Delft University of Technology
Parallel and Distributed Systems Report Series

The *Rotan* Reference Guide

Leo Breebaart

report number: PDS–1997–002

PDS

ISSN 1387–2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems
Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.twi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.twi.tudelft.nl/

Chapter 1 — Introduction

This document describes various technical and historical aspects of the *Rotan* system, and is intended both for people who intend to use *Rotan* as well as for those who will need to do future development of the system.

Chapter 2 describes the *Rotan* source file configuration, and explains how to check out, compile, and extend the system.

Chapter 3 is more of historical then of immediate interest, and consists of the text of documents that were produced at the start of the project.

Chapter 2 — The *Rotan* Source Files

The *rotan* source files are quite well documented. In principle, therefore, all knowledge necessary for understanding how *Rotan* works can be deduced from looking at the sources (which apart from the usual code comments also contain many notes on specific design decisions). However, such a bottom-up approach will always work better if there is also more high-level information available about the set-up and aims of the system, meeting the implementation documentation halfway, so to speak. This chapter is intended to be such a top-down reference, and does therefore *not* aim to be exhaustive down to the file level.

2.1 Checking out and compiling the sources

Rotan is stored in CVS as the module *rotan*. A copy of the source tree can be checked out by giving the command:

```
cvs -d ~lspace/CVS co rotan
```

The makefile structure for rotan assumes that the environment variable `RULECOMPILER` will be set to the newly created top level *rotan* directory. The binary directory `$RULECOMPILER/Bin` needs to be in your path.

Once this is set up, the rotan system can be compiled by simply saying

```
cd rotan; make all
```

Rotan has been developed and tested using the experimental *egcs* C++ compiler. It will also compile, but not link, with KAI C++.

2.2 Directory Layout

2.2.1 The `rotan/` Directory

At the top level, the *rotan* directory tree has the following layout (only 'interesting' files and subdirectories are shown here):

```
rotan/
      Makefiles/
      Bin/
      Lib/
      Sources/
      Rules/
      Src/
      TODO
      BUGS
```

- *Makefiles* contains the shared Makefiles for the project.
- *Bin* contains executables such as the compiled *rcc*, or any auxiliary scripts e.g. for controlling the loading of rules.
- *Lib* contains the libraries created by the project, and is initially empty.

- *Sources* is a default directory for loading *rcc* sources (i.e. files containing *Tm* data structure instances) from. Example sources are stored here.
- *Rules* is a default directory for loading *rcc* rule sources (i.e. files containing RL 2.0 rules) from. Example rules are stored here.
- *Src* is the directory containing all the *Rotan* (including the *rcc*) sources.
- *TODO* is the global todo-file for the entire project.
- *BUGS* is the global bugs-file for the entire project.

There is no *NEWS* or *ChangeLog* file, because CVS (and its auto-mailing of commits) takes care of that to our satisfaction (although eventually a real ChangeLog file might be a good idea, if this code is ever to be distributed to third parties).

2.2.2 The `rotan/Src/` Directory

The source directory is divided into the following subdirectories (also known as libraries, since each subdirectory leads to one library file in *rotan/Lib* after compilation):

```
rotan/Src/
          Oopl/
          Dlib/     -- with subdirectory Dlib/Defs/
          Rlib/
          Rcc/
          Calc/     -- with subdirectory Calc/Defs/
          Vnus/     -- with subdirectory Vnus/Defs/
```

- *Oopl* (Object Oriented Programmers' Library) contains stand-alone utility classes that are used throughout the system but which are not system-specific.
- *Dlib* (Domain Library) forms half of the heart of *Rotan*. *Dlib* contains two broad categories of classes: first, those classes starting with the prefix *rotan*, which implement C++ representations of *Tm* domain constructs, such as *tuple* or *class*. These classes make up the parse tree that is created when a *Tm* source is parsed by *Rotan*. Second, those classes starting with the prefix *domain*, which implement an administrative shell for managing multiple domains, and for applying functions to individual parse trees.
- The subdirectory *Dlib/Defs/* contains *Tm* templates that are used to generate the actual C++ classes and code for a specific domain. *Make* does not descend into this directory, but each separate domain directory has a *Defs* subdirectory of its own, where the files in *Dlib/Defs/* are referenced, and where *make* will put the code-generation process in motion.
- *Rlib* (Rotan Library or Rule Library) forms the second half of the heart of *Rotan*. *Rlib* too, contains two broad categories of classes: first, those starting with the prefix *rule*, which implement an administrative shell for managing rules and applying functions to individual rule trees. Second, the other classes (un-prefixed),

which implement the *Rule Language*. These classes make up the parse tree that is created when a rule file is parsed by *Rotan*.

- *Rcc* (Rotan Commandline Compiler) is the interactive environment for working with domains and rules. This code implements a simple command-line shell, where commands can be given to load in source files (i.e. parse *Tm* data structures), load in rule files (i.e. parse *Rule Language* programs), and apply rules to the data structures. *Rcc* is, in effect, a layer on top of *Rlib* and *Dlib*, and which provides an interface for the manipulation and administrative functions found in those libraries.
- *Calc* and *Vnus* are two examples of actual *Tm* domains, adapted for use by *Rotan*. Although the *rcc* must have at least one domain in order to function properly, neither of these two directories are mandatory by themselves. *Any Tm* data structure definition file can be used to create a *Rotan* domain directory. In this case, the domain *Calc* was taken from the *tmdemo* example set that comes with the *Tm* distribution, whereas *Vnus* is the intermediate programming language from the *ParTool* project.

2.2.3 Creating a new domain

In order to create a new domain library for use in *Rotan*, all that is needed is a copy of the *Tm* data structure description file (or *ds* file, after its conventional extension). Let us assume that this domain will be called *foo*, and that the ds-file is therefore *foo.ds*.

Take the following steps:

- Go to **`rotan/Src`** and create the directories **`Foo`** and **`Foo/Defs`**.
- From **`Dlib/Defs/Newdomain`**, copy the file **`Makefile.dom`** to **`Foo/Makefile`**, and the files **`Makefile.defs`** to **`Foo/Defs/Makefile`** and **`domain.t`** to **`Foo/Defs/foo.t`**. Also copy your **`foo.ds`** to **`Foo/Defs`**.
- Now edit all the files copied from Dlib, and follow the instructions therein.
- Go to Rcc/rcc.cc, and follow the instructions there to add the appropriate initialisation code for the new domain.
- Compile.

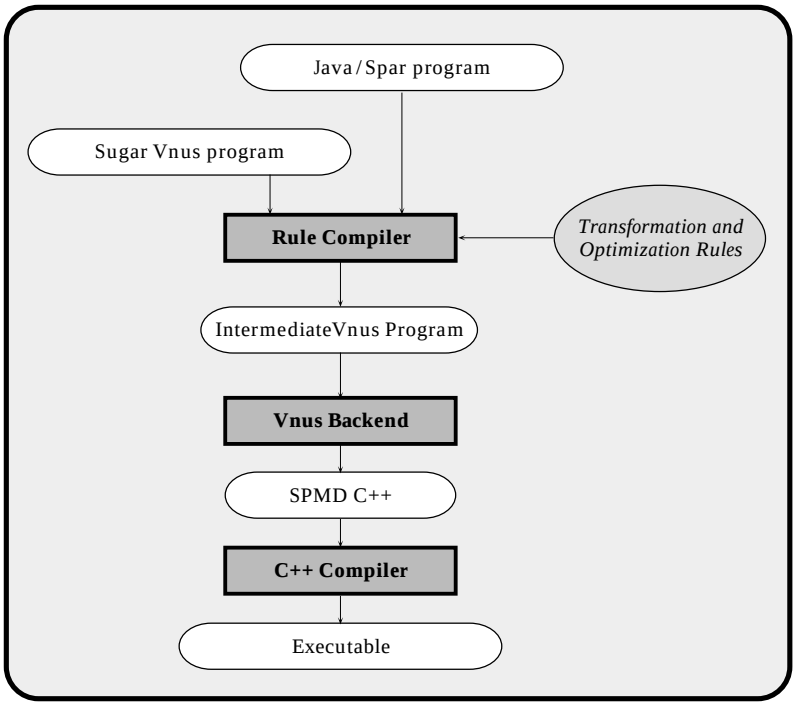
I originally intended to give a much more exhaustive description of the domain-creation process here, but I finally realised that this information will be of much use in the files themselves than separately on paper here.

Chapter 3 — The Early Rotan Documents

3.1 The *Rotan* Proposal

In order to provide a complete paper track record for the *Rotan* project, I am reproducing the 'technical' part of the original proposal here. This text can be seen as a baseline requirements specification, even though its intended goal was to obtain funding, rather than be a technical document as such. At the time of writing, the name *Rotan* had not yet been established.

3.1.1 Objectives, Workplan, and Results



Traditional compilers for programming languages are built as black boxes: a source program is transformed into target code by the compiler. Apart from setting a few compiler options, the programmer does not have the possibility of steering or influencing the translation process much.

When translating for parallel architectures, the quality and efficiency of the generated code is influenced by many more possible parameters than in the sequential case. Under these circumstances, the conventional compiler approach is too limited and restrictive, and contributes to the consistent lagging behind of parallel software in relation to the hardware developments.

In the I-Cal project, several approaches were used in an attempt to get a better grip on the translation process. One such approach focused on developing a calculus for data-parallel languages, and basing an intermediate language on that calculus. High level languages are first translated to this intermediate language, optimised, and only then further translated to target code. Complementary to this approach was the development of a more advanced compiler technology, leading in effect to a high-level, programmable compiler.

When translating a parallel program, a user is now able to program the compiler itself with rewrite rules (using the theoretical basis of the I-Cal calculus), which act on the program's representation in the intermediate format. In this manner, it becomes possible to easily enhance the compiler with hardware-specific or problem-specific optimisation engines, which themselves take the form of programs, written in the so-called Rule Language.

The current Rule Compilation System is a rough prototype version, created in aid of the author's Ph.D. research into semi-automatic translation of high-level parallel languages. As a prototype, it has been successfully used to implement various transformations and optimisations, based on the I-Cal calculus.

However, as a general-purpose tool it is currently not robust, complete, or user-friendly enough to be used as a valid support technology by anybody but its author, for any other areas than its current area of application. In addition to this, experience with and evaluation of the current prototype has brought to light a number of implementation problem and design issues that need to be addressed before the system can be used to its full potential.

The current project intends to address these issues, which are mainly concerned with re-design and re-implementation. The actual research work has already been performed and found valid.

3.1.2 Deliverables

Specifications

- The *Rule Language 2.0*. The design and grammar of a new transformation language for program optimisation / translation.

Programs & Tools

- The *Rule System Transformation Engine*. Core technology for applying the transformations specified in a Rule Language program.
- The *Rule Compilation System*. A full-featured implementation of a parameterisable compilation system, built around the Rule Transformation Engine.
- The *Rule Development Environment*. An interface to the Rule Compilation System, intended for interactively programming, debugging and running Rule Language programs.

Documentation

- The Rule Language 2.0 *User Guide and Reference Manual*.
- The Rule Compilation System *Technical Reference Guide for Implementators*.
- The Rule Development Environment *User Manual*.

3.1.3 Publications relevant to the proposal

[1] Edwin M. Paalvast, Leo C. Breebaart and Henk J. Sips, The Booster Language: A User Language Reference, Report Nr. 90 ITI 1864, TNO Institute of Applied Computer Science (ITI–TNO), Delft, The Netherlands, November 1990.

[2] Edwin M. Paalvast, Henk J. Sips and Leo C. Breebaart, *Developing and Tuning Parallel Algorithms with Booster*, Report Nr. 90 ITI A 5, TNO Institute of Applied Computer Science (ITI–TNO), Delft, The Netherlands, January 1990.

[3] Leo C. Breebaart, Edwin M. Paalvast and Henk J. Sips, "The Booster Approach to Annotating Parallel Algorithms". In *Proceedings 1991 International Conference on Parallel Processing*, 1991.

[4] Edwin M. Paalvast, Henk J. Sips and Leo C. Breebaart, "Booster: a high-level language for portable parallel algorithms", *Applied Numerical Mathematics*, 8, pp. 177-192, 1991.

[5] Leo C. Breebaart and Peter Doornbosch, *The Rule Language*, Report Nr. 92 TPD/ZP 1024, TNO Institute of Applied Physics (TPD), Delft, The Netherlands, September 1992.

[6] Leo C. Breebaart, Edwin M. Paalvast and Henk J. Sips, "A Rule Based Transformation System for Parallel Languages". In *Proceedings Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, ACPC/TR 92–8, 1992.

[7] Edwin M. Paalvast, Leo C. Breebaart and Henk J. Sips, "An Expressive Annotation Model for Generating SPMD Programs". In *Proceedings Scalable High Performance Computing Conference*, Williamsburg, VA, IEEE Computer Society Press, 1992.

[8] Leo C. Breebaart, "Experiences with Rule-based Compilation". In *Proceedings Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993.

[9] Joachim A. Trescher, Leo C. Breebaart, Paul F. G. Dechering, Arnaud B. Poelman, J. P. M. de Vreught and Henk J. Sips, "A formal approach to the compilation of data-parallel languages". In *Proceedings 7th annual LCPC workshop*, 1994.

[10] Leo C. Breebaart, Paul F. G. Dechering, Arnaud Poelman, Joachim Trescher, Hans de Vreught and Henk J. Sips, *The Booster Language — Syntax and Static Semantics*, Report Nr. CP-95-02 (Technical report), Delft University of Technology, November 1995.

[11] Paul F. G. Dechering, Leo C. Breebaart, Frits Kuijlman, Kees van Reeuwijk and Henk J. Sips, “The Meaning and Implications of the Forall Statement in V-nus”. In *Proceedings 2nd Annual Conference of the Advanced School for Computing and Imaging, ASCI ‘96*, Lommel, Belgium, 1996.

[12] Paul F. G. Dechering, Leo C. Breebaart, Frits Kuijlman, Kees van Reeuwijk and Henk J. Sips, “A Sound and Simple Semantics of the FORALL Statement within the V-nus Compiler Framework”. In *Proceedings 6th Workshop on Compilers for Parallel Computers*, Aachen, Germany, 1996.

[13] Paul F. G. Dechering, Leo C. Breebaart, Frits Kuijlman, Kees van Reeuwijk and Henk J. Sips, “A Generalized Forall Concept for Parallel Languages”. In *Proceedings 9th Annual Workshop on Languages and Compilers for Parallel Computing*, San Jose, California, USA, to appear in *Lecture Notes in Computer Science,* Springer Verlag, 1996.

[14] Paul F. G. Dechering, Leo C. Breebaart, Frits Kuijlman, Kees van Reeuwijk and Henk J. Sips, *A Generalized Forall Concept for Parallel Languages*, Report Nr. CP-96-003 (Technical Report), Delft University of Technology, 1996.

[15] Paul F. G. Dechering, Leo C. Breebaart, Frits Kuijlman, Kees van Reeuwijk and Henk J. Sips, “Semantics and Implementation of a Generalized FORALL Statement for Parallel Languages”. In *Proceedings IPPS*, Geneve, Switzerland, 1997.

[16] Leo C. Breebaart, *Rule-based Compilation*, Ph.D. thesis, Delft University of Technology [to be published], 1997.

3.1.4 Epilogue October 1997

Although you could quibble over the word “full-featured”, and although the *rcc* never evolved to the point where it became sophisticated enough to warrant a separate User Manual, or even a paragraph in the Reference Guide, I’d say that about 80 % of the project’s goals as originally proposed have been reached.

3.2 Tools for Rules — report of an investigation

At present count, the Rule Compiler consists of about 45 hand-written classes implementing the Rule Engine, about 50 hand-written support classes, and about 130 *Tm*-generated classes representing the *Vnus* domain.

The key characteristic of the support classes is that they provide general purpose functionality (ranging from string handling classes to object container classes such as *List* or *Vector*) that is not specific to the project, but can be used or built upon by any program.

One of the first decisions that needs to be made for the *RTTDS* project (*Rule-based Transformation of Tm Data Structures*, of course!), is determining what support classes we are going to use.

I have spend some time investigating the possibilities, and have identified what I think are our four main options. They are:

1. Continue with the currently used *Object Oriented Programming Library* (*Oopl* or *Ooplib* for short).
2. Create a new version of the *Oopl* from scratch.
3. Use the *GNU C++ Library* (*libgen* or *libg++* for short).
4. Use the HP/SGI *Standard Template Library* (*STL* for short).

I will provide a bit more info for each option, as well as my recommendation with respect to the usefulness of choosing it.

3.2.1. Current *Oopl*

Advantages:

- Those methods and classes we currently use from the *Oopl* are pretty well debugged — or at the very least the bugs (and possible workarounds) are *known*.
- With a little luck, no new coding effort will be required, especially not in the short term.

Disadvantages:

- Ancient legacy code from the dawn of time.
- Extremely ill-designed and badly programmed.
- Maintainability, documentation, and extendibility are all but non-existent.
- Portability problems.
- Follows the 'tree' model of support classes (one common *Root* object that everything is derived from), making integration with other classes difficult.

- Not up to the task of providing proper support for *RTTDS*, because of lack of support for modern C++ constructs such as templates.

Conclusion:

> It would be extremely unwise to continue with the *Oopl*. Now is the time to switch to something better. This will cause extra effort in the short term but will *so* be worth it in the long term that the question is almost a non-issue.

3.2.2. New *Oopl from* scratch

Advantages:

- Sources under our control, making it easy to tailor this library to our own specs, leaving out all unnecessary baggage, but including functionality we can benefit from.

Disadvantages:

- Writing a good *Oopl* duplicates a lot of effort and thought that has already been spent on these issues by the rest of the C++ world.
- There is no time for writing our own *Oopl* from scratch.
- Portability problems.

Conclusion:

> The 'no time' argument pretty much clinches this, as far as I am concerned. Of course it will be always be possible to add our own custom support classes if necessary (*UniqueId* comes to mind) — I just do not think that we can afford the time to write the really basic stuff (strings, containers, iterators, etc.) ourselves.

3.2.3. GNU Libgen

Advantages:

- Libgen follows the 'forest' model of support classes (no common *Root* object), making it easy to pick-and-choose, or to integrate with other approaches.
- We already have some experience with some of the *libgen* classes, and apart from the 'template' classes, they are pretty easy to work with, and appear quite stable.

Disadvantages:

- Has not been updated in years. Development appears to have stopped.
- Duplicates functionality now found in *STL*.

- The template 'mechanism' used by *libgen* is outdated (it predates C++'s intrinsic templates) and badly implemented (as we know from experience).
- Integration with GNU g++ itself is so bad it won't even compile correctly at all levels of optimisation.

Conclusion:

> Using *libgen* exclusively is a bad idea: the software is too old, too shaky, and too unsupported for reliable use (sort of like our own *Oopl*, but now written by other people). Certain classes, however, (in particular the *String*/*Regexp* classes) may well be suited as ingredients in a mix 'n match approach to support classes.

3.2.4. The STL

Advantages:

- Officially adopted by ANSI, and therefore expected to be a portable, reliable, ubiquitous standard for the foreseeable future.
- Excellent documentation available in the form of books, web pages, example programs.

Disadvantages:

- *STL* implements a novel algorithmic approach (called *generic programming* — for a quick introduction see attached article copies) which will have a learning curve attached, not only when it comes to how to use it, but particularly when it comes to writing our own additions and integrating it into our own software.
- Not *STL*'s fault, but nevertheless a potential problem: GNU g++ 2.7.2 (the most current version) has such lousy support for C++ templates, that *STL* will not compile out of the box with gnu. Instead, we need specially modified versions, and even then the results aren't 100 % (as I have already determined empirically).
- As yet, I do not get the impression that *STL* is *that* widely used. As such, its status is perhaps a bit more experimental still than we would have liked.

Conclusion:

> Going over all the information I've read and reviewed for this survey, I can only come to the conclusion that STL is the way to go, despite the disadvantages.

3.2.5 Epilogue October 1997

The decision to go with STL has, as far as I'm concerned, indeed been the right one, particularly so when we were able, halfway through the project, to switch to the experimental *egcs* compiler, a development C++ compiler based on the conventional GNU sources, but many lightyears ahead in terms of C++ (and STL) support.

Although the old GNU compiler actually did better than expected on compiling the rudimentary kind of STL usage needed for *Rotan*, the switch to *egcs* enabled me to start writing some *really* proper C++ code, without the need for workarounds and kludges every three lines. The fact that the *Rotan* code can also be compiled by the very strictly conforming KAI C++ compiler is a tribute to this.

3.3 The Name of the Game

3.3.1 Introducing: *Rotan*

A project needs a name. For historical reasons, the first generation of the rule compiler system never really acquired one, and this has been a frequent source of confusion, both in day-to-day conversation amongst ourselves, and when trying to communicate with others. Now that we are going to spend six months building the second generation of the system, the time seems right for a once-and-for-all fix of the nomenclature issues.

After considerable thought and experimentation, I propose to use the word *Rotan* as the canonical 'top-level' name for referring, in general, to the new project.

Rotan as a word has no special meaning, and is intended as neither acronym nor abbreviation. Instead it is a neutral, short, bilingual word that resonates comfortably in both Dutch and English with the concepts it is supposed to encompass. Elements of words like 'rotate', 'transform', 'translate', 'language', 'rules' and even 'data (structures)' can all be found in *Rotan*.

Originally, I was of the opinion that the word would look best when written as ROTAN, in small caps, but upon reflection I think that the added effort, however minimal, necessary to achieve this typesetting effect consistently in all our documents will simply not be worth the trouble. As to the remaining alternatives, spelling it in all caps as ROTAN is something I have never found aesthetically pleasing in other languages or systems. Using bold text — **Rotan** — is cute, but makes the word stand out too much from the surrounding text. In the end I have compromised and chosen to simply follow the convention as exemplified in e.g. *Booster*, *ParTool* and *Vnus*, and write the word in italic text, capitalised: *Rotan*.

3.3.2 Further Terminology

Within the *Rotan* project, it is still going to be necessary to identify certain sub-systems by separate names.

I have tried not to fiddle with the currently used names of these subsystems too much — too many radical changes and everything becomes confusing again, defeating the original purpose of the exercise. However, even the existing names have not been properly documented and explained anywhere yet, and could do with a little bit of polishing up. Hence this section.

The terminology used for the *Rotan* components is as follows:

The Rule Language

This is the actual name of the programming language that transformation rules are written in. The current version of the language (as described in Chapter 4 of my PhD thesis) is version 1.0, the new version will be version 2.0.

The *Rule Language* can be abbreviated as *RL* (or *RL 1.0* / *RL 2.0*, as necessary). Programs written in this language will reside in files having a *.rl* extension.

Since one of the key characteristics of the *Rule Language* is that it can be parameterized, it will occasionally be useful to refer to the *Core Rule Language*, that is: only those elements of the *Rule Language* that are invariant, i.e. available across all possible *instantiations* of the language.

The Rule Engine

This refers to the 'implementation' of the *Rule Language*, the actual code for performing the transformations specified by a rule. The I/O code for reading rule files and parsing rules is also considered to be part of the *Rule Engine*. The *Rule Engine* is not in itself, however, an executable program.

The *Rule Engine* can be abbreviated as *RE*, but is historically more often referred to as the *Rlib*, after the name of the library which houses its code.

The Domain Libraries

A domain is, in the abstract sense, nothing more than a *Tm* data structure, as described in a *Tm* data structure file. In the *Rotan* context, the *Tm* data structure file is called the *domain file*, and a *Domain Library* consists of all the code generated from a domain file, plus any additional handwritten code for custom methods, such as printing and parsing.[1]

For unfortunate historical reasons, the *Vnus* domain library is currently known as the *Nlib*.[2] In *Rotan*, this will be renamed into the more appropriate *Vnuslib.*

The Rule Compiler

The most ambiguous name of all. To begin with, up until now, the phrases 'Rule Compiler' and '*rcc*'[3] (after the name of the Unix executable) have often been used to signify the entire project, as in: "We're going to use Leo's Rule Compiler". With the advent of the top-level name *Rotan,* this usage is officially deprecated.

Even so, confusion has arisen because three closely related, yet subtly different concepts have at times all been referred to as 'Rule Compiler' or *rcc*. The common denominator is that in each case the Rule Compiler in question is the actual executable used to trans-

---

[1]Printing and parsing will be automatically generated as well, as soon as *Tm* fully supports object oriented languages, and *Rotan* fully supports *Tm* domains — but those are issues outside the scope of this memo.

[2]Once upon a time there used to be a *Vista* domain library already called *Vlib*. When I needed to create a *Vnus* domain library, I therefore had to think of something else to use: I simply chose the next letter in the word 'Vnus'.

[3]Or rather '*ncc*' to be more historically correct, see footnote 2.

form an input file into an output file according to an RL program. The difference lies in how many of the various parameters are considered to be *part of* the executable.

Although it is usually clear from context which concept is being referred to, it will be good to have an 'officially sanctioned' way of referring to each of the three possibilities. Therefore:

1. The *Core Rule Compiler* or *rcc.*

   In analogy to the Core Rule Language, this is the most skeletal form of the executable, consisting of library interface code, command-line parsing code, and the interactive user interface code. The Core Rule Compiler is that part of the executable that is invariant across all possible executables created by the *Rotan* system. In functional terms, its usage is:

   rcc domain rules source

2. An *Instantiated Rule Compiler*.

   In analogy to an Instantiated Rule Language, this is an executable that is configured to process both source files and rule files associated with a specific domain. It consists of the Core Rule Compiler code and a domain library. An Instantiated Rule Compiler for a domain *D* is also often referred to as "the *D Compiler*". In functional terms, its usage is:

   (rcc domain) rules source

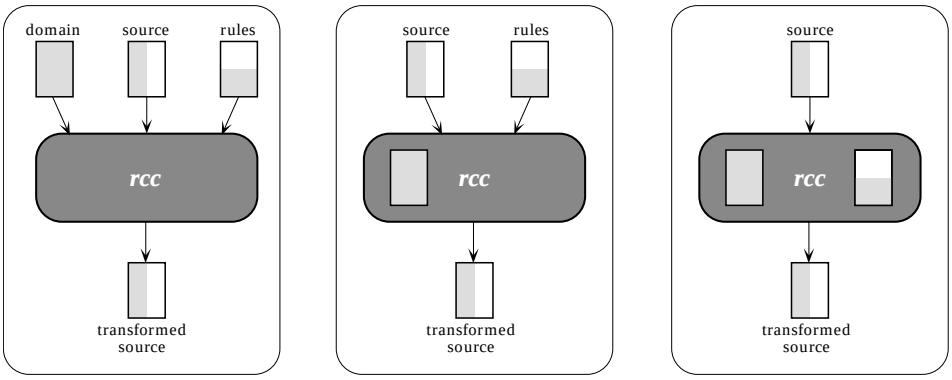
3. A *Specific Rule Compiler* or *Transmogrifier*

   This is an executable that is configured to process source files associated with a specific domain, according to a specific rule program. In effect, this is the fully instantiated rule compiler, equivalent to the traditional compiler-as-a-black-box. Specific Rule Compilers are typically referred to as "the <*adjective*> *D Compiler*", for example: "the shared-memory *Vnus* compiler", but basically the executable can at this point be given any appropriate name and simply regarded as another Unix command. In functional terms its usage is:

   (rcc domain rules) source

I hope the above classification will make some sense to the reader. The Core Rule Compiler is a single entity created by my programming in C++. There are infinitely many Instantiated Rule Compilers possible; one can be created at any time and by anybody, simply by writing a *Tm* domain file and plugging the resulting domain library into the Core Rule Compiler. Each Instantiated Rule Compiler in turn can be turned into an infinity of Specific Rule Compilers; one of which can be created at any time and by anybody, simply by writing a program in the Rule Language and plugging that in turn into the Instantiated Rule Compiler.

If this functional approach to the Rule Compiler seems needlessly complex, it should perhaps be pointed out that the person writing domains does not have to be the same person as either the one writing rules, or the one writing the source files to the compiler.



Core Rule Compiler **Instantiated Rule Compiler** **Specific Rule Compiler**

3.3.3 Meanwhile, in a Parallel Universe

The software engineering aspects of *Rotan* are complicated somewhat by the fact that *Rotan* is not immediately going to supersede all that went before. The current implementation of the rule system is going to remain in existence, on a separate revision branch, for purposes of my PhD thesis. It will therefore be necessary to have a way of referring to and differentiating between these two software branches in conversation and writing.

With this in mind I propose to use the phrase *Rotan BC* for the 'old' system.

3.3.2 Epilogue October 1997

The proposed names have not been very difficult to assimilate in day-to-day conversation. In hindsight, I think that especially the name *Rotan* itself has been a particularly good choice.

3.4 *Rotan* and *Tm*

In this memo I will describe the relationship between *Tm* and *Rotan*, specifically in the light of the requirement specifications for the latter. Used terminology is as per *R002*.

3.4.1 A comparison

From the very beginning *Rotan BC* has included (by way of the *Libgen* utility and its processing of domain files) a crude functionality similar to that which *Tm* provides.

A *Rotan BC* domain file contained compact, language-independent specifications of elementary, record-like data structures. The *Libgen* utility parsed the domain file, parsed a template file, and used the C preprocessor to generate from these inputs a set of C++ classes — one instance of the template file for each structure defined in the domain file. Each class contained the member fields defined in the original record, plus a set of generated methods allowing the rule engine to traverse a parse tree made up out of instances of these classes.

A *Tm* data structure file contains compact, language-independent specifications of several different kinds of data structures, one of which is an elementary, record-like structure called a tuple. The *Tm* compiler parses the data structure file, parses a template file, and outputs code according to elaborate meta-commands sprinkled throughout the template file. Depending on the template, the result can be anything. Default templates yielding C code are included with *Tm*. These templates generate a number of standard functions (ranging from allocation to parsing to printing) that work on each structure in the data structure file.

The similarities between these two descriptions are striking. As a full-fledged system, *Tm* is obviously much more powerful and generic than the special-purpose *Rotan BC* approach, but the basic idea of generating code based on the specific combination of data structure definitions, template files, and a substitution-based tool is exactly the same.

Even more interesting is the complimentary nature of the primary differences between the two approaches. In *Tm* an entire template language is available to direct the code generation; in *Rotan BC* we never made that leap, trying to make do with elaborate makefiles and shell commands instead. *Rotan BC*, on the other hand, offered an entire rule language to direct manipulation and transformation of the generated data structures, which is something *Tm* has no direct support for. In other words:

> *Tm* is almost unlimited in what it can generate, but strongly limited in what it can manipulate, whereas *Rotan BC* is almost unlimited in what it can manipulate, but strongly limited in what it can generate.

An idea was born...

3.4.2 Integration: the First Step

Our first attempt to bring *Tm* and *Rotan BC* closer together occurred about a year and a half ago, when we converted the *Rotan BC* domain file into an equivalent *Tm* data structure file and used *Tm* itself rather than the exceedingly untrustworthy *Libgen* utility to generate the C++ classes and the tree traversal routines. No attempt was made at this point to develop generally useful C++ templates (e.g. with support for the other *Tm* data structures, and with support for e.g. parsing and printing methods). At this point all we wanted was to simply achieve a one-to-one mapping of existing functionality.

This succeeded beyond all expectation. Only one extension had to be made to the *Tm* data structures: support for inheritance — a crucial feature of C++ classes. Likewise only one simplification had to be made to the *Rotan BC* data structures: abandoning the mapping that allowed rule language keywords to be different from their domain counterparts. After these two changes, the two formats matched, and *Libgen* was successfully replaced by *Tm*.

3.4.3 *Tm + Rotan BC = RTTDS = Rotan*

In *Rotan* we intend to take the fusion between *Tm* and *Rotan BC* to its logical conclusion, which is the creation of a generic system for Rule-based Transformation of Tm Data Structures. *RTTDS* was in fact an acronym I considered using for a while, but in the end I decided that *Rotan*, while not as appropriate, can at least be (a) remembered and (b) pronounced.

In *Rotan* we want to be able to accept any valid set of *Tm* data structures as a domain, and allow a user to write transformation rules in the *Rule Language* instantiated for that domain, and apply those rules to source files using an *Instantiated Rule Compiler* for that domain.

In order to successfully do this, the *Rule Language 2.0* will have to be designed more explicitly with *Tm* data structures in mind, and the *Rule Engine* will have to be modified accordingly. Both these design issues will be the subject of future memos in this series.

It appears increasingly likely that *Tm* itself will have to be modified, too. Although inheritance is possible, the current 'flat' nature of *Tm*'s constructor types precludes their easy integration into the 'deep' inheritance chain that is one of the defining characteristics (and strong points) of the *Rule Language*.[1] More information on this issue will be the subject of a future memo by Kees.

[1] In *Rotan BC*, for instance, we had to rewrite the *Vnus* data structure file in such a way that no constructor types, but only tuples were used. Although this process could perhaps be automated so that it can be applied to arbitrary data structure files, it leads to redundancy and a loss of cohesion in the rewritten file. Not good.

3.4.3 Epilogue October 1997

Tm has indeed undergone a major revision during the course of the *Rotan* project, and now contains support for a true *class* type that makes everything described above possible.

These changes have been documented in the new version of the *Tm User Manual,* also available in the PDS report series.