-to-be.
4.	The refinement level: i.e. to try to keep all sub-propositions of a proposition at similar levels of abstraction.

These supports are are discussed in the following sub-sections.

Type. A goal oriented proposition is a refutable statement written in natural language which as shown in Figure 2 we specialize into five types.

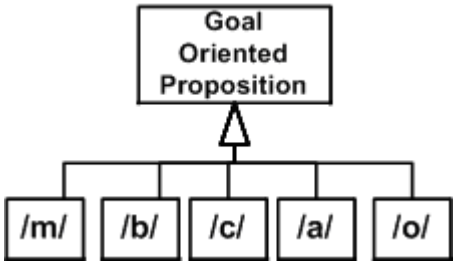


Figure 2. Goal types

/m/ is a motivation goal representing the agreed-upon concerns of the stakeholders that motivate the project; in terms of KAOS they are likely to be "soft goals". They may harbour refinement implications that require refinements in different problem contexts with different time-spans (e.g. project time or system run-time). Even when they apply to a single context they can only be satisficed [17] and their refinement should include an assumption giving the justifying world-view (similar to weltanschauung [18]). For example the refinement of 'achieve greater efficiency' might include goals such as 'provide data at point of need' but it would depend upon an assumption linking the provision of data to greater efficiency.

/b/ is a behavior goal explicitly required by the stakeholders or by force of circumstance and necessary for completeness. It 'affords' [19,5] an option or freedom to a user, whether or not the user chooses to exploit it. It combinines the capability and condition elements of a 'well formed requirement' in the IEEE recommendations on systems requirements [20].

/c/ is a constraint: a nonfunctional requirement that limits the possible system implementation solutions. It is a factor that is imposed on the solution by force or compulsion and may limit or modify the design changes. This is consistent with [20].

/a/ is an assumption: something that is stated on trust but is necessarily true for the rationale to present a defensible argument; i.e., it is 'load bearing' [21]. An assumption may be a simplifying argument used to find an acceptably easy goal refinement argument or it may be a KAOS domain property [3].

/o/ is an obstacle used as in KAOS to oppose the satisfactory fulfilment of any other proposition. These propositions are not discussed further in the following models as they remain substantially as used in KAOS.

The /b/ and /c/ propositions are strictly bound to system run-time whereas the /a/ propositions may be either project or run time.

Problem Domains and Context. Jackson ties requirements statements to domains in a rigorous fashion [16] as illustrated in Figure 3 where the requirement is understood as referring to phenomena in the domain.

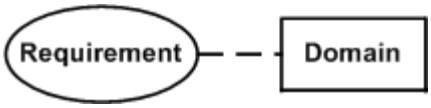


Figure 3. Requirements context

A requirement straddling multiple domains is shown by a dashed line to each [16] and the associated phenomena are referenced exclusively in each domain.

In goal sketching we advocate tying GOPs to their relevant domains in a similar fashion. In such diagrams, see Figure 4, we use { } to show that we mean a GOP.

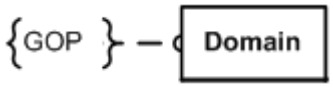


Figure 4. GOP context

The domain may be a large domain such as a business operation. Inside the business operation there might be sub-domains to which we attach lower level GOPs. It is in the nature of /m/ propositions that they may imply references to phenomena in the domain with different enactment contexts. We identify three such contexts in Table 3.

Table 3. Three common contexts

1. The system-to-be.
2. A system to manage the life-cycle of the system-to-be.
3. The project to manage business change and the construction of the 'kit' [22].

In goal sketching we choose goal refinement steps that lead rapidly to referencing phenomena of single contexts. This technique helps to distinguish constituent domains that would obfuscate the G-R graph.

Starting with an agreed outer problem domain the *problem context* is established by attaching the root GOPs and any global constraints and assumptions (as a set of /m/, /a/ and /c/). This can be refined by then exposing the important inner domains and then attaching agreed GOPs (this time possibly including /b/ types). This technique echoes the work in [23] where problem frames are used to guide a goal refinement using business process modelling but is more lightweight.

We find that just as the goals require *sketching* (especially early-on) it is usual to sketch the domains; again adding precision on a just good enough basis.

Owner. In colloquial use a goal would be owned by a person (or group of people). For example: *Owner* "To make a profit."; or *user* "To reserve a book". This kind of ownership dominates use-case based GORE [4,5]. In branches of system engineering it may also be said that a goal is owned by the system (or indeed a machine) in the sense that it is an embedded objective. For example:the goal of a heat seeking missile is to find its target. This is the usual kind of ownership in KAOS. Generally in goal-sketching ownership passes from people to system as operationalization is approached.

Structurally Complete Refinement. In goal sketching the aim is to capture the logic moving from /m/ to operationizable propositions which will be a collection of /b/ and /c/ propositions and *en passent* it may be necessary to add assumptions /a/. We apply the following rules to guide the construction of a structurally complete refinement such as illustrated in Figure 1.

Table 4. Goal sketching rules for a structurally complete G-R model

1.	The roots of a goal graph must be /m/ propositions.
2.	The leaves of a goal graph can only be /b/ and/or /c/ and/or /a/ propositions.
3.	Every leaf of type /b/ or /c/ must have a complete set of responsible actors (see below) assigned.
4.	Any /b/ or /c/ proposition may be refined into combinations of sub-propositions of types /b/ and/or /c/ and/or /a/ in the same temporal and contextual mode
5.	Any /a/ may only refine into /a/ sub-propositions.
6.	Any /m/ may refine into i. /m/ sub-propositions, or ii. /b/ and/or /c/ and/or /a/ sub-propositions. In case (ii) there must be at least one /a/ that expresses the binding/justifying world view.
7.	Refinement arguments must satisfy the refinement argument criterion.

Note that as information improves it may become necessary to convert one type of proposition into another and then reconsider the refinement arguments and reapply the rules. This is typically the refactoring mentioned above in terms of stage and system graphs.

3.4. Lazy Refinement

Refinement should always halt when just enough detail is exposed to allow safe operationalization. Hence in goal sketching the degree of refinement applied is kept to a minimum. Often, especially early in a project, it must be halted owing to a lack of information. In terms of sprint based agile development there is an implied set of such goals pending exploration at a suitable time in the future. But it is important to capture such lack of information in a context that is informative to the sponsors and other stakeholders. In goal sketching this is left as a refinement TBD (to be determined) and is explicitly recorded on the graph.

In the interests of efficiency refinement can be halted at a relatively abstract level where the implications of operationalization are well known; i.e. they are normal [24] to the community (the key stakeholders). On the other hand where they are not understood (perhaps radical [24] to the community) a more rigorous refinement may be called for; this can be provided as problem frames and,or fully dressed use case analysis [4] and,or the usual methods of KAOS.

3.3. Operationalization

In Figure 1 the actors (aka agents in KAOS) are entities of the system-to-be that can take responsibility for the necessary enactments of the leaf goals. For example: Actor 1 is responsible to enact, effect or be whatever goal proposition S requires. In this case no other actor is involved. In the case of goal proposition R it requires two actors in collaboration. The nature of the collaboration will be interpreted from the specification of R.

For lazy refinements the specification may be informal such as: a simple statement, a software engineering template specification [25], an eXtreme style story or a use case. It is typical in lazy refinement to have multiple actors collaborating.

In full refinement, as in KAOS, the objective is to have a unitary relationship between a requirement or expectation (equivalent to goal propositions) and an agent (actor in goal sketching). Alternative methods of achieving and specifying full refinement, which we prefer include Jackson's Problem Frames [16], activity diagrams [8] and use-cases. These are also illustrated in the example below.

3.5. Composition

When creating clear refinement arguments goal sketching favors a strict policy of separation of concerns. This implies decomposition and thus necessitates a late re-composition [16] as cross-cutting concerns (e.g. collaboration between responsible agents to indicate necessary superimposition of capabilities, constraints and conditions). In our experience this approach minimizes the number of goals with multiple parents and thus reduces visual tangling in the goal graph. The price for this benefit is that the composition concerns are not explicit. However a lightweight solution is to annotate the assigned responsibilities using a system of *composition tags* (see Figure 5). In contrast KAOS uses object and operation models to accommodate composition concerns. This can be rigorous but tends to be heavyweight.

Figure 5. Responsibility annotation

Figure 5 shows three versions of the responsibility assignments. Each is shown as an oval with the name of an assigned agent followed by a full stop. The architectural precision of the agent depends upon the underlying domain analysis being used; e.g. an object in a UML model or a sub-domain of a Jackson context diagram [16]. An optional system of semantic tagging is allowed after the full stop. Each tag is written in the form "<MYTAG>" or <@MYTAG>. Any responsibility with a given tag (say <MYTAG>) is a target for composition with a similar named tag including the "@". Thus a responsibility marked <@MYTAG> composes with all responsibilities tagged with <MYTAG>; i.e. the goal associated with the '@' symbol is added to or changes the goal associated with the other responsibilities. This feature allows strict separation of cross-cutting concerns and subsequent re-composition. The semantic tags are created and managed by the analyst either manually or with tool support.

Any conflicts that emerge through this composition will need to be resolved by design or by negotiation.

3.6. Accelerating Functional Goal Sketching

In [7] and [8] we mention problems that people may experience with functional goal refinement: for example the tendency to interpret 'how' as project flow and elaboration that is unjustified in the circumstances. In [8] we introduce the idea of dual use of goal graphs and activity diagrams. The former give coherence and the latter facilitate refinement of functionality.

The approach depends on the idea that an activity diagram has a goal that is satisfied by its activities plus a special goal to guarantee its logic (guards, flow etc). Thus an activity diagram such as Figure 6can be said to have an objective GO and will be a goal proposition of type /b/. Similarly the objectives of the activities A1, A2 and A3 are G1, G2 and G3. This gives the corresponding goal graph shown in Figure 7.

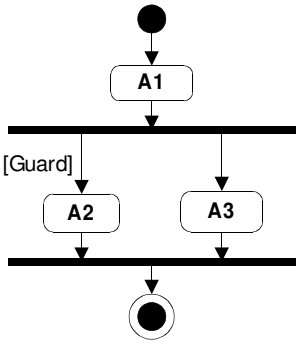


Figure 6. An activity diagram representing a /b/

It is important to note the goal in Figure 7 'Impose Process A' as a /b/ type proposition. Its purpose is to represent the need to guarantee the flow of the activity diagram as a leaf goal in the structure. If the activity diagram is informally drawn then the logic to be guaranteed in 'Impose Process A' can be correspondingly informal (the use of such informal sketches is an area we are currently investigating).

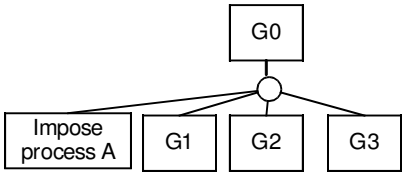


Figure 7. Goal graph corresponding to Figure 6

If any of the activities A1 to A3 in Figure 8 have sub-activities these are appended to their goals in Figure 6. In this way nesting of activities is a dual of goal refinement. This approach has been used in one of our industrial examples.

4. Example

To illustrate the technique we will use an example involving the calculation of body mass index.

> The *customer, WeighCom, wishes to develop new walk-on scales that can be installed in public places and used by any passers-by to measure their weight, height and body mass index (BMI) and receive a business card sized printed record on the spot. Normal operation is for the user to step onto a pressure mat facing an instruction screen and stand under an acoustic ranger. The measurements are made once the user pays a fee of 1 Euro into a receptor.*
>
> *WeighCom specifies that the solution must use certain components: pressure mat (PM); coin receptor (CR); acoustic ranger (AR) and integrated processor with alpha numerical visual display and user selection touch screen (IP). All of these are to be controlled through software using an API. These components support an existing assembly in which the whole is weather proof and vandal proof.*
>
> *WeighCom currently installs personal weighing equipment in public places for coin operated use by the public. They have an excellent reputation, which is of paramount importance to them, for always providing a reliable service or repaying. They have a call centre which customers can call if their installations appear to be malfunctioning.*

Figure 8. Problem statement

Scrutiny of the problem statement suggests the following primary concerns:-

- Operation in public places.
- Normal operation (i.e. accepting payment through to printing a card)
- Use of prescribed components.
- WeighCom's reputation.

From the problem statement we can also reasonably place these in context as shown in Figure 9 where we can see that there are likely to be concerns associated with the call centre. Further we might speculate that there is a maintenance problem domain for which we have no expressed concerns. Table 3 shows that we might associate the use of prescribed components to an additional problem domain concerned with the project but since there are no other concerns stated here we will ignore the project problem domain. We also have no express concerns about an installation problem domain; which would probably affect a maintenance domain. What matters is that we can agree with the stakeholders that Figure 9, with the attached assumptions, represents the problem under discussion.

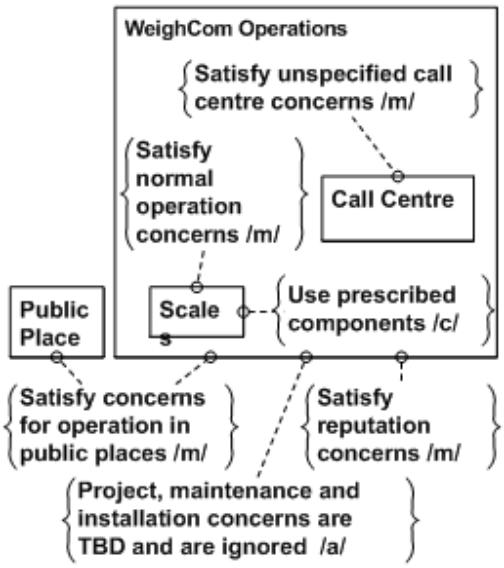


Figure 9. Context of WeighCom goals

The corresponding G-R modelling is shown in the first level refinement in Figure 10 where all the GOPs are owned by stakeholders.

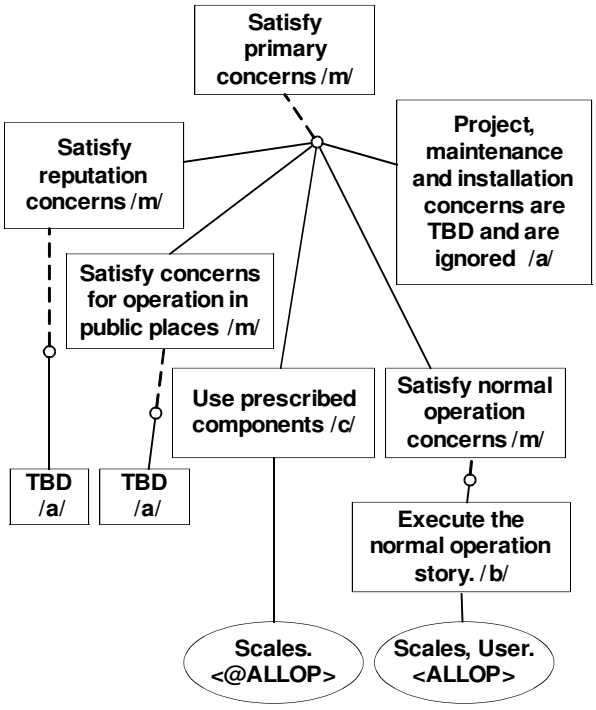


Figure 10. Structurally complete G-R model

If and when the assumption 'project maintenance..' is reversed a separate refinement argument would be created for other problem domains (project, maintenance etc.) and these are likely to crosscut as constraints on the responsibilities in Figure 10 .

Figure 10 has been made structurally complete by adding a lazy refinement ('execute..') and assumptions declared TBD. In this case the transition between the / m/ and /b/ goals has not needed a weltanschauung assumption as the /m/ goal itself applies to the scales domain in which all activity in the /b/ goal takes place. Note also that the responsibility to use prescribed components rests on the actors of the scales. The tag ALLOP was created, and catalogued, to refer to all normal operation behavior.

The refinement of the 'execute...' is an example of functional refinement. There are two potential problems (see Table 2) when creating a stable refinement argument at consistent levels of abstraction and getting business stakeholders to review the argument. This is a good opportunity to use the activity diagram technique. A plausible analysis is shown in Figure 11.

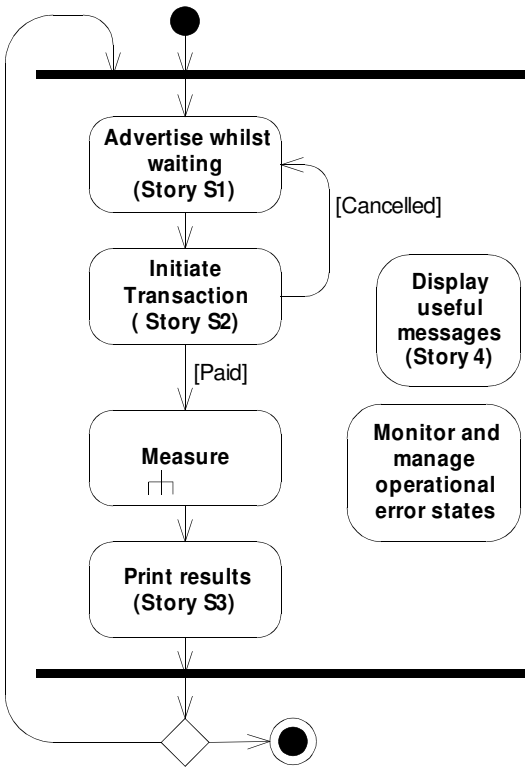


Figure 11. Normal operation story as an activity diagram sketch

Figure 11 itself is a sketch as there is no pretence at full rigour. However it is suitable for discussion with stakeholders to reveal the required activity. Thus the activity diagram allows the stakeholders to design a solution rationale. Each activity must be supported either by a specification (here stories have been used) or a further level of refinement (e.g. as in 'measure'). Our experience is that this approach is easier for stakeholders to comprehend than looking at mixed 'case' and 'milestone' KAOS refinements. The two floating activities are read as occurring concurrently with all other activities between the fork/join lines.

Figure 12 Shows all the activities in Figure 11 plus an 'impose..' goal as the refinement of 'execute..' in Figure 10.

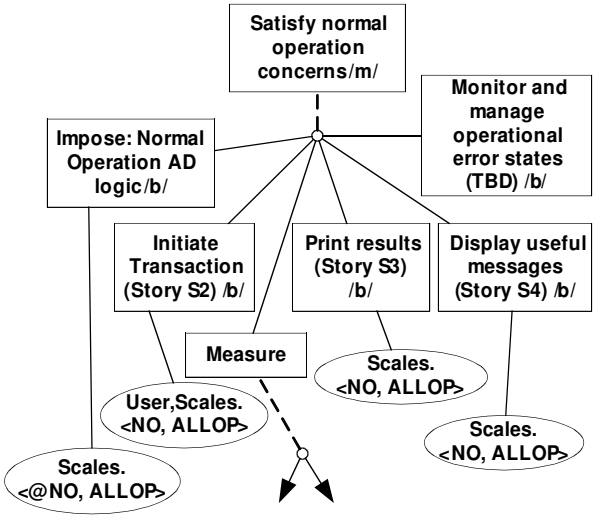


Figure 12. Goal sub-refinement for Figure 11

The 'impose process..' goal in Figure 12 emphasizes the need for the glue logic and can be developed to an appropriate level of precision (on a scale from leaving it to the developer's intuition, to detailed narrative, up to fully developed UML or formal logic). The 'measure' goal is further refined (not shown in detail here). The figure will be structurally complete provided that the refinement of 'measure' is actually complete and that the 'monitor and manage.' goal is replaced by an assumption that it is not to be implemented in the current stage. All these matters being negotiated and prioritized as apart of stakeholder negotiation for a stage of the development.

An example simple story is provided in Figure 13. The level of precision shown would be enough for many developments. If more precision becomes necessary then the story may be replaced with one in more detail, a use-case, a problem frame or by a full KAOS refinement.

"When a customer pays €1 into the CR they may either confirm the payment or cancel the payment. If they cancel then the CR refunds the payment. If they confirm then the service is initiated."

Figure 13. Transaction initiation (Story S2)

This simple example has allowed a demonstration of the techniques. In the next section concerning real projects we can observe how well we meet the objectives set down in Table 1.

5. Industrial Projects

We have improved our method using a number of industrial applications. These include products supported by venture capital, a management information system (MIS) for a food processing company, a university infrastructure project and support for services in healthcare. We start here with some general observations and then look at some details of a healthcare project and a venture capital project. The first is chosen because it seems representative of the general method and the second because it makes a slightly different use of our method and shows a situation that often arises in agile backlog driven projects. Most of the projects have been mentioned in [7,8,9].

The staffing profiles for these projects involved managers, executives, developers and testers; all with very different perspectives and analytical abilities. In all cases the managers and executives were not involved with detailed requirements analysis, whereas the developers and testers were.

The analyst (one of the authors) worked with key staff members (project and/or product managers). From the beginning it was clear that our industrial colleagues were not familiar with goal based requirements methods. In order to reach out to the executive and other non-technical stakeholders, whose participation was essential, we developed an approach which used the familiar sales terminology: 'pain' (things that are presently unsatisfactory in the problem domain) and 'gain' (new opportunities to improve the problem domain) [26]. To this we added 'maintain' (things that should not change as a result of dealing with the pain and gain concerns).

Thus armed, our first step involved analyzing the problem domain and the stakeholders' concerns (i.e. the root GOPs). Inevitably lower level concerns (design fragments, particular functions etc) arose but they were put aside until the root problem was agreed. One of our projects was a retrospective study and it is clear that the project lacked shared understanding.

We find that the cost of reaching an agreed problem domain and root GOPs is only a few staff days unless there are conflicts that need to be resolved. The smaller (health-care) projects took only a few hours to reach this point. Importantly what they all established firmly were the 'load bearing' assumptions.

Figure 14 illustrates the root problem for one of the healthcare projects. This project was motivated by a benefactor organization wishing to sponsor a tool to be supplied to assist the care of patients with a particular disease. A group of physicians (the Forum) were to be the initial beneficiaries. They would be called upon to help specify an initial product, limited by budget, and would use the product as a support to their normal consultations and supplementing their usual medical system (MedSys).

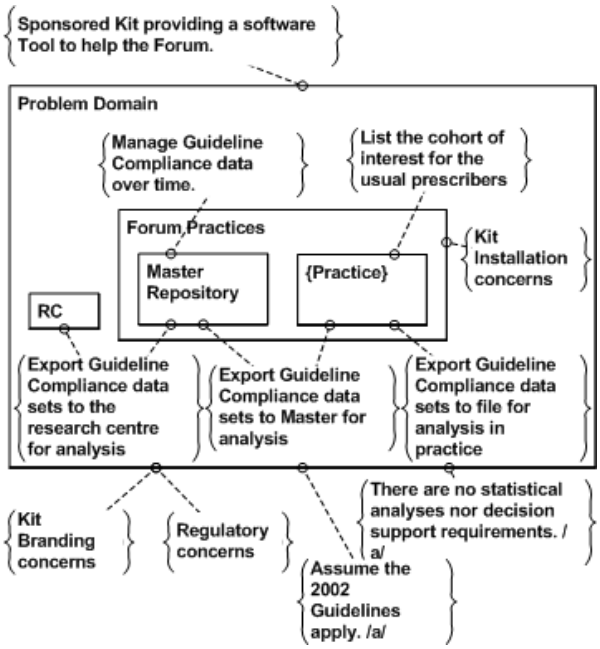


Figure 14. The agreed problem statement

The figure shows a key simplifying assumption arrived at after negotiation and constraints arising from data confidentiality and security protocols and from the wishes of the benefactor and Forum to have their roles acknowledged in product branding (logos and style etc). All these concerns attached to the outside of the problem domain box affect (cross-cut) everything inside the box. A research centre (RC) and the Forum Practices are the principal sub-domains of the problem domain and inside the Forum Practices are a Master repository (in one of the practices) and the MedSys and medical staff sub-domain (in all of the practices as

indicated by notation {Practice}). Inside the problem domain box are more localized concerns including a concern about 'Kit Installation'; an example of a domain context that has a different time-scale to the normal system-operation (see Table 3).

Figure 14 was constructed on the basis of a two hour discussion between stakeholders and remained stable throughout the development. It satisfied our third objective (Table 1)and laid a foundation for the first. In our earliest attempts at goal sketching we did not realize the importance of first obtaining an agreed problem domain and concerns diagram and invariably paid the price of taking much longer to establish root GOPs and this compromised our second objective.

After achieving Figure 14 it was a straightforward process to finish a structurally complete GR model by pair sketching and cross-checking with the stakeholders. We proceeded rapidly to a complete G-R model (taking about one day) but here whilst being confident that we are satisfying three of our objectives for goal sketching it must be noted that the third is challenged as it remained fully understandable only to a subset of stakeholders. The situation was remedied by a two stage process: (1) talk through the contents and (2) debate the correctness of the contents. Nevertheless the project manager could always use the representation to ensure that the right questions were asked and to ensure that the key assumptions were recorded.

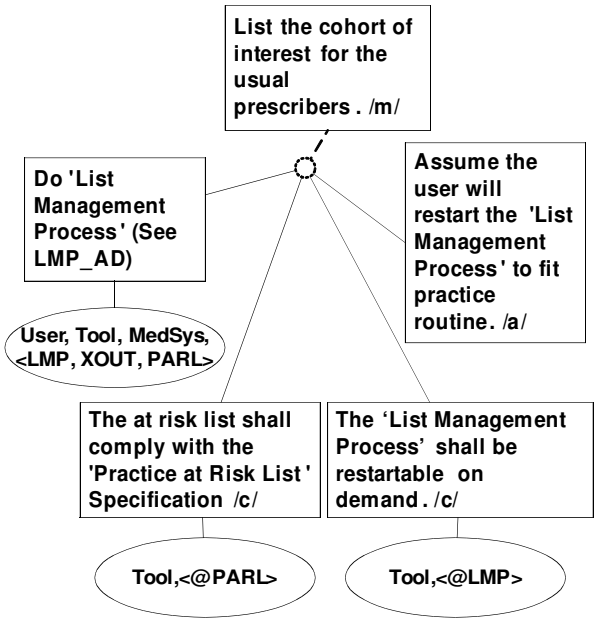


Figure 15. Figure 2 G-R model for one concern of Figure 14

Figure 15 illustrates one of the concerns from Figure 14and amounts to about 1/6th of the whole G-R model. It includes the responsibilities (which we usually only expose to the technical stakeholders): The actor MedSys is a sub domain of the Practice Domain. The semantic tag XOUT reflects cross-cutting of the regulatory concerns. The PARL and LMP tags show constraints acting on the "Do 'List .." goal. This particular goal is interesting as it is an example of hiding a detailed refinement that was constructed and negotiated using the activity diagram approach shown in Figure 16. The full G-R accommodates this figure in the manner illustrated in the scales example above.

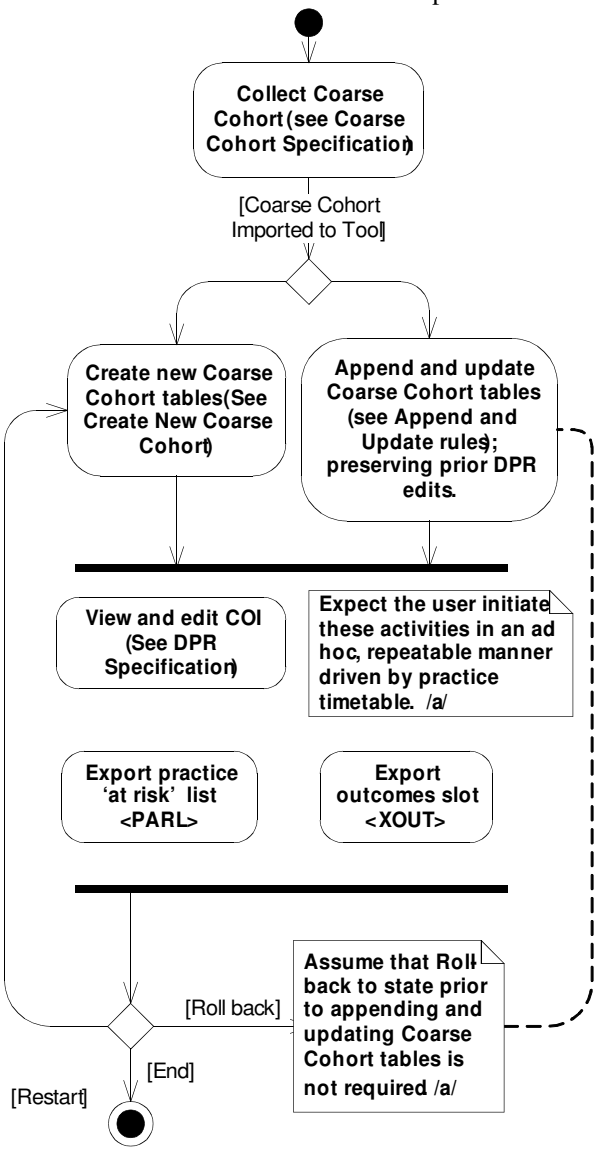


Figure 16: List management process

There is a significant simplifying assumption in Figure 16 agreed by all stakeholders for this stage of

development (to set aside roll-back). The need to surface such assumptions can easily be missed in less disciplined approaches. But where feasibility and adequacy are in conflict, as they were here, it is crucial to help the stakeholders make a decision. Figure 16 was reviewed on several occasions by the stakeholders; and thus improves our score on our third objective (Table 1).

Our tool support for our method allows us to annotate the G-R model. Included is a traffic-light annotation on each GOP to indicate our confidence in the refinement argument and/or its feasibility. Further we can export the leaves of the model into a project management spreadsheet to define the developments and procurements to be accomplished and the load-bearing assumptions to be monitored. This has been helpful to project managers.

In one of our venture capital supported development projects the main use of our method was to guide the development of acceptance testing. The test team found that working from a requirements backlog failed to provide sufficient understanding of the behavior that was being warranted by their product (see [8]. After several backlogs driven sprints the coherent picture of the intended user experience became unclear. This made test design very difficult and led to problems of product regression. The remedy was to use activity diagrams in the manner described here to reverse engineer the entire functionality of the product. This produced a set of four level nested activity diagrams upon which the acceptance tests could be designed. Converting these to a G-R model showed that they needed to pay more attention to the 'Impose Logic' goals described in previous sections. It also allowed the cross-cutting effects of the non-functional requirements to be included systematically in the tests. Recently the company has applied formal inspections, guided by the G-R model, to guarantee that the activity diagrams comply with all engineering and product management stakeholders' expectations. We will report further on this separately.

6. Related Work

We have mentioned some related work in the introduction. In addition we comment here on related requirements engineering material.

Work has been done on how some of the best practices of requirements engineering could enrich agile approaches [27]. The practices described include customer interaction, requirements analysis, non-functional requirements and managing change. The paper suggests that ways of adapting requirements management practices for agile processes are needed. However note that [27] simply describes how to include requirements engineering methods in an agile development process, rather than describing a method for requirements engineering that is agile. Similarly Nawrocki et al propose a way in which documented requirements could be introduced into XP through the use of automated tools, the Web and on-line documentation [28].

Cao and Ramesh have reported on how agile requirements engineering differs from traditional requirements engineering [29]. Their study showed that the agile case is more dynamic and adaptive than the traditional.

Orr suggests that it is possible to combine requirements and agile development by using up-to-date hardware and sophisticated graphical software [30]. Prototypes are suggested as a way to improve the process of defining requirements. However this work emerged from practice rather than from a theoretical technique such as goal-oriented requirements engineering.

Ambler describes an agile approach to modeling requirements, utilizing approaches such as the planning game of Extreme Programming and the Scrum methodology [31]. Similarly Leffingwell and Widrig discuss an agile requirements technique that is based on use-case specifications [32]. They also provide guidelines for selecting which requirements method (extreme, agile, or robust) is right for a particular project. However, again these approaches do not have a formal method such as goal-oriented requirements as a basis.

7. Further Work

The work reported here concerns the basics of the goal sketching technique. We are undertaking the following investigations to advance the work:-

1. Application to more industrial projects to confirm the applicability and practicality of the method for use in Agile projects.
2. The relationship between SSM[18] and the problem of transforming stakeholders concerns into goals.
3. Development of tools to accelerate the speed of sketch drafting and refactoring. In this area we are currently exploring the use of UML diagrams such as activity diagrams as these are well suited to the problem of determining behavioral goal refinements.
4. Development of metrics and supporting tools to exploit the structure of goal graphs in conjunction with expert judgments to quantify the adequacy and

feasibility of the intention expressed in a goal graph. It is anticipated that this will contribute significantly to the better planning of project stages and the improved sharing of expectations amongst the key stakeholders.

5. Tools to export goals sketches into KAOS for cases that justify upgrading from a goal sketch to a rigorous KAOS analysis.

8. Conclusion

In this paper we began by observing the problem of helping stakeholders set realistic expectations and take decisions. The problem is particularly pronounced in agile projects but is not limited to them. We have proposed a disciplined method of goal responsibility modelling as the basis for supporting stakeholders but also argue that success depends upon a set of practical objectives. We have also presented a goal sketching technique aimed to satisfy these objectives. Our experience shows that goal sketching in its present state performs well against our objectives although more validation is still needed.

The emphasis of goal sketching has been to provide a disciplined method of appraising the validity of a set of requirements for a project. Our method can be used alongside other requirements methods (especially use-cases) and can play an important part in reinforcing the coherence of agile requirements engineering based on backlogs.

9. Acknowledgements

The authors would like to acknowledge their industrial collaborators. In particular: Nick Gradwell, Product Manager of ClearPace Ltd; Dr Steve Moyle and James Wilson of Secerno Ltd., Ian Lycett KTP Associate at Image Farm Ltd; and Sean O'Mahoney, Martin Roskell and Richard Olearczyk of Oskis Informatics Ltd.

10. References

[1] D. Nevo, and M. Wade, "How to avoid disappointment by design", Communications of the ACM, 2007, Vol 50, No.4 .

[2] A. Desilets, "The Agile Physician", letter, IEEE Software, vol. 24, No. 3, 2007.

[3] A. Dardenne, A. van Lamsweerde, and S. Fikas, "Goal-Directed Requirements Acquisition", Science of Computer Programming Vol. 20, pp. 3-50, North Holland., 1993, pp. 3-50.

[4] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Boston, 2001.

[5] I.F., Alexander, and R. Stevens, *Writing Better Requirements*, Addison Wesley, 2002.

[6] A.van Lamsweerde, "Goal-Oriented Requirements Engineering: A Round trip from Research to Practice", 12th IEEE International Requirements Engineering Conference (RE'04), 2004.

[7] K. Boness, and R. Harrison, "Goal Sketching: Towards Agile Requirements Engineering," ICSEA, International Conference on Software Engineering Advances (ICSEA 2007), 2007, pp.71-6.

[8] K. Boness, and R. Harrison,"Goal Sketching with Activity Diagrams," ICSEA, International Conference on Software Engineering Advances (ICSEA 2008), 2008

[9] K. Boness, and R. Harrison, and A. Finkelstein, "A lightweight technique for assessing risk in requirements analysis", Software, IET, 2008, Volume: 2, Issue: 1 pp. 46-57.

[10] I. Sommerville, and P. Sawyer, *Requirements Engineering – a good practice guide*, Wiley, Chichester. 1997.

[11] S. Adolph, and P. Bramble, *Patterns for effective Use Cases,* Addison-Wesley, 2003.

[12] A.M. Davis, *Just Enough Requirements Management*, Dorset House Publishing, New York, 2005.

[13] T.P. Kelly, and R.A. Weaver, "The Goal Structuring Notation –A Safety Argument Notation." Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases, July 2004.

[14] L. Rising, and N. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software* July/August 2000.

[15] B. Boehm., "A Spiral Model of Software Development and Enhancement", *Computer*, May 1988, pp. 61-72.

[16] M. Jackson, Problem *Frames: Analysing and Structuring Software Development Problems,* Addison Wesley, 2000.

[17] J. March, and H.A. Simon, *Organisations*, New York: Wiley, 1958.

[18] P. Checkland, and J. Scholes, Soft *Systems Methodology in Action,* John Wiley and Sons, 1990.

[19] J. Gibson, "The Theory of Affordances". In Perceiving, Acting, and Knowing, Eds. R. Shaw, and J. Bransford, ISBN 0-470-99014-7. 1977.

[20] IEEE, "IEEE Guide for Developing System Requirements Specifications" IEEE Std-1223 (1998).

[21] J.A. Dewar, and C.H. Builder, et al., "Assumption-Based Planning: A Planning Tool for Very Uncertain Times", Santa Monica, RAND. 1993.

[22] I. Alexander, "A Taxonomy of Stakeholders," Int'l J. Tech. and Human Interaction, vol. 1, no. 1. 2005.

[23] S.J. Bleistein, K. Cox, and J. Verner, "Requirements Engineering for e-business Systems: Integrating Jackson Problem Diagrams with Goals Modelling and BPM", Proceedings of 11th Asia-Pacific Software Engineering Conference (APSEC'04) IEEE, 2004.

[24] W. G. Vincenti, "What Engineers Know and How They Know It:", Analytical Studies from Aeronautical History: The Johns Hopkins University Press, 1990.

[25] T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

[26] S. Deep, and L. Sussman, *Close the Deal: 120 Checklists for Sales Success,* Sandler Institute.

[27] A. Eberlein, and J. Cesar Sampaio do Prado Leite, "Agile Requirements Definition: A View from Requirements Engineering", International Workshop on Time-Constrained Requirements Engineering TCRE'02, Essen, Germany, Sep, 2002.

[28] J. R. Nawrocki, M. Jasiński, B. Walter, and A. Wojciechowski, "Extreme Programming Modified: Embrace Requirements Engineering Practices", Proceedings of the 10th Anniversary IEEE Joint international Conference on Requirements Engineering (September 09 - 13, 2002). RE. IEEE Computer Society, Washington, DC, 303-310.

[29] L. Cao, and B. Ramesh, "Agile Requirements Engineering Practices: An Empirical Study". IEEE Software. 25, 1 (Jan. 2008), 60-67. 2008.

[30] K. Orr, "Agile Requirements: Opportunity or Oxymoron?" *IEEE Software, 21, 3* (2004), 71-73.

[31] S.W. Ambler, *Agile Modelling: Effective Practices for eXtreme Programming and the Unified Process*, John Wiley & Sons, 2002.

[32] D. Leffingwell, and D. Widrig, 2003 *Managing Software Requirements: a Use Case Approach*. 2. Pearson Education.

Verification of Evidence Life Cycles in Workflow Diagrams with Passback Flows *

Osamu Takaki, Takahiro Seino, Izumi Takeuti, Noriaki Izumi and Koichi Takahashi
National Institute of Advanced Industrial Science and Technology (AIST)
2-41-6 Aomi, Koto-ku, Tokyo 135-0064, Japan
{o-takaki, seino-takahiro, takeuti.i, n.izumi, k.takahashi}@aist.go.jp

Abstract

We introduce the Evidence Verification Algorithm (EVA) in this paper, which verifies consistency of life cycles of evidences (evidence documents) in workflow diagrams. We used the AIST Workflow Language (AWL) as the syntax for workflow diagrams, which has additional information about evidences. A workflow diagram of AWL is essentially the same as a Unified Modeling Language (UML) activity diagram. EVA verifies the existence of consistent sequences of flows between the occurrences of evidences in a workflow diagram. It is important to verify consistency of life cycles of evidences, since some defects in the workflow diagram itself can be found by checking inconsistent life cycles of evidences in a workflow diagram.

Keywords: workflow diagram, verification, evidece life cycle

1. Introduction

We define a consistency property of life cycles of evidences described in a workflow diagram in this paper and introduce an algorithm that verifies the consistency property. Here "evidence" is a technical term which means an annotation on a workflow diagram, which denotes a document on which information is written, and/or with which something is approval, during the process of an operation. Workflow diagrams play a central role in describing business processes in the development of large-scale information systems, especially in analyzing the requirements of these systems. There have been several standard workflow languages such as BPMN [1] or XPDL [15], and numerous investigations into methodologies of verifying several consistency properties in workflow diagrams (e.g., see [12]).

*The authors are grateful to anonymous referees for their fruitful comments. This work was supported by 'Service Research Center Infrastructure Development Program 2008' from METI and Grant-in-Aid for Scientific Research (C) 20500045.

However, verifying consistency properties in the life cycles of documents, which are described in workflow diagrams, has not been sufficiently investigated. In large organizations such as large enterprises or governments, documents such as order forms, estimate sheets, specification descriptions, invoices, and receipts play significant roles for purposes of feasibility, accountability, traceability, or transparency of business. Tasks involve workers with different roles in such organizations, and these are carried out by circulating documents. Such documents are considered as kinds of evidences for the purposes above. We describe such documents with evidences in this paper. For simplicity, we often call such documents themselves "evidences".

Some evidences require office workers to carry out various tasks. Some evidences are manuals that teach workers how to conduct tasks. Some workers check evidences and sign them in when they accept their content. Therefore, numerous actual operations are currently based on evidences even if they are carried out with information systems. Consequently, it is important to consider workflow diagrams in which one can concretely and precisely describe the life cycles of evidences to analyze requirements in developing large-scale information systems.

When someone develops workflow diagrams, they often make errors in describing evidences, because their states are subtly affected by other evidences around them. Moreover, many inconsistencies in evidences come from inconsistencies in constructing the diagrams. The larger a diagram becomes, the harder it is to find inconsistencies in evidences that is in it.

However, verifying the consistency of evidence helps us to confirm correctness of the diagrams. In fact, we can find numerous defections and redundancies in flows by finding inconsistencies in evidences around the flows. Therefore, it is worth verifying evidences formally and/or automatically.

We define the consistency property of life cycles of evidences in a workflow diagram in this paper, which is described with a new language for workflow diagrams called AIST Workflow Language (AWL), and introduce the Evidence Verification Algorithm (EVA), which verifies the con-

sistency property of a given workflow diagram.

A workflow diagram of AWL is essentially the same as a Unified Modeling Language (UML) activity diagram. The control flows in the workflow diagrams of AWL as well as major workflow languages are described by arrows. Moreover, one can describe how to control evidences by using the arrows in the workflow diagrams of AWL. Since most operations are carried out with some specific evidences, it is rational for a single arrow to denote both a control flow and a flow of an evidence. By describing the flows of evidences with appropriate control flows, one can easily describe and understand the life cycles of the evidences. This paper targets AWL instead of UML, because AWL is used in the actual development of large-scale information systems at the National Institute of Advanced Industrial Science and Technology (AIST) [9].

Roughly, the life cycles of evidences mean a series of states of the evidences, and consistency of evidence life cycles in a workflow means that the workflow has no inconsistent life cycles of evidences. A consistent evidence life cycle is defined in a workflow diagram with correct structure, where "correct structureff is defined in [4] and [10].

EVA receives a workflow diagram as input data and returns a list of subgraphs as output data, each of which destroys a consistent evidence life cycle. EVA checks the "local evidence conditions", which we will define in this paper, to verify the consistency of evidence life cycles in a workflow diagram.

EVA is designed to verify life cycles of evidences in *acyclic* workflow diagrams. We introduce the Removing Algorithm of Passback Flows (RAPF), where a "passback flow" in a workflow diagram denotes a kind of flow that appears at the boundary of a "main stream" and the "replay of an operation" in the workflow diagram in order to apply EVA to cyclic workflow diagrams. We will define passback flows in a workflow by using the graph-theoretical properties of the workflow diagram, and define RAPF, which translates a workflow diagram into various workflow diagram(s) with no passback flows, which can be applied to EVA.

We implemented both EVA and RAPF, and used them to verify workflow diagrams in actual development at AIST. Although both EVA and RAPF target workflow diagrams of AWL, one can easily apply these algorithms to activity diagrams of UML.

This paper is based on a previous paper [7] but differs from it in three respects. First, we discuss the consistency property of life cycles of evidence and its verification algorithm based on workflow diagrams in AWL instead of UML activity diagrams. Second, we introduce RAPF to apply the verification algorithm to cyclic workflow diagrams. Third, we add proofs of the main lemmas, which we omitted from [7].

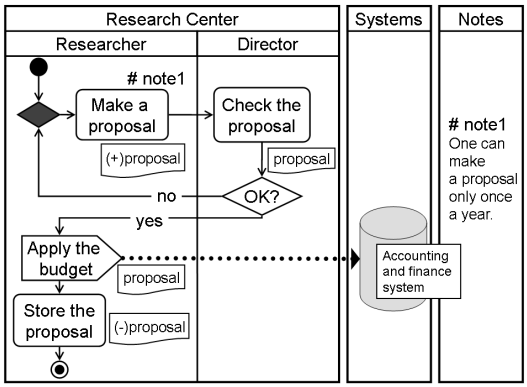


Figure 1. An example of a workflow diagram

The remainder of this paper is organized as follows. We introduce AWL in Section 2 and explain the consistency of life cycles of evidence in workflow diagrams in 3. We introduce EVA in Sections 4 and 6. We define local evidence conditions in Section 5 and introduce RAPF in Section 7. Section 8 explains the experimental results, by using an implementation of EVA. The experimental results indicate EVA could effectively be used to test and verify the consistency of life cycles of evidence to substantiate the construction of the workflow itself was consistent.

2. AIST workflow language

In this section, we explain a language of workflow diagrams [9], which is called "AWL" (AIST Workflow Language). AWL is defined to be appropriate to compose workflow diagrams for human workflow easily, and to verify consistency of evidence life cycles in workflow diagrams.

2.1. Overview of AWL

Comparing standard workflow languages such as BPMN or XPDL, the main feature of AWL is that in a workflow diagram of AWL one can assign to each activity a list of evidences (evidence documents) which are used in the activity.

The figure 1 is a workflow diagram describing a work of planning of a research. In the diagram, the rectangles and the pentacle denote operations needed for planning of a research. The figures near the polygons above denote evidences (evidence documents) used on the operations.

In this example, first a researcher composes a proposal of a research, and then a director checks the proposal. If the proposal passes the checking, then the proposal is returned to the researcher and he/she applies the budget on an accounting and finance system based on the proposal, and finally the proposal is stored by the researcher. If the

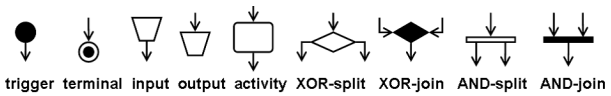


Figure 2. Shapes of nodes in workflow diagrams

proposal does not pass the checking, then the proposal is returned to the researcher and he/she remakes the proposal.

In this paper, we discuss only control flow and evidence life cycles in a workflow diagram. Therefore, we omit notions that are not relevant to control flow of workflows or evidences. For example, in this paper we do not consider data flows or actors in workflow diagrams.

2.2. Control flow of AWL

In the perspective of control-flow, workflow diagrams of AWL are similar to those in BPMN or XPDL, which are the most standard workflow languages, or workflows in previous researches such as [3], [5] or [13].

Definition 2.1 A workflow in the perspective of control-flow denotes a directed graph $W := (\mathbf{node}, \mathbf{flow})$ that satisfies the following properties.

1. $\mathbf{node}$ is a non-empty finite set, whose element is called a node in W.
2. $\mathbf{flow}$ is a non-empty finite set, whose element is called a flow in W. Each flow f is assigned to a node called a source of f and another node called a target of f.
3. Each node is distinguished, as follows: start, end, activity, XOR-split (branch), XOR-join (merge), AND-split (fork) and AND-join (rendezvous).
4. Whenever a flow f has a node x as the target (or the source) of f, x has f as an incoming-flow (resp. an outgoing-flow) of x. The numbers of incoming-flows and outgoing-flows of a node are determined by the type of the node. We itemize them in the following table.

	incoming-flows	outgoing-flows
start	0	1
end	1	0
activity	1	1
XOR-, AND-split	1	$\geqq 2$
XOR-, AND-join	$\geqq 2$	1

Table 1. Numbers of incoming- and outgoing-flows of a node

5. W has just one start and at least one end.
6. Activity diagrams are "simple" graphs, that is, for each activity diagram $\mathcal{A}$, and for each nodes N_1, N_2 in $\mathcal{A}$ there is *one* edge (flow) from N_1 and N_2 at most. Moreover, each activity diagram has *no circle edge*, that is, there is no flow from a node N to the same node N.
7. For a node x in W, there exists a path on W from the start node of W to x, where a path from s to x denotes a sequence $\pi = (f_0, \ldots, f_n)$ of flows in W such that the source of f_0 is s, the target of f_n is x and that the target of f_i is the source of f_{i+1} for each $i < n$. Moreover, there exists an end node e and another path on W from x to e.

2.3. Evidence

By "an evidence" in workflow diagrams one means a paper document or a data (a data file) of a document. In this paper, we regard an evidence as a paper document, which is composed, referred, re-written, judged, stored or dumped in some activities. Unlike data files, an evidence does not increase. Though one can make a copy of it, the copy is regarded not to be the same thing as the original evidence. Moreover, unlike data in a system multiple people can access simultaneously, an evidence can not be used by multiple people at the same time.

In formulating workflow diagrams, especially, those for human workflows, evidences are still very important even through a lot of paper documents are replaced by data (data files) in information systems. In a workflow diagram of AWL, evidences used in an activity is explicitly described in the activity, in order to describe and verify life cycles of evidences more correctly.

In the technical perspective, a list of evidences with length at least 0 is assigned to an activity, and an evidence E is defined to be a triple $(e, created, removed)$, where e is a label, and $created$ and $removed$ are boolean values. In what follows, we fix a non-empty set $\mathbb{E}$.

Definition 2.2 *Evidence* is a triple $(e, created, removed)$, where e is an element of $\mathbb{E}$ and $created$ and $removed$ are boolean values, that is, they are elements of $\{true, false\}$. For each evidence $E := (e, created, removed)$, we call e the *evidence label* of E.

Remark 2.3 In what follows, we denote E by the following ways.

(i) If $created = \text{false}$ and $removed = \text{false}$, then we abbreviate E to "e".

(ii) If $created = \text{false}$ and $removed = \text{true}$, then we abbreviate E to "$(-)e$".

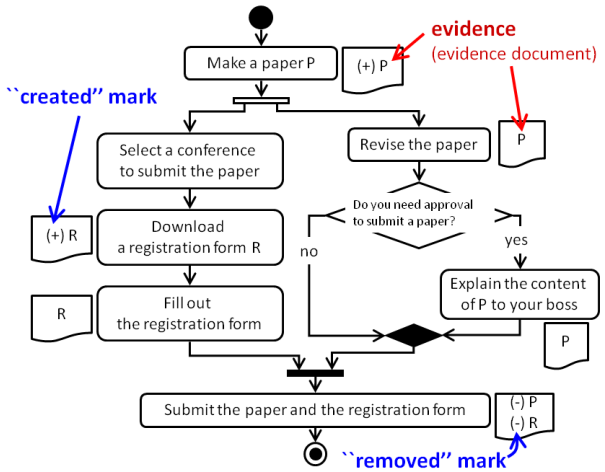


Figure 3. A workflow diagram of paper submission

(iii) If $created = \text{true}$ and $removed = \text{false}$, then we abbreviate E to "$(+)e$".

(iv) If $created = \text{true}$ and $removed = \text{true}$, then we abbreviate E to "$(+)(-)e$".

For a workflow diagram W, we consider an allocation which assigns to each activity in W a string of evidences. Note that such an allocation may assign to some activities the empty string, i.e., the string with length 0. By using workflow diagrams, one can express a lot of workflows. In order to explain evidences, we give an example of a workflow diagram which explains how to submit a paper, as follows.

For each workflow diagram W, each activity A in W and for each evidence E in the string assigned to A, we call E an evidence *on* A and call A an activity *having* E. Moreover, if the evidence label of E is denoted by e, we often call e the evidence label *on* A and call A an activity *having* e.

Moreover, for simplicity, we often identify each evidence with its evidence label. So, in what follows, *we often abbreviate "an evidence label" to "an evidence"*.

Remark 2.4 In what follows, we assume that, for each workflow diagram W and each activity A in W, A does not have multiple evidences sharing the same evidence label. We call the condition *the basic evidence condition*. [1]

Since each workflow diagram W is assume to satisfy the basic evidence condition, if an activity A in W has an evidence label e, A has just one evidence E with label e. So, we often say that e *is created (or removed) on* A if A has an evidence E having the $(+)$-mark (or the $(-)$-mark, respectively).

[1] We assume that each evidence can be differentiated from others even if some evidences share the same content. For example, if $\mathbf{E}$ is the set of documents, and if an evidence e is copied in an activity A, then one should not consider that A has two es, but should consider that A has e and a copy of e.

3. Consistency properties of workflow diagrams

The main subject of this paper is to verify consistency property of evidence life cycles in a workflow diagram. However, the consistency property is closely related to another consistency property of control flow of a workflow diagram. Thus, in this section, we first explain consistency property of control flow of a workflow diagram, and then we explain consistency property of evidence life cycles in the workflow diagram.

In the following subsections of this section and the three coming sections 4∼6, we will treat only *acyclic* workflow diagrams. That is, we assume that any workflow do not have a loop, where a loop denotes a sequence of flows

$$N_0 \xrightarrow{f_0} N_1 \xrightarrow{f_1} \cdots \xrightarrow{f_{n-1}} N_n \xrightarrow{f_n} N_0.$$

We will treat *cyclic* workflow diagrams in Section 7.

3.1. Correctness of Workflows

Consistency verification of workflows on the control flow perspective is one of the most important issues in research area of workflow verifications. There are a lot of researches of consistency properties of workflows in the viewpoint of control flow of them such as [2], [4], [5], [10], [11], [13] and [14].

An inconsistency of structures of workflows comes from a wrong combination of XOR-split/join nodes and AND-split/join nodes. Such inconsistencies are known as "deadlock" and "lack of synchronization" [5]. An acyclic workflow which is deadlock free and lack of synchronization free is said to be "correct" [13].

Definition 3.1 For an acyclic workflow W, an *instance* of W denotes a subgraph V of W that satisfies the following properties.

1. V contains just one start node. Moreover, for each node x in V, there exists a path on V from the start node to x.
2. If V contains an XOR-split c, then V contains just one outgoing-flow of c.
3. If V contains a node x other than XOR-split, then V contains all outgoing-flows of x.

Definition 3.2 Let W be an acyclic workflow.

1. An instance V of W is said to be *deadlock free* if, for every AND-join r in V, V contains all incoming-flows of r.

2. An instance V of W is said to be lack of *synchronization free* if, for every XOR-join m in V, V contains just one incoming-flow of m.

Definition 3.3 An acyclic workflow W is said to be *correct* if every instance V of W is deadlock free and lack of synchronization free.

Instances of a workflow diagram are not used not only to define correctness property but also to define consistency property of evidence life cycles in the workflow, which we will define in the next section.

3.2. Consistency property of evidence life cycles

Roughly, the "life cycle" of an evidence means that a series of states of the evidence. To be more exact, the life cycle of an evidence e (in $\mathbb{E}$) means when e is created, how e is moved to some activities, and when e is removed (archived or destroyed).

In order to define consistent life cycles of evidences in workflow diagram in a rigorous manner, we introduce some new concepts.

Since workflow diagrams have XOR-split nodes, one can regard each activity diagram $\mathcal{A}$ as an gathering of flow-sequences, each of which is obtained from $\mathcal{A}$ based on XOR-split nodes in $\mathcal{A}$. Based on the point, we introduce the following definition.

Definition 3.4 Let W be a workflow diagram and C the set of all XOR-splits on W. Then, a *phenomenon* on W denotes a function $\psi : C \to \mathbf{flow}(W)$ satisfying that $\psi(c)$ is an outgoing-flow of c for each $c \in C$, where $\mathbf{flow}(W)$ denotes the set of all flows in W.

Lemma 3.5 For a workflow diagram W and a phenomenon ψ on W, there exists a unique instance V of W such that for every XOR-split c in V the outgoing-flow of c in V is $\psi(c)$. We refer to the instance V as $W(\psi)$.

Conversely, for a instance V of W, there is a phenomenon ψ with $V = W(\psi)$.

The lemma above is easily shown.

One can consider each $W(\psi)$ as the workflow diagram obtained from W by extracting all activities and flows which take place under the phenomenon ψ.

Definition 3.6 For a workflow diagram W, a *line* in W is a sequence of flows in W

$$L = (A_1 \xrightarrow{f_1} A_2 \xrightarrow{f_2} \cdots \xrightarrow{f_{n-1}} A_n)$$

which satisfies the following properties.

(i) A_1 is an activity or the start in W.

(ii) A_n is an activity or an end in W.

(iii) $A_2, \ldots, A_{n-1}$ are nodes in W, each of that is not any activity, the start, nor any end.

For a line L above, A_1 is called the *source* of L, A_n the *target* of L and f_{n-1} the *target flow* of L.

Definition 3.7 A line L is said to be *equivalent* to another line L' if L and L' share the source and the target.

Definition 3.8 A sequence π of lines is said to be *equivalent* to another sequence π' of lines if there exist lines $L_1, \ldots, L_n$ and $L'_1, \ldots, L'_n$ such that

$$\pi = (A_1 \xrightarrow{L_1} A_2 \xrightarrow{L_2} \cdots \xrightarrow{L_{n-1}} A_n)$$

$$\pi' = (A_1 \xrightarrow{L'_1} A_2 \xrightarrow{L'_2} \cdots \xrightarrow{L'_{n-1}} A_n)$$

and that, for each $i = 1, \ldots, n$, L_i is equivalent to L'_i.

$L \sim L'$ (or $\pi \sim \pi'$) denotes that L is equivalent to L' (π is equivalent to π', respectively). Note that every line is equivalent to itself, and so is every sequence of lines.

Definition 3.9 Let W be an acyclic workflow diagram, ψ a phenomenon of W and let e be an evidence in W. Then, the *consistent life cycle* of e on $W(\psi)$ is the sequence π of lines in $W(\psi)$

$$\pi := (A_0 \xrightarrow{L_0} A_1 \xrightarrow{L_1} \cdots \xrightarrow{L_{n-1}} A_n)$$

which satisfies the following properties.
(i) Every activity A_i has e.
(ii) e is created on A_0.
(iii) e is not created on A_i for any i with $0 < i \leqq n$.
(iv) e is removed on A_n.
(v) e is not removed on A_i for any i with $i < n$.

Definition 3.10 An acyclic workflow diagram W is said to *have consistent evidence life cycles* if, for each phenomenon ψ of W, each activity A in $W(\psi)$ and for each evidence e on A, there is an essentially unique consistent life cycle π of e which contains A.

The statement "there is an essentially unique consistent life cycle π of e containing A" means that there is a consistent life cycle π of e containing A and that $\pi \sim \pi'$ for each consistent life cycle π' of e containing A.

4. Verification algorithm of evidence life cycles

The main body of this paper is to introduce an algorithm verifying consistency of evidence life cycles in a given acyclic workflow diagram, which is called *EVA* (Evidence Verification Algorithm).

Although details of EVA is described in the section 6, we here explain input and output data and the main property of EVA.

Input data of EVA: a correct workflow diagram W and all instances $\{W(\psi)\}$ of W.

Output data of EVA: a string consisting of pairs (L, e), where L denotes a sequence of flows and e an evidence.

Each (L, e) in output data denotes a defect of an evidence life cycle in input data.

Theorem 4.1 For each correct workflow diagram W, if the output data of W by EVA is the empty string, then W has consistent evidence life cycles, and vice versa.

This theorem is a corollary of Lemma 6.2 described in Section 6.

Remark 4.2 Here we omit explanations about how to extract instances from a workflow diagram and how to verify correctness of it. For details, see [6]. One can also refer to [2], [10], [11], [13] and [14],

5. Local evidence conditions

Actually, in order to verify consistency of evidence life cycles of a given correct workflow diagram W, EVA checks whether or not W satisfies "local evidence conditions". In order to explain there conditions, we first introduce a proposition and some definitions.

Proposition 5.1 For each workflow diagram W, each line L and for each evidence e contained in L, just one of the following properties holds.
(1) The source S and the target T of L share e, and e is not removed on S and e is not created on T.
(2) The source S and the target T of L share e, and e is removed on S, and e is created on T.
(3) The source S of L has e, e is removed on S, and the target of L does not have e.
(4) The target T of L has e, e is created on T, and the source of L does not have e.
(5) The target T of L has e and e is not created on T. Moreover, if the source S of L has e, then e is removed on S.
(6) The source S of L has e and e is not removed on S. Moreover, if the target T of L has e, then e is created on T.

One can easily show the proposition above.

Remark 5.2 Let L be an line with source S and target T activities, and let e be an evidence, for example, a document used in S or T or both of S and T. Moreover, assume that L contains no AND-split and no AND-join. Then, if S has e and e is not removed on S, e should exist on T. Therefore, (6) above can be regard as an wrong state. Similarly, if T has e and e is not created on T, e should "come from" S, and hence, (5) above can be regard as an wrong state as well as (6).

We next assume that L has an AND-split F and another line L' has S as its source. Then, when S has e, e is not removed on S, and when T does not have e, it is possible that e "pass" from S to the target of L'. In such a case, one can not assure that (6) is wrong. One can consider several similar cases.

Definition 5.3 A pair (L, e) of a line L and an evidence e is called a *line-evidence*.

Definition 5.4 To each line-evidence (L, e), we assign one of the following states.
(1) SCS (State of Consistent Succession) if (L, e) satisfies (1) in Proposition 5.1.
(2) SIR (State of Inconsistent Redundancy) if (L, e) satisfies (5) in Proposition 5.1.
(3) SID (State of Inconsistent Defection) if (L, e) satisfies (6) in Proposition 5.1.
(4) SCNS (State of Consistent Non-Succession) if (L, e) satisfies one of (2)$\sim$(4) in Proposition 5.1.

Definition 5.5 For a correct workflow diagram W, W is said to satisfies *local evidence conditions* if, for each phenomenon ψ of W, the restricted graph $W(\psi)$ satisfies the following conditions.
(1) For each line L in $W(\psi)$ and for each evidence e, if (L, e) is assigned SIR, then there exists a line L' sharing the target with L such that (L', e) is assigned SCS.
(2) For each line L in $W(\psi)$ and for each evidence e, if (L, e) is assigned SID, then there exists a line L' sharing the source with L such that (L', e) is assigned SCS.
(3) There are not two line-evidences (L, e) and (L', e), which are assigned SCS, and which share the source (or the target), but which do not share any node as their targets (or their sources, respectively).

The following lemma indicates that, for a workflow diagram W, if W is correct, local evidence conditions suffice to verify consistency of evidence life cycles in W.

Lemma 5.6 For each correct workflow diagram W, if W satisfies local evidence conditions, then W has consistent evidence life cycles, and vice versa.

We show this lemma in Appendix A of this paper.

6. Definition of EVA

By virtue of Lemma 5.6, in order to verify consistency of evidence life cycles of a given correct workflow diagram W, it is sufficient to check that W satisfies local evidence conditions. So, we establish EVA as an algorithm finding line-evidences (L, e) which violate local evidence conditions of a given correct workflow diagram.

Definition of EVA

(I) Input and output data of EVA are described in Section 4.

(II) The content of EVA is defined, as follows.
(EVA.1) Prepare three empty sets $\mathbf{Suc}$, $\mathbf{Inq}$ and $\mathbf{Res}$, which are provided for recording line-evidences.
(EVA.2) Execute EV.2.1 and EV.2.2 below in parallel.
(EVA.2.1) For a given W, starting the start of W, search all flows f in W by an appropriate graph search algorithm, [2] and all lines with target flow f.
(EVA.2.2) For each line L and for each evidence e contained in L, check the state of (L, e), and classify (L, e), as follows.

Case (i) where (L, e) is assigned SCS.

If L contains an AND-split or an AND-join, put (L, e) into $\mathbf{Suc}$. Otherwise, dump (L, e).

Case (ii) where (L, e) is assigned SIR.

If L contains an AND-join, put (L, e) into $\mathbf{Inq}$. Otherwise, put (L, e) into $\mathbf{Res}$.

Case (iii) where (L, e) is assigned SID.

If L contains an AND-split, put (L, e) into $\mathbf{Inq}$. Otherwise, put (L, e) into $\mathbf{Res}$.

Case (iv) where (L, e) is assigned SCNS.

Dump (L, e).

(EVA.3) For each phenomenon ψ of W, execute (EVA.3.1)$\sim$(EVA.3.3) below.
(EVA.3.1) Set $\mathbf{Suc}(\psi) := \mathbf{Suc} \cap W(\psi)$ and $\mathbf{Inq}(\psi) := \mathbf{Inq} \cap W(\psi)$, where $W(\psi)$ is regarded to be the set of line-evidences on $W(\psi)$.
(EVA.3.2) For each element (L, e) of $\mathbf{Inq}(\psi)$, check whether or not there exists an element (L', e) of $\mathbf{Suc}(\psi)$ such that (L, e) and (L', e) satisfy the property (1) or (2) in the definition of local evidence conditions. If (L, e) does not have such a (L', c), put (L, e) into $\mathbf{Res}$.
(EVA.3.3) Check whether or not there exist multiple elements of $\mathbf{Suc}(\psi)$ violating the property (3) in the definition of local evidence conditions. If there exist such elements, put them into $\mathbf{Res}$.
(EVA.4) Output $\mathbf{Res}$.

[2] We use a depth first search algorithm for implementation of EVA.

Remark 6.1 In most cases, it does not need to prepare all instances, since some instances $W(\psi), \ldots, W(\psi')$ share the same figure even though $\psi, \ldots, \psi'$ are not the same.

At the last, we show correctness of EVA.

Lemma 6.2 (Correctness of EVA) For each correct workflow diagram W, EVA terminates in finite steps. Moreover, all lines violating local evidence conditions of W are contained in the output of W by EVA, and vice versa. In particular, Theorem 4.1 holds.

We show this lemma in Appendix B.

7. Application of EVA to cyclic workflow diagrams

Until now, we have dealt with acyclic workflow diagrams. From now, we will discuss verification of correctness and consistency of evidence life cycles over cyclic workflow diagrams. In order to apply EVA to cyclic workflow diagrams, we extend the definitions of consistency properties in the previous sections to those over cyclic workflow diagrams. Thus, in order to extend the definitions of consistency properties, we consider a translation of cyclic workflow diagrams.

The main body of this section refers to [8].

Before defining the translation, we explain an observation of "real" workflow diagrams.

7.1. Observation of real workflow diagrams

By virtue of investigation of about 460 workflow diagrams, which have been composed in development of real large-scale information systems, we have the following observations about real workflow diagrams.

- **Observation 1.** Most loops in workflow diagrams contain flows which we call "passback flows".

A passback flow in a workflow diagram denotes a kind of a flow which appears at a boundary of a "main stream" and a "replay of an operation" in the workflow diagram. For example, in the figure 1, the main stream in the workflow is described by the sequence of flows between activities: "Make a proposal", "Check the proposal", "Apply the proposal" and "Store the proposal". On the other hand, if the proposal does not pass in the activity "Check the proposal", the proposal will be turned back to the previous activity "Make a proposal" via the flow labeled "no". In this case, the researcher have to replay the activity "Make a proposal". So, the flow "no" is a passback flow.

Workflow diagrams in the investigation above contain no loop which is considered in usual programs and assured its

termination property. That is, most loops in the workflow diagrams express replays of operations in the workflow diagrams. Thus, most loops contain passback flows.

- **Observation 2.** Every information described in a workflow diagram is treated as a "static" information. In particular, no information about operations after an operation is turned back via a passback flow.

For example, in the figure 1, in the case where a proposal is turned back to the activity "Make a proposal" via the passback flow "no", the researcher should not "make" a proposal, but "remake" the proposal. However, there is actually no modification for such a case in a workflow diagram.

It maybe possible to reconstruct such a workflow so that there is not such an inconsistency above which occurs on the target of the passback flow. Thus, one can select a policy which prohibit inconsistencies on the targets of passback flows like the example above. However, in many cases, this constraint is so strong that workflow diagrams have too large size or too complex structure. Therefore, in the real development, they do not select such a policy. Thus, we do not consider inconsistency between the target and the source of a passback flow in a workflow. Actually, from the real workflow diagrams in the investigation, we also have the following observation.

- **Observation 3.** The change of evidences during a passback flow is not described in the workflow diagram. In other words, even if there is some inconsistency between the evidences on the source node of (the first flow of) a path containing a passback flow and those on the target node of (the last flow of) the path, one should not regard it as an error.

In principle, evidences on each node in a workflow diagram are described only when a job stream arrived at the point for the first time. For example, the activity "Make a proposal" in the figure 1 has an evidence "(+)proposal", which is information described at the first time when the job stream arrives at the activity node from the start node. The evidence "(+)proposal" is not what is described in the case where a job stream arrives at the activity via the passback flow "no". Therefore, we should not consider that the evidence "proposal" in the activity "Check the proposal" changes to the evidence "(+)proposal" in the activity "Make a proposal" via the flow "no".

The observations above indicate that a verification algorithm should not output inconsistency between evidences on the source node of a path containing a passback flow and the target node of the path as an error. As a consequence of the discussion, we take the stance to deal with passback flows as special ones.

There maybe a workflow diagram in which a passback flow must not be regarded as a special one. We can not apply our methodology to such cases. We do not care about that, because such cases are rare.

7.2. Translation of cyclic workflow diagrams

In order to translate a cyclic workflow diagram into those to which EVA are applicable, we remove all passback flows in the cyclic. In this section, we first formalize passback flows in a workflow diagram, by using only graph-theoretical properties of the workflow diagram. Moreover, we introduce an algorithm which removes all passback flows in a given workflow diagram. For more details, see [8].

A path $(f_1, \ldots, f_n)$ in a workflow diagram W satisfying the following properties is called a *lariat path*.

1. The source of f_1 is a start node in W.
2. The target of f_n is the source of one of $f_2, \ldots, f_n$.
3. For each i and j with $i \neq j$, f_i and f_j do not share the same source.

The last flow f_n of a lariat path $\sigma := (f_1, \ldots, f_n)$ is called the *tail* of σ.

A path $(f_1, \ldots, f_n)$ in a workflow diagram W satisfying the following properties is called a *directly ending path*.

1. The source of f_1 is a start node in W.
2. The target of f_n is an end node in W.
3. For each i and j with $i \neq j$, f_i and f_j do not share the same source.

In the following two paragraphs, we consider what a passback flow is.

First, a passback flow heads in the opposite direction to a primary job stream and reaches such a job stream. Therefore, a passback flow is the tail of a lariat path.

Next, consider a flow contained in some directly ending path. Then, the flow is considered a member of a directly ending path, and the change of evidences during the flow is described in the workflow diagram, even if it is the tail of a lariat path. Therefore, the flow is not considered a passback flow.

By virtue of the discussion above, we can formalize passback flows, as follows.

Definition 7.1 A flow f in a workflow diagram W is called a *passback flow* if f is the tail of a lariat path in W and there is no directly ending path in W which contains f.

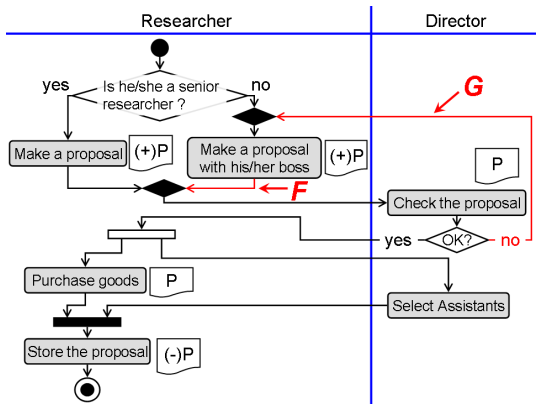


Figure 4. A workflow diagram of proposal submission

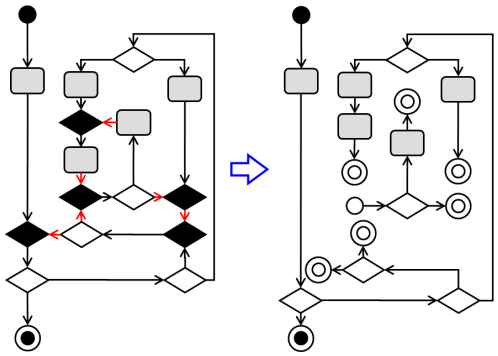


Figure 5. RAPF may divide a workflow diagram into multiple workflow diagrams

For example, in the figure 4, G is a passback flow, while F is not, even though F is the tail of a lariat path.

Now we define an algorithm *RAPF* (Removing Algorithm of Passback Flows) [8], which translates a workflow diagram W into some workflow diagram(s) with no passback flows, as follows.

1. Detect all flows in W and record tails $t_1, \ldots, t_n$ of all lariat paths in W.

2. Detect all directly ending paths and let $p_1, \ldots, p_m$ be tails each of which is not contained in any directly ending path. These tails are passback flows in W.

3. Replace targets $T_1, \ldots, T_m$ of $p_1, \ldots, p_m$ by new end nodes $E_1, \ldots, E_m$, respectively. These end nodes are called *additional end nodes*. Note that the number of incoming flows of each T_i decreases.

4. Execute the following operations corresponding to the number of incoming flows of each T_i. (Note that each T_i is an XOR-join or AND-join node.)

   (a) If T_i has more than or equal to two incoming flows, then leave T_i as it is.

   (b) If T_i has just one incoming flow, then replace the target of the incoming flow by the target of the outgoing flow of T_i, and then remove T_i as well as the outgoing flow of T_i.

   (c) If T_i has no incoming flow, then replace the source of the outgoing flow of T_i by a new start node and remove T_i. The new start node is called a *additional start node*.

RAPF makes no new lariat path. Moreover, any flow in a directly ending path does not become not to be contained in any directly ending path by RAPF. Thus, we have the following proposition.

Proposition. RAPF translates a workflow diagram W into RAPF(W), each element in which is a workflow diagram with no passback flows.

RAPF may output multiple workflow diagrams. We show an example in the figure 5. The operation on this example is executed mechanically by RAPF.

By RAPF, one can extend the definitions of correctness and consistency of evidence life cycles of acyclic workflow diagrams to those over cyclic workflow diagrams. In order to do so, we only have to redefine instances defined in Definition 3.1.

Definition 7.2 For an cyclic workflow W, an *instance* of W denotes a subgraph V of RAPFW that satisfies the properties 1)$\sim$3) in Definition 3.1.

By using the instances above, one can extend the definitions of correctness and consistency of evidence life cycles of acyclic workflow diagrams to those over cyclic workflow diagrams.

The role of RAPF is to extract a designer's intention of a workflow diagram. Here, the "designer's intention" is the intention which flows to be exceptional ones around which evidence life cycles should not be checked, and which flows to be usual ones around which evidence life cycles should be checked. Such a designer's intention is not completely formalizable, and hence, RAPF may not extract the designer's intention completely. However, from the observation of actual workflow diagrams, we claim that RAPF has enough ability to extract designers' intentions about passback flows correctly.

RAPF is not an algorithm which translates cyclic workflow diagrams to acyclic ones. Cyclic workflow diagrams are not contained in the range of EVA. Moreover, some

cyclic workflow diagrams are not translated into acyclic ones by RAPF. One can not apply EVA to such diagrams. However, such diagrams are rare, and we do not care about them.

8. Experimental results

The algorithm EVA is implemented as a java program called "evidenceVerifier", the input data of which is an xml-file expressing a workflow diagram, and the output data of which is an xml-file expressing a string of line-evidences which violate life cycles of evidences. [3] More properly speaking, evidenceVerifier has functions which execute the following for each input data W.
(1) verification of syntax and graph-theoretical properties (for example, connectivity) of W.
(2) translation of W to (an) acyclic workflow diagram(s) W^* by RAPF in Section 7.
(3) extraction of instances $\{S_1, \ldots, S_n\}$ of W^*.
(4) verification of basic and local evidence conditions of W^* based on $\{S_1, \ldots, S_n\}$.

In this paper, we omit explanations of algorithms which (1) and (3) are based on (one can refer to [6] for the algorithm of (3)).

By using the program, we verify a set of real workflows, which is a subset of the set of workflows used to design a real large enterprise application system of AIST. The subset consists of 60 workflows. Each of them has 5∼34 nodes and 5∼31 flows. All workflows are reviewed in a manual way in advance.

The output data of the workflows above by evidenceVerifier are classified into the following types.

(i)	*defects coming from complications of structures of workflows*	**6**
(ii)	*defects coming from inconsistencies of structures of workflows*	**8**
(iii)	defects by trivial mistakes	26
(iv)	superfluous error messages	10

Every execution time is within 0.5 second.

Important defects are those in (i) and (ii).

Defects in (i) come from unexpected flows of activities in workflows. For example, 2 defects in (i) come from a gap coming from changing flows of activities in a workflow in the later phases. It is difficult to find such defects by manual.

Defects in (ii) come from designers' essential misunderstanding on some activities or inconsistent structure of workflow diagrams. In particular, we could find 3 inconsistent flows of activities by the defections in (ii) above. This indicates that, in order to verify construction of a workflow, it is worth to verify consistency of evidence life cycles in it.

Defects in (iii) are defects which come from forgetting adding (+)-marks, forgetting removing (+)-marks, typos of evidence labels or forgetting describing evidences.

The error messages in (iv) consist of 3 messages by designers' informal omission and 7 messages by the relations to other workflow diagrams. As for designers' omission, designers sometimes omit some evidences on purpose. Our program does not corresponds to such omissions. As for the relations to other workflow diagrams, the program has not yet been implemented any function analyzing relationships between multiple workflow diagrams. For example, there is a workflow A which is a successive part of another workflow B, and an evidence E which is created in B and still occurs in A. Then the designer does not add (+)-mark to the first occurrence of E on A. However, our program outputs an error message to that. This is a superfluous error message.

9. Conclusion and future work

In this paper, we have defined a consistency property of evidence life cycles of a workflow diagram in AWL (AIST Workflow Language), and an algorithm EVA (Evidence Verification Algorithm), which verifies the consistency property of each workflow diagram. We also have defined local evidence conditions which are necessary and sufficient conditions for each workflow diagram to have consistent evidence life cycles (Definition 5.5 and Lemma 5.6). Moreover, we have shown that, for each correct workflow diagram W, EVA can determine whether or not W has consistent evidence life cycles (Theorem 4.1).

In Section 7, in order to apply EVA to cyclic workflow diagrams, we introduce an algorithm to translate them into acyclic ones. We first formalize a passback flow, which expresses (a boundary of a main stream and) a reply of an operation in a workflow. We formalize this flow with only graph-theoretical properties of the workflow diagram. We then give an algorithm RAPF, which detects and removes all passback flows in a given workflow diagram, and which translates cyclic workflow diagrams to acyclic ones, which EVA is applicable to.

In Section 8, we experimented with an implementation "evidenceVerifier" of EVA and 60 workflow diagrams of a real large enterprise application system, and have shown that evidenceVerifier could find 38 defects of evidence life cycles of the workflow diagrams and that we could find several defects of structure of the workflow diagrams from the defects of evidence life cycles of them.

As a future work, we are developing the way to verify

[3] Properly speaking, input data express workflows corresponding to workflow diagrams. Moreover, output data have several additional informations.

other kinds of life cycles of workflow diagrams, for example, life cycles of operations and data in databases which are described on activities.

References

[1] Business Process Management Initiative (BPMI). *Business Process Modeling Notation (BPMN) Version 1.0*. Technical report, BPMI.org, 2004.

[2] H. Lin, Z. Zhao, H. Li, and Z. Chen. A novel graph reduction algorithm to identify structural conflicts. In *Proceedings of the 35th Annual Hawaii International Conference on System Science (HICSS)*. IEEE Computer Society Press, 2002.

[3] R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. In *Proceedings of 3rd International Conference on Business Process Management (BPM)*, LNCS 3649, pages 268–284. Springer, 2005.

[4] W. Sadiq and M. E. Orlowska. On correctness issues in conceptual modeling of workflows. In *Proceedings of the 5th European Conference on Information Systems (ECIS)*, pages 943–964, 1997.

[5] W. Sadiq and M. E. Orlowska. Analyzing process models using graph reduction techniques. *Information Systems*, 25(2):117–134, 2000.

[6] O. Takaki, T. Seino, I. Takeuti, N. Izumi, and K. Takahashi. Algorithms verifying phenomena independence and abstracting phenomena subgraphs of UML activity diagrams. In *Software Engineering Saizensen 2007*, pages 153–164. Kindaikagaku-sha (in Japanese), 2007.

[7] O. Takaki, T. Seino, I. Takeuti, N. Izumi, and K. Takahashi. Verification algorithm of evidence life cycles in extended UML activity diagrams. In *Proceedings of The 2nd International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE Computer Society Press, 2007.

[8] O. Takaki, T. Seino, I. Takeuti, N. Izumi, and K. Takahashi. Quality improvement of workflow diagrams based on passback flow consistency. In *Proceedings of the 10th International Conference on Enterprise Information Systems (ICEIS 2008)*, pages 351–359. INSTICC, 2008.

[9] O. Takaki, T. Seino, I. Takeuti, N. Izumi, and K. Takahashi. Workflow diagrams based on evidence life cycles. In *Proceedings of the 8th Joint Conference on Knowledge - Based Software Engineering 2008 (JCKBSE 2008), Frontiers in Artificial Intelligence and Applications*, pages 145–154. IOS Press, 2008.

[10] W. M. P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, LNCS 1248, pages 407–426. Springer, 1997.

[11] W. M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[12] W. M. P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. In *Lectures on Concurrency and Petri Nets*, LNCS 3098, pages 1–65. Springer, 2004.

[13] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek. An alternative way to analyze workflow graphs. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE)*, LNCS 2348, pages 535–552. Springer, 2002.

[14] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44(4):246–279, 2001.

[15] Workflow Management Coalition (WfMC). *Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface - XML Process Definition Language (XPDL)*. (WfMC-TC-1025), Technical report, Workflow Management Coalition, Lighthouse Point, Florida, USA, 2002.

A. Proof of Lemma 5.6

In this subsection, we first show that, for a correct workflow diagram W, if W satisfies local evidence conditions, then W has consistent evidence life cycles, and then we show the converse proposition.

Here we consider only an acyclic workflow diagrams.

Definition A.1 For an acyclic workflow diagram W and a node n in W, the *degree* of n in W denotes the maximum of the lengths of the paths from the start node of W to n. Moreover, the *height* of n in W denotes the maximum of the lengths of the paths from n to some end node of W.

Let W be a correct workflow diagram, ψ a phenomenon of W, A an activity in $W(\psi)$ and let e be an evidence in A.

Now we construct a consistent evidence life cycles F of e on $W(\psi)$

$$F := (A_0 \xrightarrow{L_0} \cdot\cdot \xrightarrow{L_{n-1}} A_n = A = B_0 \xrightarrow{R_0} \cdots \xrightarrow{R_{m-1}} B_m),$$

where $L_n, \ldots, L_0, R_0, \ldots, R_{m-1}$ and R_m are lines.

Claim 1. There is an essentially unique sequence of lines $A_0 \xrightarrow{L_0} \cdot\cdot \xrightarrow{L_{n-1}} A_n = A$ which satisfies the following properties

(i) Every activity A_i has e.

(ii) e is created on A_0.

(iii) e is not created on A_i for any i with $i > 0$.

(iv) e is not removed on A_i for any i with $i < n$.

Proof of Claim 1. We construct $A_0 \xrightarrow{L_0} \cdot\cdot \xrightarrow{L_{n-1}} A_n$, by using induction on the degree of A in W.

(1) If e is created on A, then we set A_0 to be A.

(2) Assume that e is not created on A. Then, by local evidence conditions, there exists an essentially unique line L with target A. Moreover, the source S of L contains e,

which is not removed on S, and the degree of S is less than that of A. Therefore, by induction hypothesis, there is an essentially unique sequence of lines $S_0 \rightarrow \cdots \rightarrow S_k = S$. Thus, we obtain the desired sequence $S_0 \rightarrow \cdots \rightarrow S_k \xrightarrow{L} A$. Moreover, by the definition of equivalence relation on sequences of lines, the desired sequence is essentially unique. □

Claim 2. There is an essentially unique sequence of lines $A = B_0 \xrightarrow{R_0} \cdots \xrightarrow{R_{m-1}} B_m$ which satisfies the following properties

(i) Every activity B_i has e.

(ii) e is removed on B_m.

(iii) e is not created on B_i for any i with $i > 0$.

(iv) e is not removed on B_i for any i with $i < m$.

Proof of Claim 2. One can show the claim in the similar way to the proof of Claim 1, by using induction on the height of A in W. □

By Claims 1 and 2, we obtain an essentially unique consistent life cycle of e in $W(\psi)$. Therefore, we have shown that if W satisfies local evidence conditions, then W has consistent evidence life cycles.

Now we assume that W has consistent evidence life cycles and show that W satisfies local evidence conditions. That is, we show that $W(\psi)$ satisfies the properties (1)∼(3) in Definition 5.5.

(1) Let L be a line in $W(\psi)$ and (L, e) a SIR line-evidence. Then, the target T contains e but e is not created on T. Thus, by consistency of evidence life cycles in W, there exists an essentially unique consistent evidence life cycle

$$A_0 \rightarrow \cdots \rightarrow A_{k-1} \xrightarrow{L_{k-1}} A_k = T \rightarrow \cdots \rightarrow A_n$$

with $k > 0$. Since A_{k-1} contains e but e is not removed on A_{k-1}, (L_{k-1}, e) is a SCS line-evidence. We have the result.

One can show the properties (2) and (3) in the similar way to the proof of (1) above. □

B. Proof of Lemma 6.2

Since it is clear that EVA terminates in finite steps, we show that, for a given correct workflow diagram W, the output of W by EVA is the set of all line-evidences which violate local evidence conditions of W.

We first remark that the division of line-evidences in the step EVA.2.2 in the definition of EVA does not depend on instances, since the state of a line-evidence (L, e) is determined only by the state of E on the source and the target of L.

We now show that every line-evidence which violates the property (1), (2) or (3) in Definition 5.5 is contained in the set $\mathbf{Res}$ at the step (EVA.4) in the definition of EVA. Let $W(\psi)$ be an instance of W and (L, e) a line-evidence in $W(\psi)$.

(1) If (L, e) violates the property (1) in Definition 5.5, (L, e) is assigned SIR, but there exists no SCS line-evidence (L', e) such that L' is contained in $W(\psi)$ and shares the target with L. Thus, if L contains no AND-join, (L, e) is put into $\mathbf{Res}$ in the step (EVA.2.2).(ii). If L contains an AND-join, then (L, e) is put into $\mathbf{Inq}$ in the step (EVA.2.2).(ii), and moved to $\mathbf{Res}$ in the step (EVA.3.2). So, (L, e) is contained in $\mathbf{Res}$ in the step (EVA.4).

(2) If (L, e) violates the property (2) in Definition 5.5, one can obtain the same result in the similar way to (1) above.

(3) If (L, e) violates the property (3) in Definition 5.5, (L, e) is assigned SCS, and there exists another SCS line-evidence (L', e) such that L' is also contained in $W(\psi)$ and shares the target or source with L. Thus, (L, e) and (L', e) are put into $\mathbf{Suc}$ in the step (EVA.2.2).(i), and moved to $\mathbf{Res}$ in the step (EVA.3.3). So, (L, e) is contained in $\mathbf{Res}$ in the step (EVA.4).

We finally show that every line-evidence in $\mathbf{Res}$ violates the property (1), (2) or (3) in Definition 5.5. Let (L, e) be a line-evidence in $\mathbf{Res}$. Then, it is put in $\mathbf{Res}$ in the step (EVA.2.2.(ii)), (EVA.2.2.(iii)), (EVA.3.2) or (EVA.3.3).

(1) If (L, e) is put in $\mathbf{Res}$ in the step (EVA.2.2.(ii)), (L, e) is assigned SIR but L contains no AND-join. Thus, for any instance $W(\psi)$ which contains L, there is no line which shares the same target with L, since L contains no AND-join. So, (L, e) violates the property (1) in Definition 5.5.

(2) If (L, e) is put in $\mathbf{Res}$ in in the step (EVA.2.2.(iii)), one can show that (L, e) violates the property (2) in Definition 5.5 in the similar way to (1) above.

(3) It is clear that every line-evidence which is put into (EVA.3.2) or (EVA.3.3) violates one of the properties in Definition 5.5.

Thus, we have completed the proof of Lemma 6.2. □

A FRAMEWORK FOR THE MODULAR DESIGN AND IMPLEMENTATION OF PROCESS-AWARE APPLICATIONS

Davide Rossi and Elisa Turrini

Dept. of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
I-40127, Bologna, Italy
{rossi | turrini}@cs.unibo.it

ABSTRACT

Process-aware software systems are establishing themselves as prominent examples of distributed software infrastructures. Workflow Management Systems, web service orchestration platforms, Business Process Management support systems are relevant instances of process-aware software systems. These systems, because of their own nature, are all characterized by presenting a behavioral perspective, that is a perspective describing the steps that can be executed during the enactment of the process.

In this paper we present a framework for the design and implementation of the behavioral perspective in modular process-aware architectures. This approach can be applied across different application domains. The modularity of the resulting architectures is well-known as a key factor in achieving software qualities such as reliability, extensibility, robustness, maintainability and ease of use.

Our framework is based on EPML [21][22], an executable process modeling language, and on its enactment engine. EPML has been designed with the aim of promoting separation of concerns, easing the modular approach to the design of process-aware software architectures. A notable advantages of the presented approach is that of using the same modeling language and the same modular software component to support the behavioral perspective across different application domains. As an example we show how it is possible to use EPML and its engine in the modeling and the implementation of a wide spectrum of software architectures, from those supporting business process simulation to those supporting web service orchestration.

Keywords

Process-aware systems, Process modeling, EPML, Software engineering.

1. INTRODUCTION

Process-aware software systems are establishing themselves as prominent examples of coordination-based software infrastructures, following a trend that sees computer systems shifting their focus from data to processes. Workflow Management Systems (WfMSs), web service orchestration platforms, Business Process Management support systems are relevant instances of this class of systems. A modular approach to the design of architectures supporting process-aware systems is that based on perspectives (a concept introduced in [11] and later refined in [13] and [28]). A process can be characterized by different perspectives: the functional, describing what has to be executed; the organizational (or resource), describing who (software component or human) is in charge of the execution; the informational (or data), describing what data has to be processed and the behavioral (or process, or control-flow) describing when part of the process has to be executed during its enactment. In this work we focus on the latter perspective, by presenting a framework to address it, based on an effective separation of concerns. This promotes modularity and allows the design of process-aware architectures able to support a wide spectrum of applications. Our approach is based on EPML, a graphical, executable, process modeling language and on its enactment engine. As a result the very same notation can be used to model the process perspective in workflow systems, business process simulation systems, web service orchestration systems, process-aware web applications and other process-aware software systems. Moreover, the software architectures supporting these systems can be designed in a modular fashion, composing together application-domain specific components and the EPML engine.

This paper is structured as follows: section 2 outlines the main issues we tried to address in designing our proposal; section 3 introduces EPML and its enactment engine. The sections 4, 5, 6 and 7 show sample methods and architectures to design and implement process-aware applications

using our modular EPML-based framework. Section 8 discusses our proposal with respect to related work. Section 9 concludes the paper.

2. COORDINATION FOR PROCESS-AWARE APPLICATIONS

The Babel of business process notations, workflow and web service orchestration languages keeps growing day by day. Most of these languages/notations have not been designed to provide improvements in the description of the dynamic behavior of process-aware applications but, rather, to address some domain-specific issue (like supporting web services invocation to coordinate an orchestration or managing resources in a workflow). This forces the members of the development team to learn different tools to address the same aspect: the behavioral perspective. Moreover, it may well be the case that in a single, large application, coordinating users connected to different software systems and remote components, the behavioral perspective has to be addressed with different languages/notations at the design level and with different enactment engines at the implementation level.

A partial solution to this problem is provided by BPMN [16]. BPMN is a graphic notation to model business processes that has not an associated formal semantics and that does not produce executable specifications. A BPMN diagram could, however, be transformed into different executable notation. BPMN's specifications, for example, suggest a mapping to transform a diagram into BPEL. From a conceptual point of view the same approach can be taken with respect to different "target" notations, providing a unified high-level modeling tool. This would imply that, at least at the design level, a single modeling notation can be used to be later transformed into one or more executable ones. As discussed in section 4, however, this solution suffers several limitations. It is our opinion then that, while BPMN is a highly valuable contribution for high level process modeling in early phases of the software development process and for documentation purposes, an executable notation (and possibly a single one used within one process) has clear advantages in the design-related activities.

These are the reasons that lead us to the development of a framework to address the behavioral perspective in different application domains using the same tools. This approach is based on EPML, an exogenous [7] coordination language that has been designed in order to obtain an effective separation of concerns, easing the modular approach to the design on process-aware software architectures. It should be noticed that EPML defines the interactions that can take place between the actors in the system, but cannot change them at run time like, for example, Manifold [8] and other control-driven [18] coordination languages can do. Process modeling languages, in fact, can be instances of different coordination models (control-driven, data-driven, space-based, rule-based and so on) and, while EPML's characteristics are shared with most existing workflow languages and process modeling languages (that are *flow languages* in the broad sense of languages that describe the process in term of, potentially concurrent, flows of executions, their interactions and their synchronizations) there are contexts in which different paradigms are more suitable and should be preferred.

EPLM is a graphical, executable language with a formal semantics and high level of expressiveness. It is our opinion that these characteristics are essential in this kind of tools. While some of the reasons for this are rather obvious (a diagram is easier to understand than a sequence of text lines) some are less immediately apparent. As an example of this consider the following issue: WSBPEL [6] is a textual (XML-based) web service orchestration language that has no graphical representation but most of the available tools that support WSBPEL development provide a graphical modeler based on a proprietary notation. The diagrams produced with these tools become artifacts of the software development process introducing non-standard notations in the process and causing potential vendor lock-in problems.

Details about EPML and its engine can be found in section 3; in this section we focus on how a process modeling language can be designed to maximize separation of concerns. Separation of concerns is a well-known topic in software engineering in general and has specific relevance in the research area related to coordination models and languages. Separation (orthogonality) between coordination and computation is a cornerstone for this area [12]. In a coordinated process the computation is carried out by software components or human beings (*actors*) participating in the process; the coordination is carried out by a coordination runtime. The distinction between coordination and computation however is, in our opinion, too coarse. Coordination should really be split in *interaction model* and *process logic*. The interaction model defines the execution flows, i.e. the possible interactions among the involved entities. The process logic defines which, among all possible execution flows, have to be activated. To show the relevance of the process logic and the fact that its importance is often neglected we use BPEL again as a paradigmatic example (in this article we use BPEL to refer to either BPEL4WS [3] or WSBPEL, the two versions of the language). BPEL uses XPath expressions as predicates to support control flow decisions; there are cases in which, however, these decisions imply a computational effort for which XPath is not well suited. In these cases the decisions are delegated to activities that have to be created ad hoc for this task; as a consequence two different perspectives get mixed together (besides the impact on software qualities such a solution also implies that new web services have to be created and

hosted somewhere in order to support the behavioral perspective). Implicitly acknowledging this limitation all existing commercial BPEL engine implementations we are aware of, include custom extensions that allow to activate software components implementing the process logic. The latest version of BPEL, WSBPEL, also provides a standardized extension mechanism for this. For the same reason, BEA and IBM (some of the partners that supported the development of BPEL) proposed an extension of BPEL4WS: BPEL-J [2]. In BPEL-J, Java code *snipets* can be embedded in a BPEL specification reducing the need to delegate to external components. The main drawback of this approach is that the process logic can be expressed using a specific language only. This implies that only users proficient in Java can benefit from BPEL-J; moreover, from a technological point of view, workflow engines must be able to activate Java programs.

To better support separation of concerns, in EPML the computation, the interaction model and the process logic are addressed by distinct elements, that is, respectively, *activities*, *processors* and *processors' logic*. The first two are (graphical) elements of the language; processors' logic is a program fragment (expressed with any suitable language) supporting the decisions associated to processors. This clear distinction between these three aspects is propaedeutical in promoting a high level of modularity allowing us to use the framework based on EPML to support the behavioral perspective in several different application domains as we show in this paper.

Another aspect which is essential for the wide applicability of our framework is related to the expressive power of EPML. The expressive power of workflow languages has been subject to several investigations in the last few years, these studies can be (and have been) applied to process modeling languages as well. The main problem addressed in this research field is the fact that there is not a formal metric to evaluate the expressive power. One of the most successful approaches is the one based on the workflow patterns [5]; this is an analysis strategy that evaluates the languages with respect to their ability to model a set of predefined (sub)processes. Expressive power is a critical parameter to increase the suitability of a process modeling language (for the quite obvious fact that a language that cannot easily model the interactions within a process surely poses large limitations to its own usage). Most existing workflow languages, for example, cannot easily model a large number of common real-world interactions. A workflow pattern-based analysis shows that EPML supports all the 20 patterns described in [5] (with the partial support of implicit termination because of a design decision); an extended pattern set has been presented in [25], EPML support most of these 42 patterns with minor exceptions (like interleaved parallel routing and critical section); extending the semantics of the language to capture all patterns is possible but that would make the language itself more complex so, at this time, we decided to not support them. In the design of EPML, in fact, we addressed the workflow patterns but we also strive to achieve a good compromise between simplicity and power.

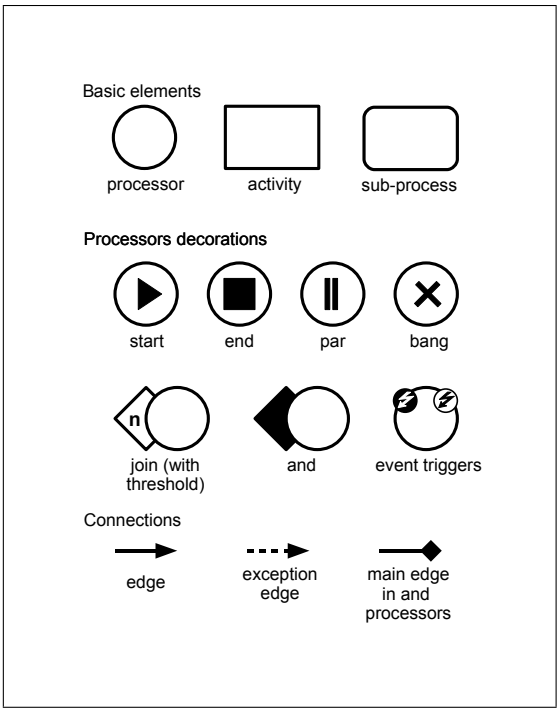


Fig. 1. The components of a EPML diagram

3. EPML

EPML is a graphical process modeling language that enables the representation of a process interaction model using a directed graph in which oriented edges are used to define the execution flow structure.

In this paper we call EPML *specification* a process modeled with EPML. The specification can be a *diagram* or an XML document. It is possible to translate an EPML diagram in its XML representation and vice versa. A process specification (in form of XML document) can be executed by an *engine*; we refer to a specification in execution as a *process instance*.

The purpose of this section is not to provide a detailed description of EPML (interested readers can refers to [21]) but only to briefly introduce its main features.

The components of EPML are shown in Fig 1.

Two types of nodes exist in EPML: *activities* (represented with squares) and *processors* (represented with circles); subprocesses (represented with rounded rectangles) really are just folding of subgraph with specific characteris-

tics. Activities are elements of computation: they can be either external applications or work items allocated to a workflow participant (possibly a human actor). An activity can have an entry edge, an exit edge, and, potentially, an exception edge. Processors are elements of coordination: they implement the process logic possibly using a standard programming language. A processor is like a gateway: it can perform synchronizations and uses the process logic to perform routing decisions; as such it can have multiple entry edges and multiple exit edges.

Each execution flow is represented by a *token*. A token contains the *data* associated to the flow and produced and/or used by activities and processors. Moreover tokens also contain identifiers that allow to implement flows synchronization. Similarly to other formalisms (e.g. place-transition networks), the set of tokens present in a given moment, and their location in the diagram, represent the state of the system.

Each time a flow (i.e. a token) reaches a node, and its associated entry condition, if present, has been satisfied, the node *activates* (a new *node instance* is created). Concurrent activations of the same node are possible inside the same process.

In EPML, two types of edges exist : the *standard edges* and the *exception edges*. The latter are activated only in the case of anomalies (an exception raised by an activity or by an unsatisfiable join associated to a processor). Activating an edge means sending a flow over it, (i.e. putting the token representing the flow into the edge's destination node).

A processor can be decorated with the decorations shown in Fig. 1; the entry decorations (and, join, join-with-threshold) and the exit decorations (bang, par) can be used for representing control-flow operations like join, split and sub-flow creation.

In the following paragraphs we briefly describe the semantics of every decoration that can be associated to the processor nodes. The decoration(s) a processor supports characterize(s) the type, and then the behavior, of the processor itself.

The *start* processor and the *end* processor (obtained decorating a processor with a *start* and *end* decoration, respectively) represent the start point and the end point of a process.

A processor with no decorations is a *simple processor*. It manipulates the data received by the previous node(s) in order to select the exit edge on which the execution flow will be routed.

A processor with a bang decoration (*bang processor*) enables to split a single process execution flow in parallel flows, and route them on one or more exit edges. The exit edges of a par processor can be labeled with a cardinality notation (à la UML) indicating the minimum and/or maximum number of process flows that can be routed in parallel

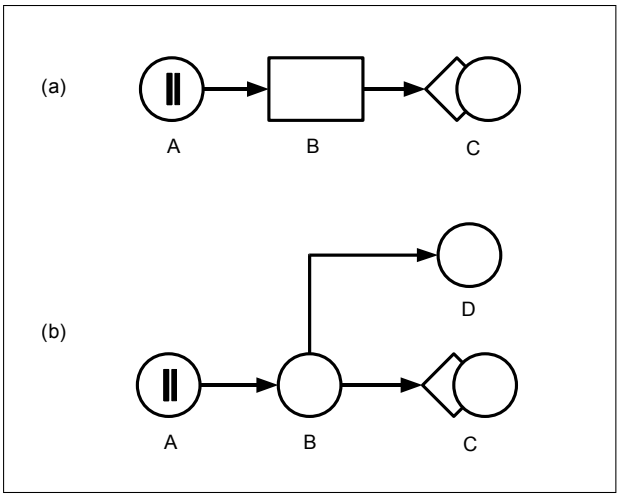


Fig. 2. Examples of how par and join processors can be composed

on the edge(s). If such a notation is not present, no limitation about the number of flows is imposed.

A processor with a par decoration (*par processor*) has a behavior which is similar to that of a bang processor, with the difference that the flows produced by this processors are sub-flows of the entry flow, that can be later synchronized. Technically this is accomplished by extending the token's identifier (which is a sequence) with new unique elements shared among all the related sub-flows.

The goal of a processor decorated with a join decoration (*join processor*) is to synchronize process flows (generated by one or more par processors) that are executing in parallel. The join semantics in EPML is quite sophisticated and it significantly contributes to make powerful the EPML expressiveness. For this reason, we present the peculiarities of the join semantics and discuss some examples. Intuitively, we can say that a join processor activates when all flows produced by the same instance of a par processor reach it, or when at least one flow is arrived and the other flows produced by the same par processor instance cannot reach it any more (for example because they have been canceled or they have been routed elsewhere). The join processor removes from the flow(s) the identifiers that have been used to implement the synchronization and merges the sub-flows in a unique flow.

The first example we discuss is shown in Fig. 2*a*. In this graph, when the flow reaches the par processor *A*, an instance of it is created and the associated process logic is executed. Let us suppose that this instance generates two flows marked with the same identifier, e.g. A_1. Whenever one of these flows reaches the activity *B*, a new activity instance is created. When an activity instance completes, the flow is routed on the exit edge (notice that the termination

order of the activities instances can be different from the activation order). The processor *C* activates when it receives both flows marked with A_1 (that is when both executions of B complete).

It is important to remark that the synchronization operated by the join processor is related to the sub-flows generated by the same par processor instance. For example, let us suppose the par processor *A* activates twice; the first time it produces two flows (with identifiers A_1), while the second time it produces three flows (with identifiers A_2). In this case the join processor activates both when it receives the two flows marked with A_1 and when it receives the three flows marked with A_2 (again, notice that these flow can interleave in different ways).

We would like to point out that, according to the join semantics, a join decoration is satisfied even in the case at least one flow has reached the join processor and the other flows produced by the same par processor instance cannot reach the join processor any more. Cases like that can occur, for example, in a process fragment like the one depicted in Fig. 2*b*. Let us suppose that the par processor *A* generates two flows marked with the same identifier, e.g. A_1. Both flows arrive in the simple processor *B*, then it activates twice. Let us suppose that one of the flow is then redirect to the join processor *C*. This processor receives the flow but it cannot activate, since another flow marked with A_1 (that at the moment is in *B*) can reach it. In this situation *B* could:

- route the flow on the edge that conduces to the processor *D*;
- route the flow on the edge that conduces to the join processor *C*;

In the first case, the join processor *C* can activate as soon as the flow is sent on the edge; In the second case the join processor *C* can activate as soon as it receives the flow. In both cases, indeed, the activation of the join processor *C* is possible as no other flows generated by the same par processor instance can reach the processor anymore.

In a EPML graph, nodes can be connected in a free structure; this implies that, in general, there is not a 1-to-1 relation between a par processor and a join processor. A join processor can then synchronize flows coming from different par processors. To this end, a join processor maintains a list of par processors it refers to (we named this list *par set*) and activates only when all process flows generated by the par processor present in the par set reach the join processor. The par sets are calculated by means of an algorithm based on a network coloring mechanism (for sake of conciseness the algorithm is not described in this paper). This algorithm runs before the process specification starts to execute; the par set of a node is the same for every process instance and does not vary during the process execution. The rationale of this algorithm is that the par set of each join processor must

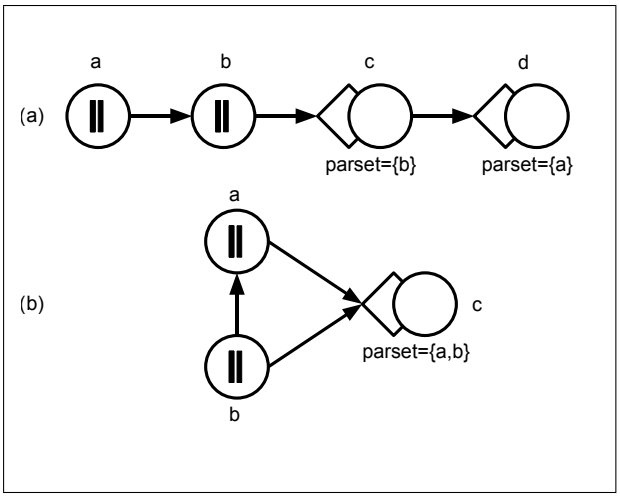


Fig. 3. Examples of structured and unstructured par-join composition

contain all par processors that are directly connected to it (in the sense that there is a path between the two with no other join processor in the between) or indirectly connected (in the sense that there is a path between the two in which each par processor has a complementary join processor later in the path).

In Fig. 3 are shown two examples of structured and unstructured par-join composition. In Fig. 3*a* the parset of the join processor *C* contains the par processor *B*, and the parset of the join processor *D* contains the par processor *A*. This means that *C* synchronizes the flows produced by *B* while *D* synchronizes the flows produced by *A*. In Fig. 3*b* the par set of the join processor *C* contains both the par processors *A* and *B*. This means that *C* synchronizes the flows produced by both *A* and *B*.

A process decorated with a *join-with-threshold* decoration (*join-with-threshold processor*) is a special type of join processor. The threshold (i.e. the number n written inside the decoration) indicates the number of process flows the join processor has to wait before activating. The threshold can be positive, negative or zero. If it is positive, the processor waits for n flows; otherwise it waits for the number of generated flows minus $|n|$. Note that, if the threshold is zero, the joinWT processor waits for all the flows produced by the par processor instance. When the joinWT decoration can not be satisfied (i.e. no other flow can reach the joinWT processor), the joinWT does not activate and the flow is routed on the exception edge (if present).

A processor decorated with an *and* decoration (*and processor*) implements a different kind of synchronization. The and processor must have a main entry edge and can activate only when the following conditions are satisfied: (1) flows must arrive in all its entry edges; (2) incoming flows must

be sub-flows of the flow coming from the main edge. Only the flow coming from the main edge is driven forward, the other flows are simply discarded.

A processor decorated with an *event* decoration (*event processor*) activates only if the entry decoration (if present) has been satisfied and an event is arrived. Events can be permanent or transient. Permanent event can be stored and used subsequently; on the contrary transient events are lost if the processor can not be activated at the moment the event arrived.

EPML supports *subprocesses*. A subprocess is just folding of subgraph with specific characteristics. When an execution flow enters in a subprocess, a new subprocess instance is created; the entering flow is considered as a new flow, that is all identifiers associated to that flow are ignored (they will return valid only when the flow exits from the subprocess). Processors inside a subprocess can add identifiers to the flow, however those identifiers are valid only inside the subprocess instance that created them, and are canceled when the flow exits from the subprocess.

EPML has a cancellation construct that enables to define a *cancellation area*. A cancellation area is a set of nodes and can be graphically depicted with a dashed perimeter connected to a node. When the node is activated, the flows and the node instances inside the cancellation area are removed or forced to terminate, respectively (in general, the flow that activates the cancellation and the canceled flows have to be generated by a common par processor instance).

EPML also makes available language elements that are just syntactic sugar, this is the case for the messaging (or asynchronous) activities and for timed activities and subprocesses. Messaging activities are depicted like activities decorated with the same glyph used to represent events in event triggers. Despite their graphical appearance, these really are processors whose process logic generates events (by interfacing with the enactment engine run-time system). When these events are used to communicate with components that have to be coordinated the usage of an activity-like element remarks the interaction with the functional perspective. A timed activity is an activity for which a deadline is set as soon as its execution starts. If the execution does not end before the deadline the activity is canceled. An exception edge can exit from a timed activities; when a time-out occurs the entry flow is routed on this edge. Being syntactic sugar a timed activity is internally expanded into regular EPML elements that support the aforementioned semantics. Timed sub-processes behave in a similar fashion.

As hinted above, EPML also includes elements related to the informational perspective. In this perspective we find two classes of data: production data and control data. Production data comprise all data that are essential for an application area. Control data can be either information related to the internal state of a process or pointers to production data (allowing process elements to access the information that has to be processed). In EPML a *data bag* (an associative map) associated to each token is used to manage the control data. By using special shared and transient entries, EPML provides a simple way to manage global data (that is data that has to be shared among all the processes hosted by an engine); process data (data that has to be shared by all the instances of a specific process); instance data (data that has to be shared within a single process or subprocess); and transient data (data that is automatically removed after the execution of an activity). For example process data has to be set (and retrieved) using the `$process` entry automatically added to the data bag of all generated tokens. This entry is a new associative map shared among all the instances of a specific process.

A formal semantics for EPML, based on a transition system, is described in [21] (although it is not updated to the latest version of the language). A formal semantics can be the starting point for the formal verification of specific properties (reachability, liveness, etc.). It is our opinion, however, that this argument is of less importance with respect to the ability to define the behavior of the language with no ambiguities. Anyone that had the chance of working with these languages to model complex processes went through the "try and see what happens" approach: when interactions get complex, the only way to be sure about the actual behavior of the system is to enact the process in a testing environment (and, quite frequently, the observed behavior is not the one expected by reading the manuals).

3.1. The EPML engine

A process modeled with EPML can be enacted by means of a software component called EPML engine; it takes as input an XML representation of the EPML specification and it executes it. The EPML engine is a software component written in Java 1.5 (and thus portable to most platforms). It has been designed as an event-based architecture: it consumes events and produces both events and state transitions. An event can represent an external event, the termination or the activation of a node instance. A state transition can entail modifying tokens or moving them in the network.

In many situations, the engine is expected to interact with human actors and/or software components. This implies the integration with a software architecture designed for a specific application domain. This integration has been implemented managing the events that the engine produces (*output events*) and generating the events that the engine consumes (*input events*). The input events can be *start events*, *external events*, and *end-activity events*. The start event triggers the execution of a new process instance; the external events are those caught by an event processor and typically are generated by the environment in which the process is enacted, and the end-activity events notify to the engine that an

activity has terminated its execution. The output events can be *termination-process events* or *start-activity event*. They notify the external component that the process is terminated or that a new activity instance has been created and can start its execution.

In our system, the event notification is achieved exploiting the *implicit invocation* principle. To notify an input event to the engine the software architecture invokes an engine method that adds the event to its input-event queue; it will be then examined and proper actions (e.g. processor activation) will be taken. To notify an exit event to an external component the engine invokes a method on it.

The activities are not part of the engine, but they are external components that interact with the engine by means of a Java class that extends an `ActivityWrapper`. This class has a method (named `run`) that is invoked by the engine to start the activity execution. The run method contains the code that enacts the activity execution. Such execution can imply, for example, either the invocation of an external software component or the addition of a task into an actor task list. When the activity has terminated its execution, the `ActivityWrapper` produces a termination-activity event and passes it to the engine.

The architecture just described is very flexible as it enables the interaction among most kinds of external components. The engine integration in an existing architecture does not require any modification in the engine itself, but can be achieved by extending the `ActivityWrapper` class and putting into it the code that enables the interaction with the external component.

The activities execution is coordinated by the processors and specifically by the process logic. Technically speaking, the process logic can manipulate both production and control data. In some context, the process logic could also be required to decide which actor(s) the activity(-ies) has(have) to be assigned to. Such feature is typically related to the *resource perspective* in a workflow system. It can be integrated in EPML in a way transparent to the engine, that is specifying, in the token's data bag, which actor is in charge of executing the activity.

The process logic has to be specified by means of a suitable formalism. This could be a generic programming language, a scripting language or a language based on XML (e.g. XQuery or XSLT), provided that specific adapters implementing proper bindings are provided to interface to the engine. It is also possible to specify the name of a Java class implementing the process logic. The engine makes available to the process logic mechanisms for manipulating the tokens' data bags; the process logic has to inform the engine about which exit edge(s) has(have) to be activated.

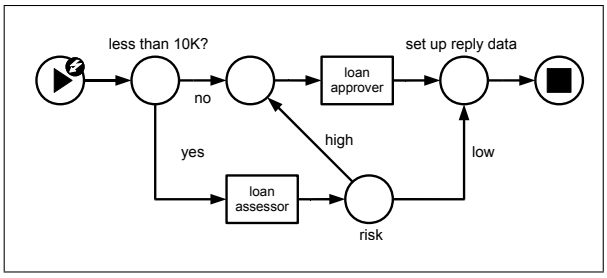


Fig. 4. A loan process

4. EPML.WS

EPML.WS is a software architecture for web services orchestration based on EPML. In order to support this specific applications class EPML.WS complements EPML with components that address the interaction with synchronous and asynchronous web services and allow a EPML-enacted process to be accessed as a web service.

EPML.WS is designed to be hosted inside a JEE architecture supporting JSR 109 (Implementing Enterprise Web services) and JSR 181 (Web Services Metadata for the Java Platform).

A very thin layer of software adapters have been provided in order to create the EPML.WS architecture. This includes:

- an adapter to transform incoming web services invocations (SOAP messages) into external events that are feed to the EPML engine;
- an activity wrapper (`ActivityWSWrapper`) to implement activities execution as web services invocation;
- a process logic wrapper (`WSXQueryLogic`) to use XQuery for routing decisions and for process data management.

Web services that have to be orchestrated are mapped into activities and the data perspective is managed using the simple integrated data perspective of EPML (that is using the data bags associated to the tokens). As usual, with EPML, the logic of the processor can be implemented with several languages. Given the fact that the data model used by web services is based on XML we provided a specific process logic adapter that allows to use XQuery (`WSXQueryLogic`). This adapter first creates the XML document against which the provided XQuery expression has to be run, it executes the XQuery expression and then parses the generated output to activate one or more exit edges and to modify the token's data bag (used to store process-related data and parameters for subsequent web service invocations).

We now use an example to give a basic idea of how to set up a web services orchestration with EPML.WS.

Consider the process depicted in Fig. 4. The activities `loan approver` and `loan assessor` are external web services that participate in a basic loan approval process. The process is initiated with the event associated to the start processor; in our case this event has been mapped to an incoming web service invocation by using the aforementioned adapter. The `less than 10k` processor has two duties: implement a routing decision (activating the outgoing `yes` or `no` edges) and prepare the parameters needed in the invocation of the two subsequent web services (associated to the `loan assessor` and the `loan approver` activities). Specific entries of the tokens' data bag are used to store incoming SOAP request messages and to set up invocation parameters. `less than 10k` has then to access the data associated to the SOAP request received at the beginning of the process to implement the routing decision and has to prepare the parameters for the forthcoming invocations. `less than 10k` has been set up to use `WSXQueryLogic` and it uses the same XML document associated to the original incoming SOAP request for the XQuery (element properties in the XML process description are used to specify how the XML document has to be produced from specific entries in the token's data bag; in this case a special `$wsin` elements containing the SOAP message of the incoming call). The XQuery expression, in this case, is as follows:

```
<xq_result>
  <edges>
  {
    if(data(//amount) > 10000)
    then <edge>yes</edge>
    else
      <edge>no</edge>
      <wsout>
        <param name="name">
          {data(//name)}</param>
        <param name="first name">
          {data(//firstName)}</param>
        <param name="amount">
          {data(//amount)}</param>
    </wsout>
  }
  </edges>
  <data>
    <entry name="name">
      {data(//name)}</entry>
    <entry name="first name">
      {data(//firstName)}</entry>
    <entry name="amount">
      {data(//amount)}</entry>
  </data>
</xq_result>
```

Once the query has been executed `WSXQueryLogic` parses the result in order to extract the information about the outgoing edge that has to be activated (this information is in an `edge` element), stores in the token's data bag all the name-value pair contained in `data` and sets up a special entry, `$wsout` in the aforementioned data bag (that is used to set up the parameters of the invocation of either `load approver` or `loan assessor`), by analyzing the `wsout` element.

If we assume that the body of the original SOAP request is as follows (namespace references have been omitted for clarity):

```
<body>
  <firstName>John</firstName>
  <name>Doe</name>
  <amount>1000</amount>
</body>
```

the previous XQuery expression produces:

```
<xqresult>
  <edges>
    <edge>no</edge>
  </edges>
  <data>
    <entry name="name">Doe</entry>
    <entry name="first name">John<entry>
    <entry name="amount">1000</entry>
  <data>
  <wsout>
    <param name="name">Doe</param>
    <param name="first name">John<param>
    <param name="amount">1000</param>
  </wsout>
</xqresult>
```

The information in this XML fragment is then processed as described before, activating the `no` outgoing edge and putting the name, first name and amount data both in the data bag (these are used later by the `reply` activity) and in the special entry used to set up the invocation parameters for `loan approver`. Before the process completes, the `set up reply data` processor, linked to the end processor, sets up (in the data bag) a special entry that is used by the web service adapter to create a reply message to the original process invocation.

EPML.WS poses itself as an alternative to systems based on WSBPEL. With respect to these systems EPML.WS has the following advantages:

- graphical notation;
- higher expressive power;
- ability to implement complex process logics without relying on external components.

It should be noticed that WSBPEL has not a "native" graphical notation but there are specific guidelines about the

usage of BPMN to model BPEL processes. From a practical point of view, however, the mapping of a process modeling language into another is a very complex issue. The details for the mapping of BPMN into BPEL, for example, are largely incomplete and it is well possible to create a BPMN diagram for which the translation into BPEL is undetermined. Even when using techniques which are more advanced with respect to those presented in BPMN's specifications, the resulting BPEL code (when it possible to obtain it) is often a bad example of "spaghetti BPEL", which does not only pose a readability problem, it also creates monitoring problems (at which conceptual point of the process is the program executing this piece of spaghetti BPEL?) and modification tracking problem (modifications in the resulting BPEL code can hardly be reported into the BPMN diagram). For an in-depth analysis of this problem the interested reader can refer to [17]. These kind of problems are not only limited to the BPMN to BPEL mapping, but are usually found whenever a transformation between two process modeling languages have to be performed. It is easy to realize this is the case considering that most languages have different expressiveness (as defined in [5]), so most of the times a transformation has to map interaction patterns that are easily supported in the original notation and not easily (if at all) available in the target one.

To test EPML.WS against BPEL we set up a simple experiment. We used the very same process described above (which is a sample from the ActiveBPEL [1] distribution) we "unplugged" the BPEL engine and we replaced it with EPML.WS obtaining a working orchestrated process presenting the very same behavior and the very same (web service-based) interface. Another interesting part of this experiment was the analysis the effort needed to setup a BPEL-based solution and the EPML.WS one. While we did not run formal tests to access a vague parameter like "effort" our experience with computer science students shows that the same task (modeling a process of average complexity) can be accomplished using EPML.WS in about one tenth of the time with respect to BPEL, factoring out the time needed to address the link relationship problems (that are addressed in a very complex, yet powerful, way by BPEL and are not addressed at this time by EPML.WS).

EPML.WS has been mainly designed as a proof-of-concept. A working prototype has been implemented but it still has a few limitations (for example bindings have to be programmed by hand). Nevertheless EPML.WS is the proof that it is possible (and relatively easy) to design, and implement, a software architecture for a specific application domain in which advanced coordination mechanisms are required to govern interactions among distributed actors.

5. EGO

EGO (E-Game Orchestration) [20] is a software platform to deliver e-learning games based on EPML. EGO allows multiple users to be engaged in collaborative or competitive games by using a web-based interface. With EGO, games with various interaction patterns among the actors can be modeled, such as the ones occurring in turn-based and concurrent games (with or without synchronization steps). Given the large amount of possible interactions that can take place among actors, games are good candidates as case studies to test coordination models. One of the basic concepts in EGO's game modeling is the interface (in the sense of user interface). In EGO an interface is an activity assigned to an actor. The idea is that a game can be assimilated to a workflow system in which the interaction of players with their gaming interface corresponds to the execution of work items assigned to actors. By making their moves (using the interface) the players accomplish the work items. One of the main differences of this system with respect to classical workflow systems is that a single interface is usually assigned to the actors (otherwise a player might have to interact with multiple user interfaces to participate in the game). This problem can be solved with two different approaches. The first one is to explicitly cancel the activities that have been assigned (and not accomplished) to an actor right before assigning a new one. This solution entails no modification to the semantics of standard workflow languages but the specification becomes soon cluttered with cancellations. A second solution, the one we adopted, is letting the upper layer in the software architecture to take care of notifying the engine that the an actor finishes a previously assigned work item whenever a new work item is assigned to him by the engine. Please notice that while this solution can be perceived as a violation of the semantics of EPML this is not actually the case. The semantics that is violated is that of a workflow system, something that EPML is not. In EPML, in fact, the resource perspective is out of scope, since the focus is in the process perspective. Tokens' data bags can be used to transfer information related to the resource perspective, but this concept is not intrinsic to EPML.

As stated above EGO is a web-based platform and it has been designed within the JEE framework. Its structure is quite simple: the engine interacts with a web application hosted in a JEE servlet container. The interaction takes place by means of input and output events. The web application is composed by two servlets (`Dispatcher` and `Process`) and a software component that captures the start-activity events produced by the engine and maintains the association between actors and activities. The `Dispatcher` servlet queries this software component in order to obtain the activity that has to be assigned to an actor and then dispatches the associated interface to the player. When a player

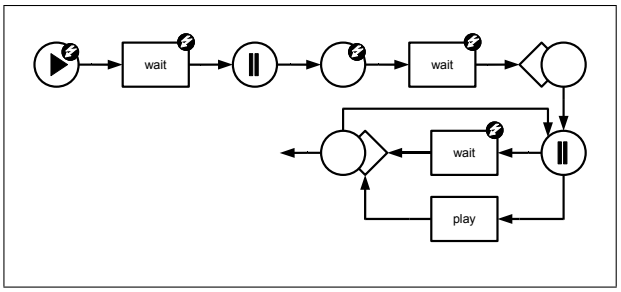


Fig. 5. The beginning of a simple turn-based game

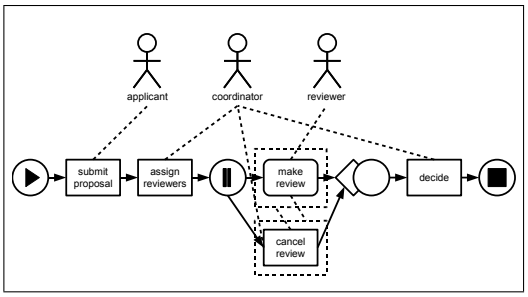


Fig. 6. A review process in EPML

submits its move, the move is processed by the `Process` servlet that produces an end-activity event embedding into it the data coming from the request sent by the player's browser. The event produced is then notified to the engine.

In Fig. 5 the first steps of a turn-based games are modeled. One player have a master role and is in charge of setting up the game environment for the other players. This player starts the game sending a message that triggers the creation of a new process instance. Other players join by sending a message that triggers the `new player` processor. The `wait` asynchronous activities are used to notify the players they have to wait for the other players to join before the game can proceed. Asynchronous activities are used to dispatch web pages for which no user input has to be reported back to the process. Once all the players have joined the game the turns can begin. During the turns one player is associated to the `play` activity (which is not an asynchronous activity, since the result of the play has to be reported to the process) while all the others players receive the `wait` interface. Once the play has been performed a new turn can start or the game can proceed with other phases.

EGO has been used to model several games and it is being currently used for e-learning purposes (with business simulation games). We also developed a version that is able to interact with AJAX-based presentation technologies.

6. PROCESS-AWARE WEB APPLICATIONS WITH EPML.WEB

EPML.WEB [23][24] is a platform (a model and a reference architecture) for the design and development of process-aware web applications based on EPML. The use of EPML in the context of process-aware web applications is motivated by the consideration that such applications should be able to support not just simple navigation and data access activities but (potentially complex) business processes. In this respect, we propose to model the business process with EPML and extend the engine architecture with software components that interact with the web navigational structure of the web application.

While navigating in the web application, users can access pages that are not related to any process or can visit process-related pages. We refer to the former as *standard navigation mode* and to the latter as *process flow mode*. To enter process flow mode the users follow specific *process-aware* hyperlinks. Process-aware hyperlinks can be used to create a (sub)process, to resume a previously started process that has been left (probably by using navigation links that drove the user outside the process flow), or to join an existing process created by another actor. In process flow mode, the sequence of the pages that are dispatched to the users may depend on the control flow of the process rather than on the navigation structure of the web application. Specifically, in our approach, the pages in process flow mode correspond to the tasks assigned to the users by the process.

Consider the process depicted in Fig. 6: it is a high level model of a simple project grant review process. In this diagram, standard EPML elements have been enriched with stick figures in order to model an elementary resource perspective. Applicants submit their projects for review, the coordinator assigns the actual reviews to a given number of reviewers, the reviewers make the reviews. A review is actually composed by two steps: a first evaluation is given considering an anonymous subset of the documents in the proposal, a second evaluation is given considering all the information related to the project, including the applicants' identity. While waiting for the reviewers to complete their work, the coordinator can decide to cancel a review (because it is delaying the process or for other reasons). When all the reviews have either been completed or canceled, the coordinator decides to reject or to fund the project. Please notice that the modeled process is a highly simplified version of what actually takes place in the real word. EPML has been designed with an high expressive power right because the authors acknowledge that real world (business) processes are far more complex than what academic papers seem to suggest. Nevertheless, given the focus of this work, an oversimplified example is reasonable.

To decide if a proposal should be funded or rejected, coordinators have to join the flow of the process generated by

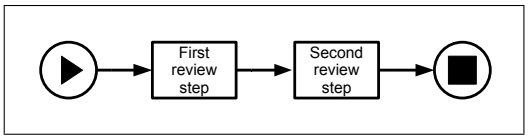


Fig. 7. The make review sub-process

the applicants when they submitted a proposal. In order to do that from within the web application, coordinators have to follow a process-aware hyperlink that drives them to a page related to the `decide` activity. This page should be available only when the `decide` activity has been assigned to the coordinator and it is a reasonable design strategy to prevent the generation of process-aware hyperlinks related to processes for which no activity is assigned to the current user. In the specific case of the aforementioned example, the `decide` hyperlink should be available from the proposal details page only when all the reviews have either been received or canceled. In general only process-aware hyperlinks used to start a new process can be always available. Even hyperlinks used to start sub-processes are not generally available (in the example, the make review sub-process can be started by reviewers only when they have been assigned the review of a proposal).

Consider now Fig. 7: it depicts the make review sub-process. As discussed above this sub-process is composed by two steps and it is very well possible that reviewers leave the sub process when finished the first step, before completing the second. As a consequence the `make review` hyperlink that is available from the pending reviews page can lead reviewers to the forms associated to the first or to the second review step depending on the state of the make review sub-process for that specific proposal.

Note that the web application framework must be able to interface to the process enactment engine in order to query the activities assigned to specific users and to signal the completion of activities. From the modeling point of view, the following issues have to be addressed:

- the modeling of process-aware hyperlinks;
- the modeling of non-navigational sequences of views in process-flow mode and
- the modeling of associations between the pages in the navigation model and the related activities in the process model.

In Fig. 8 is depicted a model for the make review (sub) process that uses a simple extension of a WAE [10] navigation diagram (a stereotyped UML class diagram that is part of the user experience model) and a EPML diagram. We decided to use WAE for mainly two reasons: first, to remain agnostics with respect to other web applications development methods like [9, 15, 26, 27] that have been extended

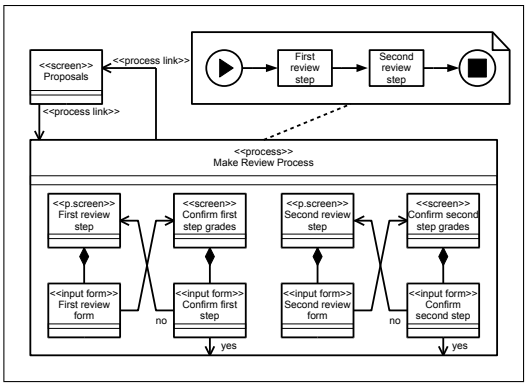


Fig. 8. Modeling the review sub-process

in order to model process-aware applications and, second, because it is a quite "lightweight" method that easily allows simple extensions. This is mostly because WAE is not targeted at model driven development and, thus, the design models are not overloaded with details.

From this example it is easy to see how the aforementioned issues can be addressed. Process-aware hyperlinks are modeled with navigational associations that connects to a *process*-stereotyped class. In order to remark the specific behavior of these associations we also used the *process link* stereotype, but it is not really essential. The *p.screen* stereotype is used (for inner classes inside a *process*-stereotyped class) to mark the entry points of sub-sequences related to an action in the EPML model. The active sub-sequence can be easily inferred since the name of the *p.screen*-stereotypes classes correspond to the names of the action in the process model. When a navigational subsequence in process-flow mode returns the control to the process-flow, a navigation connection is made between the last screen (or its aggregated forms) and the outer process class that contains it (as in the two input confirmation forms of the example). Process-stereotyped classes admit only one exit association that leads to the subsequent page (or process) that is visited when the (sub)process ends. In our example this association too is process link-stereotyped.

As far as implementation strategies are concerned, several options are available, depending on the used web application framework. The solution we present here is based on Java EE (which is a natural choice, since the EPML engine is written in Java) and the Struts MVC framework; similar solutions, however, can be implemented with different technologies. Basically, the functions that have to be supported by the framework in order to address the issues related to process-aware web applications that we mentioned above are:

- implement process-aware links to create, resume or join a process;

- dispatch the correct sequence of pages in process-flow mode;
- hide process-aware links that would lead to processes in which there are no actions (or there is not a specific action) assigned to the current user.

When using a MVC framework like Struts these issues can be easily addressed by writing specific process-aware controllers that, interfacing with the engine, dispatch the correct view given the current user (in our implementations we assume this information can be extracted from the user's session data) and the process the link points to. In the case of processes in which more than one action can be associated to the same user (this is an event that does not show up in our example) an additional parameter specifies which, among the available actions, has to be used as a starting point for the current flow. Process-aware hyperlinks can then be created simply pointing to a process-aware controller. The same controller can be used to dispatch the correct sequence of pages in the process flow. Finally, generic parts of a web page can be hidden by using a conditional custom tag that queries the engine about the availability of an action for the current user in the target process.

The management of the data perspective introduces some subtle issues and, while it is slightly out of the main topic of this article, we are going to discuss possible solutions. The main problem, in this context, is that almost every software system has some kind of data perspective; this is the case for EPML too, since some kind of data management is needed at least to allow the processors to take decisions when needed. It turns then out that we are trying to mix two sub-systems that have two distinct data perspectives. Duplicating the application data in both perspectives is unfeasible and error prone, so reasonable solutions are: using one of the two data perspectives and adapting the other system (if possible) to interface to the selected data management solution or using an external sub-system that addresses the data perspective and adapting the remaining sub-systems to interface to it. The correct solution depends on the specific overall architecture. In our case, for example, if we are dealing with a large business process that interfaces with several systems and is just partially participated using a web application, then using the existing data perspective and design the web application in order to interface to it is the better solution. On the other side, if we are dealing with a fully web-based business process, using the data perspective that is managed by the web application framework (in the case of Java Enterprise Edition, session data for the web tier - persistent data via EJBs or Hibernate for the business tier) is a feasible solution. Assuming the second scenario in our example application, Java Enterprise Edition-based solutions are used to address the data perspective: then, in this case the process logic of the EPML processors has to be designed so that it can access the web framework data (in other words the processors in EPML have to be able to interface to HTTPServletRequest-accessible data - query strings, form submitted-data, session data, etc. - and to the model components). Designing this solution is trivial using Java Enterprise Edition and the EPML engine in which we simply have to add the relevant references into the data bag of the token associated to the current flow each time a terminate activity event is dispatched to the engine (typically when a process-aware controller is activated as a consequence of a process-related interaction returning the control to the process flow). In the project proposal review application this happens, for example, when the reviewers confirm the first review step. A processor positioned between the *First review step* and the *Second review step* activities (which is actually present in our example but is not shown in the diagram for conciseness) takes decisions about outgoing flows by accessing the form data filled by the reviewer. In our prototype, to better separate the two sub-systems, the relevant web framework data are used to create an XML document that is put in the flow data. The processor uses XQuery as seen before for EPML.WS.

7. EPML.SIM

Business Process Simulation (BPS) is widely acknowledged as an effective technique to increase the chance for success of Business Process (re-)Engineering projects and, in general, to drive strategic business decisions.

In this context, we have designed and implemented EPML.SIM, a tool for modeling and simulating processes based on EPML. In particular, we propose to use EPML for modeling the control-flow perspective of a process, while ancillary (potentially pluggable) software systems can be used to support the remaining perspectives and to drive the discrete event simulation. This approach allowed us to design an effective simulation tool with a minimal footprint.

7.1. Architecture of the simulator

The simulation tool is built around three main components: the EPML engine, a *driver* with the responsibility of managing the (simulated) events (events generated by the environment, end task events, timeouts, ...) and a pluggable resources model. The driver also initializes the engine by specifying the process model to be enacted, its initial state and how the process logic and the code of the activities have to be overridden for simulation purposes. The tool, in fact, assumes that the EPML model is a generic model, possibly designed to support the enactment of the process in a real software system (and, in fact, a snapshot of the state of a running process can be used as the simulation's initial state).

The original process logic (that is the code associated to

the processors) could reference data and components that are not available during the simulation. As an elementary example, consider a processor that implements a routing decision by analyzing the data produced by a previously run activity. In this case the processor's code can be overridden for simulation purposes with a pre-defined component that performs the decision on a random basis given a probability for each of the outgoing branches, thus obtaining the behavior of most of the aforementioned simulation tools. If a more detailed modeling is required, the processor's code can also be overridden by a code snippet (EPML directly supports all the languages compliant with JSR 223 which includes Java, BeanShell, Groovy, JavaScript, Python, Ruby, TCL, XPath and others), even a code accessing the cell of a spreadsheet in order to take its decision (we actually used this solution in a simulation where a detailed financial model of the organization was available).

As stated above, also the code associated to the activities can be overridden. Pre-defined activities returning only duration informations on the basis of a probability distribution (chosen among constant, uniform, normal, gamma and exponential) or by historical data stored in a text file or in a column of a spreadsheet, are available. If a more detailed model is required, any kind of code can be used (as seen before for the processors), allowing the interaction with other business models (or with other software systems). Using this technique we can address the functional perspective.

The data perspective is addressed by using the basic data handling capabilities provided by the EPML engine. In EPML tokens are used to keep track of the status of a process. The EPML engine allows tasks to access a *data bag* (implemented as an associative map) associated to the token(s) that activate the task. The data bag is also used to carry information about the items processed during the simulation. This is implemented by specifying which events are used to generate new items (for example the event representing the arrival of a new request to be processed can be tagged as a request generator). Tokens enabled by these special events are tagged (by adding the relevant information to their data bag) and are subject to statistical recording by the driver component.

The organizational perspective is implemented by a pluggable component (a Java class implementing a specific interface). The driver queries this component when a new activity has to be executed in order to obtain the needed resources (and to know their costs). The component is also notified when a resource is (or more resources are) released (because of the end of an activity). A basic component is provided which implements a simple role-resources matrix combined with an availability matrix. Yet again, if a more detailed organizational model is required a new Java class implementing the details of this specific model has to be created. Notice, however, that this solution does not support a resource model with preemption (that is a resource model in which a resource can be reclaimed by a high priority task while performing a lower priority one, which is canceled). This is because in EPML.SIM the canceling of a task is possible only from the control-flow perspective, that means that when such a behavior is required it has to be implemented in the process model (by using the cancellation support of EPML).

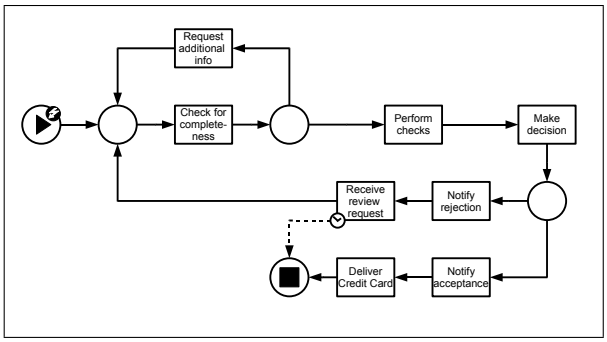


Fig. 9. The credit card application process

7.2. Simulating a Business Process with EPML.SIM

In this section we show how a Business Process can be effectively simulated by using EPML.SIM. To this end we use, as an example, the credit card application process depicted in Fig. 9 (inspired to the example from [29]).

The behavior of the modeled process is quite straightforward; the event associated to the start processor represent a new incoming application and is used to generate a new instance of the process; the clock icon associated to the `Receive review request` represents a time-out: if the activity is not performed within a specified amount of time the path exiting from the clock icon (an exception path) is activated.

This diagram corresponds to an XML file (that can be produced with EPML modeler, a graphical editor created with Adobe Flex). In this example we call this file process.xml. This XML file, along with another XML file (simulation.xml) specifying which (and how) processors/activities have to be overridden and, optionally, with an initial state description, are used by EPML.SIM. In our example we suppose that the activity `Check for completeness` is characterized by a duration depending on a normal probability distribution. We then override the behavior of this activity in simulation.xml with the following XML fragment:

```
<activity activityId="CheckForCompleteness"
```

```
    classname="epml.simulator.executors.
               ProbabilityActivityDuration">
  <param name="distribution">normal</param>
  <param name="unit">minute</param>
  <param name="mean">5</param>
  <param name="stdDev">2</param>
</activity>
```

where `ProbabilityActivityDuration` is the name of a pre-defined Java class that returns the duration of the activity by using a probability distribution (normal, in this case, with mean 5 and standard deviation 2).

For the subsequent processor (implementing a choice between two possible paths in the process structure) we can use a similar approach by redefining it in simulation.xml as follows:

```
<processor processorId="choice1"
    classname="epml.simulator.executors.
               ProbabilityProcessSwitch">
  <param name="e6">.1</param>
  <param name="e7">.9</param>
</processor>
```

where `ProbabilityProcessSwitch` is the name of a predefined Java class that implements a random choice between the outgoing edges, listed as parameters, associated to different probabilities (*e6* and *e7* are the ids of the outgoing edges as defined in process.xml).

By using a similar approach for other activities and processors, and by setting up the details of the built-in resources model, it is possible to create a simple simulation. Please notice that, at the time of this writing, the file simulation.xml has to be edited by hand. An extension to the EPML modeler tool to set up the details of a simulation is in the works.

As stated above, however, EPML.SIM allows to produce more accurate simulations where more detailed models are available. As an example consider the `Make decision` activity and its subsequent processor. We could override the activity using the `ProbabilityActivityDuration` and the processor using the `ProbabilityProcessSwitch` as seen above. In this case, however, we suppose we want the decision being implemented by the action dependent on a parameter representing the request rate (calculated in another point of the process and added to the token's data bag). We can use two approaches: override the code associated to the processor with an ad hoc script fragment or override the code associated to the activity. In this second case, assuming that the processor implements the decision by analyzing the information returned by the `Make decision` action, we have to override the action so that it returns the same values assumed by the processor. This latter approach is best suited for when we use a process model created for the actual enactment of the process. An example is as follows:

```
<activity activityId="MakeDecision"
    classname="epml.simulator.executors.
    ActivityInstanceExecutor">
  <param name="language">BeanShell</param>
  <param name="code">
    <![CDATA[
    Double requestRate = ((Double)
      (((Map)tokenData.
      get("simulation")).
      get("requestRate"))).doubleValue();
    boolean accept = requestRate > 10;
    ((Map)tokenData.get("transient")).
      set("return", new Boolean(accept));
    ((Map)tokenData.get("simulation")).
      set("duration", new Double(10));
    ]]>
  </param>
</activity>
```

where *transient* is a special entry in the token's data bag that is used for information that can be discarded when a new activity is executed and is typically used to store the return values of an activity. The *simulation* entry is also a special entry that is used to carry information related to the simulation (in this case for accessing the previously calculated request rate and for setting the simulated duration, in seconds, of the activity). Please notice that the code in the example may look complex at first sight because of the way Java (and thus BeanShell) accesses maps; language verbosity aside it just sets and retrieves values from (nested) associative data structures.

Once the models have been set up, the simulation can be launched. To this end an horizon has to be decided. In EPML.SIM the analyst can decide to stop a simulation after a specified amount of (simulated) time, at a specific date/time, after having processed a certain amount of items or by using a generic script that is called after the processing of each event (when the script returns a false boolean value, the simulation ends). Support for ending a simulation when a service level agreement (related to the duration of the activities, to the utilization of the resources or to the costs) is not met is on the works. At the time of this writing EPML.SIM does not support animations to give visual feedback on the simulation's progress. At the end of the simulation a report is produced. The report can be either in plain text format or in Open Office Calc format (which includes charts and is formatted in such a way that it can be used to easily generate a PDF report). A fragment of a sample report is depicted in Fig. 10. In a report are shown the simulation parameters and results, such as the total simulation time, the total cost, the throughput, how many activities (or items) have been simulated and, for each activity, its cost and the maximum and the average queue length.

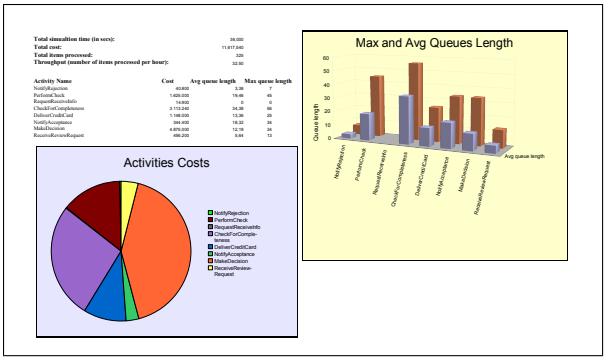


Fig. 10. A report produced by EPML.SIM

8. RELATED WORK

In the last few years, a large spectrum of workflow languages and process modeling tools ([14, 6, 4, 19, 16], to name a few) have been proposed from the industry and the academia. Most of these solutions use different strategies to model a process and each of them shows specific strong and weak points, making hard to compare the different solutions. We propose to use expressive power and suitability as reasonable metrics to this end. As far as expressive power is concerned most of the proposals show strong limitations. For example, on the basis of our hands-on experience, common real-world processes turn out to be very hard (if at all possible) to model with most existing languages without modifying its semantics or producing an overmuch complex specification. Suitability too is related to expressive power but it is also related to the ability to adapt to different application domains. This is often related to the ability to be integrated in different software architectures. Some of the most recent proposals (like YAWL and Orc [14] - all coming from the academia) try to address the problem of expressive power by implementing all (or most of) the classic workflow control-flow patterns (the first language to claim to support all the patterns is YAWL, not surprisingly designed by the same research group that originally defined the patterns, what is surprising is that this claim is still not supported by a proof). YAWL is a powerful language and comes with a formal semantics (that is actually used to check specifications properties) and a reference implementation. Its main limits are related to its suitability outside the workflow domain, both because of its Petri nets-inspired model, and because of its engine architecture.

Another proposal that deserves to be referenced is BPMN. BPMN is a OMG-endorsed specification that is receiving a great deal of attention by the industry. This is mostly due to the fact that a large number of process based applications, and a large number of software products to develop them, already exists. Most of these applications are related to business process management (ERP, workflow, supply chain management) and they are urged to support a high degree on interoperability. In this context a process modeling-related standard is badly needed. The problem with BPMN is that it tries to address too many issues. It presents itself as a tool for high level - conceptual process modeling that can be used to outline a process without defining its detailed semantics but it also claims to support MDA-like translations into executable specifications (in this case using BPEL). But in the article we already pointed out the limits of this approach. It is our opinion that BPMN is a good modeling notation (the "UML for processes") but it falls short when it comes at programming in the small.

9. CONCLUSION AND FUTURE WORK

The wide array of applications, belonging to different domains, that we designed and implemented by using our EPML-based framework witness that a modular approach to process-aware application is possible from both a design and an implementation point of view. This implies that the large number of existing process modeling languages and systems cannot be justified only by the large spectrum of application domains. In our opinion this is mostly due to the lateness of the academia with respect to the needs of the industry: the latter, lacking strong indications from the former, went its own way proposing a large number of inherently limited tools. The separation of concerns, in the form of a clear distinction between computation, interaction model and process logic, that is at the roots of EPML, provided a solid framework for achieving a high level of modularity. We hope that these concepts can help in defining the priorities around which next generation process modeling languages should be designed (or around which current languages should evolve, as in the case of a possible forthcoming executable BPMN specification).

Our work on EPML continues. While the language has reached a good level of maturity and no major changes are foreseeable the runtime-system and the support tools are subject to a continuous evolutions. For example: while a graphical modeling tool based on Adobe Flex is already available, another tool based on Eclipse is on the works. As for the runtime-system: the engine supports state saving/restoration by using a relational database as a backend; checkpointing and recovery is still not fully implemented. These are just examples of a long todo list that never shrinks as new possible applications of our framework continue to emerge.

10. REFERENCES

[1] Activebpel open source engine project. `http://www.activebpel.org/`. Accessed January 2009.

[2] BPELJ: BPEL for Java technology. White paper. Available at http://www.ibm.com/developerworks/library/specification/ws-bpelj/. Accessed January 2009.

[3] Business Process Execution Language for Web Services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/. Accessed January 2009.

[4] W. M. P. V. D. Aalst and A. H. M. T. Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[5] W. M. P. V. D. Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(14):5–51, 2003.

[6] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Liu, R. Khalaf, D. Konig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. Accessed January 2009.

[7] F. Arbab. What do you mean, coordination? Technical report, Bulletin of the Dutch Association for Theoretical Computer Science, NVTI, 1998.

[8] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[9] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.

[10] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Commun. ACM*, 35(9):75–90, 1992.

[12] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[13] S. Jablonski. Mobile: A modular workflow model and architecture. In *Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems*, 1994.

[14] D. Kitchin, W. Cook, and J. Misra. A Language for Task Orchestration and its Semantic Properties. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, pages 477–491. Springer Berlin / Heidelberg, 2006.

[15] N. Koch, A. Kraus, C. Cachero, and S. Meliá. Integration of business processes in web application models. *Journal of Web Engineering*, 3(1):22–29, 2004.

[16] OMG. Business Process Modeling Notation (BPMN) version 1.0. http://www.bpmn.org/.

[17] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst. From BPMN process models to BPEL Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)*, pages 285–292, Washington, DC, USA, 2006. IEEE Computer Society.

[18] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

[19] D. Rossi. X-Folders: documents on the move. *Concurr. Comput.: Pract. Exper.*, 18(4):409–425, 2006.

[20] D. Rossi and E. Turrini. EGO: an E-Games Orchestration Platform. In *Proceedings of the 8th annual European GAMEON®Conference on Simulation and AI in Computer Games*. EUROSIS-ETI, 2007.

[21] D. Rossi and E. Turrini. EPML: Executable Process Modeling Language. Technical Report UBLCS-2007-22, Department of Computer Science, University of Bologna, 2007.

[22] D. Rossi and E. Turrini. Using a process modeling language for the design and implementation of process-driven applications. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE Computer Society, 2007.

[23] D. Rossi and E. Turrini. Designing and architecting process-aware web applications with EPML. In *Proceedings of the ACM symposium on Applied computing (SAC 2008)*, pages 2409–2414, New York, NY, USA, 2008. ACM.

[24] D. Rossi and E. Turrini. An executable language/enactment engine approach for designing and architecting process-aware web applications. *International Journal of E-Business Research (IJEBR)*, 5(3):1–13, 2009.

[25] N. Russell, A. H. ter Hofstede, W. M. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.

[26] H. A. Schmid and G. Rossi. Modeling and designing processes in e-commerce applications. *IEEE Internet Computing*, 8(1):19–27, 2004.

[27] O. D. Troyer and S. Casteleyn. Modeling complex processes for web applications using WSDM. In *Proceedings of the Third International Workshop on Web-Oriented Software Technologies*, 2003.

[28] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 19–128. Springer, Berlin / Heidelberg, 2000.

[29] M. T. Wynn, M. Dumas, C. J. Fidge, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Business process simulation for operational decision support. In *Proceedings of the Third International Workshop on Business Process Intelligence (BPI 2007)*, volume 4928 of *Lecture Notes in Computer Science*, pages 66–77, Berlin / Heidelberg, 2007. Springer-Verlag.

Trisolda: The Environment for Semantic Data Processing

Jiří Dokulil, Jakub Yaghob, Filip Zavoral
Charles University in Prague, Czech Republic
{dokulil, yaghob, zavoral}@ksi.mff.cuni.cz

Abstract

In order to support research and development of web semantization tools, methods and algorithms we have designed and implemented the Trisolda infrastructure. It is built around a semantic repository which is supplemented by import, query and data processing interfaces. The Trisolda application server can be extended by plug-ins for advanced semantic analysis and processing. We propose the TriQ RDF query language; its compositionallity and closedness make it useful for complex semantic querying.

Keywords*: Semantic web, infrastructure, repository, query languages*

1 Suffering of the Semantic Web

One of the main goals of the Semantic Web is to create a universal medium for the exchange of data. The Web can reach its full potential only if it becomes a place where data can be shared and processed by automated tools as well as by people. For the Web to scale, tomorrow's programs must be able to share and process data even when these programs have been designed totally independently [19].

Unfortunately, it seems, this goal has not yet been reached, albeit years of research by numerous researchers and large number of published standards by several standardization organizations.

We believe the Semantic Web is not yet widespread due to three prohibiting facts: missing standard infrastructure for Semantic Web operation, lack of interest from significant number of commercial subjects (although this started to improve recently), and the last but not least absence of usable interface for common users.

A nonexistence of a full-blown, working, high-performance Semantic Web infrastructure inhibits effective research of web semantization. Whereas the 'old web' has clearly defined infrastructure with many production-ready infrastructure implementations (e.g., Apache [20], IIS [21]), the Semantic Web has only experimental fragments of infrastructure with catastrophic scalability (e.g., Sesame [4], Jena [22]).

We have tried, during our experimental research, to convince commercial subjects to make somehow their data accessible on the Internet (of course with some reasonable level of security), and they all refused to make external access to their data. Commercial subjects do not intend to participate willingly in the ideas of the Semantic Web, because for them it either means to share their business data openly or to invest a lot of time and money for securing access to them.

Current standards in the Semantic Web area do not allow it to be used by common users. Whereas any user of WWW can easily navigate using hyperlinks in an available, production-quality WWW client, a contingent Semantic Web user has only a choice from a set of complicated query languages (e.g., SPARQL [13], SeRQL [3]). These query languages are not intended for casual users, only small number of people are able to use them.

Although SPARQL is probably the most popular RDF query language in the semantic web community, its overcomplicated definition and low expressive power make it unsuitable for most web semantization projects. Therefore we propose the TriQ query language that is based on the time proven ideas behind relational algebra and SQL.

The following chapters are organized as follows: after an overview of the infrastructure there is a description of the application server in Section 3 and the query API in Section 4. Sections 5 to 7 propose the TriQ language. Two final sections contain performance comparison and conclusions.

1.1 Related Work

Of course, the Trisolda infrastructure is not the only attempt to create an infrastructure for the Semantic web. One important example is the WSMX environ-

ment [15], which also represents a different approach to building the infrastructure. Unlike Trisolda, which is centered around the RDF database and Trisolda server, WSMX is concerned with semantic web services. It's purpose is to allow discovery (using Web Service Modeling Ontology [14]), mediation, invocation and interoperation of the services.

2 Infrastructure overview

We have recognized and described the problem of a missing, standard infrastructure for the Semantic Web in [17], where we have proposed a general ideas of a Semantic Web infrastructure and later refined the proposal in [7]. During the last year we have made a significant progress: we have implemented full-blown, working, fast, scalable infrastructure for the Semantic Web called Trisolda.

The figure 1 depicts the overall scheme of its infrastructure. In this picture rectangles represent processes, diamonds are protocols and interfaces, and grey barrels represent data-storages. All solid-line shapes depicts implemented parts of our infrastructure, whereas all dashed-line shapes represent possible experimental processes implemented by researchers playing with our infrastructure.

2.1 Trisolda repository

The heart of Trisolda infrastructure is a repository. It is responsible for storing incoming data, retrieving results for queries, and storing the used ontology. It consists of the a data-storage, which is responsible for holding semantic data in any format. Import interface enables fast, parallel data storing and hides details about background a data-storage import capabilities. The query interface has two tasks: to be independent on a query language or environment and to be independent on the Trisolda data-storage query capabilities. The last part of the repository is Trisolda Application Server. It is a background worker that does the inferencing, makes data unifications, and fulfills the task of a reasoner as well. It utilizes import and query interfaces for data manipulation.

2.2 Import paths

We use two distinguishable sources of data. The simplest one is a data import through importers from external data-storages. The task of importers is mapping external data-storage data-scheme to the SemWeb repository ontology. The second source of data crawls the wild Web using a web crawler; we have used Egothor [9] in the pilot implementation. The crawled web pages are stored in a Web pages data-store, where they can be accessed in parallel by deductors, which can deduce data and their ontologies from web pages and map them to our ontology.

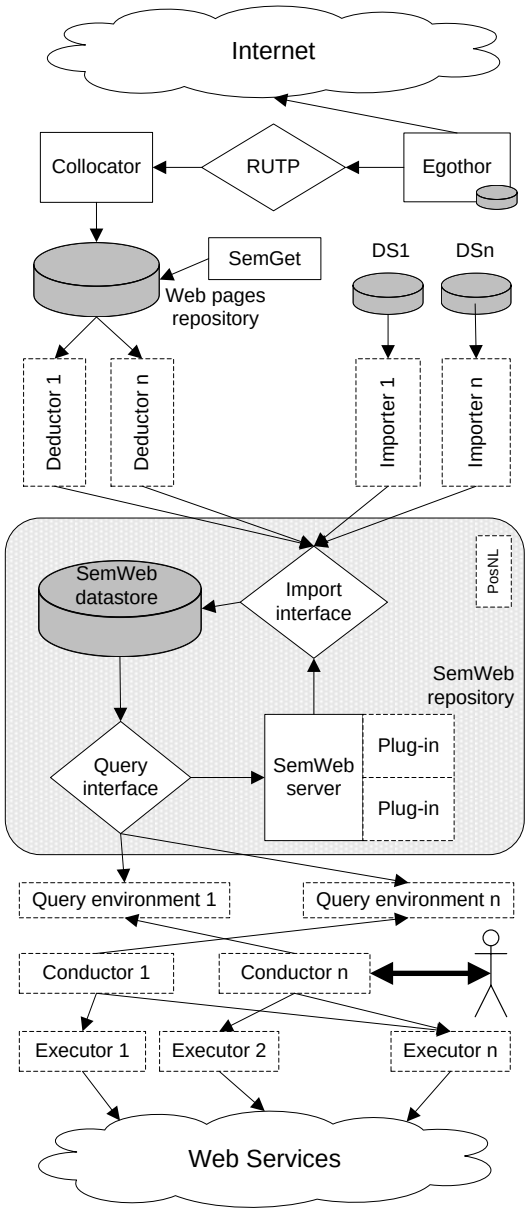


Figure 1. Infrastructure overview

2.3 Query environments

Query environments present outputs from Trisolda repository. They make queries using query API and

present results to users in any feasible manner. We have implemented a SPARQL compiler as an example, which translates SPARQL queries to the internal query API requests.

2.4 Data-storage access

We have designed and implemented an object oriented library in C++ for a data-storage access independent on a background data-storage implementation. This library is used for a low-level access to the data in all interfaces to data-storages. It allows us to change an underlaying data-storage without modifying the code of our infrastructure.

2.5 Portability

Unlike other many research projects implemented usually in Java, we have decided to implemented nearly all parts (excluding Egothor implemented in Java independently on our project) in ISO/IEC 14882 C++. The main reasons are speed, more controlled computing environment (e.g., memory management), and, although it seems absurdly comparing to Java, stability.

When properly used, using ISO C++ brings full portability among different systems and compilers. Moreover, it allows us to implement bindings to other broadly used languages, e.g., Java or C#.

3. Trisolda Application Server

The main active part of the Trisolda repository is Trisolda Application Server. It is a background worker that does the inferencing, makes data unifications, and fulfills the task of a reasoner as well. It utilizes the import and query APIs for data manipulation.

It should be noted, that the server is not a web server in a conventional meaning. It does not handle any HTTP requests.

3.1. The server's role

We believe, we do not need to have all accurate data and inferences at the moment of data import. Just like the real world, the world knowledge changes at each moment and we are not able to catch it in one snapshot. Therefore postprocessing data in the background by Trisolda Application Server and computing some additional data in the background is acceptable and feasible.

The server is only a framework offering unified connection, interface and task management for experimental plug-ins, as described in the next sections.

3.2. Server's plug-ins

Trisolda Application Server's plug-ins are independent modules, which simultaneously perform different operations on SemWeb storage in the background. Whereas import and query APIs are only libraries enabling the access to the Trisolda storage, the server allows active operations upon the storage.

Each plug-in must conform to an interface requested by the server, whereas the server offers several classes of services for plug-ins.

3.3. Implementation

Plug-ins are implemented as dynamically loaded libraries. This is an important feature, which allows selective loading and unloading of any server's plug-in without interrupting overall infrastructure operation.

Although we use C++ to implement the whole project, the interface requested by Trisolda Application Server is in a C-like style, because there is currently no possibility to make a portable C++ dynamic library interface.

3.4. Executors

The results of querying semantic data can be interpreted by many methods. From relationally oriented data-set through set of references well-known by web search engines or set of mutually semantically related entities and their attributes up to application-level services using service oriented architectures.

Traditional result representation is tightly coupled to query method. SDE displays interconnected pages containing result data together with their structure and relationships, search engine displays web links with appropriate piece of text, and SPARQL returns rows of attribute tuples.

While many researchers are satisfied with making queries, the users (based on the ideas presented in [2]) would expect more from the Semantic Web. They expect it to take care of things, not just answer queries in a Google-like fashion.

The technique of executors brings process models into this infrastructure. The task of executor is to realize semantic action, i.e. interaction of result data with an outstanding (not only semantic) world. These atomic executors can be assembled to complex composed executors. Orchestration, i.e. mutual executor interconnection to achieve more complex functionality is executed by the Conductor module.

The technique of executors may be illustrated by following example. One's mother has gone ill, she needs a

medicine. A query module searches nearby pharmacies with the medicine available. One executor is responsible for buying the medicine, while the other arranges delivery to mothers home. The Conductor orchestrates these two executors to synchronize, mutually cooperate, and pass relevant data between them.

3.5 Retrieving web documents

In the original proposal [17], there was a direct Egothor plugin for semantic data and metadata acquisition. This runtime structure would have several disadvantages.

The plugin dedicated to semantic experiments would run in the same environment as the general-purpose web robot. This would cause deficiencies in stability and performance. Moreover, it is harder to debug in this environment. Many semantic experiments need to apply several algorithms to a data set. Multiple data acquisition would cause unacceptable load to both the extractor and data providers. The web robot couldn't be dedicated to semantic data acquisition - it executes tasks for a lot of clients. Thus the delay between an initial request and document set completeness could be too long. We have decided to separate the web gathering and the semantic processing, both in time and space.

The retrieving document is converted into a stream of SAX events which enables us to process its internal structure more comfortably. This stream is then sent by a Robot UDP Transmitter Protocol (RUTP) [8] to a Collocator. The server reads document requests stored into the database, converts them into RUTP commands, sends them to the robot, receives streams of SAX events, completes them, computes tree indexes and in case of successful transmission stores each parsed document into the database.

The database stores each document in a structure similar to a XML Region Tree [11] or a NoK pattern tree [18]. The main feature of this structure is query effectivity - for a given element, all its ancestors or descendants in an element set can be queried with optimal cost.

4 Query API

The Query API is based on simple graph matching and relational algebra. Simple graph matching allows only one type of query. It consists of a set of RDF triples that contain variables. The result of the query is a set of possible variable mappings. This set can easily be interpreted as a relation with variable names used as a schema for the relation.

Relational algebra operations (e.g., joins or selection) are used on the relations created by simple graph matching. These operations are widely known from SQL, which was a major argument for this choice. Database developers are already familiar with these operations and a lot of work has been put into optimizing these operations.

So far, we decided to support only some of the common relational operations. Since the schema of elementary relations (results of basic graph patterns) consists of variable names, it is defined by the query and not in the database schema. For this reason, we use only natural joins. Variable names are used to determine which columns should the join operation operate on.

4.1 Selection

Selection operation revealed several problems specific to RDF querying. While in traditional relational algebra it is easy to maintain type information for each column, it is not possible in RDF. Even a simple query can produce a result that contains values with different data types in one column.

Having this in mind, we have to consider behavior of relational operators when it comes to different data types. For instance, in SPARQL [13] the operators should consider data types for each value separately, so one operator in one query compares some values lexicographically by their string value and some other values numerically by their value.

This is a serious performance problem that for instance makes it impossible to use indexes to evaluate expressions like $x < 5$ and especially $x < y$. On the other hand, such behavior is often not necessary because the user has certain idea about data type of x in $x < 5$. So we decided to make the type information part of the query. Then $x <_{integer} 5$ yields true for integers smaller than 5, false for integer greater or equal to 5 and error if x is not integer. This error always removes the whole row from the result.

This definition makes translation of queries to SQL more simple and efficient since it can be easily evaluated by a functional index that stores integral values. Conditions like $8 < x \; and \; x < 10$ can be evaluated by simply traversing a small part of this index.

4.2 Query language

We decided not to create yet another SQL-like query language. Since the query interface is intended to be used not by people but rather software, the query interface is actually a set of classes (an API). An example of a simple query tree:

- Natural join
  - Left natural join
    - Basic graph pattern P1
    - Basic graph pattern P2
  - Basic graph pattern P3

Had we created a query language, our form of query would basically be a derivation tree of a query in that language.

4.3 Query example

Following C++ code shows a simple query that outputs first and last name of all people that have both of them and whose last name is either "Tykal" or starts with "Dokulil".

```
Triples triples;
triples.push_back(Triple(
  Variable("x"), URI("http://example.org/lastn"),
  Variable("y")
));
triples.push_back(Triple(
  Variable("x"), URI("http://exmpl.org/firstn"),
  Variable("z")
));
Table *query_tab= new Filter(
    BasicGraph(triples),
    OrExpression( TestSubstringExpression(
        NodeExpression(Variable("y")),
        NodeExpression(Literal("Dokulil")), T, F
      ),
      EQExpression(
        NodeExpression(Variable("y")),
        NodeExpression(Literal("Tykal"))
  ) ) );

std::vector<Variable*> vars;
vars.push_back(new Variable("y"));
vars.push_back(new Variable("z"));
Query query(vars,query_tab,false);
```

The query consists of a basic graph query with two triples and three variables. Then a selection is applied to the result and finally a projection is used to return only columns with first and last name.

4.4 Query evaluation

We did not want to limit ourselves to just one system for data storage. Since the beginning of development we have been using four different data storages with several other in mind. Each of the systems offered different query capabilities from just evaluating all stored RDF triples to sophisticated query languages.

The contrast between a complex query API we wanted to give to the user and only basic query capabilities provided by the data storage system made it obvious that Trisolda must be capable of evaluating the queries itself. By implementing all operations within our system, we have reduced the requirements for the data storage engine to just one; the engine has to be able to list all stored triples. Thus the system is capable to use extremely simple storage engine that does nothing but read RDF triples from a Turtle file [1].

One of the other storage engines is an Oracle database. It would be highly inefficient to use the database only to list all triples, we want to utilize much of the Oracle optimized operations.

As a result, Trisolda is capable of evaluating any query itself, but tries to use any help the storage engine can provide. The same goes for adding new features to the query interface. Once the feature is implemented in our system, it is immediately available with all storage engines. Of course, performance of query evaluation will probably be suboptimal.

4.5 Evaluation algorithm

Since every storage engine can have specific capabilities, we could not establish a set of rules to decide what can be evaluated by the engine. Thus each engine contains an algorithm to determine whether it is able to evaluate a query. For query Q the evaluation plan is found like this:

- If the storage engine can evaluate Q then this evaluation plan is used.
- If Q is basic graph pattern then it is decomposed into individual triples, the evaluation plan for each triple is found and the results are joined together.
- If Q is an algebraic operation, evaluation plan for each operand is determined and the operation is applied to the results by Trisolda.

The limitation of this algorithm is, that it does not try to rearrange the query in order to achieve better performance either by choosing a more efficient evaluation plan or by allowing the storage engine to evaluate greater part of the query itself, which will probably be more efficient. This problem is a more complex version of optimization of relational algebra expressions and will be a subject of our further research.

4.6 Remote queries

Creating a data interface between different programming languages is not an easy task. To allow queries

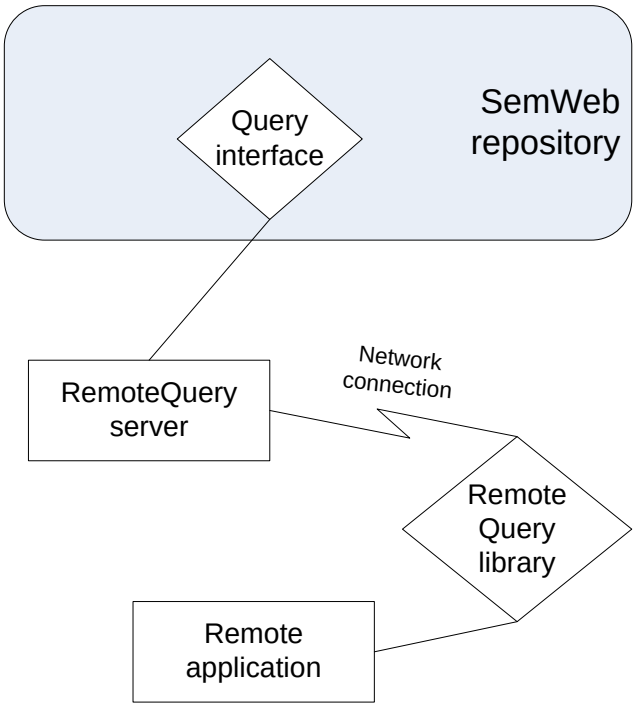


Figure 2. Remote query

from languages other than C++ we created a HTTP-based API that is language independent. The client issues a HTTP GET command that contains the query to be executed and receives a HTTP response that contains the results.

To make this possible, we had to create a way to send a query over HTTP. Since we do not use any textual query language, this required some work to be done. But due to a simple tree structure of the queries, serializing a query to a character string was a relatively easy task. The name of the class is stored in the string followed by the data. The data are either strings (stored directly) or other classes (the same serialization algorithm is recursively called to store them to the string).

An example of such encoded query could be this:

```
Query{0;BasicGraph{Triple{Variable{x};
URI{http://www.is.cuni.cz/stoh/schema/
ot_osoba#prijmeni};Literal{Dokulil;;}}
};x}
```

Furthermore, the format of the results has to be decided. To make the server as fast as possible the in-memory data format used internally by the server is used. The format is suitable for transfer over network to different platforms - it has no little/big endian or 32/64 bit compatibility issues and it only uses small amount of information other then the actual data. Only 5 bytes plus size of a URI (encoded as UTF-8) is required to transfer the URI, 5 bytes plus length for untyped literal, 9 bytes plus length of value and type for typed literal, ...

We have implemented and successfully tested a C# client library.

4.7 Complex query languages

One of the ways in which the query API can be used, is to build more sophisticated query languages. An example is a limited SPARQL [13] evaluator or the Tequila query language [10]. The languages represent very different approach to RDF querying but they both use the same API and thus access the same data.

The Tequila language is based on named patterns and supports recursive queries. An interesting example is a selecting employees from a list. The lists in RDF are recursive, which makes the following query impossible in SPARQL.

```
prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ex: <http://www.example.org/term#>

ex:list(?N)
{
  {
    ?N rdf:first ?F.
    ?N rdf:rest ?R.
    use ex:list(?R)
  }
  union
  {
    filter ?N = rdf:nil.
  }
}
get
{
  ex:department ex:employees ?list.
  use ex:list(?list)
}
```

The ex:list named pattern has one parameter and is used recursively to traverse the whole list.

Although this is only a very basic example, it clearly demonstrates the possible diversity of query languages that can be build over Trisolda.

5 TriQ

Querying is one of the important issues for any data format. In the case of RDF, many query languages have been developed, including the SPARQL language

[13], which is a W3C recommendation, or SeRQL [3], which is supported by the popular Sesame RDF framework [4].

Many are inspired by SQL (although not all of them, e.g., Lisp-like Versa [12]). But the inspiration is usually manifested (besides the fact that the queries syntax may look a bit like SQL) in the fact that the RDF graph is transformed by some graph pattern matching operation into some table-like form and these tables are then further processed. We believe that the inspiration should have been a little different. An important feature of SQL (and its theoretical background – the relational algebra [5]) is the fact, that it is a closed system. Relations are transformed into relations. This way, a result of a query can be used as an input for another, more complex query, which is impossible in SPARQL and SeRQL.

The following parts of this paper present our proposal for TriQ – a SQL-inspired, closed RDF query system. To make the system closed, we couldn't have used relations as our data model. We use RDF. But the operations are inspired by relational algebra – we use selection, projection, (inner and outer) joins, etc. The semantics of these operations is not exactly the same (after all, we have a very different data model) but the ideas behind them are the same. We believe that this (and the fact that it is closed) makes the whole query system more accessible for wider audience of developers, especially those with long SQL experience.

5.1 Data model

Since we want a closed query system, we require every operation to take some RDF graphs (zero or more) as its input and produce an RDF graph as its output. But to make the operations simple to use, we need to add some further information. We use the very RDF that contains the data for the task and add additional triples to the data – *decorate* it.

Namespaces To make decoration simple we define several namespaces. URIs starting with theses namespaces are prohibited in the queried data. The namespaces are
`dec` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/decoration`
`ptr` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/pointer`
`graph` is `http://ulita.ms.mff.cuni.cz/Trisolda/GQL/graph`

5.2 Decoration of nodes and edges

We can decorate either nodes of the RDF graph, in which case we add a new triple where the decorated node is the object of the triple, or edges, in which case we have to reify the edge (unless the triple is already reified) and then decorate the reification. To be more specific, to decorate the edge `S P O` with decoration triple D_S D_P `?` (the question mark is the decorated object) we add the following triples:

- `X dec:subject S`, `X dec:predicate P`, `X dec:object O`
- D_S D_P `X`

The X denotes an anonymous node. Note that we do not use the standard reification defined by RDF, but rather use the `dec` namespace to avoid potential "collisions". This way we can always distinguish statements added during decoration and statements that were present in the original data.

Decoration options There are two types of decoration edges. Let X be the decorated object (either a node or reification of an edge), G an URI from the namespace `graph` (each graph has a globally unique URI) and P an URI from the namespace `ptr`. The possible decoration triples are:

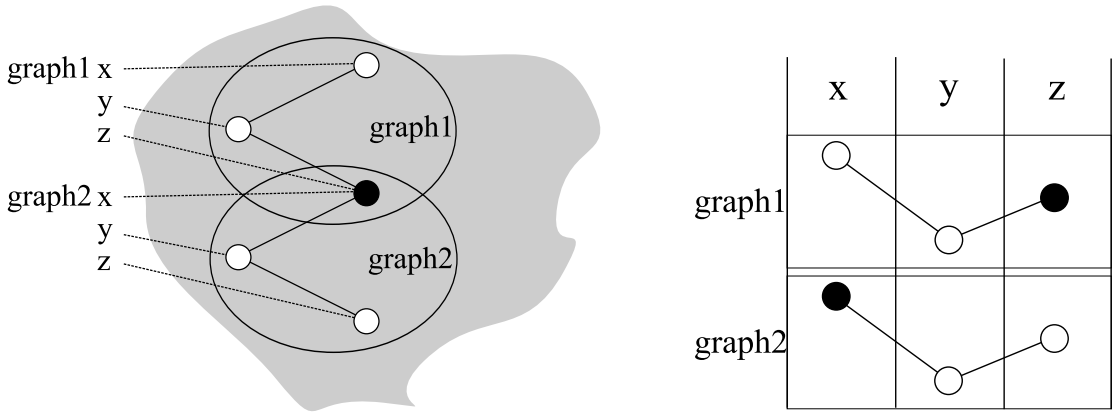
1. `G dec:contains X`
2. `P G X`

There are no restrictions for the second type of triples. The only restriction for the first type is that every node and edge of the decorated graph is decorated by at least one such edge.

5.3 Meaning of decoration

The purpose of decoration is to help user define a structure in the queried graph and exploit it to define further operations. Furthermore, we would like the whole query system to resemble relational algebra, that works with relations – sets of tuples with a well defined schema.

The first type of decoration triples is used to make the (one) RDF graph appear as if it was a (multi)set of smaller graphs so that each of the smaller graphs resembles one tuple of a relation (row of a table). The triple `G dec:contains X` tells us that the graph G contains X. So if we take all such X for one G, we get one small RDF graph.



(a) Multiple graphs in one graph

(b) Table analogy

Figure 3. Meaning of decoration

The second type of decoration triples were introduced as a parallel to schema of a relation (for an example see the Figure 3). The set of all values from the namespace `ptr` used in the graph are schema of the graph. All values of X from all triples `P G X` for a graph G correspond to a value of column P in a row G of a table. But unlike SQL that allows zero (NULL) or one value, we allow zero or more values. Although we could restrict it to just one value, we believe it would unnecessarily limit the graph-handling capabilities of the query system. For instance, we would like to be able to create such set of graphs, where each graph contains information about one person (e.g., first and last name) and all of his or her e-mails. And to make handling of the data convenient, we would like to have a pointer ptr:first-name point to the first name of the person, ptr:last-name to the last name and ptr:mail to all of the emails. That way, we could for example easily find people with more than one email or get the number of emails for each person.

6 Graph pattern operation

The graph pattern operation is in a certain sense the very basic operation of the query system. It is used to find patterns in the whole queried data. The very basic principle is that the operation specifies an RDF graph where some nodes or edges are replaced by variables. The evaluation is done by finding possible substitutions for these variables so that the we get a graph that is a subgraph of the queried data. This graph is then one member of the result set.

The operation gives the data a structure that helps us reference certain concepts in further query operations. For example, if the pattern looks like `?x ex:has-name ?y` (where ?x and ?y are variables), we know that the actual nodes bound to ?x are people and ?y their respective names (provided we have reasonable data).

Each node and edge can of the pattern can have a pointer assigned to it. In that case, the corresponding node or edge of the result is then pointed to by that pointer.

Why? Many RDF query languages (e.g., SPARQL [13], Trisolda query API [7],. . .) use some kind of graph patterns to transform the RDF graph to a table (or something analogous like set of variable mappings in the case of SPARQL). In other words, they use it to transform the queried data into some other form suitable for further processing.

At the moment, we believe this is the only operation that should work with "raw" data and that all other operations can assume to be working with decorated data. This is not as important from the technical or formal point of view, but rather from the "average user's" point of view. It would allow him or her to construct the query in two steps. First, well structured pieces of data are defined by the pattern operation. Second, the structure is exploited to combine the pieces of data into the final result. The second phase should be as close as possible to writing a query in SQL.

6.1 Definition

The previous sections briefly and informally explained what capabilities the proposed pattern matching possesses. This section gives a more formal view of the operation.

Let Uri be a set of all URIs, Lit set of all RDF literals, $Blank$ set of all blank nodes, and Var an infinite set of variables.

Let $V \subseteq Uri \cup Lit \cup Var$. The pattern P is then defined as a non-empty set of triples $P \subseteq V \times (Uri \cup Var) \times V$. V is the set of all nodes in the pattern P, $Var(P)$ denotes set of all variables used in the pattern P. The pattern can be viewed as a directed, labeled multigraph. We require that each two nodes in V are connected by an undirected path.

Each edge or node of the patter can be assigned a pointer (i.e. URI from the namespace `ptr`). The same pointer may be assigned to more objects (edges and nodes).

Let P_b be a pattern and V_b nodes of P_b. Let G be the queried data.

A variable mapping is a function $\mu : Var \rightarrow Uri \cup Lit \cup Blank$. We extend the variable to triples $t \in P_b$ and the whole pattern P_b in the natural way (each variable v used in t or the whole P_b are replaced by $\mu(v)$). We always use the variable mapping μ in conjunction with a pattern P, in which case we consider only minimal mapping, i.e. $dom(\mu) = Var(P)$.

We say, that an RDF graph M_b is a *match* for P_b iff there are mappings μ and η such that the following statement holds: $M_b = \mu(P_r) \Subset G$ where $\Subset$ is a relation between RDF graphs. The basic version of TriQ assumes that $(A \Subset B) \equiv (A \subseteq B)$.

Then we say that the tuple $\langle \mu, M_b \rangle$ is a *result* for the pattern P_b. Note that we only consider minimal μ mappings, i.e. mappings that only map variables used in P_b. The M_b graph still has to be properly decorated according to the pointers that were assigned to the pattern P_b. We add decorating triples to the set M_b in the way described in the Chapter 5.1 – if a pointer was assigned to a node or edge of the pattern, we add the appropriate decoration to its image under $\mu(P_b)$.

7 Algebraic operations

This section describes algebraic operations, that run on the decorated data and further filter and transform it. Although, strictly speaking, these operations could be run on undecorated data, but there is usually no reason to do so. In such case, each node and edge of the undecorated data would be decorated by the first type of decoration triples, which would assign the whole data to one graph.

We do not give formal definitions for these operation as they are quite straightforward and usually obvious. They would only add a few pages of not very interesting technicalities to the paper.

7.1 Selection

The selection operation has one argument and tests a condition for each graph g in the argument multiset. If the condition is true, the graph is added to the result. The basic idea is the same as in relational algebra, but there is some added complexity due to the fact that one pointer can have more than one value (within one graph) or no value at all. We use a language derived from the first-order predicate calculus to construct the expressions. The main difference from SQL is the addition of quantifiers. The quantifiers are always in the form $Q_{x \in X}$ where x is a variable, X a pointer from the schema of the operand and Q either $\forall$ or $\exists$. The rest of the expression is formed from the variables, functions, predicates and logical operators. There are some limitations. One variable cannot be used in more than one quantifier and whole expression must be closed (meaning that there is no unquantified variable and no variable is used outside of the range of the quantifier for that variable). $\forall_{x \in X}$ denotes that the quantified condition must be true for each x from $PtrVal(X, g)$ and $\exists_{x \in X}$ denotes that there mast be at least one x in $PtrVal(X, g)$ such that the quantified condition is true. $PtrVal(X, g)$ denotes values of all nodes pointed to by X in the graph g and predicates of all edges pointed to by X in the graph g.

Example: $(\exists_{p \in ptr:Payment}(p > 1000)) \wedge (\forall_{r \in ptr:Person} \exists_{c \in ptr:Customer}(r = c))$. This means that the graph must have a value p for variable $Payment$ that is more than 1000 and that for each value of $Person$ there is the same value for $Customer$.

The formal definition is very strict, but the actual query language can be more relaxed, allowing the user to write less verbose queries as long as there is a clearly defined transformation to the form defined here.

We do not attempt to list all functions and predicates. In general, we assume that there is always a (hidden) parameter that carries the currently processed graph as its value – in the strict definition of the model it is a pair containing the whole graph and subset identifier from the namespace `graph`.

Some of the functions and predicates we would like to include are:

- PathLength(x, y) that return length of a path between nodes x and y.
- Sum(A), Max(A), Min(A), Count(A) that return the sum, maximum, minimum or number of nodes that the pointer A points to.
- IsURI(x), IsLiteral(x) that check, whether the value of x is of the specified type.

- TypeOf(x) that returns the data type of the literal x.

A detailed proposal of the language would include several basic functions and data type conversion rules as well as extension mechanism that would allow implementations to add further functions.

7.2 Projection

The purpose of the projection operation is to remove unneeded parts from the data. It has one argument and the operation is specified by a set S of URIs from the namespace `ptr`. Any reasonable set S should be a subset of the schema of the argument, but it is not required. The operation removes triples from its argument. There are two versions – induced and non-induced.

The non-induced version removes all non-decoration triples that are not pointed to by a member of S and all decoration triples that represent pointers not in S. Then all decoration triples g `dec:contains` o (graph membership) are removed if there is no triple p g o where $p \in S$. Note that if we remove a decoration triple that decorated an edge, we remove the three reification triples as well.

The induced works the same as non-induced except that it does not remove edges, where both endpoints are being pointed to by members of S.

The Figure 4 gives an example where $S = \{Person, Mail\}$.

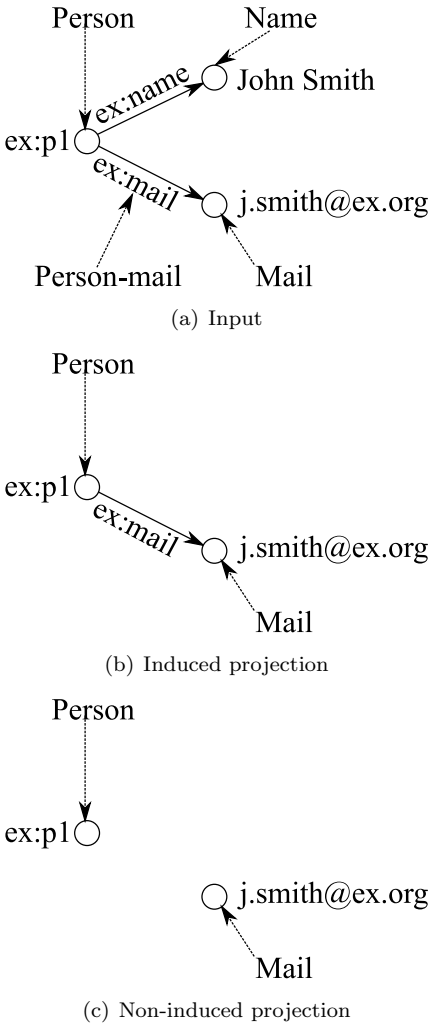


Figure 4. Projection

7.3 Distinct

So far, each operation generated a multiset of graphs. The distinct operation takes one argument and eliminates all duplicates. The equivalence of the graphs does consider decoration as well, i.e. for two graphs to be equal, even the pointers in both graph must point to equal nodes and edges.

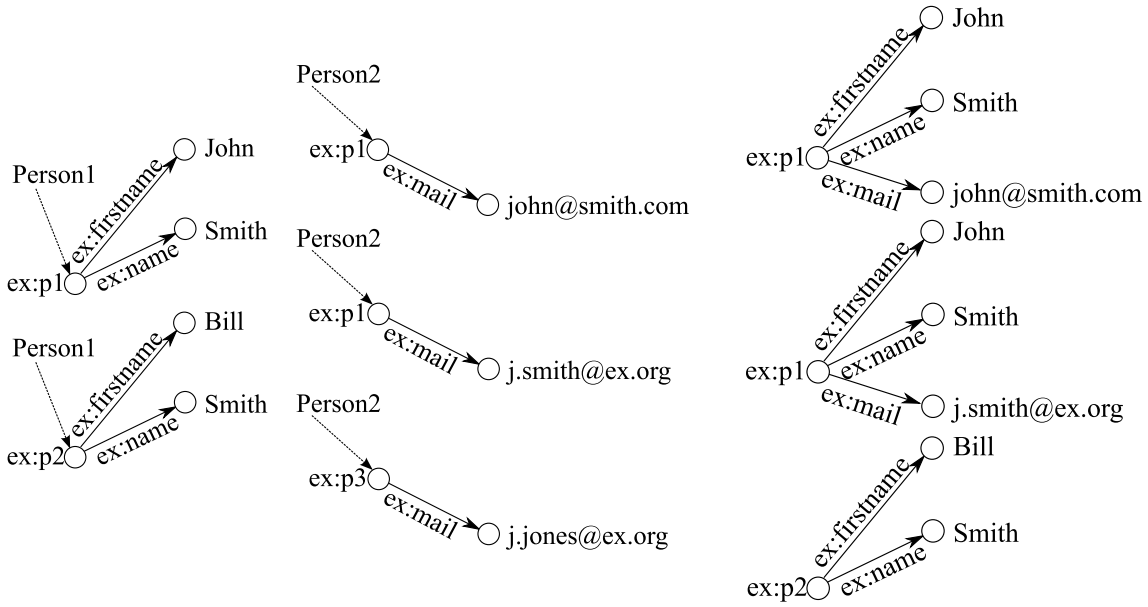
7.4 Joins

Joins are an important part of relational algebra and SQL. As we are trying to get close to these languages, we also introduce join operations. Join is a binary operation that produces results by making a Cartesian product of the arguments and then filters the results according to a condition. There are special variants of the operation – outer joins (left, right and full). The basic (inner) join could be defined as a combination of cross join (i.e. Cartesian product) and selection, but the outer joins are more complex so we have decided to include "whole" join operation.

The join can be seen as an operation that generates one small RDF graph for each pair of graphs where one is from the first argument and the other from the second argument. The graphs are union-ed together and if the produced graphs fulfills the join condition, it is added to the result.

The outer joins work just like in SQL. Consider for example left join. If there is a graph l in the left argument such that there is no graph r in the right argument that $l \cup r$ fulfill the join condition, then l is added to the result.

An example of a left join is in the Figure 5. The left and right operands are joined by a left join on a condition $Person1 = Person2$ (of course, the actual condition should contain the appropriate quantifiers).



(a) Left operand (b) Right operand (c) Result

Figure 5. Left join of names and e-mails

7.5 Group by

In SQL, the "group by" construct is almost exclusively used with aggregation. But as our data model allows more values per "row", we can use "group by" as a standalone operation that groups related data together. To be more specific, it joins (makes a union) graph, that have the exactly the same values for a specified set of pointers.

The Figure 6 shows an example where two graphs are grouped by the value of the "Person" pointer into one graph.

7.6 Aggregation

A very important feature in SQL are aggregation functions. An important difference between our data model and SQL is that a pointer can point to more than one value within each graph. So there are two possibilities, where aggregation functions can be used. They can either aggregate values within one graph or make aggregations over whole data. We decided to allow both. A *local* aggregation has the form $fnc(ptr) \rightarrow res$ where fnc is an aggregation function (min, max, sum, count, avg and "distinct" variants of sum, count and avg), ptr is a pointer and res is a pointer. The aggregation function is evaluated for each graph g and for a result v, the triples `g gql:contains v` and `res g v` are added to the data.

The *global* aggregation has the form $fnc_g(fnc_l((ptr)) \rightarrow res$ where fnc_g and fnc_l are from the same set of functions as in the local variant. The function fnc_l is used to compute aggregation over each graph and then fnc_g combines these results into one final value v. The result contains only one graph g with triples `g gql:contains v` and `res g v`. Because the data are "destroyed" more global aggregations can be specified in one aggregation operation.

An example that demonstrates why we decided to define global aggregation like this is the following. Consider the already familiar data about people and e-mails. We can use a global aggregation $max(count(Mail)) \rightarrow MaxMail$ to get the maximal number of e-mails the people have. Then, on the same source data, we run a local aggregation $count(Mail) \rightarrow MailCount$. Then we join the data on $MaxMail = MailCount$ to get information about everyone with maximal number of e-mails. Note, that since we included the aggregation functions among the function that can be used in the selection operations, we could omit the local aggregation step in the example and use $MaxMail = \textsc{Count}(Mail)$ as the join condition.

7.7 Set operations

Some set operations are also present in SQL – union, union all, intersect, and minus. The equivalent of union

(a) Input

(b) Grouped data

Figure 6. Group by

all is obvious, it simply returns union of the two graphs (since we require that graphs are identified by globally unique identifiers, there can be no "collision"). The is no such natural equivalent for the other three operations. The problem is that in our data model, "schema" is much more relaxed concept than in SQL – multiple values of a pointer, nodes and edges without pointers or with several different pointers, etc. Should the set operations be performed only with the data in graphs without regard to pointers (and what would be the pointers of the result) or with pointers? Perhaps it would be best to let the user specify a set of pointers and the operations only consider nodes and edges with these pointers.

But this creates a completely new problem. If we make an intersection of two sets and each of them contains a graph that is different, but the values pointed to by the specified set of pointers are the same. What should get into the result? The first or the second? That would make the intersection operation non-commutative.

We decided to use the following definition for the operations (A and B are operands, S is a set of pointers, $g[S]$ denotes a projection of graph g to the set of pointers S):

- $A\ union_S\ B$ is a shortcut for grouping operation with columns S applied to $A\ unionall\ B$
- $A\ minus\ B$ are all graphs g of A such that there is no graph g' in B for which $g[S] = g'[S]$ holds. Projection $g[S]$ is either induced or non-induced – the exact version is specified by the user.
- $A\ intersect\ B$ is not included as an operation. Although we came up with several possible semantics, none of them seemed more natural than the others. This and the fact that they could all be transformed into some combination of other operations led us to the decision not to include any of them as a build-in operation.

7.8 Constructors

Transformation from one RDF graph to a different one is a big problem for all RDF query languages. Many of them contain some kind of CONSTRUCT concept – a graph pattern is specified and new RDF graphs are generating by substituting each row of the query result into the pattern. Since we have no rows in the result (only an analogy that cannot help us in this case), we cannot use this approach.

Such operation would be extremely useful addition to our query system, since we could use it anywhere in the query and immediately perform other operations on the transformed data. Unfortunately, we have yet to find a simple and convincing definition for the operation. It is one of our immediate goals, perhaps the most important one.

7.9 Implementation concerns

We have defined operations of a RDF query system. We have not defined a query language, nor are we going to do so in this section. However important it may seem, it is in fact only a technical problem of defining a suitable textual representation for the presented operations. It has to be done and it has to be done carefully, since a bad language with complex grammar that makes it unclear and unreadable would certainly

discourage developers from using the whole query system, no matter what the underlying operations and data model are.

From the database point of view, the implementation of the query system would probably be more complex than in the case of the table-oriented RDF query languages, that can often be easily transformed into SQL queries over some representation of the RDF triples in a relational database. Although there are some significant performance issues involved, they can be greatly reduced. And the huge amount of work and money that have been spent on improving reliability and performance of RDBMS products are a great advantage of such solutions.

Although storage and transaction handling for an implementation could most likely be built on top of some existing solution, query processing and optimization will have to be written from scratch.

8 Performance tests

We have made three sets of tests. The first one was a comparison between load time into one of existing RDF repositories based on a relational database and Trisolda data store that is also based on relational database. The second one was designed to predict the load time curve for large semantic data and the last one compared query times between Trisolda RDBMS-based and non-RDBMS-based data stores. Tests used different data described in Table 1.

8.1 Test environment

The test environment consist of two machines. The first one hosts a Oracle db server (2xCPU Xeon 3.06 GHz, DB instance was assigned 1.0 GB RAM) and the second one is an application server (2xCPU Quad-Core Xeon 1.6 GHz, 8GB RAM).

All tests used relatively large data containing 2.365.479 triples (303 MB Turtle [1] file).

Name	Description
DATASET_1	2.365.479 triples, 184.461 URIs, 53.997 literals, 303 MB Turtle [1] file
DATASET_2	26.813.044 triples, 2.020.212 URIs, 1.043.337 literals, 3396 MB Turtle file

Table 1. The data used in tests.

8.2 Data import

The main goal of this test was to compare Trisolda data store with an existing solution based on a relational database. As an example of an existing Semantic Web data store was chose the Sesame v1.2 due to its popularity in the Semantic web community. New version of Sesame (Sesame v2.0) doesn't support relational databases. Both Sesame-db and Trisolda data store were connected to a local instance of Oracle database.

We tried to load 150 000 triples DATASET_1 into both of them. The Trisolda data store loads this data in 780 seconds. The Sesame-db finished loading near 118 000 loaded triples and failed with a database error. The error reported was low space in the TEMP tablespace.

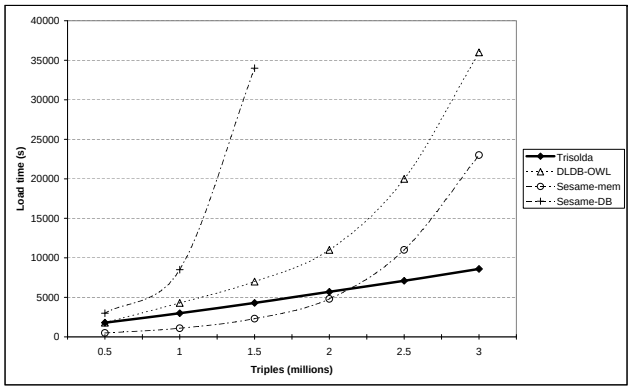


Figure 7. Data import comparison

Load times for the Sesame-db and the Trisolda data store are shown on Figure 7. The load time of Trisolda data store has almost linear dependency on the size of the processed data, but the Sesame-db exhibits rather exponential growth. The behavior of Sesame-db is expected and it is the same as described in [16].

One of the major design goals of Trisolda was storing huge semantic data. On the other hand, the Sesame database schema and SQL statements are not very suitable for loading huge data.

According to the test, smaller data (up to 110 000 triples in the machine configuration we used) may be loaded in Sesame-db, but it is not suitable to use the Sesame-db for larger data.

8.3 Huge data load

The main goal of this test was to determine whether the Trisolda data store is capable of loading huge RDF data (DATASET_2). During the implementation, we tried to identify possible bottlenecks and were able to

eliminate some of them. There are still some performance issues; it is one of the subjects of our further work.

The data was loaded in 100k triples batches. Whole load took 22 hours and 54 minutes, out of which 13 hours and 44 minutes were spent transferring data from source data file to temporary tables in the database and another 30 minutes were spent on cleanup actions.

8.4 Query performance

Although we have tried to implement the algorithms used in query evaluation in an efficient manner the algorithms themselves are only basic versions so the performance of the query evaluation leaves a lot of space for improvement.

We have tested three storage engines: BerkeleyDB based storage that stores triples in a B-tree, fully in-memory engine, Oracle-based RDF storage.

First, we measured the performance of evaluation of the query presented in section 4.3.

The BerkeleyDB-based storage engine required 1.8 seconds to complete the query, while in-memory engine took only 0.7 seconds. The performance of Oracle-based engine was the worst, requiring 6.4 seconds.

We have expected these results. The current in-memory engine is read-only and is optimized for best performance in queries similar to the one we tested. On the other hand, we used the Oracle database only to provide us with plain RDF triples and performed the join operations in our system. But this is not the main reason for the bad performance. The problem is, that the Oracle database is placed on another server and network delays for each returned triple add together. Had we used the Oracle database to join and filter the results the performance would have been much better due to smaller network trafic and better optimization of joins in Oracle. Our measurements showed that time required to evaluate this query is around 0.2 seconds.

8.5 Oracle query performance

We have performed several performance tests over our Oracle-based RDF store. The queries were completely translated to SQL and then evaluated by the Oracle server. An example of a very basic query (basic graph pattern with one triple) looks like this:

```
SELECT x_l.lit_rec_type AS x_kind,
      x_l.lit_value AS x_value,
      (SELECT lng_value
         FROM adt_lang
        WHERE lng_id = x_l.lit_lang_id)
       AS x_lang,
      (SELECT dtp_value
         FROM adt_data_type
        WHERE dtp_id = x_l.lit_type_id)
      AS x_type
FROM (SELECT x
  FROM (SELECT tri_subject_lit_id AS x
    FROM dat_triple t, dat_literal s,
         dat_uri p, dat_literal o
    WHERE t.tri_subject_lit_id=s.lit_id
      AND t.tri_object_lit_id=o.lit_id
      AND t.tri_predicate_uri_id
          = p.uri_id
      AND tri_predicate_uri_id =
         (SELECT uri_id
            FROM dat_uri
           WHERE uri_value
                 = :p1_predicate)
      AND tri_object_lit_id =
          (SELECT lit_id
             FROM dat_literal
            WHERE lit_value
                  = :p1_object))) q
      LEFT JOIN dat_literal x_l
             ON q.x = x_l.lit_id
```

In the following text, some queries are said to complete *instantaneously*. This means, that their evaluation time was comparable to the network latency (the database resides on a different server).

The queries in the following text are written as triples, where ?x denotes variable x, $<uri_1>$ denotes a URI with a value 'uri' and "value" denotes literal with value 'value'. The actual values are not given, as they are rather long and would be meaningless to the reader without deeper knowledge about the data used in the experiment.

The first query consists of basic graph pattern with one triple in the form ?x, $<uri>$, "literal". This query returned 10 rows and evaluated instantaneously.

The second query contained two triples: ?x $<uri_1>$ "literal1", ?x $<uri_2>$ "literal2". The query evaluated instantaneously and returned one row.

The next query was ?x ?y "literal". This query required 8 seconds to evaluate and returned 4 rows. On the other hand, the query $<uri>$?x ?y evaluated instantaneously returning 28 rows.

A more complex query ?x $<uri_1>$ "literal1", ?y $<uri_2>$?x, ?y $<uri_3>$?z, ?y $<uri_4>$ "literal2", ?y $<uri_5>$?w, ?w $<uri_6>$ "literal3" that returned only one row took as much as 200 seconds to evaluate. With the knowledge about the structure of the data, one could easily come up with an evaluation plan that would evaluate (nearly) instantaneously. But due to

the way that data are stored in the database, the statistics that the Oracle server utilizes are unable to provide this. Dealing with this problem will be one of the subjects of our future research.

All queries presented so far only returned small result sets. We also measured one query ?x <uri_1> ?y1, ?x <uri_2> ?y2, ?x <uri_3> ?y3 that returned 88964 rows. This took 70 seconds.

Another 'big' query was ?x <uri_1> ?y, ?z <uri_2> ?x, z<uri_3> ?w and produced 184179 rows in 66 seconds.

The main reason for relatively long evaluation times is not caused by transferring the results from the Oracle database over the network. This transfer is just a matter of seconds even for the largest result set. Most of the time was spent on the actual evaluation of the query by the Oracle database.

The experiments have shown, that queries like "give me first and last names of all people in the database" are much slower than what they would be if the data was stored in a traditional relational database. The fact that each triple is stored separately and table join has to be performed is one obvious factor. Less obvious but just as important is the fact that the statistics used by the Oracle optimizer to create query evaluation plans do not work well if the data is stored like this (all triples are stored in one table) and the optimizer makes wrong assumptions. This means, that the optimizer works with inaccurate estimations of the size of data at most places of the evaluation tree. This makes the optimizer select wrong order of the joins and also inefficient methods (like using nested loops to join large relations). The problems are very similar to those identified in [6].

9 Conclusion

We have implemented and thoroughly tested the infrastructure for gathering, storing and querying semantic data. We have focused our efforts on efficiency, extensibility, scalability and platform independence. Both our experiences and benchmarks show that this goal is feasible.

Trisolda is currently used as a platform for further web semantization research. We expect to enhance both interfaces and functionality to support these semantic experiments.

Our immediate goal is to implement the TriQ evaluator within the Trisolda environment.

We have two long-term goals. The first one is an implementation of a Semantic Web-specialized distributed parallel data-storage, which can significantly improve the behavior and performance of the Semantic Web repository.

As the second long-term goal, we plan to interconnect diverse semantic repositories, possibly with different implementation. Such interface-based loosely coupled network could become a nucleus of really usable semantic web, both for academic and practical purposes.

Acknowledgement

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 - XML Data Processing.

References

[1] D. Beckett. Turtle - terse rdf triple language, 2004. http://www.dajobe.org/2004/01/turtle/.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[3] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *Proceedings of the Workshop on Semantic Web Storage and Retrieval*, Netherlands, 2003.

[4] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference*, pages 54–68, Italy, 2002.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[6] J. Dokulil. Evaluation of SPARQL queries using relational databases. In I. Cruz, editor, *5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006*, volume 4273 of *Lecture Notes In Computer Science*, pages 972–973, 2006.

[7] J. Dokulil, J. Tykal, J. Yaghob, and F. Zavoral. Semantic web repository and interfaces. In *SEMAPRO07: International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 223–228, Los Alamitos, 2007. IEEE Computer Society.

[8] L. Galambos. Robot UDP Transfer Protocol, 2007. http://www.egothor.org/RFC/RUTP-v02.pdf.

[9] L. Galamboš. Dynamic inverted index maintenance. *International Journal of Computer Science*, 2006.

[10] J. Galgonek. Query languages for the semantic web. Master thesis at Charles University in Prague, 2008.

[11] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE03)*. IEEE, 2003. 1063-6382/03.

[12] M. Olson and U. Ogbuji. Versa, 2002.

[13] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Working Draft, 2005. http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/.

[14] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, pages 77–106, 2005.

[15] T. Vitvar, A. Mocan, M. Kerrigan, M. Zaremba, M. Zaremba, M. Moran, E. Cimpian, T. Haselwanter, and D. Fensel. Semantically-enabled service oriented architecture: Concepts, technology and application. *Journal of Service Oriented Computing and Applications*, 2007.

[16] S. Wang, Y. Guo, A. Qasem, and J. Heflin. Rapid benchmarking for semantic web knowledge base systems. Technical Report LU-CSE-05-026, CSE Department, Lehigh University, 2005.

[17] J. Yaghob and F. Zavoral. Semantic web infrastructure using datapile. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Itelligent Agent Technology*, pages 630–633, Los Alamitos, California, 2006. IEEE. ISBN 0-7695-2749-3.

[18] N. Zhang, V. Kacholia, and M. Tamer. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *Proceedings of the 20th International Conference on Data Engineering (ICDE04)*. IEEE, 2004. 1063-6382/04.

[19] W3C Semantic Web Activity Statement, 2001. http://www.w3.org/2001/sw/Activity.

[20] The Apache Software Foundation. http://www.apache.org.

[21] Microsoft Internet Information Services. http://www.microsoft.com/WindowsServer2003/iis.

[22] Jena A Semantic Web Framework for Java. http://jena.sourceforge.net/.

Modeling Security-Critical Applications with UML in the SecureMDD Approach

Nina Moebius, Wolfgang Reif, Kurt Stenzel
Department of Software Engineering and Programming Languages
University of Augsburg
86135 Augsburg, Germany
{moebius, reif, stenzel}@informatik.uni-augsburg.de

***Abstract*—Developing security-critical applications is very difficult and the past has shown that many applications turned out to be erroneous after years of usage. For this reason it is desirable to have a sound methodology for developing security-critical applications. We present our approach, called SecureMDD, to model these applications with the unified modeling language (UML) extended by a UML profile to tailor our models to security applications. We automatically generate a formal specification suitable for verification as well as an implementation from the model. Therefore we offer a model-driven development method seamlessly integrating semi-formal and formal methods as well as the implementation. This is a significant advantage compared to other approaches not dealing with all aspects from abstract models down to code. Based on this approach we can prove security properties on the abstract protocol level as well as the correctness of the protocol implementation in Java with respect to the formal model. In this paper we concentrate on the modeling with UML and some details regarding the transformation of this model into the formal specification. We illustrate our approach on an electronic payment system called Mondex [1]. Mondex has become famous for being the target of the first ITSEC evaluation of the highest level E6 which requires formal specification and verification.**

***Index Terms*—model-driven software engineering, UML, security, cryptographic protocols, verification**

I. Introduction

We focus on secure applications such as electronic ticketing or electronic payment systems. In this paper we concentrate on smart card applications. To guarantee the security of these (usually) distributed applications security protocols based on cryptographic primitives are used. Since it is very hard to design such protocols correctly and without errors, we propose to use formal methods for verification.

UML describes different views on various parts of a system. There exist several kinds of diagrams emphasizing different aspects of an application. In our approach we use use cases to describe the functional and security requirements of the system under development. Class diagrams are used to model the static view of an application. To design the protocols resp. to define the interaction steps between the components of the system we use sequence diagrams. To define the processing of messages and internal behavior of components we additionally use activity diagrams. The communication structure of the system and the abilities of the attacker are modeled by deployment diagrams. At the moment, we only model functional behavior, security properties are added on the formal level.

In the paper we only introduce the models showing the final view of the system which is used to generate code and the formal model. Of course, the creation of these models is a process that consists of several iterations and the UML diagrams evolve step-by-step. A disadvantage of UML is the lack of a comprehensive semantics directly usable in a verification system. This leads to difficulties for verification of models as well as for generation of code. This is solved by defining a mapping from the semi-formal to a formal presentation using abstract state machines (ASM) [2]. These have a well-defined and relatively simple semantics [3] [2]. Our formal specification is a combination of algebraic specifications and ASMs. Algebraic specifications are used for the description of the used data types as well as the attacker model. ASMs are used for the protocol dynamics. For verification we use the interactive theorem prover KIV [4].

Furthermore, we generate Java resp. Java Card code for smart card applications. Our group proposes a method to prove that an implementation is a refinement of the abstract formal model [5] by using the Java Calculus [6] [7] implemented in KIV.

The major advantage of our approach with respect to other existing techniques (e.g. [8]) is that we give a method seamlessly integrating modeling, formal methods as well as an implementation.

In this paper we describe the first part of the development process, i.e. the modeling of the application with UML. It is an extended and improved version of [9]. Our approach is focused on an easy to learn, and intuitive way of building the required models, and abstracts from details of the formal specification or the implementation. To model internal behavior, we extend activity diagrams with a UML-like language and use a syntax that is close to the one of an object oriented programming language. Our approach provides an opportunity to generate a formal model as well as runnable code without paying attention to the specifics of the formal specification and implementation which are harder to create and understand than the UML models.

Section II gives an overview of our SecureMDD approach. In Section III the SecureMDD UML profile and the used security data types are presented and a short introduction to

the Model Extension Language (MEL) is given. Our modeling technique is illustrated by an electronic payment application called Mondex that is introduced in Section IV. In Section V we present the modeling of a security-critical application on the platform-independent level in detail and describe the platform-specific model in Section VI. Section VII gives some details about the MEL syntax and grammar. In Section VIII we shortly address some specifics regarding the generation of Java Card code, Section IX exemplifies some details of the transformation from UML into the formal model. Section X addresses related work and Section XI concludes.

II. The SecureMDD Approach

In this section we give an overview of our framework which aims to develop secure applications (see Fig. 1). The approach is based on model-driven software development (MDSD) methods. The developer creates a UML model of the system under development. Then, several model-to-model (M2M) and model-to-text (M2T) transformations are applied and finally, Java(Card) code as well as a formal model are generated.

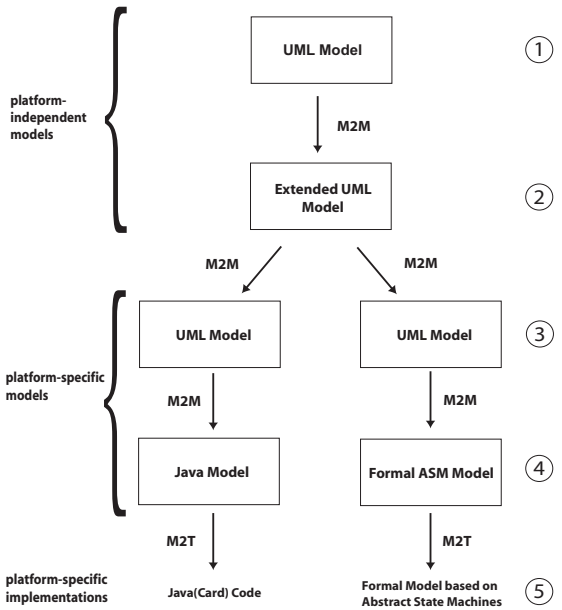


Fig. 1. Overview of the SecureMDD Approach

The approach starts with the modeling of a security-critical application with UML. We model the complete application, i.e. the static view, the structure of the system as well as the dynamic parts of an application. Since UML does not provide abilities to model the whole dynamic view, we extend the UML, especially UML activity diagrams, by a language called Model Extension Language (MEL). This language allows for modeling of e.g. assignments and creation of objects.

In the first step, the developer creates a UML model of the system under development (①). This model is platform-independent, i.e. it does not contain any specifics regarding the formal model or Java(Card) code. To model the flow of information and the processing of messages, activitiy diagrams extended with MEL expressions are used.

In a next step, the MEL expressions are parsed and stored in an abstract syntax tree. The 'Extended UML Model' is an instance of the UML metamodel which is extended by an abstract syntax tree of the MEL language (②). The generation is done automatically using model-to-model transformations.

Afterwards, as well with model-to-model transformations, different platform-specific models (PSM) are generated (③). On this level, the UML meta model is used. On the one hand, a model showing the smart card specific information is generated. This includes primitive types used in Java Card, Java Card expressions in activity diagrams as well as the translation of the stereotypes used in the previous model to Java classes. More details about the PSM can be found in Sect. VI. A smart card application always consists of one or more cards as well as a terminal with a card reader that communicates with the smart card. The terminal can be implemented using any programming language but Java is used in our approach. Since in this paper we concentrate on the modeling of the smart card part of an application, we omit the platform-specific model for generating the terminal code. On the other hand, we generate a platform-specific model containing details regarding the formal model which is based on algebraic specifications and abstract state machines (ASM). The expressions given as a MEL model are translated into syntactically correct ASM rules.

In a next step, a 'Java Model' resp. a 'Formal ASM Model' is generated from the platform-specific models. The Java model is an abstract syntax tree of Java whereas the ASM model is an abstract syntax tree of ASMs. Then, in a model-to-text (M2T) transformation, these models are transformed into Java Card code resp. a formal specification (⑤). The latter can be used to prove security properties of the modeled application using our interactive theorem prover KIV [10]. For hand-written formal models we already developed a method to prove security properties [11] [12].

The model-to-model transformations are implemented with the language QVT [13] and all model-to-text transformations with XPand [14].

III. The SecureMDD Profile and the Model Extension Language

In this section some security related data types and a UML profile which is tailored to cope with specifics regarding security-critical smart card applications are introduced. Furthermore, the Model Extension Language (MEL) that is used to extend UML activity diagrams is explained.

A. Predefined Security Datatypes

To model a security-critical application with UML it is expedient to define a few data types that are useful in these applications. Figure 2 shows the data types defined for the SecureMDD approach.

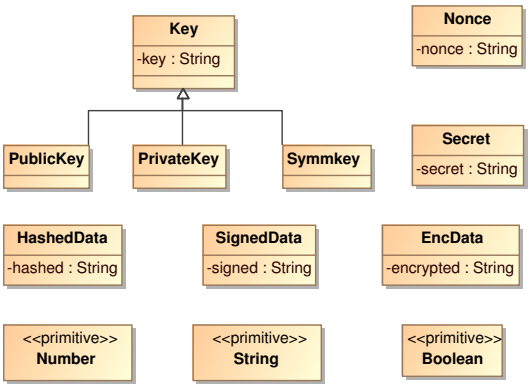


Fig. 2. Security Datatypes defined for the SecureMDD Approach

One important aspect is the use of keys. Thus, we define an abstract class `Key` that contains a cryptographic key. To capture the difference between symmetric and asymmetric encryption, i.e. public and private, keys, three subclasses of the `Key` class exist. Furthermore, a class `Nonce` representing nonces, i.e. random numbers used only once, is given. For example, nonces are used in cryptographic protocols to avoid replay attacks. Besides we define a type `Secret` which contains values that have to be kept secret, e.g. pin numbers or pass phrases. We explicitly distinguish secrets from primitive strings because this simplifies the formal verification of security properties. The classes `HashedData`, `SignedData` and `EncData` represent data that is hashed, digitally signed resp. encrypted. To facilitate the modeling on an abstract level without committing to an implementation language we additionally use primitive classes called `Number`, `String` and `Boolean` that represent numbers, strings as well as boolean values.

B. The SecureMDD Profile

Since UML is designed only to model standard application scenarios there is a need to extend it to specific application domains. For this reason the Object Management Group (OMG) [15] provides a mechanism to extend the scope of UML in a lightweight way by defining UML profiles. A profile extends the UML meta model and defines a set of stereotypes, tagged values and constraints.

In this section the SecureMDD UML profile is introduced.

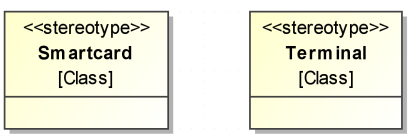


Fig. 3. UML stereotypes defining the components smart card and terminal

Figure 3 illustrates the stereotypes defined for the components of a smart card application, i.e. one stereotype to annotate a class representing a smart card and one stereotype to label a class representing a smart card terminal with a card reader. These stereotypes are used in class diagrams to describe the static view of the application as well as in deployment diagrams to define the structure of the system. In deployment diagrams we use the meta model element *Node* to describe the components of the system. Since the *Node* element is derived from the meta model element *Class* it is sufficient to extend the meta class *Class* with the stereotype.

In the SecureMDD approach the message types exchanged during a protocol run are modeled as classes instead of operations. This is motivated by the fact that data in smart card applications is sent from resp. to the card in the form of sequences of bytes. Thus, the idea is to have a message as a (serialized) object instead of a remote method call. In Figure 4 the stereotypes annotating message classes are given.

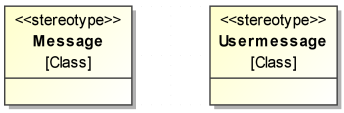


Fig. 4. UML stereotypes annotating message classes

Here, we distinguish message objects exchanged between the card and the terminal and message objects sent from the user of the system to the system, for example by entering data using a GUI. The latter is explicitly modeled because for verification we need a formal model of the whole application, including the user inputs. Since the messages are defined in the class diagram, the stereotypes extend the meta class *Class*.

Figure 5 shows the stereotypes to label data classes and constants.

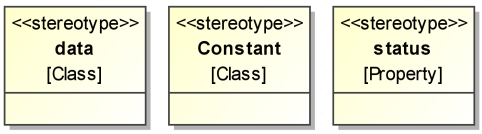


Fig. 5. UML stereotypes defining data, constants and status

These classes extend the meta class `Class`. Classes annotated with stereotype ≪data≫ are non-cryptographic data types. Classes not annotated with any stereotype are considered as ≪data≫ data type. To define constants used in the models the stereotype ≪Constant≫ is used. The stereotype ≪status≫ indicates the state of a component. While executing a protocol it is often essential to keep track of the step in the protocol that must be executed next. Depending on this step, the component may react differently by processing the next message or abort if the received message differs from the expected one. All possible states are modeled as an enumeration. An association between the component class, i.e. the terminal or the smart card, to the state class (annotated with stereotype ≪status≫) indicates the state of the component.

Figure 6 shows the stereotypes defined for digital signatures, encryption and hashing.

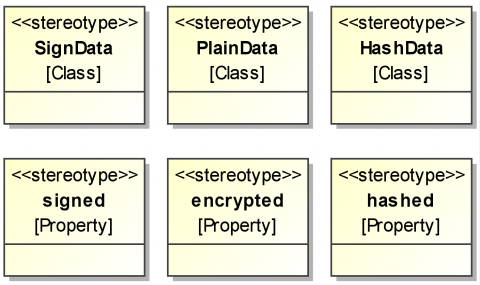


Fig. 6. UML stereotypes for encryption, hashing and signatures

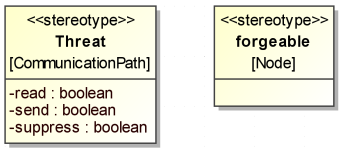


Fig. 7. UML stereotypes specifying the attacker capabilities

If data modeled in the diagram is going to be signed, encrypted or hashed, it is annotated with stereotype ≪SignData≫, ≪PlainData≫ resp. ≪HashData≫. These stereotypes extend the meta class `Class`. Furthermore, we define stereotypes that denote the signing, encryption resp. hashing of data. If data is going to be encrypted during a protocol run, the data class is marked with stereotype ≪PlainData≫. If this data is encrypted and the result stored in a field of, e.g. the smart card or a message object, the corresponding association between this object and the PlainData object is annotated with stereotype ≪encrypted≫. In the class diagram we do not specify which key is used for encryption. Since this is a dynamic aspect, the concrete encrypt operation including the specification of the used key is specified in activity diagrams. Note that the generation of the formal model and Java Card code would also be possible if we omit the use of the stereotypes ≪SignData≫, ≪PlainData≫ and ≪HashData≫, i.e. all required information is already given when using the remaining stereotypes. However, we feel that it is good practice to use them because they increase the readability of the platform-independent models.

To verify certain security properties that have to hold for the modeled system it is necessary to describe a possible attacker resp. his abilities. An attacker may be able to interfere with the communication between smart card and terminal. This can be modeled appropriately with deployment diagrams. We use the communication path element to annotate the capabilities an attacker has to affect the communication. For this purpose we define the stereotype ≪Threat≫. The stereotype has three tags `read`, `send` and `suppress` that indicate if the attacker is able to read messages sent over that path, send or suppress messages. In some scenarios an attacker may try to forge a component, e.g. he may program his own smart card. If a fake component is conceivable it is annotated with stereotype ≪forgeable≫. The stereotypes defined to describe the attacker are shown in Figure 7.

C. The Model Extension Language (MEL)

The Model Extension Language (MEL) is used to extend activity diagrams. It is a simple language whose expressions are used in `Action` elements, `SendSignalActions`, `AcceptEventActions` as well as in `guard`s to model e.g. object creation, assignments, conditions, or the sending of a message. The aim is to have a language that can be used to model cryptographic protocols and at the same time is more abstract than a programming language. For example, MEL has a copy semantics and the developer does not have to take care about memory management and object creation which must be handled with care on smart cards. Since MEL is tailored to model the protocols of security-critical applications, it contains several keywords resp. predefined methods to express e.g. encryption, decryption, the generation of nonces and hash values. More details about MEL are given in Section VII.

IV. MONDEX

The SecureMDD approach is illustrated with the Mondex application which is introduced in this section.

Mondex cards are smart cards that are used as electronic purses with the aim of replacing coins by electronic cash. Mondex is owned by Mastercard International [1]. The main field of application is the secure transfer of money from one smart card to a second card. To perform a transfer both cards are inserted into a smart card terminal that also acts as user interface. The security properties that have to be verified for Mondex are that no money can be created and any value must be accounted for. In detail, this means that no money can be loaded onto a Mondex card without subtracting it from another card. Furthermore, if a transaction fails, no money should be lost. The Mondex case study recently received a lot of attention because its formal verification has been set up as a challenge for verification tools [16] that several groups [17] as well as our group [18] [19] worked on. For Mondex, several approaches dealing with formal methods and verification (model-checking, theorem proving and constraint solver) exist. But, they are not combined with an engineering discipline for system development. Rather, they use only formal techniques for specification and verification of the Mondex application. In the SecureMDD approach software engineering techniques and formal methods are integrated.

The Mondex application is another example that the design of security-critical systems is difficult. While verifying the security of the application our group has found a flaw in the original protocol [16]. Exploiting this flaw it is possible to cause a denial of service attack that fills the memory of the card. In this state the card is disabled unless the owner returns it to the bank. More details about the flaw can be found in [18]. The protocol given in this paper is a slight modification of the original protocol introduced in [20] and avoids the denial of service attack.

V. Modelling of security-critical smart card applications with UML

In this section our method to develop a security-critical application is introduced. All steps and artefacts are exemplified by the Mondex application. In subsection V-A the description of functional and security requirements using use cases is given. In subsection V-B our methodology to describe cryptographic protocols on a very abstract level is introduced. In subsection V-C the modeling of the static view using class diagrams is presented, subsection V-D describes the specification of the dynamic behavior using activity diagrams and the Model Extension Language. Subsection V-E introduces the modeling of the communication model as well as the attacker abilities using deployment diagrams.

A. Use Cases describing functional and security requirements

Use cases are used to capture functional requirements of the system in an informal way. As in a traditional software engineering process one or more use cases are written that describe the interaction between the system and external actors or systems. They describe the application in a way that can easily be understood. In our modeling method, use cases are the basis for the sequence and activity diagrams that are used to build the formal model as well as executable code. Below five of the use cases for Mondex are given. The first one, Person-to-Person Payment, is then used as running example in the following subsections.

Person-to-Person Payment
Basic Flow:

1) The customer of a shop wants to pay with his Mondex card.
2) He as well as the shop owner insert their cards into the corresponding card reader.
3) The shop owner enters the amount to pay.
4) The customer confirms the amount and starts the transfer of money.
5) The entered amount is transferred from the card of the customer to the card of the shop owner.
6) The system confirms the transfer by returning a receipt.
7) Both participants remove their cards from the reader.

Alternative Flows:

- 3) The entered amount is wrong: The shop owner cancels the process.
- 4) The customer does not agree with the entered amount: He cancels the transfer and the system aborts.
- 5) The balance of the customer card is lower than the amount to pay: The systems aborts and returns an error message.
- 5) The entered amount added to the current balance of the shop card exceeds the maximum value that can be loaded: The system aborts and returns an error message.
- 5) An error occurs while transferring the money or one of the participants removes his card too early: The system aborts and returns an error message. If the amount was already reduced on the customer card but has not been added to the card of the shop owner this is recorded on both cards. To recover the original balance of the customer card both cards have to be shown at the bank (see use case ”Recovery of Money”).

Security Requirements:

- No money is lost: If a transfer fails, either no money is charged from the customer card or if money was already charged it can be recovered correctly.
- An attacker is not able to program his own card such that he can use it as customer card and pay with it.
- It is not possible to load money onto a card without subtracting the same amount from a second card, i.e. no money can be created.
- It is not possible that a shop owner debits a higher amount than has been agreed.

Payment using Internet
Basic Flow:

1) A customer wants to pay with his Mondex card using an internet shop.
2) He inserts his card into his card reader (which is connected to his PC) and opens the web presentation of the shop.
3) He selects the products he wants to buy, enters his postal address for shipment and selects that he wants to pay now.
4) A connection to the remote card reader of the shop owner is established. The Mondex card of the shop owner is in this reader.
5) The amount to pay is transferred from the card of the customer to the card of the shop owner.
6) The system confirms the transfer.
7) The customer removes his card from the reader.
8) The shop owner sends the goods to the customer.

Alternative Flows:

- 3) The balance of the customer card is lower than the amount to pay: The systems aborts and returns an error message.
- 4) The entered amount added to the current balance of the shop card exceeds the maximum value that can be loaded: The system aborts and returns an error message.
- 5) An error occurs while transferring the money or one of the participants removes his card too early: The system aborts and returns an error message. If the amount was already reduced on the customer card but has not been added to the card of the shop owner this is recorded on both cards. To recover the original balance of the customer card both cards have to be shown at the bank (see use case ”Recovery of Money”).

Security Requirements:

- see use case ”Person-to-Person Payment”

Recovery of Money
Basic Flow:

1) If a transaction fails (i.e. money was charged from the customer card but has not been added to the shop card) both participants of the transfer go to the bank.
2) Showing their Mondex cards it can be discovered if and what amount of money was reduced from the customer card.
3) The system adds the corresponding amount to the customer card.

Alternative Flows:

- 3) If the amount added to the current balance of the customer card exceeds the maximum balance of the card the amount will be paid out in cash.

Security Requirements:

- If money was lost it can be recovered only once, i.e. showing the cards again it is not possible to force a recovery again.
- It can be detected if the transfer has been aborted after the amount was added to the shop card. In this case no money is recovered.

Recharge of money at an automatic teller machine (ATM)
Basic Flow:

1) The card owner goes to the ATM (within his bank) and inserts his Mondex card.
2) The card owner specifies the details of his bank account.
3) He authorizes by entering his PIN number.
4) The system checks that the PIN is correct.
5) The card owner enters the amount he wants to recharge.
6) The entered amount is debited from the bank account of the card owner and loaded onto the card.
7) The card owner removes his card from the terminal.

Alternative Flows:

- 4) The entered PIN is not correct: The system returns an error message and asks for retry. After three times entering a wrong PIN the card is locked.
- 5) The balance of the owners bank account is less than the entered amount: The system returns an error message and requests to enter a lower amount.
- 5) The entered amount added to the current balance of the card exceeds the maximum value that can be loaded: The system returns an error message and requests to enter a lower amount.

Security Requirements:

- The amount loaded onto the card equals the one charged from the bank account. It is not possible to load money onto a card without reducing the bank account by the correct amount.

Discharge at an ATM
Basic Flow:

1) The card owner inserts his card into an ATM at the bank.
2) He selects that he wants to have repaid the money.
3) The ATM pays out the amount currently stored onto the card and sets the current balance of the card to zero.
4) The customer removes the card from the reader.

Alternative Flows:

- 3) The customer removes his card from the reader too early: No money is paid out.

Security Requirements:

- The amount paid out in cash equals the balance of the card.
- If returning the cash to the card owner the balance of the card is set to zero.

Other use cases cover the viewing of the last transactions, storing money of different currencies on the same card or payments using mobile phones. Also the recharge of money using the internet or the use of money in cash instead of a bank account for recharge is possible. Since the entire application is too large to present here we only model the transfer of money between a shop owner card and a customer card (Use Case Person-to-Person Payment).

B. The Protocol Description

Our goal is to give an intuitive way to model security protocols. A reader of the model should be able to understand the protocol without getting lost in details. We use sequence diagrams to specify the protocol steps and the flow of information. The idea is to start with a very abstract view of the possible protocols and refine these sequence diagrams step by step. The diagram shown in Fig. 8 shows the final sequence diagram for ”Person-to-Person Payment”. At this point the protocol which is later implemented is already elaborated. This diagram is used as basis to develop the complete dynamic behavior of the system using activity diagrams. Note that we do not show the diagrams that were drawn while working out the final models.

The sequence diagram contains one lifeline for each component participating in the protocol and additionally one lifeline for the ”user”. The user represents the customer of the service and usually initiates a protocol, i.e. ’sends’ the first message. For Mondex, we distinguish the card of the shop owner (in the following called *to* purse) and the card of the customer (in the following called *from* purse). Since a Mondex card can act as *to* card as well as *from* card this distinction is only to achieve a better readability of the diagrams.

The protocol used for payments between two persons (see Fig. 8) works as follows:

The user, i.e. the shop owner, initiates the protocol run by sending the value to be transferred to the terminal (`UTransferMoney`). Afterwards the terminal queries the *to* purse to provide its data, e.g. its name (= unique number), by sending the instruction `getData`. The *to* purse returns this data (message `ResGetData`). In a next step the terminal sends a message called `StartFrom` to the *from* purse which initiates the transfer on the `from` purse. This message contains all information required to start the transfer, i.e. the value to be transferred as well as the unique data of the other purse. Then, the *from* purse sends a `StartTo` message to

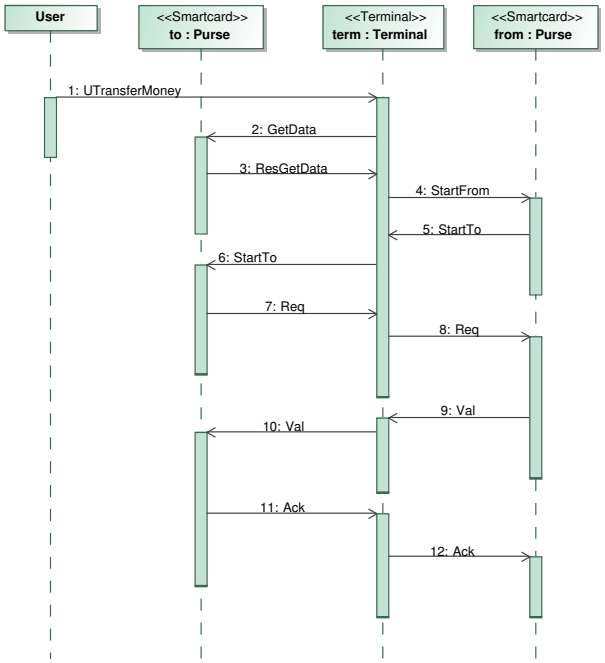


Fig. 8. Protocol Description for Person-to-Person Payment

the terminal which forwards it to the *to* purse. This message contains all data required to run a transfer and, after receiving it, the *to* purse initiates the transfer. Note that from now on the terminal only forwards message that it receives, i.e. if receiving a message from the *from* purse, it forwards it to the *to* purse without modifying the message or its state. In a next step, after checking that the received transfer information is correct, the *to* purse generates a `Req`(uest) message to request a transfer, i.e. requests the decrease of the balance of the *from* purse. After receiving this message the *from* purse decreases its balance and sends back a `Val`(ue) message which states that its balance has been decreased. Then, the *to* purse increases its balance and sends back an `Ack`(nowledgement) message that confirms the transfer.

C. Static View of the System

In the following the modeling of the static view of a smart card application is introduced. To model specifics regarding the domain of security-critical applications we use the UML profile as well as the security data types defined in Section III. The method is exemplified by the Mondex application but is applicable for smart card applications in general.

Fig. 9 illustrates the class diagram of the Mondex application. Note that the diagram only shows the part of the static view which is needed for Person-to-Person payments, other parts e.g. regarding the recovery or recharge of money are omitted.

Every component of the system, i.e. smart card and terminal, are represented by a class annotated with stereotype ≪Smartcard≫ resp. ≪Terminal≫. This distinction is necessary because the generated code (e.g. Java Card vs. Java) and the formal model differ depending on the type of component. In the Mondex application we have the class `Purse` which is representing the smart card as well as the `Terminal`.

The message types are modeled as classes. Here, we use an abstract class annotated with stereotype ≪Message≫ from which all concrete message classes are derived. In Fig. 9 several concrete message classes, e.g. `Req`, `Val` and `Ack`, are defined. Note that these messages are derived from the messages modeled in the corresponding sequence diagram (see Fig. 8).

All data types are modeled as classes and annotated with corresponding stereotypes, i.e. ≪data≫ for non-cryptographic data types and ≪PlainData≫, ≪HashData≫ and ≪SignData≫ for data that is going to be encrypted, hashed or signed. In the Mondex model we have defined the data class `PurseData` that consists of the unique name of the purse as well as a sequence number that increases after every protocol run and ensures the uniqueness of every `PayDetails`. A `PayDetails` object records the details of the current transaction, i.e. the participating purses as well as the amount to transfer. Furthermore, we define one class called `Msgcontent` that is going to be encrypted and thus annotated with stereotype ≪PlainData≫. This class contains the pay details of the current transaction and a message flag denoting if the (encrypted) data belongs to a `Req`, `Val` or `Ack` message. If this flag is omitted, the following atack is possible.

An attacker captures and suppresses a `Req` message and uses the contained encrypted data to send a correct `Val` message to the sender. Receiving this message, the sender of the `Req` message, i.e. the *to* purse, assumes that the *from* purse has decreased its balance correctly and increases its balance. Then, the balance of the *to* purse has been increased without decreasing the balance of the *from* purse.

Since an object of type `Msgcontent` is encrypted and afterwards sent with a `Req`, `Val` or `Ack` message, the corresponding associations are annotated with stereotype ≪encrypted≫. To denote the types of used attributes we use the self defined primitive types `Number`, `Boolean` as well as `String` and the security data types described in III-A. To cover associations with multiplicity greater than one we use a predefined list. For example, the `Purse` class has an exception log for failed transactions. This is modeled by an association with multiplicity 0..LOGLENGTH. This exception log is translated to a list that can be accessed with predefined methods e.g. to add an object to the list. These predefined operations are later used in the activity diagrams.

The possible states a component may be in are defined as an enumeration. An association from a component to this enumeration, annotated with stereotype ≪status≫ defines the states of a component. A purse may be in state `IDLE`, `EPR` (expecting request), `EPV` (expecting value) or `EPA` (expecting acknowledge). Since the terminal simply forwards messages to the cards and accepts all kinds of messages, it needs no state.

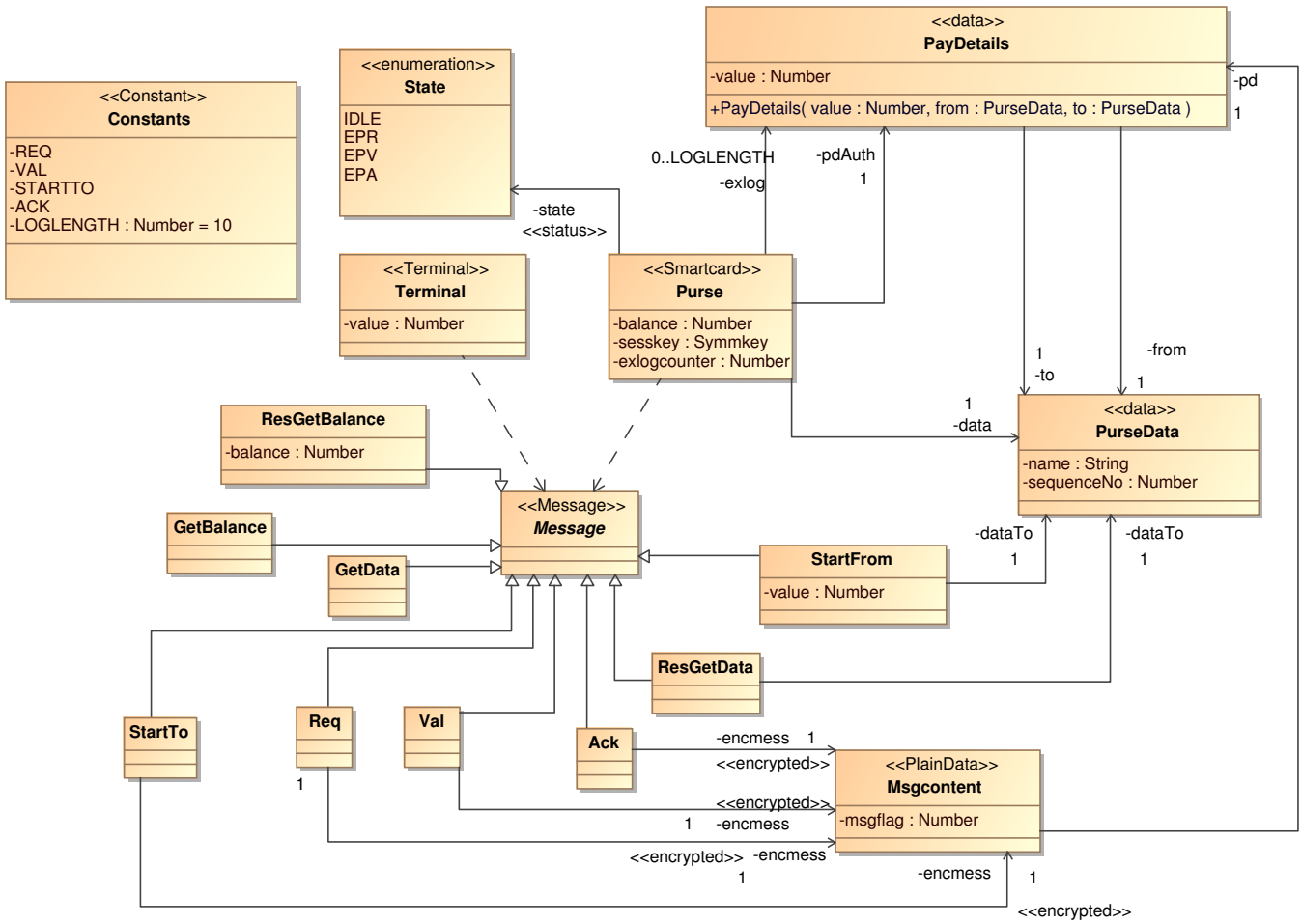


Fig. 9. Static View of the Mondex application

D. Dynamic Behavior

Sequence diagrams describe the sequence of messages that is exchanged between components but do not capture internal actions or the behavior in case an error occurs. For this reason we additionally use activity diagrams that extend the sequence diagrams and describe changes in the internal state of the components after processing a message. The activity diagram describes the communication as well as the sequence of actions taken as a result of receiving a message. At this point we use our Model Extension Language (MEL) which was shortly introduced in Section III. MEL allows to describe e.g. creation of objects, assignments or guards of conditions. We use activity diagrams instead of UML state diagrams because they turned out to be hard to read and confusing for applications we focus on (with many condition checks).

For each use case we define one activity diagram. For a better readability we additionally allow the definition of sub activities that are called within an activity. In Fig. 10 one part of the activity defining the protocol executed for Person-to-Person payments is given. The whole activity diagram can be found in the appendix.

For each component participating in the protocol one swim lane exists in the diagram. As in the sequence diagram we have a swim lane for the user, the *to* as well as *from* purse and for the terminal. A protocol can be divided into segments where one segment consists of one protocol step. A protocol step has the following parts: A component receives a message, performs several tests to check whether the message is correct and can be handled and processes the data. Finally, the component may send a message to another component. We use `SendSignalActions` to denote the sending of a message, `AcceptEventActions` to indicate the receiving of a message as well as `Action` elements to denote MEL expressions like object creation, assignments and calls of predefined operations.

The segment in Fig. 10 shows the swim lane of the terminal on the left as well as the one of the *from* purse. The terminal sends a `StartFrom` message to the *from* purse. This message contains the `value` to be transferred as well as the data of the *to* purse. The *from* purse receives this message. The content of

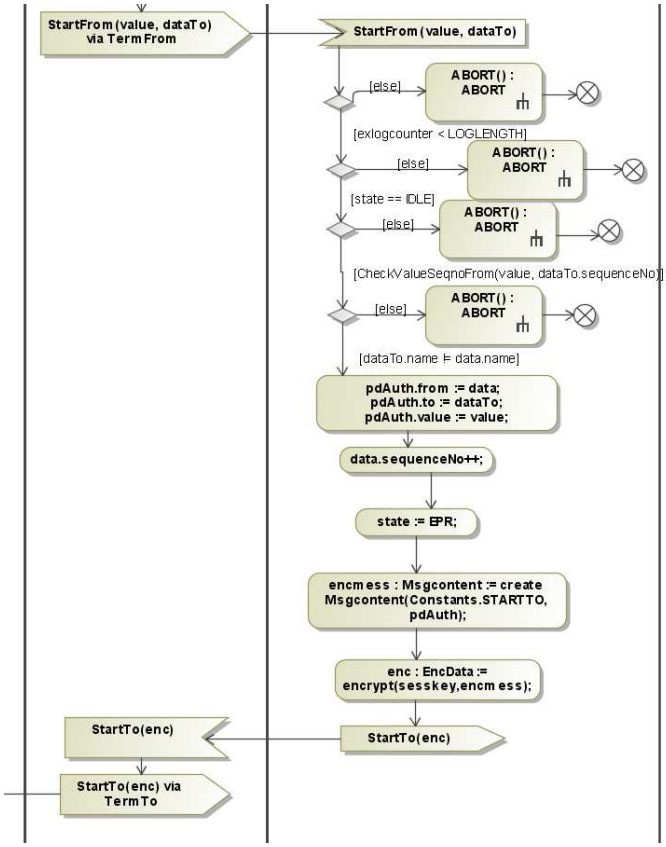


Fig. 10. Mondex Activity Diagram showing the sending, receiving and processing of a `StartFrom` message. On the left side one can see the swim lane of the terminal, on the right side the one of the *from* purse

it, i.e. the value and data, are handled as local variables. Then, the purse checks if the counter which counts the exception log entries is less than the possible maximum length. If not, the protocol aborts. The abort step is defined in a separate activity diagram and is called from this protocol (defined by a rake element). A sub activity has access to the properties of a component but not to the local variables. If the condition is satisfied it is tested whether the `state` of the purse is set to `IDLE`. Next, it is checked if the received value and sequence number of the *to* purse fulfill certain conditions, for example that the value to be transferred is greater than zero. These checks are also defined in a separate activity `CheckValueSeqnoFrom` which has two parameters and returns a boolean value with the result of the tests. Since a sub activity has no acces to the local variables, these have to be passed as arguments. The sub activity can be found in the appendix. If one of the checks fails the `ABORT` sub activity is called. Otherwise, the purse modifies some fields, e.g. the field `pdAuth` is filled with the current pay details, the purse's `sequenceNo` is increased and the `state` is updated to `EPR`. Our Model Extension Language has a copy semantics but updates of fields modify the fields of the original object. In a next step, a local variable `encmess` of type `Msgcontent` is created and, in the next action, encrypted with the symmetric key stored in field `sesskey`. The `encrypt` method is predefined in MEL and used for symmetric and asymmetric encryption. The result of the encryption of data is an object of type `EncData` that consists of a string containing the encrypted data (see Section III). This `EncData` object is stored in a local variable `enc`. Afterwards a `StartTo` message containing the created `enc` object is sent to the terminal. The terminal receives this message and forwards it to the *to* purse. The keyword `via` denotes to which components the message is sent resp. denotes the used port (see subsection V-E for more details). If the communication path is unique, e.g. the purse only communicates with the terminal, the `via` keyword can be omitted.

Activity diagrams are used to define the communication between the different components as well as the processing of a message, i.e. they are used to model cryptographic protocols. In applications with large protocols it may be desirable to add some code by hand after generating the modeled parts of the system instead of creating activity diagrams for the whole application. For this reason the developer can add own method calls where the corresponding method bodies are added later by hand on code level. Note that this causes problems resp. inconsistencies when verifying the security of the system using a formal model automatically generated from the UML models. To ensure that the security properties which are proved on the formal model also hold on code level, the formal model

has to be a suitable representation of the code. This means that all changes and additions which are made on the code (by hand) have to be done on the formal model as well.

E. Attacker and Communication Model

To verify cryptographic protocols it is necessary to formally specify the communication infrastructure as well as an attacker model. Almost all formal approaches (e.g. [21] [22]) for verifying cryptographic protocols use a rather simple model of communication and the Dolev-Yao [23] threat model. There, no constraints regarding the communication structure are given and it is assumed that the attacker may access all communication links, i.e. he can read all messages sent over that link, suppress them or write messages on that channel. In these approaches (mainly addressing internet protocols) it is ignored that certain components cannot communicate directly with other components for physical reasons.

Also, the possibility that some connections are secure against eavesdropping and others are not, is abstracted away. In contrast, our formal model is not limited to Dolev-Yao attackers. The main reason for an attacker model with reduced (but more realistic) abilities is that it becomes possible to have simpler protocols still preserving the desired security properties.

In our approach we explicitly model the existing connections. For each connection we denote if the attacker is able to read or suppress messages and whether he can send messages over that channel. But these annotations do not suffice to describe all possibilities an attacker might have. For example, an attacker could program his own forged smart card. If the protocol has a flaw such that the forged card takes advantage of the weakness of the protocol it may be possible that the attacker gains some information e.g. about secret keys.

UML provides the use of deployment diagrams to define the physical structure of a system. In our approach we use them to describe the communication structure as well as the attacker model of our application. Fig. 11 shows the deployment diagram for the Mondex application.

The components participating in the application are modeled as nodes. The terminal has one connection to the *to* purse, one to the *from* purse as well as one to the user. If a component sends a message it has to be determined which connection is used for sending. To be able to reference the connections the connection ends, also called ports, are named. If multiple connections exist between two components, the connection that is used for sending is addressed using the `via` keyword in the activity diagram.

For Mondex we assume that an attacker may have full access to the connections between terminal and cards. Thus, these connections are marked with `read`, `send` and `suppress`. Furthermore, an attacker may program his own smart card and use it as a Mondex card to attack the system.

VI. Platform-specific Smartcard Model

Based on the platform-independent model of the application a platform-specific model (PSM) is generated for each

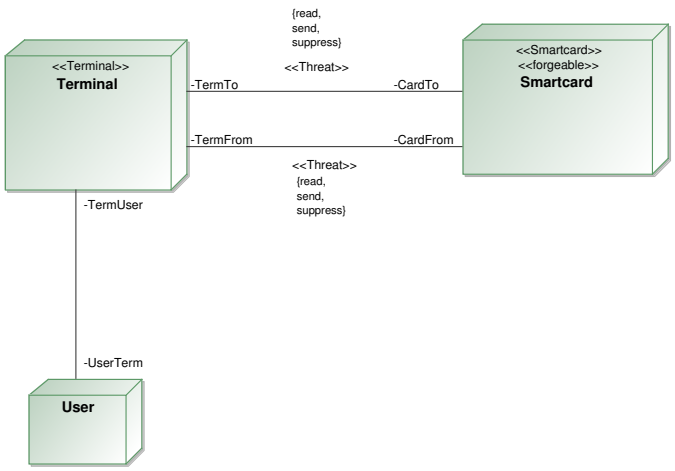


Fig. 11. Deployment Model for Mondex

platform. For the Mondex application, we distinguish three platforms: one for the terminal, one for the smart card as well as one for the formal model. In this section we present the static view of the platform-specific model for the smart card in more detail.

Figure 12 shows the platform-specific class diagram of the Mondex application.

In the class diagram the abstract data types for the smart card are replaced by Java Card [24] specific data types. Note that Java Card does not support integers or strings. Thus, all fields of type `Number` are translated to shorts, `String`s are translated into byte arrays and `Boolean` are replaced by the Java type boolean. Furthermore, for each class a constructor is added.

One main aspect of the PSM is the removal of stereotypes dealing with cryptography which were used in the platform-independent model. Instead, some classes and interfaces are added. The resulting class diagram is close to the structure of the Java Card code but omits some technical details. Remember that in the platform-independent model the encryption of data was modeled by adding a stereotype named ≪encrypted≫ to the corresponding association. The referenced class is then annotated with a stereotype ≪PlainData≫ which denotes that this data type can be encrypted. In the platform-specific model we add an interface called `PlainData` which is implemented by all classes that were marked as ≪PlainData≫ in the PIM. Moreover, we add the data type `EncData` that represents encrypted data. This class has a field `encrypted` of type byte array which stores the encrypted data. Since the Java Card Crypto API operates on byte arrays, it is of type byte[]. The class has two static methods, `encrypt` and `decrypt`, which correspond to the predefined methods of the same name defined in MEL. The `encrypt` method takes an object of type `Key` and a `PlainData` object and returns an `EncData`. The `decrypt` method operates on a `Key` and an `EncData` object and returns the decrypted `PlainData`. The classes `StartTo`, `Req`, `Val`

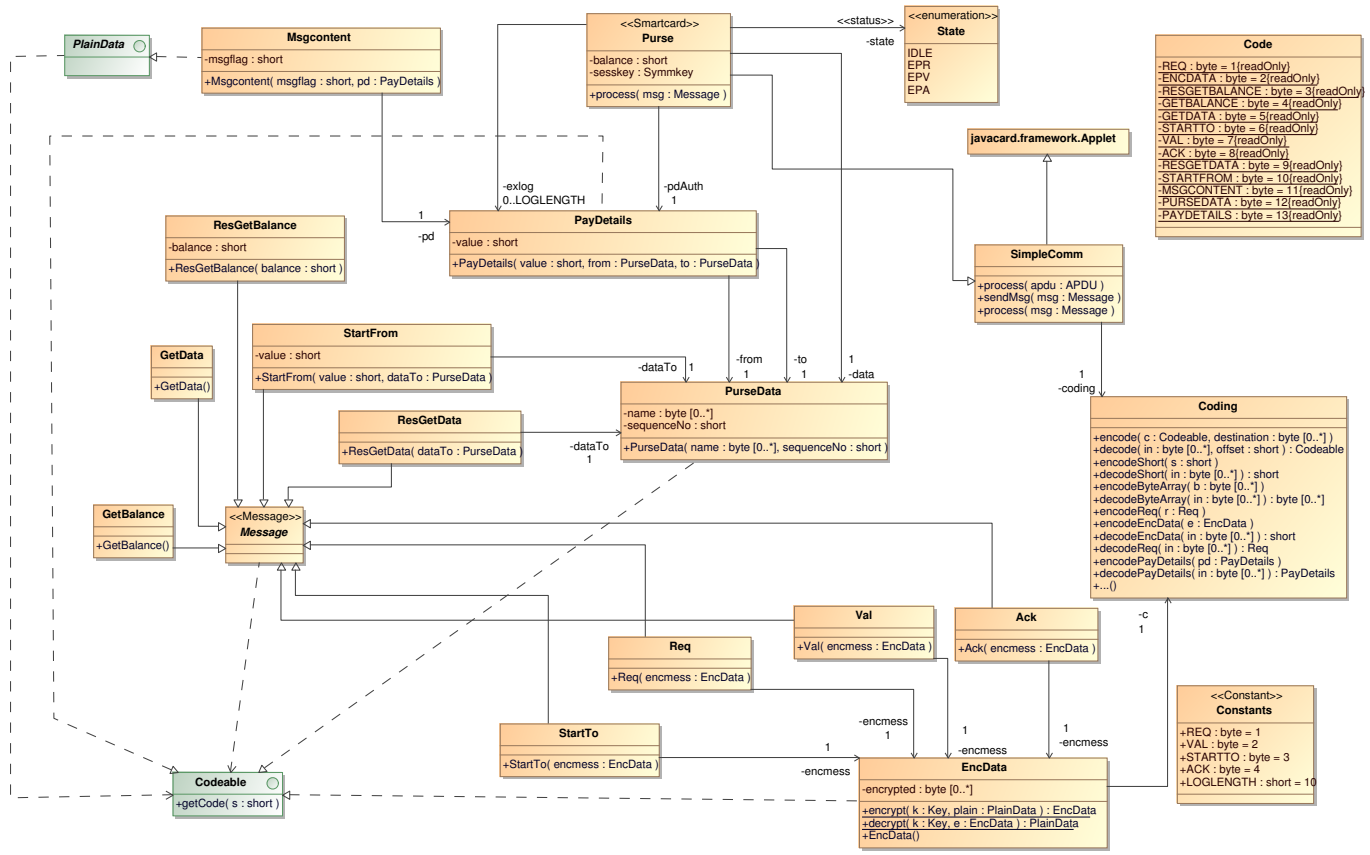


Fig. 12. Smart card-specific class diagram of the Mondex application

and `Ack` were defined in the PIM with associations to the class `Msgcontent` (annotated with ≪PlainData≫). Now, these classes have associations to the class `EncData` and hence reflect the implementation with Java Card.

To communicate with the terminal we add a class `SimpleComm` which defines two methods to receive and process a message as well as a method for sending a message. This class extends the class `Applet` defined in the Java Card API. The class `Purse` that represents the smart card extends the class `SimpleComm`.

Since the communication between card and terminal is based on byte arrays we additionally need a serialization mechanism that serializes the objects that are sent to the terminal. This is realised by a class named `Coding` that defines methods for serialization and deserialization of each object which is sent during a protocol run. More details about the implementation in Java Card as well as the generation of code can be found in [25].

The activity diagrams of the platform-specific model still have the same structure but the MEL expressions are parsed and replaced by Java Card expressions.

It is easy to see that our platform-independent model is an abstracted view of a security-critical smart card application that can be created without knowing technical details about programming with Java Cards. It is possible to model an application using the predefined stereotypes without thinking about a possible implementation. Then, in a next step, these abstract models are translated into more Java Card specific models automatically.

VII. The Model Extension Language

In this section the MEL language is presented in detail. The syntax of MEL is shown in Fig. 13. It is based on Java, but a little bit more UML-like.

The description of the grammar can be read from top to bottom. MEL can be used in UML `Action`s, in UML `guards`, and in UML `SendSignalAction`s and `AcceptEventAction`s which are treated differently. A (normal) action can contain either one expression, or a list of statements. A statement in MEL is simply an expression followed by a semicolon. Java statements like conditional, loop, return etc. are not supported, but must be modeled with activity diagram elements. MEL expressions and types are a subset of Java expressions and types. The most obvious omissions are arrays and generic types. The idea is to use more abstract data types like lists or sets instead of arrays. Generic types may be added for non-Java Card applications in the future. MEL contains an **else** expression that may only be used on top-level in a guard (UML also defines **else** as a special guard). A local variable declaration (locvardecl in

```
Start = Action | Guard
      | SendSignalAction | AcceptEventAction

Action = Expr | Stm*
Guard = Expr
SendSignalAction = Expr
AcceptEventAction = Expr

Stm = Expr ;
Expr = Locvardecl | Assignment | CreateExpr | MethodCall
      | BinaryExpr | UnaryExpr | LiteralExpr | FieldAccess
      | Name | ( Expr ) | else
ExprList = ε | Expr[, Expr]*

Locvardecl = Identifier : Type | Identifier : Type := Expr
Assignment = Expr := Expr
CreateExpr = create Identifier ( ExprList )
MethodCall = Identifier ( ExprList )
           | Expr . Identifier ( ExprList )
BinaryExpr = Expr Binop Expr
UnaryExpr = Unop Expr | Expr Unop
LiteralExpr = true | false | NumberLiteral | StringLiteral
FieldAccess = Expr . Identifier
Name = Identifier | Name . Identifier

Identifier = Legal Java identifier (JLS 3.8)
Type = Name
Binop = == | != | < | > | <= | >= | + | - - | * | / | %
      | and | or | via
Unop = + | - - | ++ | - - | not | #
NumberLiteral = Legal Java integer literal (JLS 3.10.1)
StringLiteral = Legal Java string literal (JLS 3.10.5)
```

Fig. 13. The MEL language used in activity diagrams

Fig. 13, technically not an expression in Java) has a UML-like syntax, similarly an assignment (**:=** instead of simply =). Logical operations must be written as **and**, **or**, **not** instead of **&&**, ||, **!**. A new prefix operation is **#** that denotes the length of a list or the size of a set. Another new operation is **via** that may only be used on top-level in send and accept actions and specifies the communication paths over which a message is sent or received.

After parsing a MEL expression an annotated abstract syntax tree in the form of a model is created in the same manner as by a Java compiler. Annotating MEL requires a context (the classes of the class diagram), and a current class (the swim lane of the activity diagram), and must be done in sequential order following the control flow of the activity diagram to capture the scope of local variables. Identifier are classified as either local variables, fields, classes etc. and for every method call a suitable method declaration must exist (either in the class diagram, or in the predefined types, or in a sub activity diagram), and so on.

An `AcceptEventAction` must be used as an entry point into a swim lane. It must contain a method call of the form *Classname(id1,id2,...)*, optionally followed by a **via** *Identifier*. The *Classname* must name a message class, and the identifier *id1*, *id2*, ... are interpreted as local variables with the types of the attributes and associations of *Classname*. For example, `StartFrom(val,pd)` means that a `StartFrom` message is received. `val` becomes a local variable of type `Number` that is initialized with the `value` attribute, and `pd` becomes a local variable of type `PurseData` containing `value.dataTo` (see the class diagram in Fig. 9). The scope of a local variable ends at the border of a swim lane.

MEL has a do-what-I-mean flavor that is very convenient for modeling. This can be considered as syntactical sugar. For example, the static members of a class can be accessed without a classname: The name resolution will interpret `state == IDLE` (see Fig. 10) as `state == State.IDLE`. Furthermore, MEL ignores object identities. In a communication scenario with cryptographic protocols objects are almost never identical, because messages treat objects as data. Therefore, `==` can be used to compare objects, and is interpreted as an equals test that compares attributes.

The annotated abstract syntax tree is essential for error checking as well as for the correct generation of code (e.g. `==` may become an `equals` method call). The idea is to make the MEL language easy to use for a modeler, but still as precise as a programming language. In the future, MEL can be extended if it is useful, for example with OCL-like constructs for collections. However, control flow should be modeled with activity edges.

VIII. Generation of Code

Smart cards are small, secure computers with a size of 1 × 1 centimeters and a thickness of less than 1 millimeter. For example, the subscriber identity module (SIM) of mobile phones is a smart card, the new electronic passports contain a contactless smart card, and smart cards are used as payment cards, health cards, for access control. Java Card [26], [27] is a version of Java [28] tailored to smart cards. More than 3.5 billion Java smart cards have been issued up to now [29].

Java Card has the same syntax and semantics as Java, but the programming style is usually very different from 'normal' Java programs. The reason for this are the severe resource restrictions (memory and speed) of smart cards. Java Card has no Strings, no floating point arithmetic, and no Integers. Furthermore, threads and garbage collection are not supported. The missing garbage collection means that the programmer must be very careful when he creates objects or arrays because the allocated memory will never be freed.

The communication with a smart card is realized by using APDUs [30] (application protocol data units), essentially sequences of bytes in a predefined format. The Java Card API for the communication works with byte arrays. The missing garbage collection and the communication API induce a programming style that is usually not object-oriented. Typically, Java syntax is used to manipulate byte arrays directly omitting object-oriented paradigms like modularization and encapsulation. Examples can be found in [31] that contains two different Mondex implementations based on byte arrays.

In our opinion, one challenge of model-driven code generation approaches is to reduce the gap between input and target platforms. For this reason, we decided to make further use of the classes defined in the platform-independent models (and later transformed to platform-specific classes) instead of transforming the object-oriented view of the application into a program consisting of byte array representations for each object resp. class. Thus, the purse class implementing the protocol steps of the cryptographic protocol operates on the data types defined in the platform-independent model by the developer.

However, the communication is still based on byte arrays. This means, to transmit data between a smart card and a terminal the message objects as well as associated objects must be converted into byte arrays and back again. The easiest way to do so is to serialize each message object before sending it and after receiving a byte array message to convert it into the corresponding message object. This is done using an encoding similar to a TLV encoding [32], [33]. This encoding is highly application dependent because Java Card does not support reflection. Therefore it is ideally suited for automatic code generation.

Another challenge is the missing garbage collection. The required objects cannot be created during the protocol runs but must be allocated once beforehand and reused. In our approach we generate code for an object store that allocates the required objects and manages them, i.e. if an object is needed it is requested from the store. More details on the code generation can be found in [25].

IX. Generation of a Formal Model for Verification

To prove the security of the system under development we automatically generate a formal model based on algebraic specifications and abstract state machines suitable for our interactive theorem prover KIV. The static aspects of the modeled application are defined by algebraic specifications whereas the dynamic part of the system is translated into an abstract state machine (ASM) [2]. The formal model uses the application-dependent data types which are defined in the class diagram, i.e. specifications exist for the messages, plain data and so on. We use application-dependent types instead of a generic type as used in [34] [12]. Since the formal model is used for interactive verification, it is very helpful to have a formal model that is close to the UML models.

To model the attacker we define the attacker knowledge which contains all (relevant) data known by the attacker during a protocol run, similar to [34] and [35]. The attacker knowledge contains all data that is part of a message and can be analyzed by the attacker. In the Mondex example this includes the encrypted content of the `Req`, `Val` and `Ack` messages. If the attacker does not know the key he cannot decrypt the content, but with an insecure protocol he may later learn the key, and then decrypt the data. All non-security critical data such as the amount to load is not explicitly stored in the attacker knowledge because this data is not secret and assumed to be known by the attacker.

In the formal model the components of the systems are defined as different *agents* that communicate by exchanging messages. The formal model captures the behavior of the real world that is related to the application. In the real world, many Mondex cards exist. To model the transfer of money, at least two cards (agents in the formal model) are needed. Indeed, it may be possible that there exists an attack on the protocol that needs three or more cards, and does not work with only two cards. In this case a formal model with only two cards would be grossly flawed, because the proofs of the security properties would succeed for a protocol that is in reality insecure. Therefore the formal model has an arbitrary, but finite, number of cards (more precisely: instances for each agent type). To represent the communication we explicitly model the possible connections between two agents. Since more than one communication path between two agents may exist, we additionally use ports to distinguish the paths. The information about communication paths and ports is taken from the deployment diagram. To model the sending and receiving of messages in the formal model we use inboxes (essentially queues) for each component and port. An inbox is of type message list and contains all messages that were received by an agent but not yet processed.

The dynamic part of the system is modeled as an abstract state machine (ASM). The state of the ASM consists of the states of all agents. In the Mondex example the state of the purse consists of the values of the attributes and associations of the `Purse` class. A step of the ASM applies one ASM rule and transforms the state. A run of the ASM is a sequence of single steps and creates a trace, i.e. a sequence of its states. A trace models arbitrary protocol runs that could happen in the real world. Since many different events occur in the real world (e.g. the attacker may choose to interfere with a communication or not) an adequate formal model is the set of all possible traces. If the protocol is secure for all possible traces we assume that the protocol is secure in the real world. Therefore the ASM must allow the same choices that are possible in the real world, i.e. the ASM must be indeterministic. We model the real world by defining an ASM rule that nondeterministically chooses an agent which – if possible – executes a protocol step. If for example the `Purse` agent is chosen, it is checked whether the inbox (of the connection to the terminal) is non-empty. If so, the first message is taken and processed. If the inbox is empty, another agent is chosen by the ASM. If the first message is of type `StartFrom`, the ASM rule describing the processing of a `StartFrom` message is executed. This rule is shown in listing 1. To generate the ASM rule, the activity diagrams are used as input (see Fig. 10).

It is not the purpose of this paper to describe the syntax and semantics of the ASM rules as they are used in the KIV system. Therefore, we give just an informal overview of the example rule. The content of the `StartFrom` message, i.e. the `value` and the `PurseData` of the `to` purse, are stored in local variables (lines 2 and 3 in listing 1). Next, it is checked

```
1  STARTFROM#
2  let value = inmsg.value,
3      dataTo = inmsg.dataTo in
4   if exlogcounter(ag) < LOGLENGTH
5   then
6    if state(ag) = IDLE
7    then
8     ...
9     pdAuth(ag) .from := data(ag);
10    pdAuth(ag) .to := dataTo;
11    pdAuth(ag) .value := value;
12    data(ag) .sequenceNo := data(ag)
13                    .sequenceNo + 1;
14    state(ag) := EPR;
15
16    let encmess = mkMsgcontent(
17            STARTTO,pdAuth(ag)) in
18    let enc = encrypt(
19              sesskey(ag),encmess) in
20     outmsg(ag) := mkStartTo(enc);
21   else ABORT#
22  else ABORT#
```

Listing 1. ASM rule of processing a StartFrom message

if the exception log has free entries (line 4). The expression `exlogcounter(ag)` is specific for the formal model. `ag` is a variable for a `Purse` agent. As mentioned previously, the formal model contains an arbitrary number of `Purse` agents, and `ag` is the agent chosen in this ASM step. Agents are modeled with dynamic functions in the formal model, i.e. `exlogcounter` is a function that maps a `Purse` agent to the value of its `exlogcounter` attribute. It can be read as `ag.exlogcounter`. Similar functions exist for all attributes and associations of `Purse` (`pdAuth(ag)`, `state(ag)`, ...). Then the ASM rule checks whether the state of the card is set to `IDLE` (line 6) and performs some additional checks. If all tests succeed, several attributes and associations of the considered agent `ag`, in this case the purse agent, are updated (lines 9 - 14). An update means that the corresponding dynamic function is modified (therefore the function is called 'dynamic'). In a next step, a local variable `encmess` of type `Msgcontent` is created with the `msgflag` `STARTTO` that indicates a `StartTo` message and the current pay details (line 16). Then, this variable is encrypted by using a predefined encrypt function (line 18). The dynamic function `outmsg` that is generated automatically for each agent stores the message that is going to be sent after termination of the ASM rule for processing a `StartFrom` message. In our case, a `StartTo` message is sent next and stored `outmsg` (line 20). If one of the checks made in the beginning fails, the protocol aborts (line 21 and 22). The abortion is defined in a separate ASM rule called ABORT#. It can be seen that the structure of the ASM rule follows the structure of the activity diagram, but uses a different syntax, and has a semantics that is similar to MEL (e.g. copy semantics), but not identical (dynamic functions and inboxes are not part of MEL).

One relevant security property for Mondex is that the sum of money stored on all Mondex cards plus the sum of money stored in all (valid) exception logs does not increase or decrease over the time. This implies that no money is lost or created during a transfer of money, even in the presence of an attacker. This property can be formulated as a theorem in the formal model and proved with our theorem prover KIV. Of course, since a card may be recharged, this holds only for the use case 'Person-to-Person Payment'.
In previous work Haneberg [12] [36] developed a formal model based on ASMs and verification techniques to prove the security of an abstract model. This approach was successfully used in several case studies. The formal model introduced in this section is based on the one by Haneberg but uses application-dependent data types instead of a generic data format.

X. Related Work

Basin et al. [37] [38] present a model-driven methodology for developing secure systems which is tailored to the domain of role-based access control. The aim is to model a component-based system including its security requirements using UML extension mechanisms. To support the modeling of security aspects and of distributed systems several UML profiles are defined. Furthermore, transformation functions are defined that translate the modeled application into access control infrastructures. The platforms for which infrastructures are generated, are Enterprise JavaBeans, Enterprise Services for .Net as well as Java Servlets.

Another approach that is related to ours is UMLSec developed by Jan Jürjens [8]. As in our approach he proposes to use UML for the development of security-critical applications. UMLSec defines a UML profile which adds security-relevant information to the UML diagrams. Security properties are expressed by using stereotypes. Jürjens provides tool support for verifying properties by linking the UML tool to a model checker resp. automated theorem provers. By doing so, the security properties mainly addressed are those that are expressed by the predefined stereotypes. The relevant formal model reflects an abstracted view of parts of the entire system. In our approach we concentrate on a transformation process that generates a formal model of the entire application which can be used for interactive verification of all system aspects. Based on the generated formal model, we can express and prove application dependent security properties such as "No money can be created within the Mondex application". In contrast to UMLSec we additionally focus on the generation of running Java Card code as well as the proof that this code is a refinement of the formal model.

In [39] Kuhlmann et al. model the Mondex system with UML. Only static aspects of the application including method signatures are defined by using UML class diagrams. To specify the security properties that have to be valid the approach uses the object constraint language (OCL). The specified constraints are checked using the tool USE (UML-based Specification Environment). USE validates a model by

testing it, i.e. it generates object diagrams as well as sequence diagrams of possible protocol runs. The approach neither considers the generation of code nor the use of formal methods to prove the security of the modeled application. The models are only validated by testing.

Alam et al. [40] present a model-driven security engineering framework for B2B-workflows. They introduce a domain-specific language for specifying access control policies which is used in the context of UML models. Furthermore, a UML profile for trust management is defined. After modeling a B2B application with UML, it is then translated into low-level web service artefacts using model-to-model and model-to-text transformations.

Deubler et al. present a method to develop security-critical service-based systems [41]. For modeling and verification the tool AutoFocus [42] is used. AutoFocus is similar to UML and facilitates the modeling of an application from different views. Moreover, the tool is linkable to the model checker SMV. The approach focuses on the specification of an application with AutoFocus and, in a next step, the generation of SMV input files and formal verification using SMV. The generation of secure code is not part of the approach.

XI. Conclusion

We presented our SecureMDD approach for the modeling of security-critical systems, especially smart card applications, with UML. Using this model-driven method UML models can be automatically translated into a formal model that is used to verify the security of our models. Furthermore, executable code can be generated automatically. In this paper we focused on the modeling with UML, i.e. the use of our UML profile which is tailored to security-critical applications and our Model Extension Language that we use in activity diagrams to describe cryptographic protocols. We propose a modeling technique that is easy to learn and abstracts from specifics regarding the formal specification or implementation. One disadvantage of UML is that it is only semi-formally defined. Since in our approach the UML models are translated into abstract state machines, we give them a formal semantics. We do not define a semantics for UML in general but only consider those parts that are used in our approach and which are interpreted in the context of security-critical applications. Our technique has evolved over several case studies. E.g. we have analyzed an application where a smart card is used as a copycard for a library [35]. Another case study deals with an application to buy cinema tickets using a mobile phone [12].

At the moment our approach is tailored to smart card applications but we are going to extend it, e.g. to service-oriented architectures, in the future. For example, the german electronic health card which consists of smart card parts as well as services that are realized as SOA, would fit into this domain. Another focus of future research is to build in the expression of security properties on the level of platform-independent modeling, for example by supporting the use of OCL expressions.

References

[1] *Mondex*, MasterCard International Inc., URL: http://www.mondex.com.

[2] E. Börger and R. F. Stärk, *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[3] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford Univ. Press, 1995, pp. 9 – 36.

[4] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal system development with KIV," in *Fundamental Approaches to Software Engineering*, T. Maibaum, Ed. Springer LNCS 1783, 2000.

[5] H. Grandy, K. Stenzel, and W. Reif, "A Refinement Method for Java Programs," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, ser. LNCS, vol. 4468. Springer, 2007.

[6] K. Stenzel, "A formally verified calculus for full Java Card," in *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, C. Rattray, S. Maharaj, and C. Shankland, Eds. Springer LNCS 3116, 2004.

[7] K. Stenzel, "Verification of Java Card Programs," Ph.D. dissertation, Universität Augsburg, Fakultät für Angewandte Informatik,URL: http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/,or http://www.informatik.uni-augsburg.de/forschung/dissertations/, 2005.

[8] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[9] N. Moebius, D. Haneberg, G. Schellhorn, and W. Reif, "A Modeling Framework for the Development of Provably Secure E-Commerce Applications," in *International Conference on Software Engineering Advances (ICSEA) 2007*. IEEE Press, 2007.

[10] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel, "KIV 3.0 for Provably Correct Systems," in *Current Trends in Applied Formal Methods*, ser. LNCS 1641, Boppard, Germany. Springer-Verlag, 1999.

[11] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif, "Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol," *Formal Aspects of Computing*, vol. 20, no. 1, January 2008.

[12] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel, "Developing Provably Secure M-Commerce Applications," in *Emerging Trends in Information and Communication Security (ETRICS)*, ser. LNCS, G. Müller, Ed., vol. 3995. Springer, 2006, pp. 115–129.

[13] "Eclipse Modeling Project," http://www.eclipse.org/modeling/.

[14] "Open Architecture Ware," http://www.openarchitectureware.org/.

[15] Object Management Group (OMG), "The unified modeling language," 2006. [Online]. Available: www.uml.org/

[16] J. Woodcock, "First steps in the verified software grand challenge," *IEEE Computer*, vol. 39, no. 10, pp. 57–64, 2006.

[17] C. Jones and J. Woodcock, Eds., *Formal Aspects of Computing*. Springer, January 2008, vol. 20 (1).

[18] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif, "A Systematic Verification Approach for Mondex Electronic Purses using ASMs," in *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, U. G. J.-R. Abrial, Ed. Springer LNCS 5115, 2008.

[19] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif, "The Mondex Challenge: Machine Checked Proofs for an Electronic Purse," in *Formal Methods 2006, Proceedings*, ser. LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 16–31.

[20] S. Stepney, D. Cooper, and J. Woodcock, "AN ELECTRONIC PURSE Specification, Refinement, and Proof," Oxford University Computing Laboratory, Technical monograph PRG-126, July 2000.

[21] L. C. Paulson, "Inductive analysis of the internet protocol TLS," Computer Laboratory, University of Cambridge, Tech. Rep. 440, Dec. 1997.

[22] G. Lowe, "Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR," in *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*. Springer LNCS 1055, 1996, pp. 147–166.

[23] D. Dolev and A. C. Yao, "On the security of public key protocols," in *Proc. 22th IEEE Symposium on Foundations of Computer Science*. IEEE, 1981, pp. 350–357.

[24] *Java Card 2.2 Specification*, Sun Microsystems Inc., 2002, http://java.sun.com/products/javacard/.

[25] N. Moebius, K. Stenzel, H. Grandy, and W. Reif, "Model-Driven Code Generation for Secure Smart Card Applications," in *20th Australian Software Engineering Conference*. IEEE Press, 2009.

[26] *Application Programming Interface Java Card Platform, Version 2.2.1*, Sun Microsystems Inc., URL: http://java.sun.com/products/javacard/.

[27] Sun Microsystems, "Java Card 3.0 Platform Specification," http://java.sun.com/javacard/3.0/specs.jsp, 2008.
[28] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java (tm) Language Specification, Third Edition*. Addison-Wesley, 2005.
[29] Sun Microsystems, "Press release," April 22 2008. [Online]. Available: http://www.sun.com/aboutsun/pr/2008-04/sunflash.20080422.1.xml
[30] *ISO 7816-4 – Identification Cards – Integrated cicuit(s) cards with contacts – Part 4: Organization, security and commands for interchange*, International Standards Organization, 1995.
[31] H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, and W. Reif, "The Mondex Case Study: From Specifications to Code," University of Augsburg, Technical Report 2006-31, December 2006, uRL: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.
[32] H. Grandy, R. Bertossi, K. Stenzel, and W. Reif, "ASN1-light: A Verified Message Encoding for Security Protocols," in *Software Engineering and Formal Methods, SEFM*. IEEE Press, 2007.
[33] O. Dubuisson, *ASN.1 - Communication Between Heterogeneous Systems*. Elsevier-Morgan Kaufmann, 2000.
[34] L. C. Paulson, "The Inductive Approach to Verifying Cryptographic Protocols," *Journal of Computer Security*, vol. 6, pp. 85–128, 1998.
[35] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn, "Verifying Smart Card Applications: An ASM Approach," in *International Conference on integrated Formal Methods (iFM) 2007*, ser. LNCS, vol. 4591. Springer, 2007.
[36] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn, "Verifying Security Protocols: An ASM Approach," in *12th Int. Workshop on Abstract State Machines, ASM 05*, D. Beauquier, E. Börger, and A. Slissenko, Eds. University Paris 12 – Val de Marne, Créteil, France, March 2005.
[37] D. Basin, J. Doser, and T. Lodderstedt, "Model Driven Security: From UML Models to Access Control Infrastructures," *ACM Transactions on Software Engineering and Methodology*, pp. 39–91, 2006.
[38] T. Lodderstedt, D. A. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security," in *UML 2002 - The Unified Modeling Language, 5th International Conference*, 2002, pp. 426–441.
[39] M. Kuhlmann and M. Gogolla, "Modeling and validating Mondex scenarios described in UML and OCL with USE," *Formal Aspects of Computing*, vol. 20, no. 1, pp. 79–100, January 2008.
[40] M. Alam, R. Breu, and M. Hafner, "Model-Driven Security Engineering for Trust Management in SECTET," *JSW*, vol. 2, no. 1, pp. 47–59, 2007.
[41] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel, "Sound development of secure service-based systems," in *Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM, 2004, pp. 115–124.
[42] M. Broy, F. Huber, and B. Schätz, "AutoFocus - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme," *Informatik, Forschung und Entwicklung*, vol. 14, no. 3, pp. 121–134, 1999.

APPENDIX

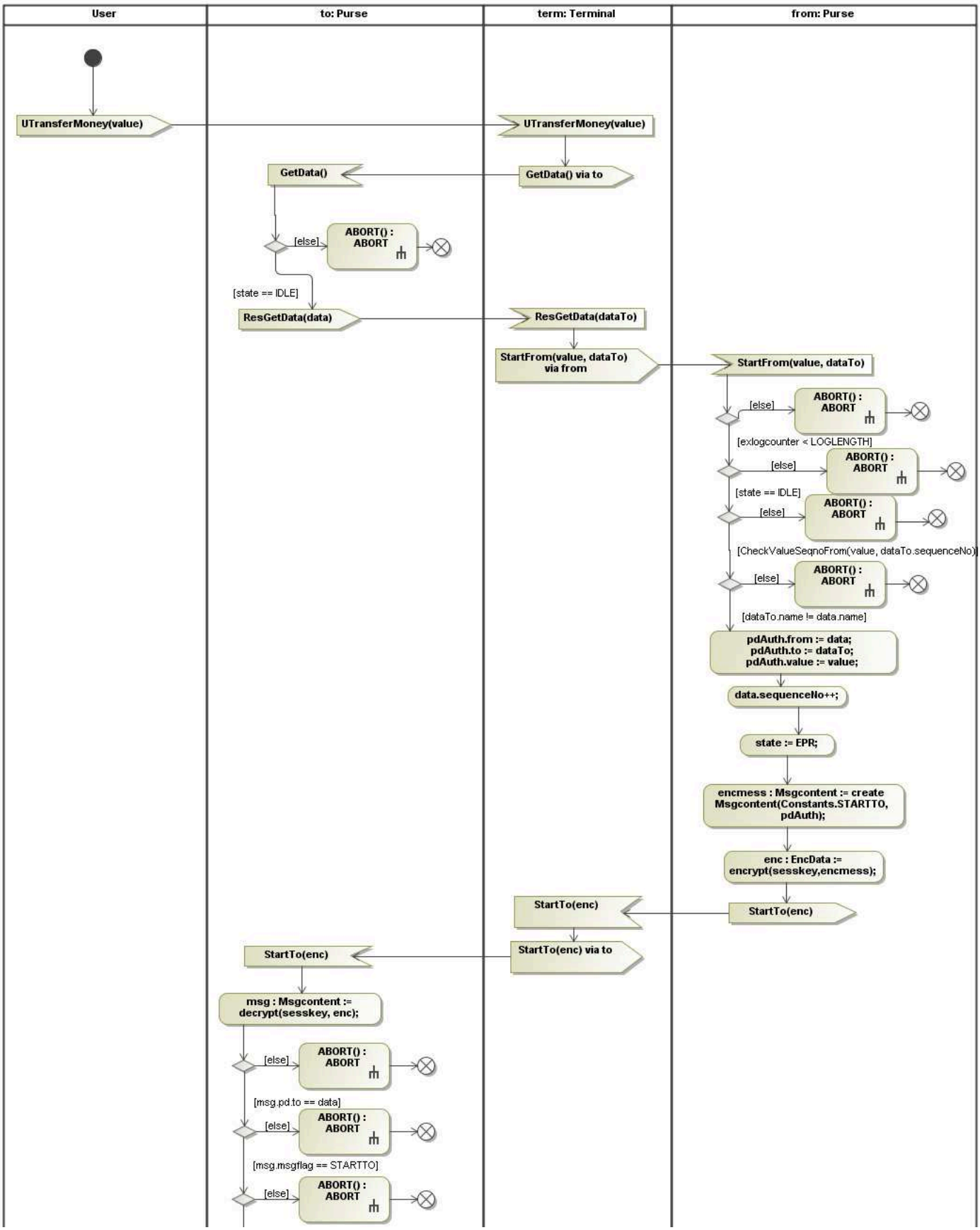


Fig. 14. Mondex Activity Diagram for Transferring Money, Part 1

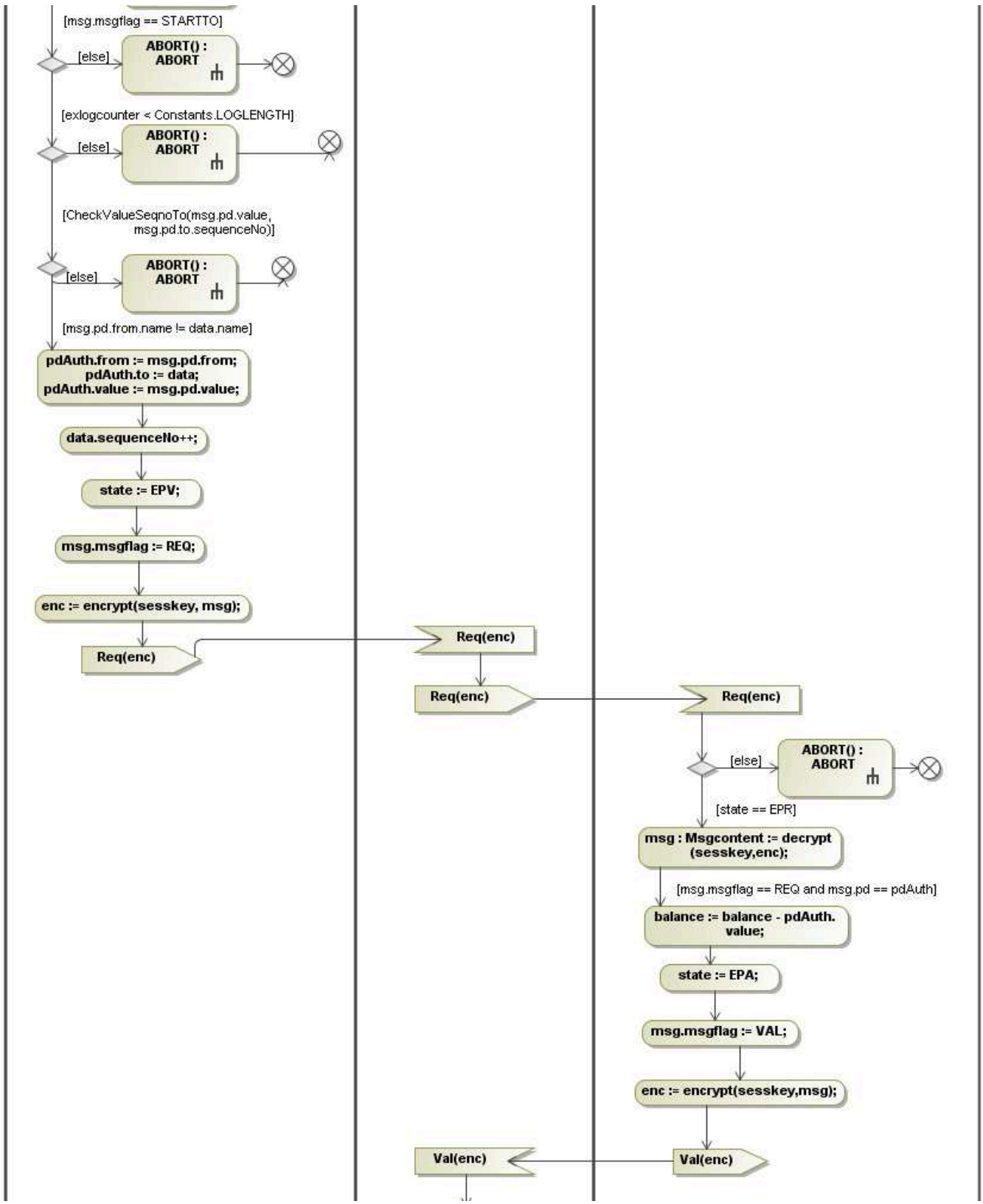


Fig. 15. Mondex Activity Diagram for Transferring Money, Part 2

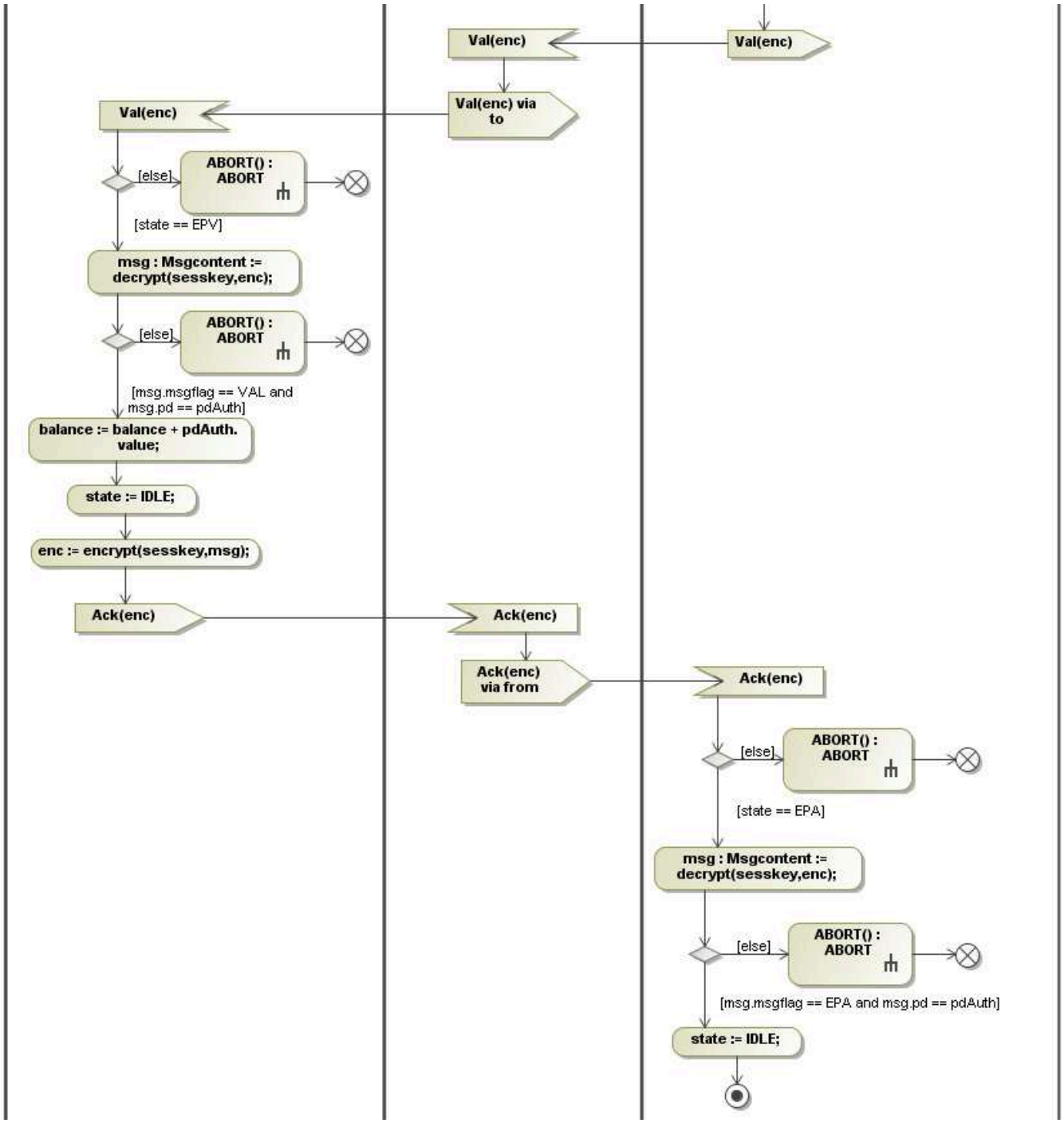


Fig. 16. Mondex Activity Diagram for Transferring Money, Part 3

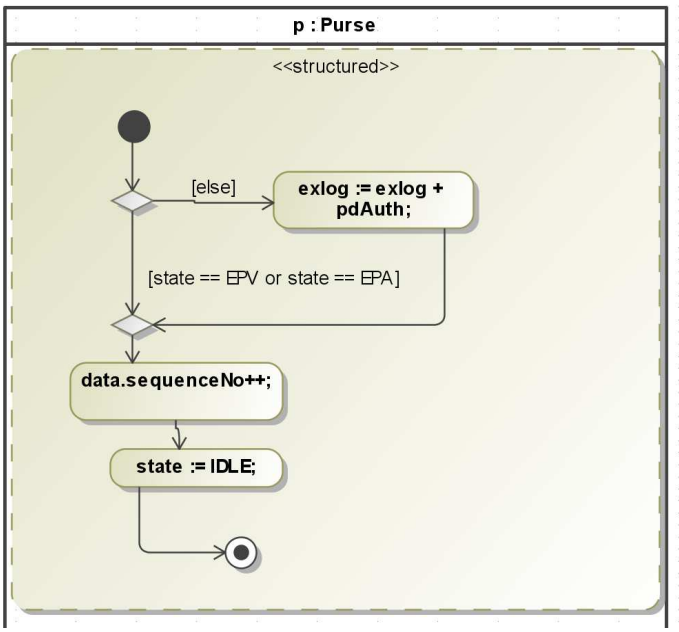


Fig. 17. Mondex Activity Diagram for Subactivity Abort()

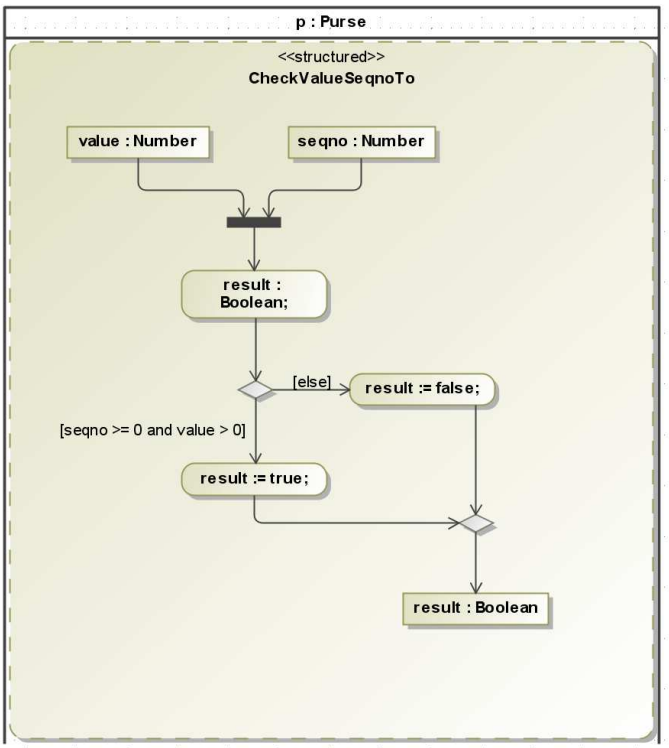


Fig. 18. Mondex Activity Diagram for Subactivity CheckValueSeqnoTo(value : Number, seqno : Number): Boolean

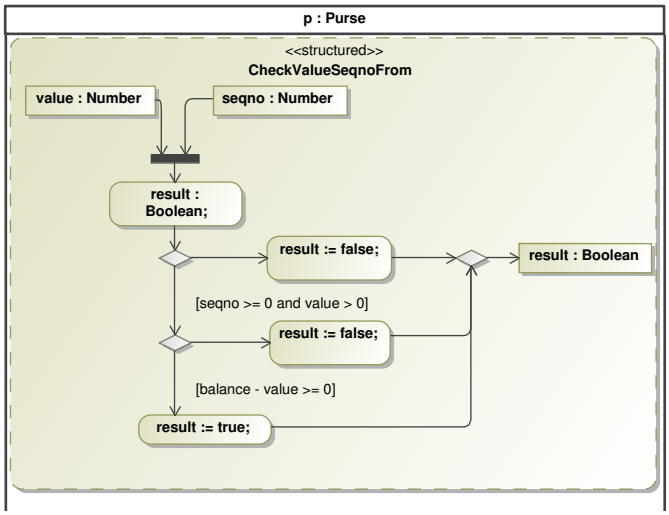


Fig. 19. Mondex Activity Diagram for Subactivity CheckValueSeqnoFrom(value : Number, seqno : Number): Boolean

Preliminary 2009 Conference Schedule

http://www.iaria.org/conferences.html

NetWare 2009: June 14-19, 2009 - Athens, Greece

- SENSORCOMM 2009, The Third International Conference on Sensor Technologies and Applications
- SECURWARE 2009, The Third International Conference on Emerging Security Information, Systems and Technologies
- MESH 2009, The Second International Conference on Advances in Mesh Networks
- AFIN 2009, The First International Conference on Advances in Future Internet
- DEPEND 2009, The Second International Conference on Dependability

NexComm 2009: July 19-24, 2009 - Colmar, France

- CTRQ 2009, The Second International Conference on Communication Theory, Reliability, and Quality of Service
- ICDT 2009, The Fourth International Conference on Digital Telecommunications
- SPACOMM 2009, The First International Conference on Advances in Satellite and Space Communications
- MMEDIA 2009, The First International Conferences on Advances in Multimedia

InfoWare 2009: August 25-31, 2009 – Cannes, French Riviera, France

- ICCGI 2009, The Fourth International Multi-Conference on Computing in the Global Information Technology
- ICWMC 2009, The Fifth International Conference on Wireless and Mobile Communications
- INTERNET 2009, The First International Conference on Evolving Internet

SoftNet 2009: September 20-25, 2009 - Porto, Portugal

- ICSEA 2009, The Fourth International Conference on Software Engineering Advances
  - SEDES 2009: Simpósio para Estudantes de Doutoramento em Engenharia de Software
- ICSNC 2009, The Fourth International Conference on Systems and Networks Communications
- CENTRIC 2009, The Second International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services
- VALID 2009, The First International Conference on Advances in System Testing and Validation Lifecycle
- SIMUL 2009, The First International Conference on Advances in System Simulation

NexTech 2009: October 11-16, 2009 - Sliema, Malta

- UBICOMM 2009, The Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies
- ADVCOMP 2009, The Third International Conference on Advanced Engineering Computing and Applications in Sciences
- CENICS 2009, The Second International Conference on Advances in Circuits, Electronics and Micro-electronics
- AP2PS 2009, The First International Conference on Advances in P2P Systems
- EMERGING 2009, The First International Conference on Emerging Network Intelligence
- SEMAPRO 2009, The Third International Conference on Advances in Semantic Processing