How to make a bridge between transformation and analysis technologies?

J.R. Cordy and J.J. Vinju

July 19, 2005

1 Introduction

At the Dagstuhl seminar on "Transformation Techniques in Software Engineering" we had an organized discussion on the intricacies of engineering practicle connections between software analysis and software transformation tools. This abstract summarizes it. This discussion contributes mainly by explicitly focussing on this subject from a general perspective, and providing a first sketch of a domain analysis. First we discuss the solution space in general, and then we compare the merits of two entirely different designs: the monolithic versus the heterogeneous approach.

Acknowledgements There were many people attending this discussion, among others: James R. Cordy, Thomas R. Dean, Rudolf Ferenc, Michael Godfrey, Robert Hirschfeld, Kim Mens, Tom Mens, Alberto Pettorossi, Markus Pizka, Roel Wuyts, Jurgen J. Vinju. The subject has been discussed earlier at the STS workshop in Vancouver, collocated with GPCE 2004.

Two domains We start from the premise that transformation of source code and analysis of source code are two different domains. For each domain specific tooling has been constructed with success. From practical experience, we have concluded that analysis and transformation technology complement eachother naturally when applied to general automated software engineering activities. Examples of transformation tooling are the domain specific languages TXL [2], ASF+SDF [1] and Stratego [8]. Tools like Grok [5], SQL, or even Prolog are considered to be analysis tools from the perspective of this discussion.

Source code representations Each domain is specialized on different kinds of source code representations. On the one hand, transformation tools work on representations of source code that closely follow the structure of the original source code. Examples of such representations are the files themselves, parse trees and abstract syntax trees or graphs. On the other hand, analysis tools deal with more abstract representations, that are frequently referred to as "facts"

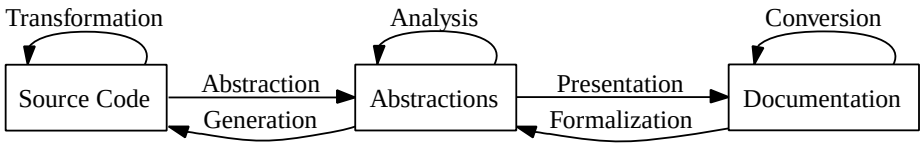

Figure 1: Three source code representation tiers

about source code. Such facts are usually represented in data-bases containing relations between selected source code artefacts.

Question Figure 1 illustrates possible connections between source code representation tiers. Apparently, such connections play a central role in the design and implementation of automated software engineering tools. The following quality attributes are most important for these bridges: transparency, efficiency, and consistency. Secundary quality attributes we consider are ease-of-use, separation of concerns and maximal reuse of existing technologies. The overall question we pose is:

> *Given one arbitrary transformation tool, and one arbitrary analysis tool, how to construct a high quality bridge between the two?*

We do not exclude the possiblity that the choice of transformation and analysis tools heavily influences the answer to this question. However, we do wish to focus on the actual bridge, not the intricasies of the separate tools.

2 The solution space

By discussing several examples of bridges that have been applied successfully in the past we analyzed the solution space. In the interest of brevity, we present here only the results of this analysis, not the particular examples we discussed to obtain it. The examples discussed were TXL/Grok, Soul [6], Columbus [4], ASF+SDF/Rscript, and Stratego.

We first identified the fact that we deal mainly with *Computation* in both the transformation and analysis domains, and *Communication* between these domains. We explicitly ignore the solution space and variability of the computations. However, there is one requirement that the respective computations must satisfy that is virtually impossible to separate from the actual computation.

2.1 Computation

The concept of *identity* of source code artefacts is pivotal in the construction of a bridge. A source code artefact is a selected piece of information from the source code, such as a class name or a method body. To be able to communicate, both computations need to agree on a *unique* encoding of each artefact. Uniqueness

ensures consistency of facts across the bridge, such that each fact refers to the right source code artefact(s) of the source code in either domain.

> *Obtaining and maintaining unique encodings of source code artefacts is hard.*

The complete syntax, static semantics and for some programming languages even dynamic semantics may be needed to compute a unique representation of a source code artefact. For example, *scoping rules* influence computing an identify of a source code artefacts, because names may be reused in different scopes. A method name is not unique enough to identify a particular method.

Note that an encoding can be *application specific*, *object language specific* but *application independent*, or completely *object language idendependent*. In our examples, we saw mainly language specific solutions, that provided a specific mapping from source code artefacts to unique representations for one particular object language. One system used a language independent mapping; each artefact was identified by a *location* encoded by filename, with the offset and length in characters in the file for each artefact.

2.2 Communication

Figure 2 illustrates the variability of the communication process that we identified. Of course this communication process is governed by general communication principles. Here we will briefly touch upon each branch in Figure 2, and detail its importance in the context of our general question.

Protocol Regarding the communication protocol, we identified two possibilities. Either it is implemented using *Co-routines* between the two computational processes, or by staging the computation processes seperately and communicating via serialized representations (files). In the co-routine case, we must select a *driver* computation that calls the other computational process at the appropriate times. The *direction* is two-way in this case, parameters are given to the co-routine and results are returned back from it. In the batch case, we may freely choose the direction depending on the application in mind.

On the one hand, the co-routine solution is attractive because the bridge is completely transparant and consistency of facts may be easily obtained due to the tight integration. However, the construction of co-routines either requires a complete integration of two tools, or a tailor-made low level communication protocol between to tools, or the use of possibly complex middleware.

On the other hand the batch approach ensures reuse of existing tools without much plumbing. The process may be easily streamlines using simple middleware solutions (XML). However, maintaining data consistency may be hard because the two computations are so independent, and the serialization process that comes with it may become hard to understand (not transparent).

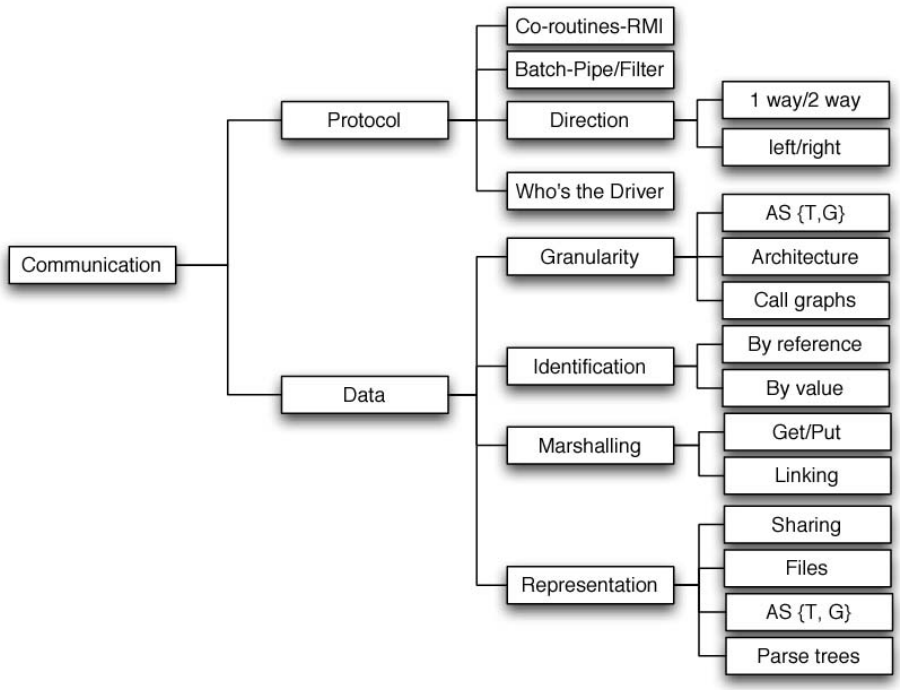

Figure 2: How to communicate source code artefacts

Granurality The usefullness of a bridge is influenced by the granularity of the facts that communicated over it. The efficiency of the bridge also depends on this measure. High granularity often implies a lot more facts that need to be communicated. Three classes of granularity of source code fact extraction have been identified in a previous Dagstuhl seminar, in order of decreasing granularity: low level, mid-level and architectural [7]

Identification We discussed the identification issue above, since it influences both the computation and the communication processes. In terms of communication we witnessed examples of identification by reference, and identification by value. Using by value, a representation of an entire artefact is passed to the other computation. Using by reference, some coded identifier is given to each source code artefact before communication starts. These identifiers may be *stable* or *unstable* across runs of the transformation and analysis tools.

Marshalling The actual transport of data, the marshalling can be done by serialization, *putting* the data somewhere such that the other computation may *get* it. It may also be done by linking the two processes together and using shared memory. In both cases, transformation of the data from one representation into another may be necessary. An examples of such a transformations is an export

filter to GXL [9].

Note that the type of marshalling that is needed depends highly on the communication protocol that is chosen. A co-routine based protocol combines easily with shared memory, while batch protocols require serialized data.

Representation The representation chosen to communicate may highly influence quality attributes of a bridge. For high granulirity facts, a *compact* or even *shared* representation may avoid a communication bottleneck. On the other hand, a *readable* representation improves the transparency of a connection (for debugging purposes).

Most systems use a *generic* representation mechanism, such that it can be reused in several automated software engineering application. Examples of file based representations are RSF and GXL. ATerms are used to easily construct maximally shared representations.

3 Monolithic versus Heterogeneous

We compared two designs. The *monolithic* design links the two computations together in one process and uses co-routines to alternate. The data is marshalled using an API that translates source code artefacts on-the-fly to either representation.

The *heterogeneous* design picks a batch approach, staging the two communications by filtering serialized representations. It uses import/export filters to marshall the serialized representations.

Monolithic approach Advantages are that fact consistency is easier, and the marshalling layer can be extremely thin. The disadvantages are that the benefits of specializing on a particular domain (analysis or transformation) may be lost. We refer to Czarnecki [3] for the benefits of domain specific languages. Another disadvantage is the tight coupling on both the implementation level and the user level: everything depends on everything.

Heterogeneous approach One advantage is that the serialization process enforces an explicit (documented) contract between the two communication parties. Another is that a tool may be easily replaces by another in this architecture, due to the loose coupling. A big disadvantages are that the marshalling of data may be very complex. The identification issue is more complex in the batch approach (stability of identifiers). Moreover, the consistency of extracted facts is harder to validate automatically: source code may be updated which may not be reflected in previously serialized and transported facts.

Discussion We were unable to ascertain which design offers higher quality. The circumstances, or goals of an automated software engineering tool, influence this. For example, in an *interactive* setting, the monolithic approach may be

advisable to optimize response time. The heterogeneous approach is much more flexible, and may therefore be better applicable to *object language independent* settings.

4 Conclusion

We have briefly touched upon several issues regarding the construction of bridges between analysis and transformation tools. The contribution of the discussion is that we have identified this as a subject of study, as opposed to considering it a trivial engineering issue. The quality of automated software engineering tools depends on it.

Future work We should invest in a rigorous domain analysis of this subject, including requirements, trade-offs and constraints, with practical solutions that may be applied in different circumstances. We may use a thorough analysis of several existing systems from this perspective to validate our recommendations.

References

[1] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[2] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[3] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[5] R.C. Holt. An introduction to the Grok programming language. Technical report, University of Waterloo, May 2002.

[6] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, December 2002.

[7] Erhard Ploedereder Timothy Lethbridge, Sander Tichelaar. The Dagstuhl Middle Metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

[8] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[9] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.