A lightweight infrastructure to support experimenting with heterogeneous Transformations

Wolfgang Lohmann
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
wlohmann@informatik.uni-rostock.de

Günter Riedewald
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
gri@informatik.uni-rostock.de

Thomas Zühlke
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
thomas.zuehlke@uni-rostock.de

ABSTRACT

We report on a class library called Trane, which provides an infrastructure to support experimenting with transformations interactively. Transformations here mean algorithms, which take software artifacts as input and output manipulated artifacts. Trane supports easy combination of transformations available in different languages, libraries and tools. Several combinations can be presented at the same time, parameters can be visually changed, and results can be compared. New transformations can be easily added. Generated transformations from experiments can be integrated into the experiments at run-time.

The paper presents the general model of the class library. We show how the class library profits by the features provided by .NET, such as language interoperability, foreign language interface, shell access, reflection, and web services by demonstrating five variants to integrate new transformations.

Keywords
Transformations, .NET, Language interoperability, cross-language inheritance, visual programming, component-based transformation systems, platform independence

1. INTRODUCTION

We report on a lightweight infrastructure developed to support experimenting with transformations interactively. Here, transformations mean algorithms, which take software artifacts as input and output manipulated artifacts or results of an analysis. We use .NET, as it facilitates integration and combination of heterogeneous transformations, i.e. transformations available as programs in different languages, existing command line tools, web services, libraries through a foreign language interface, and dynamic compilation and loading of DLLs resulting from a transformation.

Experiments with Transformation Nets

Some kinds of complex transformation are developed in an explorative way, where they are extended after a test with representative examples shows that the development might be on the desired way. Examples vary from combinations of UNIX command line tools such as sed, awk, grep to extract and manipulate information in text files to more sophisticated examples, such as refactoring, where there are many ways to achieve an improvement of the source code, or to achieve software evolution by transformations [Läm04, Set04]. Another example is the collection of individually changes for maintenance in batch files for later reuse in [Klu05].

We intend to use Trane to experiment with transformations on language components, e.g. grammars, semantic descriptions, and language processors, though it is not restricted to those applications. We want to extend languages stepwise during their development, explore several possibilities, how a grammar could be changed, compare the variants, extract parts of existing grammars and adapt them to form a sublanguage DSL, and directly connect the generated output to front end generators to test example programs. There are tools, but they are available in different formats, e.g. command like tools like yacc and GDK [Kor02], left-recursion removal for attributed grammars in Prolog and TXL [Loh04], grammar representations in XML, BNF etc.

However, to the user, it should not matter, whether a transformation is a command line tool like yacc, or an analysis written in Prolog, and should be represented uniformly modulo their parameters.

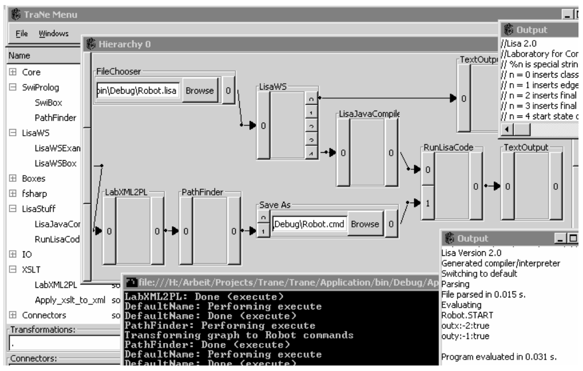
Using .NET
We were interested in an implementation on .NET mainly because it comes with the promise of language interoperability and cross-language inheritance. With C# as main implementation language, we could make use of properties, generics, delegates, reflection, and web services. The implementation was also an experiment in platform independence wrt. the availability of .NET on Linux as well as Gtk# on Windows.

Resulting Prototype
We designed a simple class model. Transformations are represented by automatically generated or self-designed boxes to be placed on a workspace, which is itself part of a box. The boxes have typed input and output ports, which can be connected using converters to describe dataflow. Boxes can provide facilities to control transformation parameters. Several sequences of transformations can be presented simultaneously, parameters are visually changeable, and results can be compared.

Trane can be extended easily with new transformations. New boxes can be any program, a web service, an encapsulated command on shell level, etc., written in any .NET language, as long as the box interface is implemented. Thus, the user creates transformation nets without paying attention to the implementation of a transformation. Due to reflection, no extra configuration files are necessary. Trane can also be seen as a wrapper architecture or an interpreter for call graphs of complex functions. It is a lightweight implementation, because .NET already encapsulates much work for the integration of transformations.

Remainder of the Paper
In Section 2 we present the concept of Trane. In Section 3 we discuss the model and the computation strategy. In Section 4 we show five categories of transformation and how they are integrated. Section 5 discusses some related work. Finally, the paper finishes with concluding remarks.

2. TRANE CONCEPT
Trane provides facilities to model **Tra**nsformation **ne**ts with heterogeneous transformations. In Figure 1, for example, an attribute grammar of a robot move language is sent to the Lisa web-service, which generates a compiler for that language. Using LisaJavaCompile (wrapper for Java at command line), the Lisa generated code is compiled. In the second sequence, a description of a maze in XML is converted to Prolog by an XSLT based transformation. A Prolog-based transformation now analyses the inherent graph and generates a program for robot moves to control its way through it. The program is saved, the filename is delivered to the generated compiler RunLisaCode for the robot language. The result of the execution, the final position of the robot relative to start position (0, 0), is delivered to the TextOutput.



```
//Lisa 2.0
//Laboratory for Cor
// %n is special strin
// n = 0 inserts class
// n = 1 inserts edge
// n = 2 inserts final
// n = 3 inserts final
// n = 4 start state c
```

```
file:///H:/Arbeit/Projects/Trane/Trane/Application/bin/Debug/Ap
LabXML2PL: Done (execute)
DefaultName: Performing execute
DefaultName: Done (execute)
PathFinder: Performing execute
Transforming graph to Robot commands
PathFinder: Done (execute)
DefaultName: Performing execute
DefaultName: Done (execute)
```

```
Lisa Version 2.0
Generated compiler/interpreter
Switching to default
Parsing
File parsed in 0.015 s.
Evaluating
Robot.START
outx:-2:true
outy:-1:true

Program evaluated in 0.031 s.
```

Figure 1. Trane in action

The underlying structure is a directed graph with nodes representing transformations. Nodes have input and output ports, which possess types, and correspond to input and output positions of the transformations. Output ports can be connected to input ports of other nodes by directed edges, assumed the types associated to the ports are equal. This way, the call graph of a composite transformation is modelled.

Connections between ports of different types can be obtained indirectly by converters. These are special transformations, which map values of a given type onto values of a related type. In the graphical representation, they are hidden behind connections to allow a simplified view on the net. For example, it should not matter that the result of a transformation is a grammar in XML format, but the next transformation expects it in a BNF style. An XML2BNF connection can transport the grammar and hide the necessary format conversion. The user simply chooses the connector with the desired type combination. Data transported can be text as in UNIX-pipes, structured data such as grammars, or file names for results in files.

Transformations can be added at run time, e.g. transformations created with Trane. Providing a new transformation means to embed a transformation into a node such that input and output ports are provided with data. To create a new converter means to provide a new transformation, which implements the desired type mapping. This requires knowledge about the structure of data.

The order of computations is determined by the dependencies between transformations in the graph. Cycles are not considered, as their role is not clear in

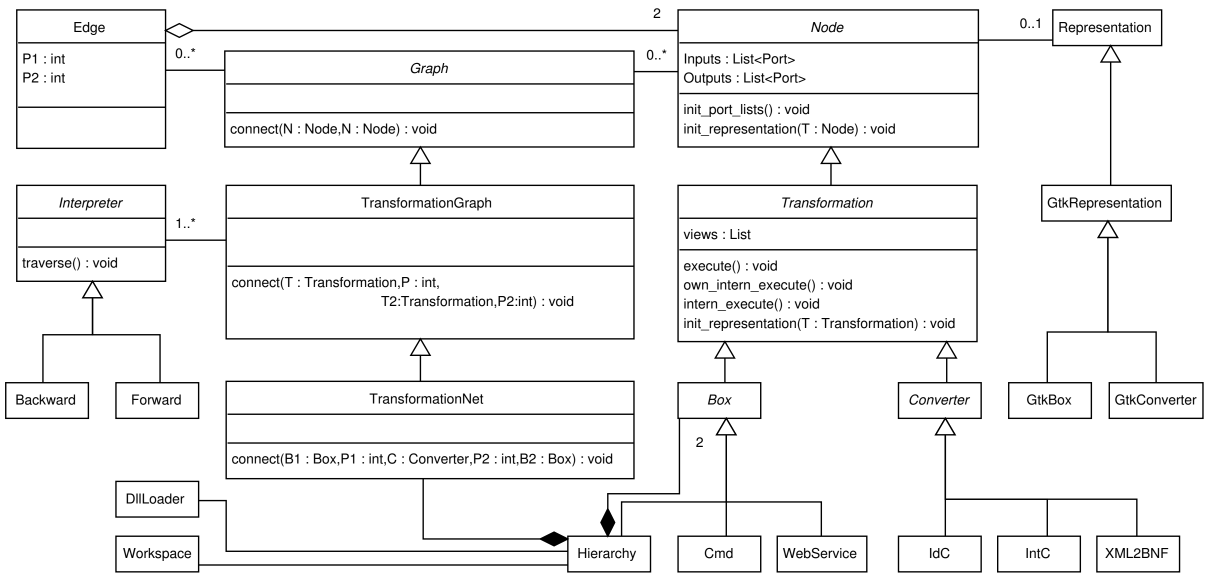


Figure 2. Class model of Trane

this setting. The computations are performed always once, when a result is demanded and the required input data for the transformation is available. Results can be queried at any output port at any transformation, thus, comparing the values of different transformations is possible. The intermediate results can be investigated, which is helpful, if the result of a transformation delivers unexpected values.

3. OBJECT-ORIENTED MODEL

Figure 2 shows the UML class diagram of the infrastructure, which largely mirrors the concept.

First Level: Combination Infrastructure

The class *Transformation* defines minimal requirements of transformation nodes. As can be seen in the class diagram, it provides lists for input and output ports. These ports manage edges connected to ports of other transformations, data, and a type annotation, which constrains data accepted. Data is packed in a separate object, which provides its value and a type. This allows for a subtype concept, i.e. the value has to be a subtype of the type of the port. The values are used as input and output values for a transformation and the object representing the transformation. To define the port lists of a special transformation, it has to override method *init_port_lists* to configure the ports (e.g. with type annotations). Port lists are extendable dynamically at run-time. Ports of transformation objects are connected using the method *connect/4* of *TransformationGraph*, which tests on type conformance, creates an edge between the ports, and keeps track of transformation objects and their connections. Edges store the nodes and indices of the ports connected.

A subclass has to override *execute*, where the actual mapping from values of input ports to values of output ports is defined or the embedded transformation is called. The computation can depend on several conditions, such as the actual computation strategy, or lazy computation (do not compute if input values have not changed). To save the user from uninteresting management work, *execute* is wrapped by methods *intern_execute* and *own_intern_execute*, which take care of the conditions, and at a suitable point in the computation call *execute*. *init_representation* associates a representation to an instance of *Transformation*.

The difference between common kinds of transformation nodes and converters is expressed by classes *Box*, which box a desired transformation, and *Converter*, whose main task is to provide some kind of type conversion. The provider of a converter will find it nice to implement it like any other transformation. They only differ from boxes through their representation and arity. This enables converters of all kinds, simple converters or arbitrary complex computations, from which the user would like to abstract in a model.

The *TransformationNet* provides a method *connect/5* to connect two objects of type *Box* using a *Converter* at the ports specified with the port index each.

We decided for overriding of some *init*-methods over configuration inside of a constructor, because in the chosen implementation language C# constructors of super classes are evaluated first before that of the actual class. For some tasks provided in the super class, e.g. for the generation of graphic representations, it is necessary that the actual class is configured already at least partially.

Second Level: Interactivity and Views

The second level provides graphical representations for transformations. In the standard representation, rectangular boxes are generated for transformations (e.g. most representations in Fig. 1 are generated.). Lists of buttons, which also activate the execution of the associated box, represent input and output ports. Converters are represented as a line, which connects two boxes. This simplifies the view on the transformation net.

If desired, the provider of the transformation can create own representations for boxes and converters by inheriting *GtkBox* and *GtkConverter* respectively. Their instances are associated to the specific transformation class by overriding *init_representation*. Objects of class *GtkBox* can be provided with additional buttons, fields, sliders, and other kinds of input/ output support for users to control the transformation.

Objects of transformation nodes can provide several views at them. The first level can already be considered as the most basic view. The main view used is the graphic representation on a workspace to combine them. In addition, more information and controlling facilities are possible, e.g. a description of the transformation represented by the object, a description of its input/output, complex tables for the user to describe or influence the way the transformation is working, status messages, and logs. Note, the workspace in Fig. 1 is just another view on a special box, allowing to create a hierarchical subnet interactively.

Providing a New Box

To create a new box, the following steps are followed: 1) Choose a box to inherit from. 2) If desired, override *init_port_lists* to redefine input and output ports by simply adding new ports to a generic list. 3) Override *execute* to describe, how values of input ports are used by the transformation to compute values and copy them into output ports. 4) If a new representation is desired, create a new subclass of *GtkBox* and redefine components or add new features to the inner frame, e.g. a button to show a new view, which can be any graphical object. Override *init_representation* in the box to assign it to the box.

Computation Strategy

There are several variants to initiate computation of the transformation net: backward and forward computation (similarly to demand-driven vs. data-driven) and direct vs. indirect data transport. The choice is realised through an instance of *Interpreter*, who performs/initiates the traversal.

With direct data transport, a transformation itself informs its successors / predecessors about results/ required results and calls their *own_intern_execute*. With indirect data transport a separate object of class *TransformationNet* controls the traversal process, e.g. calls *intern_execute*. Note, that by *connect*/5 the object keeps book about created transformations and connections. This allows intercepting and changing values for experimenting.

Backward computation is initiated by requesting the output port of the last transformation of a chain by initiating *own_intern_execute /intern_execute*, which then determine missing input values for the computation of the embedded transformation, and activate the preceding transformations. When all values are available, the wrapped *execute* is called. This strategy will be used mostly to compare several transformations at the end of a common sequence.

The forward computation strategy is thought for experiments to investigate the effect of a changed input. E.g. a composite transformation can be attached to a text editor, and show the results of a transformation chain immediately while typing e.g. a new part of a grammar (or delay start until a save-command is fired). Forward computation is simulated on top of the backward computation by calling the output ports of following transformations. This can be very expensive, though. Cycles are not allowed in the computation though we have not included a check to avoid them yet (we could think of a graph analysis based on a term generated from the net).

4. VARIANTS OF BOXES

Many transformations will only inherit from the common box type, configure the input and output ports, and define a mapping between them to create different kinds of boxes. However, using .NET, several different kinds of special box categories are viable, e.g. hierarchy boxes (to provide subnets and workspaces), web service boxes, command line tool wrappers, compilers, foreign libraries wrappers, or DLL loaders. Here we show five variants to integrate different transformations in boxes.

Web Services

As an example for a web service transformation we show in Fig. 3, how to implement the compiler generator box LisaWS used in Fig. 1. Lisa [Mer99] is a compiler generator system also available as web service. When sending an attribute grammar, it generates and delivers Java code of a compiler. The code can be compiled and the resulting compiler can be used for the programs of that language.

LisaWS gets an input port for a string value, the attribute grammar. An output port is configured to provide a string for a path (to store the generated files), and further ports, where the generated lexer, scanner, parser, and evaluator can be requested separately.

```
public class LisaWSBox : Box {
  public override void init_port_lists(){
    Inputs.Add(new Port("String"));
    Inputs[0].data =
           new ValueData(null, "String");
    Outputs.Add(new Port("String"));
    Outputs[0].data =
           new ValueData(null, "String");
       … // some more output ports
  }

  public override void  execute(){
    CServiceBeanService lisaService =
               new CServiceBeanService();
    System.Net.CookieContainer container=
         new System.Net.CookieContainer();
   lisaService.CookieContainer=container;
   lisaService.mkdir("wlohmann");

   // read file with lisa specifications
   String path = Inputs[0].data.value;
   FileStream fs = File.OpenRead(path);
   StreamReader r = new StreamReader(fs);
   String Spec = r.ReadToEnd();
   lisaService.clearError();

    // compile and save specifications
    bool OK = lisaService.compile(Spec);
    if (!OK) { … /* error */ } else {
        String scanner =
               lisaService.getScanner();
        Outputs[0].data.value = scanner;
        …  }
 }
}
```

Figure 3. A web service box

We find it especially charming to integrate remote applications into transformation nets from locally existent algorithms. Problems might be that connections are unavailable, or slow. Depending on the kind of service boxed, the transformation could require to re-compute always, even if no input values have changed.

Hierarchical Transformations

Hierarchy in transformation nets means to hide a transformation subnet *TSN* behind a box B_H, which looks and behaves like other boxes with input and output ports. Note, there are different types of hierarchy boxes. They can differ in the number of input/ output ports, or in the way they are to be used. Hiding requires mapping inputs and outputs of B_H to inputs and outputs necessary for *TSN*. This can be easily done by providing two identity boxes B_I and B_O as interface for inputs and outputs, between which *TSN* is constructed. Since transformations use properties to connect to ports, .NET helps to redirect port access to the input ports of B_H to input ports of B_I as well as output ports of B_H to those of B_O by simply overriding the definition of the properties (see Fig. 4). The graphical representation is extended by a button, which when pressed provides a second view, namely the workspace of the hierarchy box. Figure 1 shows the inner view of a hierarchical box. We additionally added a transformation browser for choosing boxes and converters. This browser makes use of reflection to analyse DLLs in a chosen directory and to create instances of provided classes.

```
public class HierarchyBox  : Box {
  public IdBox InputBox  = new IdBox();
  public IdBox OutputBox = new IdBox();

  // Hide Inputs of this box by pointing
  // to corresponding interface box
  public override  List<Port> Inputs {
      set { InputBox.Inputs = value; }
      get { return InputBox.Inputs;  }
  }

  public override List<Port> Outputs{ … }

  public override void init_port_lists(){
      base.init_port_lists();
      InputBox.Double_PortLists();
      OutputBox.Double_PortLists();
  }

  public override void execute() {
      OutputBox.ownInternExecute();
      // Input execute not necessary
  }
  // save hierarchy in a separate subnet
  private TransformationNet _TraNe =
             new TransformationNet();
  public TransformationNet TraNe {
     get { return _TraNet; }
  }
  public override void
       init_representation() {
     this.Representation = new
   Gtk_HierarchyBox_Representation(this);
 }
}
```

Figure 4. A plain hierarchy box

Use of Native Libraries

As an example for the use of existing DLLs outside of .NET we choose SWI-Prolog [Wie06], mainly because we want to use Prolog for experiments with transformation tasks similar to [Loh04, Loh03]. In Fig. 1, the PathFinder-box is based on Prolog. It determines a path through a labyrinth and generates a control program in the Robot language for it.

```
[DllImport(DllFileName)]
internal static extern uint
                      PL_new_term_ref();
     …
// make a PlTerm from a C# string
public PlTerm(string text) {
  m_term_ref = libpl.PL_new_term_ref();
  libpl.PL_put_atom_chars
                      (m_term_ref,text);
} // SwiCs.cs by Uwe Lesta
```

Figure 5. Snippet from SwiCs.cs

.NET offers the attribute *DllImport* to define access to foreign libraries. We created a DLL based on *SwiCs.cs* (cf. [Les03]) where for each exported function in the library its name is declared after the attribute (Fig. 5). The DLL provides .NET programs with methods and types to model Prolog terms and to query a SWI-Prolog engine; and is used by the box.

```
public override void execute(){
  String[] param = { @"H:\\ Projects" +
               …  "\\Application.exe"};
  PlEngine e = new PlEngine(1, param);

  // Get query as Text, call it, e.g.
  // (tell('log'),write('HiWorld'),told);
  string goal = (string)
         (Inputs[0].data.copy().value);
  PlQuery q = new PlQuery("call",
      new PlTermv(new PlCompound(goal)));
  bool b = q.next_solution();  q.free();
}
```

Figure 6. Providing direct Prolog access

Figure 6 shows how to interpret a string input as Prolog term directly and to call it. Combined with text boxes it can serve as interactive Prolog interpreter. Also, a Prolog box can provide programs that are more complex or initiate loading of a rule base.

A problem is, in our opinion, that the attribute *DllImport* expects a static string, which has to be known at compile-time. This makes replacing different versions of the Prolog DLL impossible without recompilation of the interface DLL *SwiCs.cs*, thus, reducing platform independence (the name of the dynamic libraries differ between e.g. Windows and UNIX systems).

XSLT Boxes

.NET comes with good XML and XSLT support. This offers a good basis to provide boxes to transform XML documents. Fig. 7 gives an example for the contents of *execute*.

```
  String xml_input = (String)
     ((Inputs[0].data.copy()).value);
  StringReader xml_reader =
      new StringReader(xml_input);
  XPathDocument xpath_document   =
      new XPathDocument(xml_reader);
  XslCompiledTransform transformation =
      new XslCompiledTransform();
  StringReader xsl_script_reader =
      new StringReader(Xslt_Script());

  XmlTextReader xsl_script =
    new XmlTextReader(xsl_script_reader);
  transformation.Load(xsl_script);
  StringWriter xml_output_writer = …
  XPathNavigator document_navigator =
       xpath_document.CreateNavigator();

  transformation.Transform(
       document_navigator, null,
       xml_output_writer);
  Outputs[0].data.value =
       xml_output_writer.ToString();
```

Figure 7. Apply XSLT script to input

The example takes some XML data from an input port and delivers transformed data to the output port. Note, that the XSLT script in this case is provided by a return value of *Xslt_Scipt*, a method to be overridden by subclasses to specify a concrete transformation. Other variants of XSLT boxes might expect the script itself, or a filename for the script as input at a port, or configured in another box view. A subclass of this box is used in Fig. 1 to transform the description of a labyrinth into Prolog notation.

Command Line Tools

Many transformations are available as command line tools. Examples are compilers, but also yacc, lex, awk. Additionally, there are tools like grammar deployment kit [Kor02], which could be made available through the integration in Trane. Figure 8 shows how to use the Java-compiler for Lisa-generated code (cf. Fig. 1). Here, the tool represented is hard coded into the box, but could also be provided through extra views with input fields or from input strings as part of the transformation.

```
System.Diagnostics.Process p =
                           new Process();
p.StartInfo.UseShellExecute = false;
p.StartInfo.CreateNoWindow = true;
p.StartInfo.RedirectStandardOutput=true;
p.StartInfo.RedirectStandardInput= true;
p.StartInfo.FileName = "cmd";
p.Start();
StreamWriter sw = p.StandardInput;
StreamReader sr = p.StandardOutput;
sw.AutoFlush = true;
//sw.WriteLine("dir /AD");or any cmd/tool
sw.WriteLine(@"javac –classpath lisa.jar"
                        +path+"*.java");
sw.Close();  p.WaitForExit();
Outputs[0].Data.Value=TextBuffer.Text;
```

Figure 8. Wrapping command line tools

The problem with this kind of boxes is that platform independence is restricted to the availability of the integrated tools on the platform.

Dynamic Compilation and Integration

The command line tool approach can be used to compile a transformation for Trane and make it usable at run-time. Depending on given options, the resulting executable can be started as command (maybe again wrapped in a box, as in Fig. 8), or the DLL can be examined/loaded and classes instantiated using reflection, if it is written in a .NET language. If the compiler generates .NET code itself, the resulting class can be directly instantiated instead of generating a DLL first.

F# and Other Languages

Though the above examples can use transformations written in other languages, the boxes themselves have been specified using C#. It is better to use the language of choice itself to define a box. This requires it is implemented on .NET. The resulting DLL can be used in Trane, as if C# had been used due to cross-language inheritance. Only then *the real benefit* of .NET occurs in our opinion, as the still existing problems of data conversion in approaches like command line tools or foreign libraries could be avoided.

With F# [Fsh06] we were able to inherit from C# classes of Trane (the box), to create a new box (written in F#) and to instantiate from it in Trane again. F# is functional and thus, similar to Prolog, suitable to describe transformations.

Several languages on .NET are differently suitable. We had not the expected success with P#, but this might be our fault. With Eiffel# it is necessary to take care of the naming scheme during compilation. J# is not portable on Linux as it requires DLLs available on Windows only. We would be interested in a smooth integration of Haskell. There are some attempts, but there is still a way to go.

5. RELATED WORK

Several tools provide a plugin structure and interactive placement of components. They are either large, or provide a proprietary language to extend them with new objects. Trane has mainly been inspired by Cantata, the graphical user interface for the Khoros system to analyse and manipulate graphics [You95]. Cantata allows to interactively construct such filter pipelines.

[Spi02] considers UNIX tools as components. A GUI builder is used to create the visual programming environment. The placing relation of the components describes dataflow, which is text. UNIX tools have to encapsulate as ActiveX components with much manual work. Connectors are simply a visual encapsulation of the operating system pipe abstraction. Connector and glue-type components still need to be written by hand. Trane is not restricted to one kind of data, though it is intended to be applied mainly to artifacts of language processors, i.e. data are grammars, specifications, rewrite rules, parts of parsers, etc. We provide among others a system call box, which can take the command call directly as string. A new wrapper box for a special command can be easily written on top of the system box, which can take even the options at input ports. Our converters can transport structured data of any kind, they just have to inherit from a general converter class and implement additional treatment.

Stratego/XT [Vis04] uses mainly ATerms [Bra00] to provide input and output for terms in Stratego, and to exchange terms between transformation tools. New created transformations are wrapped into stand-alone components, which can be called from the command-line or from other tools. Those tools can be used similarly to Unix pipes, but can additionally work on structured data. For compositions of complex transformations they provide the XTC model. A repository registers locations of tools. An abstraction layer implemented in Stratego supports transparent access, allowing to call and use a tool like a basic transformation step in Stratego programs. Additionally, Stratego provides a foreign language interface to call C functions. Trane is designed mainly to reuse and to combine transformations for experiments. The XT tools could be wrapped in boxes, and used for experiments. We cannot generate stand-alone tools from composite transformations.

The Meta-Environment [Bra01] also allows the combination of different tools, but separates strictly between coordination and computation. Basis is the TOOLBUS coordination architecture, a programmable software bus based on process algebra. Coordination is expressed by a formal description of the cooperation protocol between components, while computation can be expressed in any language. Meta-Environment is used to produce real life products, on the other hand, it is complex, and difficult to adapt a new tool to the tool bus.

In Trane, coordination and computation are tangled. Evaluation of a transformation net is just traversing to each node and computing as given by the inherent dependencies between transformation nodes. Transformations can be added easily by providing a wrapper, where only two methods have to be overridden.

In Eclipse, GEF allows to create similar models and associate semantics to them. However, for new parts of the model (e.g. similarly to a new box in Trane) it requires a new compilation, while Trane nets are open. We do not need to compile the net. It is directly executable. New transformations can be added dynamically. Like other plugin systems, in Eclipse a plugin needs configuration files to add a new component, while we use reflection to extract necessary information. The language plugins for Eclipse are Java classes in a JAR archive. Transformations in Trane do not need to be written in one specific configuration language, as long it is supported by .NET.

[San99] also try to spread transformation system technology over a set of reusable heterogeneous components. Using Java, CORBA and HTTP, they have instantiated a communication layer. To configure components, a description in a hybrid architecture description language is necessary.

Calling functionality from foreign DLLs is not new. However, usually the calls are determined at compile time. We offer to combine functionality, which might come from different DLLs without recompilation.

Using Trane is similar to programming in dataflow languages. We refer to [Whi94] for further reading.

6. CONCLUDING REMARKS

SUMMARY

We have presented a lightweight infrastructure, which allows to provide heterogeneous transformations with a uniform façade to combine and interact with them. The model has been given and the essential classes have been explained. We presented five categories of transformations such as integration of web services, or command line tools. Integration of new transformations is simple. Due to reflection, no extra configuration files are necessary. Trane is lightweight as a large part of the work for integration is encapsulated in .NET. The biggest advantages have languages that are implemented on .NET directly, but we still wait for more pure .NET languages, without name scheme or inheritance problems.

FUTURE WORK

We are aware that Trane is rather a proof of concept than a tool yet. The type system is currently very ad hoc. There are still conceptual. It is still matter of research, what types mean in our context. For example, for some transformations grammars of different languages are of the same type, if they are in the same format such as BNF. On the other hand, grammars can be considered as different types despite their format, if the algorithm using it is language specific. We want to design an extensible type hierarchy.

The Visitor pattern might help with flexible computations; also, to generate command line tools from a net as well as terms describing nets for analysis. As another way to integrate transformations sockets should be examined. The usability has to be increased vastly. It might be interesting to initiate the evaluation of transformations in separate threads. A classification of boxes would be nice. We need more transformations with grammar typical support to perform the experiments. We are new to F# and need more experiments with it and with other .NET languages.

7. ACKNOWLEDGMENTS

We thank the reviewers for their comments, which provided answers, literature, and suggestions for future directions of the work, though we were not able to implement most of them in this paper. We thank Damijan Rebernak for help with the Lisa web service.

8. REFERENCES

[Bra01] v. d. Brand , M.G.J., and v. Deursen, A., and Heering,J, and de Jong, H.A., and de Jonge, M., and Kuipers,T., and Klint,P., and Moonen,L., and Olivier, P.A., and Scheerder,.J., and Vinju,J.J., and Visser, E, and Visser,J. The ASF+SDF Meta-environment: A Component-Based Language Development Environment, Procs. of the 10th International Conference on Compiler Construction, p.365-370, April 02-06, 2001

[Bra00] v. d. Brand , M.G.J., and de Jong, H. A., and Klint, P., and Olivier, P. A., Efficient annotated terms, Software- Practice & Experience, 30, pp. 259-291, 2000

[Fsh06] F# Home Page (Feb.2006) http://research.microsoft.com/fsharp

[Klu05] Klusener, S., and Lämmel, R., and Verhoef, C.: Architectural Modifications to Deployed Software. Science of Computer Programming 54, pp.143-211, 2005

[Kor02] Kort, J. and Lämmel, R., and Verhoef, C. The Grammar Deployment Kit, ENTCS 65, 3, Elsevier Science Publ., 2002

[Läm04] Lämmel, R.: Evolution of Rule-Based Programs. Journal of Logic and Algebraic Programming, Special Issue on Structural Operational Semantics, 2004

[Les03] Lesta, U.: C# Interface to SWI-Prolog. http://gollem.science.uva.nl/twiki/pl/bin/view/Foreign/CSharpInterface, Version Aug. 2003

[Loh03] Lohmann, W., and Riedewald, G. Towards automatical migration of transformation rules after grammar extension. In Proc. 7th European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, March, 2003

[Loh04] Lohmann, W., and Riedewald, R. and Stoy, M. Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars, ENTCS 110 C, Elsevier, 2004

[Mer99] Mernik, M., and Zumer, V., and Lenic, M., Avdicausevic, E. Implementation of multiple attribute grammar inheritance in the tool LISA. ACM SIGPLAN not., June 1999, Vol. 34, No. 6, pp. 68-75.

[San99] Sant'Anna, M., do Prado Leite, J.C.S., An Architectural Framework for Software Transformation, Proceedings of the International Workshop on Software Transformations; STS'99', ICSE'99, 1999 http://www.dur.ac.uk/CSM/STS/

[Set04] Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004), ENTCS 127 (3), April 2005

[Spi02] Spinellis, D. Unix tools as visual programming components in a gui-builder environment. Software - Practice & Experience. 32, pp.57-71, 2002

[Vis04] Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9., in C. Lengauer et al., editors, Domain-Specific Program Generation, LNCS 3016, pp. 216--238. Spinger-Verlag, June 2004.

[Whi94] Whiting, P. G., and Pascoe, R. S. V. A History of Data-Flow Languages, IEEE Annals of the History of Computing, Vol.16(4), pp.38-59, 1994

[Wie06] Wielemaker, J. SWI-Prolog Home Page http://www.Swi-Prolog.org

[You95] Young, M., and Argiro, D., and Kubica, S. Cantata: Visual programming environment for the Khoros system. Computer Graphics 29, 1995