Babeş-Bolyai University

KEPT 2009

KNOWLEDGE ENGINEERING

PRINCIPLES

AND

TECHNIQUES

EDITORS:
Militon Frenţiu and Horia F. Pop

Cluj-Napoca, July 2–4, 2009

KEPT 2009

Important dates

Submission deadline:	March 31, 2009
Notification of acceptance:	April 15, 2009
Camera-ready copy due:	May 10, 2009
Registration deadline:	June 10, 2009
Conference dates:	July 2-4, 2009

Please note that the refereeing and publication schedule does not permit any delays in these dates.

Topics

Conference topics include, but are not limited to:

- Knowledge in Computational Linguistics
  - Corpora as knowledge bases;
  - Linguistic tools in Information retrieval and Information Extraction;
  - Text mining, Text entailment and Text summarization ;
  - Discourse and Dialogue;
  - Multi-lingual processing, machine translation;
  - Machine learning for natural languages;
  - Linguistic components of information systems;
  - Theoretical and application-oriented subjects related to NLP.
- Knowledge Processing and Discovery
  - Natural computing;
  - Metaheuristics;
  - Machine learning;
  - Computational intelligence;
  - Agent based systems.

- Knowledge in Software Engineering
  - Software design;
  - Formal verification;
  - Automated reasoning;
  - Parallel and concurrent programming;
  - Ontology-based software engineering;
  - Ontology-driven information systems;
  - Formal methods;
  - Knowledge based applications;
  - Formal concept analysis.
- Knowledge in Distributed Computing
  - Knowledge representation and processing;
  - Databases and Data Mining;
  - Web services, middleware and web technologies;
  - Semantic web;
  - Grid architecture;
  - Collaborative systems.
- Knowledge Processing in Economics
  - Collaborative Decision Support Systems;
  - Knowledge acquisition and management in e-activities and m-activities;
  - Enterprise ontologies and content management;
  - Integrated software systems, ERPs and extensions;
  - Business Intelligence and Warehousing;
  - Knowledge Management in Small and Medium Size Enterprises;
  - Multi-agent Models in Knowledge Acquisition and Management.

Aims

The aim of KEPT2009 is to serve as a forum to present current and future work as well as to exchange research ideas in the field of Knowledge Engineering.

COMMITTEES

Organizing Committee

Conference General Chairs.

- Militon Frenţiu (Babeş-Bolyai University)
- Horia F. Pop (Babeş-Bolyai University)

Section Chairs.

- Doina Tătar (Babeş-Bolyai University)
  - Knowledge in Computational Linguistics
- Dumitru Dumitrescu (Babeş-Bolyai University)
  - Knowledge Processing and Discovery
- Bazil Pârv (Babeş-Bolyai University)
  - Knowledge in Software Engineering
- Florian Mircea Boian (Babeş-Bolyai University)
  - Knowledge in Distributed Computing
- Stefan Niţchi (Babeş-Bolyai University)
  - Knowledge Processing in Economics

Secretariat.

- Grigoreta Cojocar (Babeş-Bolyai University)
- Laura Dioşan (Babeş-Bolyai University)
- Sanda Dragoş (Babeş-Bolyai University)
- Vladiela Petraşcu (Babeş-Bolyai University)
- Andreea Sabău (Babeş-Bolyai University)

Program Committee

CONTENTS

Knowledge Processing and Discovery

INVITED LECTURES

KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2009
Cluj-Napoca (Romania), July 2–4, 2009, pp. 3–6

KNOWLEDGE REPRESENTATION WITHIN AN INTELLIGENT TUTORING SYSTEM

HELMUT HORACEK[(1)]

Abstract. Intelligent tutoring systems, though quite successful in a variety of applications, are typically limited with respect to the generality of their problem-solving methods and in their communication capabilities. Aiming at exploring the prerequisites for future, more powerful tutoring systems, we present techniques for representing knowledge within such a system that allow for reasoning about more complex problem-solving situations and that enable flexible communication in natural language. The domain of application is teaching a student how to prove mathematical theorems.

1. Motivation

Intelligent tutoring systems are currently a hot topic in artificial intelligent research, and a number of widely-used application systems as well as experimental systems have been built over the last decade. Despite this success, the technology is still restricted in several ways, which has a major source in the predominant system architecture that does not support a cleanly separated and in part deep representation of different categories of knowledge.

In contrast to that, our elaborations feature modularization, based on a strict distinction between

- problem-solving knowledge
- communication knowledge
- pedagogical knowledge

and their suitable coordination. In particular, we emphasize interaction in natural language, because this way of communication constitutes a crucial asset of human tutors that intelligent tutoring systems are essentially missing - it encourages students to explicitly formulate their plans, which is an important step in learning new conceptions. Apparently, such an approach is more difficult to orchestrate, but it is believed to pay off in the long run, since it allows for the incorporation of problem-solving and natural language processing components that are developed independently of the tutoring context. Such components are supposed to be reused beneficially, and they also support the transfer to other domains of application.

2000 *Mathematics Subject Classification.* 68T50, 68T30.
Key words and phrases. intelligent tutoring systems, knowledge representation, natural language processing.

The aim of building a tutoring system on these lines is associated with a number of research challenges. In the following, we elaborate on experience made in the context of a project carried out at our university, whose goal was

(1) to empirically investigate the use of flexible natural language dialog in tutoring mathematics, and
(2) to develop a prototype tutoring system that incorporates the empirical findings.

The experimental system engages a student in a dialog in written natural language to help him/her understand and construct mathematical proofs. Example domains include elementary set theory and mathematical relations.

2. Problem-Solving Knowledge

Within a tutoring context, demands on knowledge representation for problem-solving purposes are extremely high. First of all, it should allow for incremental and flexible solution development. In addition, it should support inspection of the state of affairs, and checking possible continuations, in view of ambiguous and only partial specifications, which is what contributions of students typically are. When efficient problem-solving is the only concern, suitable representations are essentially simple and uniform, such as logical calculi for theorem proving systems. While completed results can be transformed into more abstract, communication adequate representation levels, systematic transformation methods cannot reasonably cope with incompleteness and incrementality.

In order to meet the requirements of intelligent tutoring, representations are built on an abstracted level, where individual reasonnig steps essentially correspond to the application of axioms, that is, conceptually meaningful pieces of domain knowledge, for example, applying the distributivity law. Populating such a representation with inferences suitable for the chosen subdomain requires some development effort by system engineers, which is a certain price to pay. Inference steps can be composed to build a proof graph, which enables one to check the validity of each step down to the underlying proof calculus, thus making use of a powerful and general reasoning component. Moreover, testing and completing specifications of inference steps that stem from the tutoring interaction is supported in a flexible manner, including forward and backward inferences.

3. Knowledge for Natural Language Interpretation

Within a tutoring context in a formal domain, such as mathematics, natural language interpretation is confronted with two specific fundamental challenges:

- Descriptions in these domains are characterized by an intertwined combination of exact formulas with sometimes sloppy natural language expressions, and
- descriptions produced by students may contain errors, and they may be vague and incomplete, in some cases to a greater extent than this is consistent with the domain-specific language use as found in mathematical textbooks.

Mastering these challenges is supported by classical linguistic knowledge sources, which are extended and adapted in some crucial places for our purposes. Grammar rules may contain places for mathematical expressions of different types, including some examples of linguistic structures with gaps that can be filled by mathematical expressions. Such a construct may be essential to cope with specific cases of intertwinedness, particularly in connection with some operators, such as negation. The a priori categorization of input components into formulas and pieces of natural language text, as well as segmentation into individual assertions, a quite critical decision, is done on the basis of some relatively simple heuristics - hence, limitations in these heuristics are an obvious source of interpretation failures. The meaningful interpretation of vague and incomplete descriptions is handled by a repertoire of specialized entries in the semantic lexicon, which allow filling gaps in expressions, such as interpreting domain-specific metonymic relations.

A final and crucial task in natural language analysis is the ultimate interpretation of the representations oriented on natural language in terms of representations used for domain reasoning, which differ in some crucial factors among one another. In order to accomplish this task, we have built a representation that constitutes an enhanced mirror of the domain representations. It serves as an intermediate representation between the domain and linguistic models. Measures in this mapping comprise the interpretation of vague and sloppy natural language expressions in terms of exact mathematical concepts, where ambiguities can sometimes be resolved by semantic role restrictions.

4. Tutoring Knowledge

Tutorial goals in teaching mathematical theorem proving are two-fold: teaching problem-solving methods is the essential target of tutoring, while techniques for building correct formal expressions constitute a somehow subordinate goal. Reconciling these two complementary tutorial goals is quite challenging, since pursuing one of them can in some cases interfer with pursuing the other. We support these goals by dedicated representations of knowledge. In order to address low-level errors, we use a set of replacement rules which aim at curing formal flaws in otherwise reasonable specifications of problem solving steps, such as confusing two operators or omitting parentheses. Here, tutoring knowledge interacts with natural language analysis.

In order to address teaching problem-solving methods, a multi-dimensional taxonomy of hint categories, varying in degrees of precision and content, has been developed. This taxonomy encompasses a variety of contextual factors, such as the number of trials already made by the student and degrees to which the necessary elements of the inference step addressed are also covered. Here, tutoring knowledge interacts with problem-solving knowledge. This interaction comprises the evaluation of proof-relevant parts of the utterances with respect to completeness, correctness, and relevance, where correct proof steps may not necessarily be relevant in that they do not contribute to the progress in finding the proof solution. This categorization is an integral part of our tutorial strategies.

5. Assessment

The techniques descibed have been developed and evaluated on the basis of two corpora of tutoring dialogs obtained through Wizard-of-Oz experiments. Our methods can mimic the behavior of the human tutors in these experiments in essential elements for some portions of the tutoring dialogs. As expected, a variety of limitations became apparent as well. They include

- low-level student errors whose repair is more involved than what our replacement rules can explain,
- telegraphic natural language expressions which are beyond the state-of-the-art of syntactic semantic interpretation, frequently caused by the notorious absence of segmentation markers in student utterances,
- free mixture of dialog functions, including unexpected topic changes,
- the need for a larger variety of tutoring strategies, which to some extext are limited by missing meta-knowledge about the domain itself and about problem solving techniques.

Altogether, our investigastions have illustrated the capabilities and the potential of natural language processing and reasoning methods for intelligent tutoring purposes, which have been extended in some essential ways. However, we have also found a number of quite principled limitations, which are likely to influence the design of interfaces for these kind of systems in the near future. The most serious drawback of our approach in comparison to traditional architectures of intelligent tutoring systems lies in the difficulty to build authoring tools. In existing systems, they typically comprise scripts of some sort, which are driven by surface-oriented specifications, so that domain experts who are not familiar with computational approaches can populate these scripts. In our approach with distinct knowledge sources, filling these representations appropriately requires considerable degree of computational capabilities.

[1] German Research Center for AI and Saarland University F.R. 6.2. Computer Science, P.O.Box 1150 D-66041 Saarbrcken, Germany

E-mail address: horacek@cs.uni-sb.de

KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2009
Cluj-Napoca (Romania), July 2–4, 2009, pp. 7–16

MODELING SEMANTIC KNOWLEDGE IN ERLANG FOR REFACTORING

ZOLTÁN HORVÁTH(1), LÁSZLÓ LÖVEI(1), TAMÁS KOZSIK(1), RÓBERT KITLEI(1), ANIKÓ NAGYNÉ VÍG(1), TAMÁS NAGY(1), MELINDA TÓTH(1), AND ROLAND KIRÁLY(1)

Abstract. RefactorErl is a refactoring tool for the Erlang programming language. Refactorings have to collect many kinds of information that appear scattered in the source code. Therefore, when designing a refactoring tool, the most important concern is how the program is represented so that the many layers of intertwined information can be accessed conveniently. Such information strongly depends on the language, therefore we have opted to develop a language specific model for Erlang. This model encompasses the expert knowledge necessary for refactoring Erlang programs by describing the connections between the diverse pieces of information.

1. Introduction

During a large portion of the life cycle of software, after the code has reached maturity, maintenance and improvement become prevalent. In this phase, it often becomes evident that the structure of the already produced code is not fit for the needs of further development. Thus, in order to proceed, changes have to be made. These changes may or may not involve difficult questions to consider, but most of the time they are tedious and error-prone to perform by hand. For example, in the case of renaming a variable, all of the instances of the variable have to be identified (considering shadowing), and then all of these instances have to be renamed.

Refactoring [6, 7] means changing the program code without changing what the code does. Tool support for performing refactoring is available for many object-oriented programming languages, and for some functional ones as well [9, 20]. Refactoring tools not only automate systematic transformations of programs, but also ensure that the semantics of the refactored programs are preserved. For this reason, the refactoring tool will analyse the structure of the refactored program (based on the syntactic rules of the underlying programming language), and it will also collect and use static semantical information about the program.

The paper is structured as follows. In Section 2, RefactorErl, our refactoring tool for Erlang, is briefly described. Some of the transformations implemented in the

2000 *Mathematics Subject Classification.* 68N15, 68N30.
Key words and phrases. Erlang, refactoring.
Supported by Ericsson Hungary and ELTE IKKK..

tool are presented in the course of an example. Section 3 introduces the way the tool acquires and stores information about the program code. Section 4 describes the extent of knowledge that is represented in the tool. Finally, Section 5 concludes the paper.

2. RefactorErl

RefactorErl is a tool that is developed for refactoring programs written in Erlang, both simple ones and large-scale telecom applications. The tool makes use of a grammar description based on an XML description, and specific knowledge of Erlang's semantics that provides the necessary information for refactoring.

The tool currently supports 15 refactorings. These refactorings are the following:

- Expand fun expression,
- Merge expression duplicates,
- Eliminate variable,
- Inline function,
- Extract function,
- Generalise function,
- Reorder function parameters,
- Tuple function parameters,
- Move function between modules,
- Move record between modules,
- Rename function,
- Rename variable,
- Rename module,
- Rename record and
- Rename record field.

Our refactoring tool has gone through serious modifications since its first prototype implementation. As we have gathered more knowledge about the requirements and possibilities of refactoring, we have redesigned the representation of Erlang programs inside RefactorErl. With this redesign, the refactorings have been much easier to implement than in the previous versions.

2.1. **Language specific model.** Besides a general, language-independent refactoring software infrastructure [3, 4, 21, 23], a language specific model supporting refactoring concepts proved to be useful. Based on that model a model-driven architecture can be developed. In order to guarantee consistency and to avoid ad-hoc and conflicting solutions, all the components of the refactoring tool (i.e. the parser, the semantic analyser, the code generator, the construction utilities for insertion and replacement of code parts, and the source code formatter) are either generated from, or controlled by, the same model. This is a declarative approach which maintains the refactoring-specific lexical, syntactical and static semantical rules of the investigated language as data. Modifying these data should result in the (as far as possible) automatic adaptation of the code of all the components of the refactoring tool.

```
-define(START, 1).
```

Figure 1. global.hrl

```
-module(demo).
-export([sum_n/1]).
-include("global.hrl").

sum([]) ->
    0;
sum([H|T]) ->
    H + sum(T).

sum_n(N) ->
    Lst = lists:seq(?START, N),
    sum(Lst).
```

$\longrightarrow$

```
-module(demo).
-export([sum_n/1]).
-include("global.hrl").

sum([]) ->
  0;
sum([H|T]) ->
  S = sum(T),
  H + S.

sum_n(N) ->
  Lst = lists:seq(?START, N),
  sum(Lst).
```

$\longrightarrow$

Figure 2. Merging `sum(T)`.

For example, our tool represents the model as an XML-document containing information about the lexical and syntactical rules of Erlang together with instructions for creating the internal representation of programs. This format was chosen because it is easy to maintain, should the language definition change. Furthermore, it can be handled easily with XMErl [26], a standard Erlang tool for traversing XML documents, and it can be easily transformed with e.g. XSLT tools.

2.2. **A detailed example.** The left side of Figure 2 shows a module that contains a function `sum_n/1` to sum numbers from an initial value (`?START`, defined in `global.hrl`, see Figure 1) to the argument `N`. This function creates a list that contains the values to be summed, then calls `sum/2`, which sums the elements of a list.

```
sum([]) ->
  0;
sum([H|T]) ->
  S = sum(T),
  H + S.

sum_n(N) ->
  sum((lists:seq(?START, N))).
```

$\longrightarrow$

```
sum([], Z) ->
  Z;
sum([H|T], Z) ->
  S = sum(T, Z),
  H + S.

sum_n(N) ->
  sum((lists:seq(?START, N)), 0).
```

$\longrightarrow$

Figure 3. Inlined `Lst`, generalising over zero.

```
sum([], Z) ->
  Z;
sum([H|T], Z) ->
  S = sum(T, Z),
  plus(H, S).

plus(H, S) ->
  H + S.

sum_n(N) ->
  sum((lists:seq(?START, N)), 0).
```

$\longrightarrow$

```
sum([], Z, _Op) ->
  Z;
sum([H|T], Z, Op) ->
  S = sum(T, Z, Op),
  Op(H, S).

plus(H, S) -> H + S.

sum_n(N) ->
  sum((lists:seq(?START, N)), 0,
      fun(H,S) -> plus(H, S) end).
```

$\longrightarrow$

FIGURE 4. Function `plus` extracted, generalising over `Op`.

```
sum([], _Op, Z) ->
    Z;
sum([H|T], Op, Z) ->
    S = sum(T, Op, Z),
    Op(H, S).

plus(H, S) -> H + S.

sum_n(N) ->
  sum((lists:seq(?START, N)),
      fun(H,S) -> H + S end,
      0).
```

$\longrightarrow$

```
sum([], _Op, Z) ->
    Z;
sum([H|T], Op, Z) ->
    S = sum(T, Op, Z),
    Op(H, S).

plus(H, S) -> H + S.

sum_n(N) ->
  sum((lists:seq(?START, N)),
      fun(H,S) -> H + S end,
      0).
```

$\longrightarrow$

FIGURE 5. Inlined `plus`, reordering the parameters of `sum`.

Let us first use the RefactorErl tool to extract the call to `sum(T)` in the second clause of `sum/1` to be bound by a new variable `S`. This can be done using the transformation Merge Expression Duplicates; the transformation actually merges all instances of the selected expression. The result of the transformation can be seen on the right side of Figure 2.

Next, let us eliminate the variable `Lst`. The result of the transformation can be seen on the left side of Figure 3. The place where the variable was applied, the first argument of the function `sum/1` in the body of function `sum_n/1`, has been changed, and the variable binding `Lst = lists:seq(?START, N)` has disappeared. Note that the current transformation creates a superfluous pair of parentheses; this will be improved in a later version of RefactorErl.

From the following examples on, the first three lines of the module are not repeated.

In the next step, we generalize the function `sum/1` by the expression `0` that can be found in its first clause. This transformation makes the selected expression a

```
fold([], _Op, Z) ->
    Z;
fold([H|T], Op, Z) ->
    S = fold(T, Op, Z),
    Op(H, S).

plus(H, S) -> H + S.

sum_n(N) ->
  fold((lists:seq(?START, N)),
       fun(H,S) -> H + S end,
       0).
```

FIGURE 6. Renamed `sum` to `fold`.

parameter of the function, and all function applications that call this function will have the selected expression inlined. The result of the transformation can be seen on the right side of Figure 3.

Let us continue by extracting the expression `H+S` to a new function `plus/2`. Only the name of the new function has to be given; the refactoring detects the free variables in the selection. The left side of Figure 4 shows the module after the extraction.

Let us generalize the `sum/2` function once more along the application of the newly generated `plus/2` function. This makes the function `sum/2` gain one more argument becoming `sum/3`. At the place of the function application, we have to pass the appropriate function; since the selection along which the generalization takes place may contain a more complex expression than a function call, the expression wrapped in a closure is passed. In a future version of this refactoring, the simpler `fun plus/2` expression may be used in this place. In the first clause of `sum/3`, `Op` is not used, therefore we write `_Op` to avoid compiler warnings.

As the next step (seen on the left side of Figure 5), we inline the body of the recently created fun expression in order to simplify the expression. This transformation replaces the function application with the body of the function. It is very useful if the function has only one clause, preferably with a simple body; in more complex cases, where the function contains more clauses, the transformation is still possible, but the resulting expression is harder to read.

Let us change the order of the last two parameters of `sum/3`. This transformation has to change the order in all clauses of the function definition, as well as all function applications.

As the final step our example, we notice that the function `sum/3` has indeed become more general, therefore we rename it to `fold/3`. The result of our transformations can be seen in Figure 6.

3. Knowledge representation

RefactorErl represents an Erlang program as a "program graph": a directed, rooted graph with typed nodes and edges. The skeleton of this graph is the abstract syntax tree of the program. Apart from syntactical information, the graph contains lexical and semantical information as well. These latter kinds of information are provided as additional nodes and edges in the graph. For example, each function in the program is represented as a semantic node in the graph; the definition of the function and all the calls to the function are linked to this semantic node with semantic edges. The maintenance of semantic information is useful for boosting side condition checking. Usually, the hardest part of refactoring is not the application of the requested transformation, but the evaluation of the conditions that are required to hold for the refactoring to be safe. These conditions often depend on a large amount of semantical information – which can be efficiently picked out from the program graph. Apparently, the stored semantical information, similarly to the AST, must be updated when a transformation is applied.

Lexical information, such as the tokens produced by the scanner, is also essential. Even information about the whitespace separating the tokens must be kept available so that the refactoring tool can preserve the layout of the refactored program.

The kinds of semantic information to be gathered and maintained by the refactoring tool depend on the transformations the tool supports. The RefactorErl tool is designed to be open-ended: it should be possible to implement a new refactoring with the relevant semantical analysis and add them to the refactoring framework. To achieve this goal, the semantical analyses are organized into independent modules, and result in independent sets of semantic nodes and edges in the program graph. Examples of semantic analysis modules are analysing scopes, analysing function definitions and calls, or analysing variable bindings. Also, new semantic analyses are easy to add to the system. One such planned but not already realised analysis collects connections between variables along the data flow.

Interestingly, the concept of using a program graph in order to represent semantic knowledge about source code is not bound to any specific programming language. Therefore, we have created a language independent framework for building and maintaining a program graph. Within this framework, we have implemented an Erlang specific model that reflects the semantics of this language, see Figure 7. The services of the framework enable the development of important components in the Erlang specific model such as the semantic analysis modules and the layout preserving parser.

RefactorErl also includes a query language, similar to XPath [25], for retrieving information from the program graph. Links of the graph can be traversed forwards and backwards, and filtering by semantical information is also supported.

To optimize the shape of the program graph for fast information retrieval, the syntax of the language is reflected in the tool at two levels of abstraction. In the more abstract view there are four syntactical categories: files, forms, clauses and expressions. Files (including header files) contain forms. Forms can be, among others, function definitions, which are made up of one or more clauses (clauses are basic building blocks of several compound expressions as well, such as `case`-expressions).

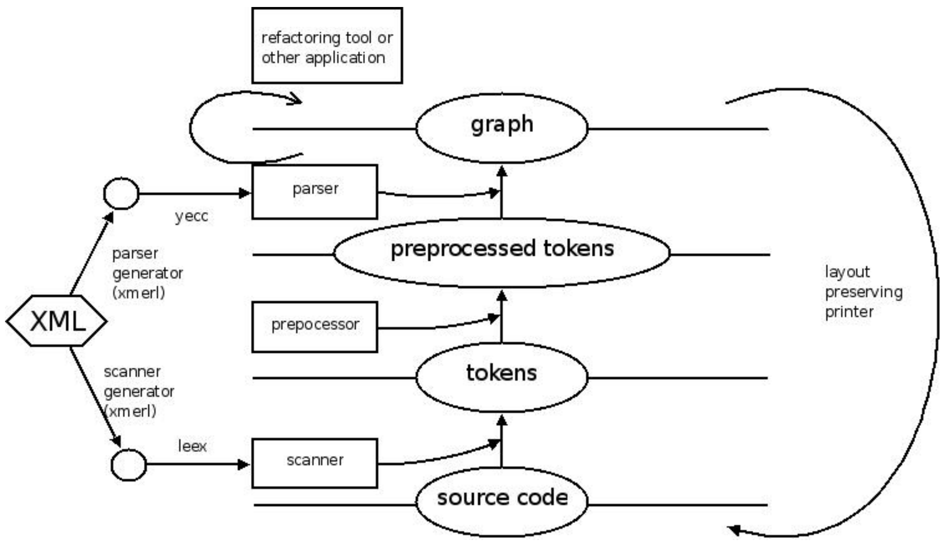

FIGURE 7. Scanning, preprocessing and parsing in RefactorErl

The right-hand side of a clause is a sequence of expressions (and the left-hand side of a clause contains further expressions such as patterns and guards). The rich syntactical structure of Erlang (reflected in the close to fifty rules of the grammar) can be abstracted into these four kinds of graph nodes. Many details of the syntax are encoded in the types of the graph edges, forming the less abstract syntactic view of the language.

The low number of types of syntax nodes improves efficiency of the queries written in the query language. Another important source of efficiency is that chains of applications of the same production rule are not represented by an unbalanced tree, but rather by a single graph node, which collects all of the syntax edges of the productions, retaining the order of the edges.

In order to improve the reusability of the refactoring infrastructure, one could design the model of the refactored programs as general and language-independent as possible. Our experience in refactoring Clean [20] and Erlang did not foster this approach. Even in these two functional languages, the syntactic and semantic differences are so significant that it is not worth to introduce a common model, not even with language-specific extensions (like in [21]). Our approach is to keep the focus of the tool on a single language, and achieve a level of precision and efficiency that is sufficient to make the tool applicable in practice.

4. THE SCOPE OF KNOWLEDGE REPRESENTABLE IN THE MODEL

Some features of the Erlang language are advantageous for refactoring: side effects are restricted to message passing and built-in functions, variables are assigned a value only once in their lifetime, and code is organised into modules with explicit interface definitions and static export and import lists. There are, however, some features that are disadvantageous for refactoring, e.g. the possibility to run dynamically constructed code, and the lack of programmer defined types. For a more detailed analysis see [8].

Refactorings, by definition, should preserve semantics. Unfortunately, it is practically impossible to guarantee this in the case of a language supporting reflection. Worse still, industrial Erlang code makes use of such facilities very frequently. On the one hand, if we design a conservative refactoring tool that always refuses to perform transformations which might alter the meaning of the refactored program, we might end up with a tool that, albeit perfectly safe, is completely useless in practice. On the other hand, a tool offering insufficient support for the preservation of semantics will never be used in practice: nobody will ever dare to refactor large programs with it. A good refactoring tool will be sufficiently safe, but not too restrictive. To achieve this, the decision mechanism in the tool should be customizable and/or interactive.

Since in general it is not possible to completely determine the meaning of an Erlang program by static analysis, a refactoring tool might decide to compensate for otherwise unsafe transformations. Inserting dynamic checks into the refactored programs often helps to bring a refactoring into effect depending on run-time information. Consider the following example. In Erlang, it is possible to construct a function call by computing the name of the function to be called and the actual arguments, and passing them to the built-in function `apply`. There are various ways of computation that are very hard to analyse by only examining the source code. It is possible to receive the function name through message passing, which requires data flow analysis in most of the cases to trace the origination. It is possible to have the user input the name of the module to be called.

Still another example, used in real life, is to store the names of the modules in a database and retrieve them from there. Let us suppose that the functions `store_mod`, `store_fun`, and `store_args` store the the name of a module, the name of a function, and arguments respectively into a database. Furthermore, let the functions `read_mod`, `read_fun`, and `read_args` retrieve the said information from the database. Then the code fragment on the left will apply `m:f/2` (that is, the binary function `f` from module `m`) on the actual arguments `1` and `2`. A refactoring tool has no chance to find out by static analysis that `m:f/2` is executed here. The only way to preserve program behaviour when refactoring `m:f/2`, for example by swapping its arguments, is to insert dynamic checks. The previous call to `apply/3` could be replaced with the expression shown on the right-hand side of Figure 8, assuming that `M`, `F`, `A`, `A1`, and `A2` are fresh variables.

5. Conclusions

In this paper, we have presented a tool for refactoring Erlang programs called RefactorErl. RefactorErl collects, stores and maintains lexical, syntactical and semantic information about Erlang source code. The tool is based on domain specific expert knowledge about semantics preserving program transformations, while improving the quality of the software product in the most common cases at the same time.

```
store_mod(m),
store_fun(f),
store_args([1, 2]),
...
apply( read_mod()
     , read_fun()
     , read_args()
     )
```

$\longrightarrow$

```
store_mod(m),
store_fun(f),
store_args([1, 2]),
...
M = read_mod(),
F = read_fun(),
A = read_args(),
case {M,F,A} of
   {m,f,[A1, A2]} -> m:f(A2, A1);
   _              -> apply(M,F,A)
end
```

FIGURE 8. Compensation for swapping the arguments of the function `m:f/2`.

We have developed a language dependent model for Erlang that serves as the basis of the tool: the model contains the semantic analyses and the semantics based program transformations. It is reasonable to refine this model step-by-step to reach the desired level of syntactic and semantical coverage; also, the model has to be flexible enough to follow changes in the language definition. We also have developed a language independent framework in which the model has been implemented. The framework enables us to produce and manipulate a program graph based on the lexical, syntactic and semantic information about the source code.

REFERENCES

[1] Barklund, J., Virding, R., *Erlang Reference Manual*, 1999. Available from `http://www.erlang.org/download/erl_spec47.ps.gz`.

[2] Brett D., et al., *Automated Testing of Refactoring Engines*, In Proc. of the the 6th joint meeting of the European software engineering conference, pages 185-194, Dubrovnik, Croatia, 2007.

[3] Charles, P., Fuhrer, R.M., and Sutton, Jr., S., M., *IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse*, In Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pages 485-488, Atlanta, Georgia, USA, 2007.

[4] Ducasse, S., Gîrba, T., and Nierstrasz, O., *Moose: an Agile Reengineering Environment* In Proceedings of ESEC/FSE 2005, September 2005, pages 99-102.

[5] Eötvös Loránd University, *Refactoring Erlang Programs* (project homepage). `http://plc.inf.elte.hu/erlang/`

[6] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code.*, Addison-Wesley, 1999.

[7] Fowler, M., *Refactoring Home Page.* `http://www.refactoring.com/`.

[8] Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., Víg, A., *Use Cases for Refactoring Erlang Programs.* To appear in Central European Functional Programming School, Revised Selected Lectures, Springer LNCS series.

[9] Li, H., Reinke, C., and Thompson, S. J., *Tool support for refactoring functional programs.* In Proceedings of the ACM SIGPLAN workshop on Haskell, Uppsala, Sweden, pages 27–38, 2003.

[10] Li, H., Thompson, S.J., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., and T. Nagy, T., *Refactoring Erlang Programs.* In Proceedings of the 12th International Erlang/OTP User Conference, November 2006.

[11] Li, H., Thompson, S.J., *Testing Erlang Refactorings with QuickCheck.* In Proc. of the 19th International Symposium on Implementation and Application of Functional Languages, IFL2007, Freiburg, Germany, Septemper 2007.

[12] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T., *Refactoring in Erlang, a Dynamic Functional Language.*, In Proceedings of the 1st Workshop on Refactoring Tools, pages 45-46, Berlin, Germany, July 2007.

[13] Lövei, L., Horváth, Z., Kozsik, T., Király, R., *Introducing records by refactoring.* In Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pages 18-28. ACM Press, 2007.

[14] Lövei, L., Horváth, Z., Kozsik, T., Király, R., and Kitlei, R., *Static rules of variable scoping in Erlang*, In Proceedings of the 7th International Conference on Applied Informatics, volume 2, pages 137-145. 2008.

[15] Mitchell, B.,S., *A heuristic search approach to solving the software clustering problem*, PhD thesis, Drexel University, Philadelphia, PA, USA, 2002.

[16] Nyström, S., *A soft-typing system for Erlang*, Proceedings of the 2003 ACM SIGPLAN workshop on Erlang, pp. 56-71, Uppsala, Sweden, 2003.

[17] Robbes, R., Lanza, M., *The "Extract Refactoring" Refactoring*, In Proceedings of WRT 2007 (1st International Workshop on Refactoring Tools), pp. 29 - 30, Berlin, Germany, 2007.

[18] Roberts, D.: *Practical Analysis for Refactoring*, PhD thesis, University of Illinois at Urbana Champaign, 1999.

[19] Roberts, D., Brant, J., and Johnson, R., *A Refactoring Tool for Smalltalk*, Theory and Practice of Object Systems. V3 N4, October 1997.

[20] Szabó-Nacsa, R., Diviánszky, P., and Horváth, Z., *Prototype environment for refactoring Clean programs.*, In The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4, 2004. Full paper is available at `http://aszt.inf.elte.hu/~fun_ver/` (10 pages).

[21] Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O., *A Meta-model for Language-Independent Refactoring*, Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00), IEEE Computer Society Press, 2000, pp. 157-167, Kanazawa, Japan, November 2000.

[22] Vinju, J.J.: *Uptr: a simple parse tree representation format*, In Software Transformation Systems Workshop, October 2006.

[23] Vinju, J.J, *Analysis and Transformation of Source Code by Parsing and Rewriting*, PhD thesis, November 2005.

[24] Wloka, J., Hirschfeld, R., and Hänsel, J., *Tool-supported Refactoring of Aspect-oriented Programs*, In Proceedings of the Conference on Aspect-oriented Software Development (AOSD), pages 132-143, Brussels, Belgium, March 31 - April 4, 2008.

[25] World Wide Web Consortium: *XML Path Language (XPath) Version 1.0.* W3C Recommendation, Nov. 16, 1999, `http://www.w3.org/TR/xpath.html`

[26] Wiger, U., *XMErl – Interfacing XML and Erlang*, In the Sixth International Erlang/OTP User Conference (EUC 2000), Stockholm, Sweden, October 3, 2000.
`http://www.erlang.se/euc/00/xmerl.ppt`

(1) Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2009
Cluj-Napoca (Romania), July 2–4, 2009, pp. 17–25

AN OVERVIEW OF DISTRIBUTED USAGE CONTROL – EXTENDED ABSTRACT –

ALEXANDER PRETSCHNER

Abstract. Usage control generalizes access control to what happens to data in the future ("delete after thirty days," "do not copy," "notify owner upon access.") Distributed usage control is about defining and enforcing usage control requirements on data after giving it away. It is relevant in the areas of data protection, the management of intellectual property, the management of secrets, and compliance with regulations. In this extended abstract, we provide an overview of the field. We introduce fundamental concepts, requirements, policy specifications, policy analyses, dissemination models, the enforcement of usage control requirements at different levels of abstraction, and the challenges ahead.

1. Introduction

Ever increasing amounts of digital data require procedures and mechanisms for secured access to and usage of that data. Because we live in an interconnected world, this problem not only extends to the original provider of a data item, but also to all those parties who have, transitively, received a—possibly modified—copy of this data item. The subject of *usage control* [13, 6] is the definition and enforcement of access control requirements that relate to actions and state information before data is released, and also to usage control requirements that relate to actions and state information after the data is released. We distinguish between two kinds of requirements, *provisions* that reflect access control requirements, and *obligations* that reflect requirements on the future usage [6]. Usage control requirements include permissions ("song may be played at most twice," "document must not be printed," "financial statements must be retained for at least five years") and duties ("data owner must be notified upon each access," "song must be paid for after listening five times," "data must be deleted after thirty days"). While both kinds of requirements are stipulated in *usage control policies*, we will only be concerned with obligations in this extended abstract.

1.1. **Relevance.**

Usage control is relevant in at least the areas of privacy, the management of intellectual property, the management of secrets, and compliance with regulations. In terms of privacy, users may want, among other things, their medical

Key words and phrases. Access Control, Usage Control, Enforcement, Policies, Trust.
This work was supported by the FhG Internal Programs under Grant No. Attract 692166 as well as by the EU under the project FP7-IST-IP-MASTER.

information, loyalty card records, telecommunication connection records, and banking information to be kept under wraps (sometimes they may not, as the current immense popularity of largely unprotected social network sites shows). At least in Europe, in addition to EU-wide legal requirements, current developments clearly indicate a potential or actual political fallout when large sets of citizen or customer data become available.

In terms of intellectual property, business processes are increasingly implemented in a distributed manner, fueled by an increasing trend to outsourcing. To create added value, this requires the exchange of usually confidential information, including blueprints, results of experiments, and the like. Companies may have an interest that such data is solely used according to their expectations. As far as digital rights management is concerned, artists may be granted a vested interest in receiving royalties for their artwork.

In terms of secrets, most administrations, state departments, intelligence agencies and the military may want to have control over how specific information is disseminated.

Finally, usage control is becoming increasingly mandatory. Regulations such as the EU directive 95/46/EC require data to be collected and used according to a specified purpose; the Sarbanes-Oxley Act (SOX) requires specific data to be retained for five years; and the US Health Insurance Portability and Accountability Act (HIPAA) stipulates strict documentation requirements on the dissemination of data [9].

1.2. **Big Picture.** We assume the following scenario. A *consumer* wants to get access to data. To do so, negotiations on the terms and conditions of usage take place with the *provider*. The negotiations result in a *policy* that encodes rights and duties related to the data item and that reflects regulations, non-disclosure agreements, or any legally binding contract. After checking if the consumer can enforce the policy, the provider sends both data and policy to the consumer. In order to prevent simply throwing away the policy, this is likely to involve some encryption mechanism. At the consumer's side, the data is stored so that only well-specified *enforcement mechanisms* can access it. This, again, is done by cryptographic means; such ideas are currently implemented at different levels of abstraction, including the operating system where so-called data caging takes place. The mechanisms are configured by the policy. Consumers attempt to use (render, process, execute, disseminate) the data. The enforcement mechanisms then either check if this is allowed, by turning the attempted usage into an actual usage, or if a policy violation has occurred, report the violation.

1.3. **Overview.** Challenges related to usage control roughly encompass the specification of requirements, including their evolution upon dissemination; the enforcement of respective policies; and the assessment of how easily these enforcement mechanisms can be circumvented. These issues are considered in the remainder of this paper, mostly from the perspective of our own work, which is explained by the very nature of this article.

2. Requirements, Policies, Analysis, and Evolution

Usage control requirements can be classified into permissions and duties. Both kinds usually specify conditions in which data may be used or in which actions need to be taken. Conditions relate to time ("within thirty days"), cardinality ("copy at most three times"), purpose ("personal use only"), events ("upon access"), and both the technical ("Windows RMS must be installed") and organizational ("virus scanner databases need to be updated every week") environments [7, 16].

Many policy specification languages have been defined (including [1, 4, 24, 23, 7]), a few of them also with formal semantics.[1] This formal semantics, by its very nature, however is restricted to the aspects that are captured by temporal, modal, and first-order logics. Propositions ("print," "copy," "delete") are, essentially because of their high level of abstraction, much harder to define precisely.

In current languages, policies are mostly specified in terms of events ("play," "copy," "delete"). For well-specified rendering devices in a DRM context that make use of a standardized ontology, this is often sufficient. However, as we will see, usage control requirements can be enforced at different levels of the software stack. One level includes the operating system; relevant events are then system calls. Exhaustively defining the notion of "deletion" in terms of sequences of system calls (unlink, mv to a null device, overwrite, ...) seems like a Sisyphean endeavor. One possible remedy is to track data flow through the system, and encoding as abstract state the (overapproximated) mapping from data containers to data items. Deletion in state-based terms means that in a given state, no container may contain the data item. Similarly, prohibiting dissemination can be expressed by stating that the a data item is in at most one data container.

It is useful to distinguish between three kinds of policies. *Specification policies* declaratively state what should be the case ("no non-anonymized data must leave the system"). The decision of how to enforce a policy—by modification, inhibition, or execution—is done in *implementation policies.* The example policy can be enforced in at least two ways: by modifying the data record's name, birthdate, and address fields into blanks; and by simply blocking all data packets which are not anonymized in the sense that their name, birthdate, and address fields are not blanks (which of course is a very rough definition of anonymization). Implementation policies usually come in the form of classical operational Event-Condition-Action rules, where the condition must be specified over the present and the past rather than the future (otherwise it could not be checked). Finally, *configuration* policies are rights objects that can directly be understood by implementations of enforcement mechanisms. In model-driven engineering terms, configuration policies are platform-specific while specification and implementation policies are platform-independent.

At the abstract level, checking if an implementation policy makes sure that a specification policy is fulfilled amounts to checking entailment of logical formulae. Respective reasoning technology can also be used to assess the consistency or subsumption relationships between policies [19].

[1]Note that we tacitly assume policies to be data-centric rather than server-centric; this is not a conceptually fundamental distinction, however.

When data is disseminated, some policy will have to be associated with it as well. In some sense, this policy should at most be a "strengthened" version of the original policy—otherwise, a subject could send the data item to itself, together with a relaxed policy that allows everything and requires nothing. Strengthening permissions can be done by restricting them; strengthening duties should then, dually, mean that they are increased. By ordering events in lattices and specifying lower and upper bounds for both permissions and duties, one can uniformly express the strengthening of policies as a combination of logical entailment and interval reduction [20, 19].

Widely applicable usage control frameworks must cater to the problem of policy management as well, including overwriting and revoking policies as well as handling conflicting policies.

3. Enforcement

The fundamental problem of enforcing usage control requirements is that a data provider usually has no control over nor inspection into the IT infrastructure of a data consumer (note that the roles dynamically change upon re-distribution of data). In order to prevent simple interceptions or retrievals of sensitive data, data must be stored and transmitted in encrypted form. Moreover, at the consumers' side, tamper-proof and trustworthy monitoring and control devices must be in place. These devices can come as special rendering software (such as the Adobe Acrobat Reader in conjunction with the respective rights management system), or as add-ons to a system, similar to malware intrusion detection systems.

3.1. Reactive and Preventive Enforcement.

One way of enforcing usage control requirements is *by observation*, or reactive. Provided that adherence to policies can be monitored, one can at least detect the violation of a policy and react by undoing the violating action, by penalizing the wrong-doer, or by performing compensating actions. This is similar to how human law enforcement works [14].

Another way of enforcing respective properties is *by control*, or preventive. The goal here is to prevent a policy violation from happening. Rather than observing events post factum, one must usually observe *requests*, anticipate the events that would be a consequence of these requests, and then either inhibit the request, modify it, or execute some action. For instance, a policy "no non-anonymized data may leave the system unnoticed" can be enforced by dropping the request that asks for a non-anonymized data item (inhibition); by anonymizing the data item (modification); or by logging the event that the data item was released (execution) [17, 16]. Whether a reactive or a preventive enforcement strategy is to be chosen cannot be decided in general; this depends on the trust relationship of provider and consumer, and also the value of the data items that are exchanged [18].

Particularly enforcement by control can quickly become impossible. This is often the case when media breaks occur. Once a song is played, i.e., transformed into sound waves and has thus left the scope of a controlled (the boundaries of which need to be defined), one can externally record and replay it. Similarly, a document can be printed or photographed from a screen, thus rendering most control devices useless. Watermarking schemes have been developed for these situations. Their purpose is to subject such data to the possibility of more or less random observation mechanisms, and hoping that non-rightful possessors of data will be deterred.

On the other hand, enforcement by control may turn out to be too intrusive. If usage control is applied to Java API calls, for instance, then one can of course block calls that are considered sensitive, e.g., calls to text message APIs in mobile phones. However, unless the original system is programmed defensively and always anticipates the potential failure of a method invocation, this is likely to lead to a crash—many exceptions tend to be caught only at the bottom-most stack frame. This problem turns out to be particularly challenging in asynchronous communication infrastructures such as service-oriented architectures.

In any case, there never is complete security in life or it would be too restrictive or too expensive anyway. We believe that increasing the barriers for accessing and using data in non-permitted ways is sufficient in a large majority of cases. Security in the sense of usage control must be subject to risk analysis, and we believe in the idea of *just-right-security*, similar to the analogous concept of *just-right-reliability* [12]. Needless to say, we are of course aware that any solution is most likely to encompass both technical and organizational means.

3.2. **Signaling, Monitoring, and Enforcement proper.** We have alluded above that enforcement mechanisms can abstractly be perceived as sets of Event-Condition-Action rules. Conditions relate to the current state of the system (which in many cases encodes past events); events are triggers; and actions define whether or not an event is inhibited (which requires the distinction into requests and actual actions), modified, delayed, or if another action is to be executed. From an architecture perspective, this necessitates three components: *signalers* that make events visible so that a *monitor* can check if the condition is true, and *enforcement components* that perform the respective action. Consider a policy that requires a movie to be played at most three times before it is paid for. A signaler must provide information on payments and whether the movie is played or attempted to be played; a monitor must count the number of times the movie is played; and the enforcer must then issue a payment or block the attempt to play the movie. Note that these components can but need not necessarily reside on one single machine [15].

Monitoring can be achieved by many different technologies, including rewriting logics, state machines that implement policies, and complex event processing technologies (where it is then usually called aggregation). For signalers, it is likely that the subsystem that generates events must be instrumented; this is sometimes but not always the case for monitors.

3.3. **Levels of Abstraction.** Enforcement, which from now on we understand to include signaling and monitoring, can be implemented at different levels of the software stack. Consequences include both the universe of discourse of the policy language (for instance, messages vs. files and natural language terms such as deletion vs. unlink()) and—of course depending on the chosen trust or attacker model—the guarantees that can be given. It is likely that a combination of several—rather than one single—enforcement mechanisms at different levels will be necessary to provide guarantees (e.g., dissemination requirements relate to both files and screenshots).

The lowest level at which software-based usage control can be applied is the *CPU/virtual machine*: specific calls or references to memory cells can be blocked or modified. At this level, data flow within an application (or business logic) or between processes can be controlled or at least monitored. If the content of a file is written to a specific memory location, for instance, then subsequent reading operations can be detected.

The next level is the interface to the *operating system*. At this level, system call interposition can take place: system calls that access specific files, for instance, can be monitored and possibly modified or blocked. Re-distribution at a rather rough level can be controlled: if a process has accessed a file, it is possible to forbid all subsequent communication with other processes, file systems, networks, etc. This is likely to be considered an impediment by the users. Examplary base technologies include systrace [22] for OpenBSD and the Detours framework for Microsoft Windows [8].

At the level of the *runtime system*, calls to libraries can be monitored, blocked, and modified. Specific library calls and their parameters for communication can be prohibited, for instance, which provides a more fine-grained control than prohibiting all future communication. One example is the Polymer system that modifies Java byte code on the go [5].

At the level of dedicated *application wrappers*, applications can be controlled on the grounds of abstractions that relate to the specific application. As an example, the UNO framework for OpenOffice allows to control copying text between specific documents rather than generally prohibiting copy and paste.[2]

At the level of *applications*, all kinds of usage control can be implemented. However, the question arises how the guarantees can be assessed. This is of course simpler if pre-defined components are installed at the consumer's side and data is only given to the client if the respective components are in place.

If we distinguish between end-user applications and *infrastructure applications*, such as the X11 server or the window manager, then control can also be exercised at the level of infrastructure applications. Controlling access to clipboards, for instance, can also take place at the level of the X11 server. Another example are *data base systems* with usage control mechanisms in place [2].

Usage control can also be implemented at the levels of wrappers for *services* [9, 3, 18]. The AXIS framework, for instance, allows the definition of handlers that intercept incoming and outgoing messages. Wrappers can also be written in an ad-hoc manner and are likely to operate at the level of messages (but need not necessarily, depending on the implementation).

Finally, usage control that operates on messages can be implemented at the level of the *enterprise service bus*. This is the least intrusive yet most vulnerable approach:

[2]The latter is all that can be expected at the level of the X11 server because the X11 server knows about windows, possibly mapping them to processes, but not about single documents that are simultaneously opened by one process.

XML tags can simply be compromised. At the level of an orchestration engine for service-oriented architectures, reactions to a policy violation can be initiated.

Note that in general the step from a lower-layer enforcement framework to a higher-layer enforcement framework is possible but non-trivial, particularly so if information flow is to be taken into account. One can monitor both the explicit and the implicit flow of sensitive data through a Java program. However, if this data is to be rendered on a screen, an AWT or Swing method will be invoked which will then call some native routines which, in case of a Unix system, will invoke X11 libraries. Respective interfaces that help identifying the flow of sensitive data across the levels must be defined.

3.4. **Assurance.** Among other things, the above logical architecture requires signalers to be complete and trustworthy. There must be means to make sure that signalers are not simply switched off. Monitors must be guaranteed not to miss events, and they must also be guaranteed to run as long as usage control is, according to the provider, to be exerted. Finally, the actions executed by a mechanism must be definitive and not easily overwritten.

Data providers likely want to know if a suitable enforcement framework is in place at the consumer's side [17]. Upon negotiation [21] over a data item to be released, the provider wants to know if the consumer can enforce the requirements as specified in the policy. At the logical level, this again boils down to an entailment problem [17]. At the more technical level, we believe that remote attestation on the grounds of trusted computing technology [3] is a promising candidate technology in this respect.

A further challenge obviously relates to the management of cryptographic keys that are the prerequisite for avoiding the simple interception and unconstrained usage of data before it enters the respective enforcement mechanisms.

4. Conclusions

Distributed usage control is about making sure that a data consumer handles data according to the rules as set forth by the data provider. Without entering the moral discussion of whether or not this is always desirable, we see usage control as an indispensable enabler when it comes to data protection or privacy, the management of intellectual property in distributed business processes, the management of secrets, and compliance with data-related regulations.

In this article, we have provided a big picture of the field, ranging from the model world (requirements, policies, analysis problems) to implementations of enforcement mechanisms that, in addition to fulfilling their functional specifications, must also be tamper-proof.

There are many open research, engineering and business problems left. Starting with the latter, it is not entirely clear what adequate business models are, and who is going to pay for usage control—the idea of usage control as an enabler must also be translated in business terms. Research problems in particular include problems related to suitable trust models; policy management schemes in different organizational settings (enterprises; the Internet); the conceptually clean connection between different levels of abstraction for enforcement; the difficult problem of de-classification when information flow is tracked; a better understanding of information flow, including quantitative measures [11, 10]; and the question of how we can, qualitatively or quantitatively, measure the guarantees that a usage controlled system can provide. Finally, engineering problems relate to efficient signaling, monitoring, and enforcement technologies at different levels of abstraction, secure key storage, and the question of how it can be assured that a particular mechanism is in place at the consumer's side.

5. Acknowledgment

We would like to thank T. Walter and C. Schaefer from DOCOMO Euro-Labs for many fruitful discussions. D. Hupel, C. Lorini, O. Maschino, M. Radulescu, and S. Willenbrock commented on an earlier version of this article.

References

[1] Open Digital Rights Language - Version 1.1, August 2002. odrl.net/1.1/ODRL-11.pdf.
[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, pages 143–154, 2002.
[3] B. Agreiter, M. Alam, R. Breu, M. Hafner, A. Pretschner, J. Seifert, and X. Zhang. A Technical Architecture for Enforcing Usage Control Requirements in Service-Oriented Architectures. In *Proc. ACM workshop on Secure Web Services*, pages 18–25, 2007.
[4] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *Proc. ESORICS*, LNCS 2808, pages 162–180. 2003.
[5] L. Bauer, J. Ligatti, and D. Walker. Composing Security Policies with Polymer. In *Proc. PLDI*, pages 305–314, 2005.
[6] M. Hilty, D. Basin, and A. Pretschner. On obligations. In *Proc. ESORICS*, Springer LNCS 3679, pages 98–117, 2005.
[7] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A Policy Language for Distributed Usage Control. In *Proc. ESORICS*, pages 531–546, 2007.
[8] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc. USENIX Windows NT Symposium*, pages 135–143, 1999.
[9] V. Lotz, E. Pigout, P. Fischer, D. Kossmann, F. Massacci, and A. Pretschner. Towards Systematic Achievement of Compliance in Service-oriented Architectures: The MASTER approach. *Wirtschaftsinformatik*, 50(5):383–391, October 2008.
[10] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. PLDI*, pages 193–205, 2008.
[11] C. Mu. Quantitative information flow for security: a survey. Technical Report TR-08-06, Department of Computer Science, King's College London, September 2008.
[12] J. Musa. *Software Reliability Engineering.* AuthorHouse, 2004.
[13] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Information and Systems Security*, 7:128–174, 2004.
[14] D. Povey. Optimistic security: a new access control paradigm. In *Proc. workshop on new security paradigms*, pages 40–45, 1999.
[15] A. Pretschner, M. Hilty, and D. Basin. Distributed Usage Control. *CACM*, 49(9):39–44, September 2006.
[16] A. Pretschner, M. Hilty, C. Schaefer, F. Schütz, and T. Walter. Usage Control Enforcement: Present and Future. *IEEE Security and Privacy*, 6:44–53, July/August 2008.
[17] A. Pretschner, M. Hilty, C. Schaefer, T. Walter, and D. Basin. Mechanisms for Usage Control. In *Proc. ASIACCS*, pages 240–245, 2008.
[18] A. Pretschner, F. Massacci, and M. Hilty. Usage Control in Service-Oriented Architectures. In *Proc. TrustBus*, pages 83–93, 2007.

[19] A. Pretschner, J. Rüesch, C. Schaefer, and T. Walter. Formal Analyses of Usage Control Policies. In *Proc. AReS*, 2009.
[20] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. In *Proc. 4th Intl. Workshop on Security and Trust Management*, pages 97–110, 2008.
[21] A. Pretschner and T. Walter. Negotiation of Usage Control Policies—Simply the Best? In *Proc. AReS*, pages 1035–1036, 2008.
[22] N. Provos. Improving host security with system call policies. In *Proc. SSYM*, pages 257–272, 2003.
[23] W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, 2005.
[24] X. Wang, G. Lao, T. DeMartini, H. Reddy, M. Nguyen, and E. Valenzuela. XrML – eXtensible rights Markup Language. In *ACM workshop on XML security*, pages 71–79, 2002.

Fraunhofer IESE and TU Kaiserslautern, Germany
E-mail address: `alexander.pretschner@iese.fraunhofer.de`