Delft University of Technology
Parallel and Distributed Systems Report Series

The Rule Language 2.0

Leo Breebaart

report number: PDS–1997–001

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 the grammar of version 2.0 of the Rule Language (RL), as has been implemented in the *Rotan* system. The Rule Language is a special purpose high level language, intended to provide an easy-to-use interface for transforming *Tm* data structures. (For information on Tm, see [4] and [5].)

The first versions of both the *Rotan* system and the Rule Language were developed as part of the *ParTool* project ([1], [6]). Those versions were not yet based on *Tm*. Instead, a data structure format of our own devising was used, limiting the general applicability of the system.

For RL 2.0 we have, apart from adapting the language to take the new *Tm* basis into account, also evaluated all the previous experiences with RL 1.0 (documented in Chapter 7 of [2]), and as a result re-designed the language to be both more powerful and more user-friendly.

For consistency's sake, the structure and style of this document purposely resembles that of the *Vnus Language Specification* ([4]). Some actual content has been borrowed from that document as well, with no disrespect to the original author intended.

Thanks are due to Kees van Reeuwijk, Frits Kuijlman and Henk Sips for their input and support during all phases of the RL 2.0 and *Rotan* development process.

Delft,
October 1997

Chapter 2 — Grammar Notation

2.1 The purpose of the grammar rules

The grammar of the RL (unless explicitly specified otherwise we will assume RL means: RL 2.0) is formally defined in appendix A. Throughout the text, parts of this grammar are shown as illustrations. For didactic purposes, these parts can sometimes deviate slightly from the official version.

The text accompanying the grammar rules is intended to annotate and clarify the language. Text in italics indicates open questions that still need an answer, or parts of the language that were unimplemented at the moment of writing.

2.2 The notation

The grammar consists of a series of *productions*, where each production specifies the derivation of a *nonterminal* symbol from terminals and other nonterminals. Nonterminal symbols are shown in an *italic* font, terminal symbols are shown in a `courier` font.

The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative production rules then follow on succeeding lines. For example, the production:

> *builtinFunctionCall:*
>   **`close(`** *tree* **`)`**

states that the nonterminal *builtinFunctionCall* produces the terminal string "**`close(`**", followed by the nonterminal *tree*, followed by the terminal ")".

To clarify the meaning of the language construct, it is allowed to prefix a nonterminal with a name, followed by a dash. This prefix is ignored in the interpretation of the grammar rules. For example, the following production is identical to the previous one:

> *builtinFunctionCall:*
>   **`close(`** *argument-tree* **`)`**

A production may have more than one alternative; each alternative is listed on a separate line, or in a separate production. For example, a tree has several possible productions:

tree:
  simpleTree
  rule-identifier **(** *tree* **)**

tree:
  builtinFunctionCall

The subscripted suffix *'opt'*, which may appear after a terminal or nonterminal, indicates an optional symbol. This suffix is a shorthand for two grammar rules, one that omits the optional element, and one that includes it. For example:

pattern:
  domainNode $predicate_{opt}$

is a shorthand for:

pattern:
  domainNode
  domainNode predicate

If a grammar rule contains more than one optional symbol, all combinations of presence and absence of the symbols are allowed.

When all the alternative productions of a nonterminal consist of single terminal symbols, a more compact notation is used, where all alternatives are listed in a contiguous list. For example:

relationOp: one of
  ==   !=

Finally, the symbol 'λ' denotes the empty production.

Chapter 3 — Language Constructs

3.1 Comments

The Rule Language supports two forms of comment: line comments, that start with '`//`', and end at the end of the line; and multi-line comments, that start with '`/*`' and end with '`*/`'.

Comment cannot be nested.

3.2 Rules

> *rules:*
>   *ruleList*
>
> *ruleList:*
>   *rule*
>   *ruleList rule*

Rules are like C functions: all are equal and each can be invoked separately by the outside world. The language does not recognise or enforce the concept of a (file-based) *program* as is usually the case in other programming languages. The hierarchical approach of RL 1.0 (rules, drivers, engines) is completely gone. Instead, the RL parser sees a source file as a sequence of individual rules.

For the Rule Language, different run-time systems are possible. The current *Rotan* implementation defines a rule *interpreter*, which dynamically reads in a rule file, and makes the parsed rules available as instances of the *Rlib* class *Rule* in an STL container.

➠ *What is still needed (and is in fact crucial for the ability to handle rules with embedded code) is the implementation of a rule* compiler*, which can be used to convert a rule-file into a true C++-library with an API that can be used from outside the* Rotan *run-time environment (*rcc*), without further need for parsing.*

➠ *In analogy to C, we can establish the convention that a rule with a special reserved name (such as, say,* **`main`***), serves as a default entry point for executing a rule 'program' if that is required.*

In order to avoid the boring but complex problems associated with scoping and name-space issues, the Rule Language recognises only a single global name-space and scope for rules. There is no module concept, and files are considered operating system artefacts that play no semantic role in the Rule Language.

The Rule Language parser performs syntactical checks on the validity of the rules, as well as a number of semantical checks. There are also run-time tests and sanity-check assertions in the code that only take effect during the actual application of a rule.

rule:

    `rule` *rule-identifier* **`begin`** $pattern_{opt}$ –> $actionList_{opt}$ **`end`**

    `repeat rule` *rule-identifier* **`begin`** $pattern_{opt}$ –> $actionList_{opt}$ **`end`**

Each rule always acts on exactly one *input tree*, which needs to somehow be specified as a parameter when the rule is applied.

The input tree is made explicitly available to the rule itself as the predefined rule variable *$ROOT*.

A rule is allowed to completely replace *$ROOT* by a new tree, with the one restriction that the replacement must be of the same type as *$ROOT* was. By 'type' we mean here one of the types defined by the Domain Library (i.e. in the *Tm* data structure file).

In RL 2.0, *applying* a rule causes an attempt by the system to *match* the rule's *pattern* to the input tree. This match is said to be successful if a mapping between the pattern nodes and the tree nodes can be found such that both *content* (node types and values) and *structure* (parent-child or sibling relationships) correspond (see Section 3.2).

The process of successfully matching a pattern results in the instantiation of a number of *rule variables* (as specified by the rule programmer in the pattern itself), each of which identifies a subtree of the original input tree.

When a condition has matched, the rule will *fire*, meaning that the list of *actions*, if present, will be *executed.* Whereas the matching process is strictly read-only as far as the input tree is concerned, an action can cause changes to the tree by using one or more of the rule variables in the left hand side of *assignment* actions (see Section 3.3).

If no pattern is present, the rule always matches. This is useful when implementing 'control flow' rules, which specify a sequence of rule applications on a single input tree (this is a good way of implementing rule 'programs' after all).

If no action list is present, the rule will be 'const', and have no effect on the input tree at all. This is useful during the writing and debugging of complex match patterns.

If the keyword **`repeat`** precedes the rule definition, the rule is a so-called *reappliccable rule*. After such a rule has been successfully applied, the system will continue to apply the same rule to the same input tree for as long as new matches are found. It is up to the programmer to avoid writing endlessly matching rules that never terminate.

3.3 Patterns

pattern:

    domainNode $predicate_{opt}$

domainNode:
  classNode
  listNode

At the top level a pattern is either a *classNode*, which stands for any type as defined in the *Tm* domain data structure file, or a *listNode*, which stands for a sequence of *patterns*.

Both *classNode* and *listNode* map directly and intuitively to their *Tm* counterparts[1], but a major advantage of the Rule Language is that it allows us to specify for these domain nodes an additional *predicate* that must hold before the match is considered successful.

classNode:
  class-identifier $rulevarDef_{opt}$

rulevarDef:
  . *rulevar-identifier*

A *classNode* consists of a single type identifier, taken from the set of types defined by the Domain Library. If a node with such a type is found at the appropriate position in the input tree, it will be matched to the *classNode*, and the portion of the input tree starting at that node will be optionally assigned to a rule variable for further reference.

A *listNode* works similarly, but consists of a sequence of patterns and regular expressions enclosed in list brackets:

listNode:
  [*listElementList* **]** $rulevarDef_{opt}$

listElementList:
  listElement
  listElementList listElement

listElement:
  pattern
  regex

regex:
  ... $rulevarDef_{opt}$ $predicate_{opt}$

Apart from patterns, it is also possible to have a limited set of regular expressions appear as listElement. Currently, this set is very limited indeed, since only the ellipsis operator (also known as the *anyspan* is implemented. This construct acts as a regular expression wildcard, representing zero or more list elements in the domain tree list to be matched.

[1]In hindsight, it would have been better to name the *classNode* a *typeNode* instead, since the *class* is only one of the *Tm* types being supported by the Rule Language.

➠ *Currently, the ellipsis wildcard is only rudimentary supported in the implementation. Improving regular expression support (and, indeed, list support itself) is a high priority item on the RL todo list.*

Once a domain node is matched, the pattern itself is still not considered successful until the *predicate* also evaluates to **true** (or is missing altogether). Note that a predicate can contain further matches, and that the patterns in a *listNode* can each have predicates of their own.

At the top level, a predicate is a boolean expression over terms called *simple predicates*:

> *predicate$_{opt}$:*
>
> λ
> **<** *predicate* **>**
>
> *predicate:*
>
> *predicate* **and** *predicate*
> *predicate* **or** *predicate*
> **not** *predicate*
> **(** *predicate* **)**
> *simplePredicate*
>
> *simplePredicate:*
>
> *match*
> *comparison*
> *binding*
> *embeddedFunctionCall*

The C-like shortcuts **&&**, **||**, and **!** can be used as alternatives for **and**, **or**, and **not**.

The 'simple' in simplePredicate should, in this context, be seen as the antonym of 'compound', and definitely not of 'complex' — the simplePredicates can be almost arbitrarily complex and deep, and there are rather a lot of them, although the division into four main groups is intended to clarify matters somewhat.

> *match:*
>
> **contains** pattern
> *member-identifier* **contains** *pattern*
> *member-identifier* **matches** *pattern*

A *match* predicate causes an attempt to find a deeper match within an already-matched *domainNode* (the one that the closest surrounding *predicate* belongs to).

The first form is called the *globalcontains*, and tries to find the specified pattern in any of the children of the *domainNode*, starting with the first (leftmost) child. For historical reasons, a globalcontains also matches if the *domainNode* itself matches the pattern.

➠ *It would be better to remove this historical meaning, and make the globalcontains a 'true' contains predicate that only looks at a node's children, not at the node itself.*

The second form of match is the *membercontains*, which tries to find the pattern in the specified member only.

The third form is the *membermatch* which only succeeds if the member in question is an *exact* match for the pattern.

If a membermatch is successful, the corresponding membercontains (i.e. with the same *member-identifier* and *pattern*) would also be successful, but not necessarily vice versa. If a membercontains is successful, the corresponding globalcontains (i.e. with the same *pattern*) would also be successful, but not necessarily vice versa.

In the Rule Language, *classNode* members are, like the types themselves, identified by the same string used to denote them in the *Tm* data structure file.

➠ *If the matched* domainNode *is a* listNode*, which does not have named members, member matches and member contains are still possible, because the rule language recognises "pseudo-members" for list classes. Instead of member identifiers, one can use number strings, which the system will convert to the appropriate child index number.*

> *comparison:*
>   *rValue relationOp rValue*
>
> *relationOp: one of*
>   **==**   **!=**
>
> *rValue:*
>   *member-identifier*
>   *simpleTree*
>
> *simpleTree:*
>   **$** *rulevar-identifier*
>   *stringLiteral*
>   *intLiteral*

The *Rotan* system guarantees that the equality operator is defined for every possible domain tree. Two trees are equal if they have exactly the same structure with exactly the same values for the leaf nodes.

An *rValue* is basically composed of constructs in the Rule Language that yield a domain tree when evaluated. It can be either a member reference (referring to a member of the nearest surrounding *domainNode*), a rule variable reference (the rule variable must have been bound to a subtree in a section of the match executed prior to the comparison), signified by prefixing the name of the rule variable by a '**$**', or a string/int-literal (which of course does not point into the existing input tree but will rather yield a newly created tree upon evaluation).

binding:
  rulevar-identifier = *rValue*

Binding or instantiating a rule variable can be done in two ways. We have already seen that a *domainNode* match can be assigned to a rule variable by use of a pattern suffix, but an explicit assignment is also possible. Such a *binding* will always yield the value **true** (barring run-time problems, such as the attempted use of an invalid member string or unbound rule variable as *rValue*).

Note that rule variable references are always prefixed by '**$**', whereas definitions are not (this is similar to the way shell programming works under Unix).

embeddedFunctionCall:
  { *codeString* **}**

➠ *The* codeString *needs to be executed, and must therefore represent a callable C++ function in the run-time environment. This poses obvious problems for an interpreter, which will always cripple an interpretative application such as the* rcc *somewhat. Rules containing embedded code must be read-in, dumped to file, compiled, and then linked back into the* rcc *itself, before such rules can be applied. If any rule containing embedded code changes, the entire* rcc *must be relinked. What happens in the codeString is entirely up to the programmer. The current state of the match (i.e. the rule variable values) will be passed to the function, so the codeString has access to all the rule variables matched up to the point of the codeString's occurrence, and will be able to add new rule variables of its own. Some simple preprocessing must be done to convert the $FOO syntax of referring to rule variables into something that is a legitimate C++ identifier.*

➠ *The C++ code used in* embeddedFunctionCall *should, since the nonterminal is used as a predicate, be suitable for using as the body to a boolean function in C++. In particular, this means that the* codeString *must end with a "*`return` `<boolean>`*" C++ statement. This is a semantical requirement on the* codeString *that cannot be enforced during parsing of a rule.*

3.4 Actions

actionList:
  action
  actionList **;** *action*

action:
  assignment
  embeddedProcedureCall

assignment:
  $ *rulevar-identifier* = *tree*
  rulevar-identifier = *tree*

embeddedProcedureCall:
  { *codeString* **}**

The *embeddedProcedureCall* is similar to the *embeddedFunctionCall* from the match, only this code should *not* return anything, but be a true procedural (void) function instead.

The rulevar-identifier in the left-hand side of an assignment can be prefixed by '`$`' or not. In the first case, the rule variable must already exist, and the subtree it points to will be replaced by the value yielded by an evaluation of the *tree* in the right hand side of the assignment. In the second case, the semantics of the assignment are similar to that of the *binding* in a match: a new rule variable is initialised (with the evaluated *tree* value) and made available to subsequent actions.

tree:
  simpleTree
  ruleCall
  builtinFunctionCall
  buildTree

The tree construct is similar to the *rValue* construct we encountered in match patterns, but contains a few extra constructs that make no sense in the match context, whereas the member identifiers allowed in the former case make no sense here.

Another difference, speaking semantically, is that rule variable references evaluated in the context of an assignment will always yield a *clone* of the original domain-(sub)tree they point to. So although assignments are the means by which the original tree is changed, these changes can never cause common sub-trees to come into existence. Sections of the original tree may become 'orphaned', but will never have more than one link pointing to them.

So if *$FOO* is a rule variable, then an assignment *$A* = *$FOO*, will cause the subtree that *$A* points to to be replaced by a clone of *$FOO*. Any other rule variables pointing into the original *$FOO* continue to point there, and any changes to those rule variables in subsequent actions will only affect the original *$FOO*, and *not* the clone that was just assigned to *$A*. Had that been the intention, the changes to those other rule variables should have been made *before* the assignment to *$A*. This cloning behaviour of the Rule Language implies that the order in which actions are written down greatly influences the final outcome of the rule.

ruleCall:
  rule-identifier **(** *tree* **)**

builtinFunctionCall:
  `close(` *tree* **)**

A *ruleCall* applies a rule (this can be the current rule) to its *tree* argument. Rule calls are side-effect free, and are guaranteed not to change their argument, but simply yield a new domain-tree as a result. A consequence of this functional behaviour is that if any rule *wants* a ruleCall to affect its argument, this must be written as e.g. "*$A = foo($A)*". This is important to remember for sequencing rules.

Currently, there is only one recognised builtin function call: the **close** function. Semantically, a function call behaves similar to a rule call: its return value is a domain tree, and its argument is not changed. The **close** primitive only makes sense in the context of a reappliccable rule. It returns a tree where subsequent applications of the current rule will always fail, no matter what pattern is being looked for. By assigning 'closed' trees to parts of the original tree, it can be assured that the rule, when re-applied, will find the 'next match' instead of endlessly looping on the same first match.

Note that the functional behaviour of **close** with respect to its argument implies that after the assignment "*$A* **= close(***$B***);**", the subtree pointed to by *$B* is *not* closed off — only the closed clone of *$B* assigned to the location of *$A* will be closed.

buildTree:
    buildClassNode
    buildListNode

buildClassNode:
    class-identifier buildMembers$_{opt}$

buildMembers:
    < *buildMemberList* **>**

buildMemberList:
    buildMember
    buildMemberList **;** *buildMember*

buildMember:
    member-identifier = tree

buildListNode:
    [*treeList* **]**

treeList:
    tree
    treeList tree

A *buildTree* builds, as the name implies, a new domain tree from scratch. It does this using a syntax that is similar to the one used to match a pattern, but with some significant differences.

One major difference between the *buildTree* and the *pattern* is that the former does not have the notion of a wildcard. The tree is constructed explicitly, node by node. Another difference is that although the predicate syntax is used, the only 'predicates' allowed here are straightforward member-instantiations for created class nodes.

Appendix A — The Rule Language Grammar

A.1 Rules

rules:
    ruleList

ruleList:
    rule
    ruleList rule

rule:
    `rule` *rule-identifier* **`begin`** *pattern*$_{opt}$ –> *actionList*$_{opt}$ **`end`**
    `repeat rule` *rule-identifier* **`begin`** *pattern*$_{opt}$ –> *actionList*$_{opt}$ **`end`**

A.2 Patterns

pattern:
    domainNode predicate$_{opt}$

domainNode:
    classNode
    listNode

classNode:
    class-identifier rulevarDef$_{opt}$

rulevarDef:
    `.` *rulevar-identifier*

listNode:
    `[` *listElementList* **`]`** *rulevarDef*$_{opt}$

listElementList:
    listElement
    listElementList listElement

listElement:
    pattern
    regex

regex:
    `...` *rulevarDef*$_{opt}$ *predicate*$_{opt}$

predicate$_{opt}$*:*
: λ
: **<** *predicate* **>**

predicate:
: *predicate* **`and`** *predicate*
: *predicate* **`or`** *predicate*
: **`not`** *predicate*
: **(** *predicate* **)**
: *simplePredicate*

simplePredicate:
: *match*
: *comparison*
: *binding*
: *embeddedFunctionCall*

comparison:
: *rValue relationOp rValue*

rValue:
: *member-identifier*
: *simpleTree*

simpleTree:
: **$** *rulevar-identifier*
: *stringLiteral*
: *intLiteral*

binding:
: *rulevar-identifier* **=** *rValue*

embeddedFunctionCall:
: **{** *codeString* **}**

A.3 Actions

actionList:
: *action*
: *actionList* **;** *action*

action:
: *assignment*
: *embeddedProcedureCall*

assignment:

    $ *rulevar-identifier* **=** *tree*
    rulevar-identifier **=** *tree*

embeddedProcedureCall:

    { *codeString* **}**

tree:

    simpleTree
    ruleCall
    builtinFunctionCall
    buildTree

ruleCall:

    rule-identifier **(** *tree* **)**

builtinFunctionCall:

    close(*tree* **)**

buildTree:

    buildClassNode
    buildListNode

buildClassNode:

    class-identifier buildMembers$_{opt}$

buildMembers$_{opt}$*:*

    λ
    < *buildMemberList* **>**

buildMemberList:

    buildMember
    buildMemberList **;** *buildMember*

buildMember:

    member-identifier = *tree*

buildListNode:

    [*treeList* **]**

treeList:
    tree
    treeList tree

A.4 Tokens and lexical entities

relationOp: one of
    `==   !=`

keyword: one of

identifier:
    identifierChars but not a *keyword*

identifierChars:
    letter
    identifierChars letter
    identifierChars digit

letter: one of:
    `_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

stringLiteral:
    " *stringChars* "

stringChars:
    stringChar
    stringChars stringChar

stringChar:
    any *char* except an unescaped ‘"’

intLiteral:
    $sign_{opt}$ *digits*

digits:
    digit
    digits digit

digit: one of:
    `0 1 2 3 4 5 6 7 8 9`

sign: one of:
    `+ -`

char:

any character
escapedCharacter

escapedCharacter:

`\b`
`\f`
`\n`
`\r`
`\t`
`\"`
`\\`
`\` *digit*
`\` *digit digit*
`\` *digit digit digit*

Bibliography

[1] 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.

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

[3] C. van Reeuwijk. Tm Users' Manual. Technical Report, Delft University of Technology, Department of Electrical Engineering, Delft, The Netherlands, 1992.

[4] C. van Reeuwijk. Vnus Language Specification, version 2.0. Technical Report, Delft University, Faculty of Technical Mathematics and Informatics, Delft, The Netherlands.

[5] C. van Reeuwijk. WWW Page, 1997. **`http://pds.twi.tudelft.nl/~reeuwijk/tm.html`**

[6] E. M. Paalvast. *Programming for Parallelism and Compiling for Efficiency.* Ph.D. thesis, Delft University of Technology, June 1992.

❖❖❖❖❖