Vanda Studio – Instructive, Rapid Experiment Development

Matthias Büchse*
Technische Universität Dresden
matthias.buechse@tu-dresden.de

Abstract

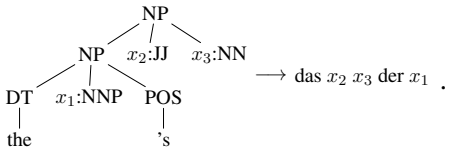
Statistical machine translation research routinely involves conducting experiments on a computer. Designing and running those experiments is tedious, because many different programs have to be operated in concert, and the threshold for novices is high. This paper reports on Vanda Studio, an integrated development environment that allows for rapid incremental design of small-scale experiments, both for teaching and prototyping.

1 Introduction

Research in statistical machine translation (SMT), as well as building an SMT system, routinely involves the task of designing and running experiments on a computer. Engineers who take on that task can be compared to conductors, because many computer programs must work in concert for an experiment to work. Unfortunately, these programs do not adhere to a coherent standard when it comes to command-line syntax, data formats, directory structures, and documentation.

Consequently, the engineer's life is made easier when the execution of these programs is automated, and when these programs are integrated into a coherent environment. Even more profoundly, so are the lives of students and researchers alike who want to enter the area of SMT.

This paper is a report on Vanda Studio, an integrated development environment that allows for rapid incremental experiment design, in particular for learning and teaching SMT. As an example of an experiment, we consider the tree-to-string approach to SMT (Yamada and Knight, 2001; Galley et al., 2004; Huang et al., 2006; Graehl et al., 2008), where the translation process follows rules such as



Using this rule, among others, one might translate a syntax tree of "the Commission's strategic plan" into "das langfristige Programm der Kommission". The rules are extracted from existing translations. Roughly speaking, the following four steps are performed (cf. Fig. 1).

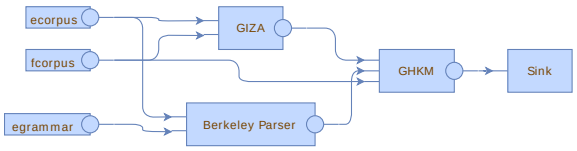


Figure 1: Rule extraction experiment, visualized as a workflow diagram.

First, obtain a parallel corpus of existing translations, e.g., from the Europarl proceedings (Koehn, 2005) or the Canadian Hansards. The parallel corpus is represented as two sentence-aligned monolingual corpora (called ecorpus and fcorpus in the figure). Second, compute a word alignment for each sentence pair using the program GIZA (Och and Ney, 2000), plus a program for symmetrizing alignments (Och and Ney, 2003, Section 4), which is not shown. We refer the reader to (Koehn, 2010, Sections "Run GIZA" and "Align Words") for details. Third, determine a syntax tree for each English sentence, i.e., convert the corpus into a tree bank. To this end, we may use the Berkeley parser (Petrov et al., 2006), egret, or some other parser. Finally, apply the GHKM rule extraction algorithm (Galley et al., 2004) to the alignments file, the tree bank, and the foreign-language corpus. An implementation by Michael Galley is available from Stanford (Galley, 2010).

*With input from Heiko Vogler, Toni Dietze, Johannes Osterholzer, and Torsten Stüber, and help from Tobias Denkinger, Kilian Gebhardt, Anja Fischer, Linda Leuschner, Hans-Jakob Holtz, and Ralf Müller.

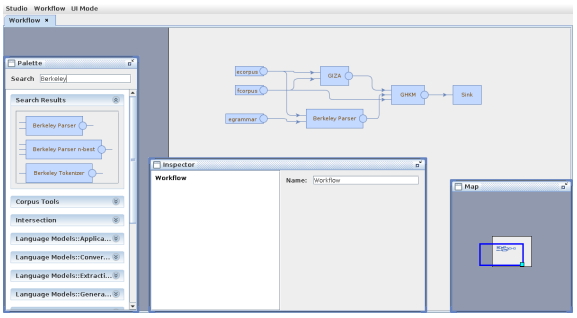


Figure 2: Vanda Studio GUI Layout.

Together these four steps comprise an experiment.

A user of Vanda Studio constructs a workflow diagram such as in Fig. 1 to describe the experiment. In fact, the figure has been exported from Vanda Studio to PDF. The user can run the experiment from inside Vanda Studio, and she can easily inspect pieces of data thanks to data visualization, e.g., for word alignments and trees. She can alter the experiment and rerun it; there are no unnecessary duplicate computations.

We envision the following use cases for Vanda Studio: playfully getting acquainted with SMT tools and their interaction, incrementally designing small-scale experiments, e.g., in front of a class or for rapid prototyping, collecting (and possibly grading) experiments designed by students.

The main part of this paper (Sections 2–8) is a tour of the basic abstractions and functions that define Vanda Studio. Section 9 shows related work, and Section 10 concludes the paper.

2 GUI Layout

The GUI consists of four major parts (see Fig. 2):

- the workspace (background). The user freely arranges the workflow elements using the mouse.
- the tool palette (bottom left). The user drags tools into the workspace in order to add a step to the experiment.
- the inspector window (bottom center). It displays information regarding the currently selected element, including data visualization.
- the map (bottom right). It is used to navigate the workspace.

3 Workflows

The basic setup of an experiment is described by a workflow such as the one in Fig. 1. A workflow is a directed acyclic graph that consists of the following three parts:

- Jobs with input and output ports. A job is either a tool instance, i.e., an invocation of a program such as GIZA, or a literal, i.e., a variable such as ecorpus, which has a fixed value during the course of one run of the workflow. In the diagram, a job is depicted as a box with input ports to the left and output ports to the right.
- Locations. A location holds a piece of data during and after the execution of the workflow; apart from a few exceptions, a location is a file. Locations and output ports are in a one-to-one correspondence. In the diagram, a location is depicted as a circle at an output port. Locations are used to inspect data.
- Edges, which indicate data flow. An edge connects a location to an input port. A location can have several outgoing edges, or none, while an input port must have exactly one ingoing edge.

Although not shown in the diagram, ports and locations are typed. As a consequence, one cannot connect a location holding a grammar to an input port that expects a sentence corpus. Likewise, one can only assign a value to a literal if the type is respected, i.e., ecorpus must be a sentence corpus, while egrammar must be a grammar for the Berkeley Parser. The type system is not only a safety measure—type information is also used by Vanda Studio to select the GUI for data visualization.

An alternative, more explicit representation for workflows is illustrated in Fig. 3. It is a finite set of formal equations of the forms

$$\{\texttt{literal}/x_1\} = \texttt{Literal[name :: type]}\{\}$$
$$\{\texttt{o}_1/x_1, \ldots, \texttt{o}_m/x_m\} = \texttt{Tool}\{\texttt{i}_1/y_1, \ldots, \texttt{i}_n/y_n\}\,,$$

where each equation corresponds to a job, each variable x_j or y_j stands for a location, the left-hand side describes bindings of variables x_j to output ports o_j (implicit in the diagram representation), and the right-hand side describes bindings of variables y_j to input ports i_j. The first form describes a literal with its `name` and `type`, the second form an instance of the tool named `Tool`. The system of equations must be nonrecursive, i.e., the data dependencies must be acyclic.

4 Tool Interfaces

Our view of a workflow is a very liberal one: it does not predefine any tool name, nor does it spec-

$$
\begin{aligned}
\{\texttt{literal}/x_1\} &= \texttt{Literal[ecorpus :: SentenceCorpus]}\{\} \\
\{\texttt{literal}/x_2\} &= \texttt{Literal[fcorpus :: SentenceCorpus]}\{\} \\
\{\texttt{literal}/x_3\} &= \texttt{Literal[egrammar :: BerkeleyGrammar]}\{\} \\
\{\texttt{alignments}/x_4\} &= \texttt{GIZA}\{\texttt{english corpus}/x_1, \texttt{french corpus}/x_2\} \\
\{\texttt{tree corpus}/x_5\} &= \texttt{BerkeleyParser}\{\texttt{corpus}/x_1, \texttt{grammar}/x_3\} \\
\{\texttt{rules}/x_6\} &= \texttt{GHKM}\{\texttt{alignments}/x_4, \texttt{tree corpus}/x_5, \texttt{sentence corpus}/x_2\} \\
\{\} &= \texttt{Sink}\{\texttt{inport}/x_6\}
\end{aligned}
$$

Figure 3: Equational workflow representation corresponding to Figure 1.

ify the input and output ports of a tool. For example, as per our definition, we could use some tool `Megatool` twice, with varying input and output ports. Naturally, it is hard to systematically assign a meaning (e.g., an executable shell script) to such a workflow. On the other hand, if our definition of a workflow were to contain tool specifications, it would be outdated pretty soon.

The concept of a tool interface constitutes a compromise. A tool interface is a collection of specifications for thematically related tools. Put in terms of an analogy, a tool interface is to a tool as a Java interface is to a function. On the one hand, tool interfaces allow the user to design her experiment to a clear and well-documented specification. In fact, the GUI shows a palette of all tools that are part of a registered tool interface, and the user inserts instances of these tools into the workflow using drag and drop. On the other hand, tool interfaces allow Vanda Studio developers to build semantics to a clear specification as well. In other words, tool interfaces act as a contract between the user and the developer.

Moreover, tool interfaces allow for a flexible, modular syntax definition. New tool interfaces can always be added as needed, and outdated tool interfaces can be superseded by newer versions. Note that, technically, a new version is no different from a new tool interface because specifications should not be altered retroactively.

5 Assignments and Data Sources

Every literal must be assigned a value in order for the workflow to be runnable. Table 1 illustrates three possible such parameter assignments, corresponding to the following scenarios: extracting rules from English to German, once with a small portion of Europarl and once with a large portion, and extracting rules from German to English, again with the small portion (notice the change in the grammar).

identifier	data source
europarl	directory; path: `/home/user/europarl`, filter: `*`, type: `SentenceCorpus`
gramm	directory; path: `/opt/bin/berkeley`, filter: `*.gr`, type: `BerkeleyGrammar`
integer	integer

Table 2: A registry of data sources.

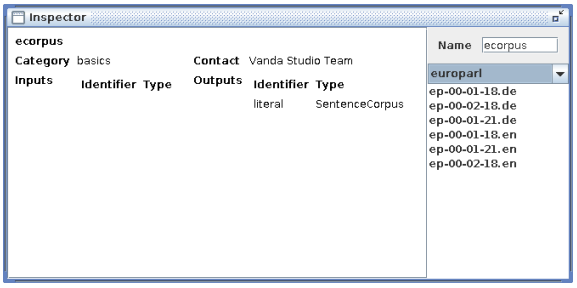


Figure 4: Editing a literal value in the inspector.

The prefixes `europarl:` and `gramm:` arise from the concept of data sources. Conceptually a data source is a set of objects, such as corpora, grammars, or just real numbers, each with a type information. In addition, a data source encompasses the GUI that allows the user to choose an object. The two most prominent data sources are the integer data source, where the user just enters an integer, and the directory data source, where the user can choose a file. A directory data source is determined by a path in the file system, a filename filter, and a type; and it consists of all files below that path matching that filter, and each file is assumed to be of that type.

Vanda Studio keeps a registry of data sources, which is just a mapping from identifiers to data sources, as illustrated in Table 2. Any object is then referenced in the way shown in Table 1.

	ecorpus	fcorpus	egrammar
(1)	`europarl:en/small.txt`	`europarl:de/small.txt`	`gramm:eng_sm6.gr`
(2)	`europarl:en/large.txt`	`europarl:de/large.txt`	`gramm:eng_sm6.gr`
(3)	`europarl:de/small.txt`	`europarl:en/small.txt`	`gramm:ger_sm5.gr`

Table 1: Three assignments for the literals.

Together with each workflow Vanda Studio keeps a table of assignments and a row index. The user can select a literal in the workflow diagram to edit its value at the current row, or she can open up the table as a whole. Figure 4 shows the former case: the combobox on the right-hand side (reading "europarl") is for selecting the data source, and the part below is for selecting an object.

6 Semantics

A workflow is a syntactic object that has to be interpreted so that it can be run. The present implementation converts a workflow, together with a parameter assignment, into a shell script. It proceeds as follows.

First, each location is assigned a value. If the location is at a literal, then the location value is the literal value, expanded according to the data source: `europarl:en/small.txt` becomes `/home/user/europarl/en/small.txt`, and `integer:10` becomes `10`. If the location is at a tool instance, then the literal value is the concatenation of the tool name, the MD5 hash of its input values, and the corresponding output port name. The same value, but without the output port, is used by the tool as a directory name for temporary files and logging. Note that due to the naming scheme, a location value uniquely determines its contents. This is crucial for avoiding duplicate computations and thus for incrementality.

Second, the jobs are translated in topological order. Literals are already accounted for, as their contribution is in the location values. An instance of tool X is translated as a call to a function named X. The values of the locations corresponding to the input and output ports are passed as arguments.

The present implementation keeps a registry of possible functions, which is automatically populated by scanning shell scripts for functions with appropriate annotations (cf. Fig. 5). Whether these annotated functions conform to a tool interface is easily verified.

Vanda Studio includes data visualization that

```
# GIZA
# IN english corpus :: SentenceCorpus
# IN french corpus :: SentenceCorpus
# OUT alignments :: Alignments
GIZA () {
  # (implementation)
}
```

Figure 5: Shell function with tool annotation.

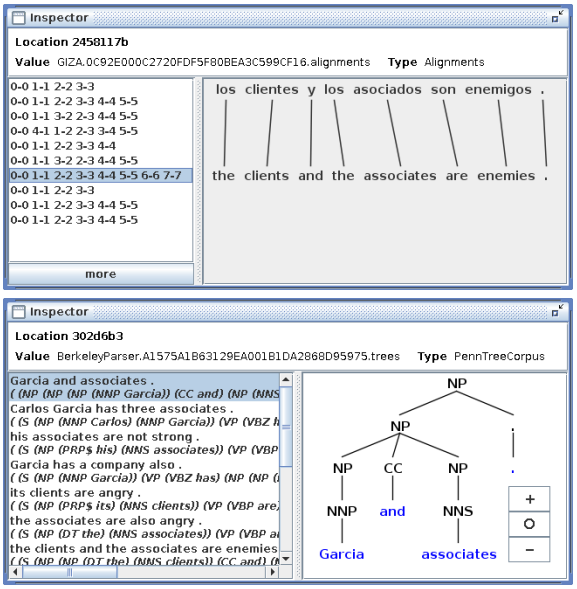


Figure 6: Data visualization: alignments, trees.

makes inspecting the contents of a location easy. Figure 6 shows the preview for an alignments file as well as for a treebank file.

7 Experiment

An experiment consists of a workflow and a selection of parameter assignments. Running the experiment amounts to creating the corresponding shell scripts and running them. Running an experiment can be done from within Vanda Studio; execution takes place on the local machine. At present there is limited run visualization, telling the user when a job is being started or when it has finished.

8 Deployment

Vanda Studio consists of the GUI, which is a Java program, and a few shell scripts. In particular, it

includes a rudimentary package management system. A typical package either automatically downloads and installs third-party software such as the Berkeley Parser, or it uses an existing installation of that software, if directed to do so. This system enables the user to restrict the amount of third-party software to be installed on the system, and it permits updating individual components.

Note that the package system is orthogonal to the tool interface system. On the one hand, the systems need not use the same granularity. On the other hand, there may be several implementations of the same tool interface, each packaged separately, and the user may decide to install any number of implementations, or none.

Packages can also be used to deploy data, such as the Europarl corpus. The installation script can download the data and perform any preprocessing steps such as stripping extraneous information or tokenizing. It can also register a corresponding data source. Alternatively, it can provide a tool that takes a language and a chapter number and outputs the corresponding corpus.

9 Related Work

We compare the qualities of six methods for running experiments that are available at present; see Table 3. The methods are

- command line,
- shell script,
- GNU make,
- the Experiment Management System (EMS) (Koehn, 2010, Section "experiment.perl"),
- LoonyBin (Clark and Lavie, 2010), and
- Vanda Studio,

and the qualities are

- incrementality (i.e., the user can develop and test the experiment one step at a time),
- ease-of-use (i.e., the method is fool-proof),
- reproducibility (on the same machine),
- portability (to a different configuration),
- robustness (i.e., detection of failures during execution, resume after failure),
- parameterizability (i.e., the experiment can be run with several parameter assignments),
- records (of the setup, legible),
- reports (of the execution, legible),
- visualization (of the setup),
- run visualization (of the execution),
- data visualization (of the data items), and
- concurrency (i.e., execution on a cluster).

	command line	script	make
Incrementality	✓		✓
Reproducibility		✓	✓
Parameterizability		✓	

	EMS	LoonyBin	Vanda Studio
Incrementality	✓	✓	✓
Ease-of-use		✓	✓
Reproducibility	✓	✓	✓
Portability			✓
Robustness	✓	✓	✓
Parameterizability	✓	✓	✓
Records	✓	✓	✓
Reports	✓	✓	✓
Visualization	✓	✓	✓
Run visualization	✓		✓
Data visualization			✓
Concurrency	✓	✓	

Table 3: Six methods of performing experiments, along with their respective qualities. A check mark is awarded when the quality in question is supported by appropriate idioms in the method.

The first three methods perform poorly, and we shall not discuss them here.

A user of the EMS writes a declarative script called `experiment.meta` to describe the general setup of the experiment. It contains a declaration for each step, which consists of its inputs, outputs, and the program to be invoked, among other things. And it uses variables to abstract from concrete input files and other parameters. A separate configuration file contains the concrete values. Execution is handled by a perl script, `experiment.perl`. The EMS contains facilities for monitoring the progress of the experiment, including a dependency graph showing which steps have been completed so far. Furthermore, the EMS can resume a crashed experiment, it keeps a log of each step, and it allows for concurrent execution on a cluster.

However, portability is limited as the configuration file may need to be adapted to accommodate for varying file locations. Moreover, the script `experiment.meta` must be meticulously hand-crafted because it is a template for the experiment as a whole. In other words, there is no template mechanism for individual steps. For example, if your experiment has two steps involving the Berkeley Parser, you have to write two almost identical declarations. In the context of the EMS's inception, which is Moses (Koehn et al., 2007), this does not seem to be a problem because the script is considered quasi fixed, and experiments only vary in the configuration file.

A user of LoonyBin constructs, using drag-and-drop operations, a workflow diagram similar to Fig. 1 to describe the experiment. As opposed to Vanda Studio, LoonyBin does not display individual input and output ports for the nodes; the actual connections are only evident when inspecting a single edge. As opposed to the EMS, LoonyBin has a template mechanism for steps: a tool like the Berkeley Parser exists as an abstract concept, and one can drag two instances of it into the workflow. Execution is handled by converting the workflow into a shell script. A single LoonyBin workflow can describe several experiments, thanks to the OR tool. In order to obtain a single experiment, each OR node is resolved by routing exactly one of its inputs to its output. The user selects the routing combinations she deems relevant. LoonyBin has facilities for logging, determining whether a step needs to be (re)run, resume after failure, sanity checks, and concurrent execution.

However, LoonyBin workflows reference concrete file names, which restricts portability. As a workaround, one can use a tool node which copies the file in question from a central source, such as a file server, to the working directory. However, it may be preferable not to repeat such a step for each experiment. Moreover, the latest release of LoonyBin was in 2010, and it does not appear very smooth. The LoonyBin authors review additional methods (Clark and Lavie, 2010, Section 5).

10 Conclusion

Vanda Studio is an SMT workbench that allows rapid incremental design of small-scale experiments, be it for teaching or rapid prototyping. Suitable abstractions such as data types, data sources, tool interfaces, and workflows on the one hand and simple details such as informative location values and package management on the other hand make Vanda Studio versatile and easy to use. For instance, an instructor can playfully and incrementally design an experiment in front of class, and students can reenact the steps on location or at home. Experiments designed by students can run on the instructor's computer as long as the data sources are configured accordingly.

As yet, concurrent execution on a cluster is not supported. This paradigm necessitates proper allocation of resources (compute nodes, memory) to jobs, which Vanda Studio's abstractions do not cover. Augmenting the abstractions accordingly is a possibility for future research. So is support for macros. In addition, it would be interesting to increase the granularity of SMT tools to a point where even a decoder can be designed graphically.

References

Jonathan H. Clark and Alon Lavie. 2010. Loonybin: Keeping language technologists sane through automated management of experimental (hyper)workflows. In *Proceedings LREC 2010*.

Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. 2004. What's in a translation rule? In *Proc. HLT/NAACL*, pages 273–280.

Michael Galley. 2010. GHKM rule extractor. `http://www-nlp.stanford.edu/~mgalley/software/stanford-ghkm-latest.tar.gz`, retrieved on March 28, 2012.

Jonathan Graehl, Kevin Knight, and Jonathan May. 2008. Training tree transducers. *Computational Linguistics*, 34(3):391–427.

Liang Huang, Kevin Knight, and Aravind Joshi. 2006. Statistical syntax-directed translation with extended domain of locality. In *Proc. 7th AMTA*, pages 66–73.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: open source toolkit for statistical machine translation. In *Proc. ACL Interactive Poster and Demonstration Sessions*, pages 177–180.

Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proc. of MT Summit X*, pages 79–86.

Philipp Koehn. 2010. Moses documentation. `http://www.statmt.org/moses/`, accessed on April 07, 2013.

Franz Josef Och and Hermann Ney. 2000. Improved statistical alignment models. In *Proc. ACL*, pages 440–447.

Franz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, March.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proc. COLING/ACL*, pages 433–440.

Kenji Yamada and Kevin Knight. 2001. A syntax-based statistical translation model. In *Proc. ACL*, pages 523–530.