



Master Thesis
Master's Program in Computer Science

**An Experimentation Laboratory for the Automatic
Parallelization of Programs written in the R Language
(ALCHEMY)**

submitted by
Michael Miroid

submitted
25.11.2011

Supervisor
Dr. Frank Padberg

Advisor
Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,

_____ Datum

_____ Unterschrift

Abstract

The R language is being widely used in fields such as bioinformatics and statistics, where it is common to process large data sets. R does not exploit the speed-up that may be gained from current multicore processors. In this thesis, I present the conception, the design, and the realization of the ALCHEMY parallelization laboratory and its integration into the R runtime environment.

I introduce the concept of *Transmutators*, small software components that apply transformations to an AST-based intermediate language called *AIR*, which is specifically designed to facilitate analysis and code transformation.

Contents

1	Introduction	11
2	Approach	13
2.1	Interfacing with the user	13
2.2	Preprocessing the R Program for Parallelization Analysis	14
2.3	Parallelization Analysis	15
2.4	Parallel Execution	18
2.5	Scope of this Thesis	19
3	Related Work	21
3.1	R	21
3.1.1	S Expressions	21
3.2	Approaches to Parallelizing R	22
3.2.1	Exploit Inherent Data-Parallelism in R	23
3.2.2	Exploit Task-Parallelism Hidden in Sequential R Programs	23
3.3	Parallel Programming Patterns (Skeletons)	23
3.4	Parallel Intermediate Representations	24
3.5	Approaches to Automatic Parallelization	25
4	Requirements Analysis	27
4.1	UC 1: Execute R Program	28
4.2	UC 2: Perform Interactive R Session	29
4.3	UC 3: Evaluate R Expression	29
4.3.1	AIR Expression	31
4.3.2	XML Representation	33
4.4	UC 4: Convert R Code to AIR	33
4.5	UC 5: Transmutate AIR	34
4.5.1	Transmutator	36
4.5.2	Transmutation Controller	37
4.5.3	Rule-Based Transmutation Configuration	38
4.5.4	Static and Dynamic Configuration	41
4.6	UC 6: Configure ALCHEMY	42
4.7	UC 7: Convert AIR to R	42
4.8	UC 8: Query AIR	42
4.9	UC 9: Modify AIR	43
4.10	Analysis Overview	43
5	ALCHEMY Software Design and Implementation	44
5.1	System Architecture	44
5.1.1	Architectural Factors	44

5.1.2	Architectural Decisions	45
5.1.3	Description of ALCHEMY's Architecture	46
5.1.4	Package: RAlchemy	47
5.1.5	Class: RUserInterface (or R UI)	51
5.1.6	Package: AlchemyAdapter	52
5.1.7	Class: RtoAIRConverter	53
5.1.8	Class: AIRtoRConverter	56
5.1.9	Package: RCore	56
5.1.10	Package: RServices	56
5.1.11	Package: Communication	56
5.1.12	Package: Logging	56
5.1.13	Package: RServer	56
5.1.14	Package: AlchemyCore	56
5.1.15	Class: TransmutationService	57
5.1.16	Class: SessionService	57
5.2	Inter-Process Communication	58
5.3	R Services	61
5.3.1	Value Service	61
5.3.2	Environment Service	62
5.3.3	AIRtoR Service	63
5.3.4	R Service Client: EnvServiceProxy class in AlchemyCore	63
5.3.5	R Service Client: ValueServiceProxy in AlchemyCore	64
5.4	AIR Design	65
5.4.1	Types	67
5.4.2	Values and Storages	68
5.5	The AIR Interface	68
5.5.1	Conversion to/from AIR XML	70
5.6	Transmutation	70
5.7	Planned: Parallelization Backends	73
5.8	Planned: Tracing, Single-Step, and Breakpoints	74
6	Implementation	75
7	Transmutators	77
7.1	FuncDefFilter	77
7.2	EMBA	78
7.3	RMulticoreBackend	79
7.4	Planned Transmutators	79
8	Experimental Evaluation	81
8.1	Test Environment	81
8.2	End-to-End Parallelization with EMBA and RMulticoreBackend	81
8.2.1	Setup	81
8.2.2	Execution Analysis	82
8.2.3	Results	84
9	Conclusion and Outlook	86

A	Installing ALCHEMY	91
A.1	Prerequisites	91
A.2	Installation	91
A.2.1	R Installation	91
A.2.2	ZeroMQ Java Binding	92
A.2.3	ALCHEMY Core Installation	92
B	Using ALCHEMY	93
B.1	ALCHEMY Configuration	93
C	Sample Core Configurations	94
C.1	Transmutation Controller Configuration	94
C.2	Type Environment Configuration	95
D	Example AIR XML Representation	97
E	Numerical Results of Evaluation Chapter	102
F	Description of AIR XML Representation	103
F.1	BinopExpr	103
F.2	BreakStmt	103
F.3	BuiltinFunc	103
F.4	ClosureExpr	103
F.5	ComponentExpr	104
F.6	ConstantExpr	104
F.7	ExprList	104
F.8	ForStmt	104
F.9	FuncCall	105
F.10	FuncDef	105
F.11	IfExpr	105
F.12	IteratorExpr	106
F.13	NextStmt	106
F.14	ParamExpr	106
F.15	Program	106
F.16	RepeatStmt	106
F.17	SkeletonExpr	107
F.18	SubscriptExpr	107
F.19	SymbolExpr	107
F.20	UnaryExpr	107
F.21	WhileStmt	108

List of Figures

2.1	ALCHEMY Overview	13
2.2	Startup of R interpreter with integrated ALCHEMY	13
2.3	Example of Complex Transmutation Configuration	16
2.4	Simple Configuration as Used in Example	17
2.5	Final output of the interactive R session	19
2.6	ALCHEMY collaborations for the introductory example	19
3.2	SEXP tree of $\sin(42)$	22
3.3	Illustration of REDUCE skeleton operation	24
3.1	SEXP tree	26
4.1	ALCHEMY Use Cases	27
4.2	Usecase Realization “UC 1: Execute R Program”	28
4.3	Usecase Realization “UC 2: Perform Interactive R Session”	29
4.4	usecase realization “UC 3: Evaluate R Expression”	30
4.5	AIR Class Hierarchy	31
4.6	Usecase Realization “UC 4: Convert R to AIR”	33
4.7	Example AIR tree	35
4.8	Usecase Realization “UC 5: Transmutate AIR”	36
4.9	Transmutator analysis class	37
4.10	Transmutation Configuration Example	38
4.11	Transmutation Configuration Example	39
4.12	Analysis Activity Diagram “Rule Processing”	40
4.13	Rule dependencies in transmutation configuration	41
4.14	Overall analysis class diagram	43
5.1	Overall logical design	48
5.2	Alchemy Deployment	49
5.3	Alchemy High-Level Collaboration Diagram	49
5.4	1st Part of Use Case 3 Realization	50
5.5	Call sequence in R with enabled ALCHEMY	52
5.6	(Design) Class hierarchy of SEXP visitors	54
5.7	(Pseudo-)sequence diagram of RtoAIRVisitor processing of $x \leftarrow 3.1$	54
5.8	Collaboration example for transmutation	57
5.9	Collaboration example for transmutation	58
5.10	R design classes participating in R - Alchemy integration	60
5.11	Example request/response cycle for the minimal ALCHEMY protocol	61
5.12	Classes involved in environment lookup	64
5.13	Execution sequence for a call to an AIRVector with proxy storage strategy	66
5.14	AIR Programs	67

5.15	Value class hierarchy	68
5.16	Call sequence for <code>query()</code>	69
5.17	Activities involved in transmutation	71
5.18	Important design classes related to transmutation	72
5.19	Design classes participating in transmutation configuration	72
5.20	Execution sequence for configuration evaluation	73
5.21	Possible backend integration using adapter transmutators	74
6.1	Physical <code>AlchemyCore</code> packages	76
7.1	EMBA Workflow	78
7.2	Example of an EMBA AIR modification	80
8.1	Execution times on a multicore machine	84
8.2	Parallel speedups for different numbers of cores for the measured problem instances	85

List of Tables

- 4.1 Meaning of Robustness Icons used in analysis class diagrams 28
- E.1 Numerical results of evaluation 102

1 Introduction

The R programming language is being widely used in statistics, machine learning, and bioinformatics. In these areas, it is common to work with large data sets, which renders processing speed an important factor. However, R being a sequentially interpreted language does not optimally exploit the speed-up that may be gained from modern, parallel hardware or cluster environments.

There are several extensions to the R language and its interpreter environment that allow programmers to explicitly distribute computations to parallel backends (see section 3.2). However, these packages are not easily applicable to the existing code base (e.g. CRAN) and require programmers to introduce code into their programs that may contain bugs and is irrelevant to the application domain.

A potential alternative is the automatic and transparent parallelization of sequential programs. This is known to be inherently difficult, but the R language provides characteristics that facilitate parallelization such as offering native vector data types or not offering reference types.

Goals of this Thesis

There are several techniques for parallelizing sequential programs that approach this problem from different directions (see [MAS05] for a brief overview of existing methods).

With this thesis, I want to present a framework, ALCHEMY, that allows researchers to experiment with the application of these techniques to R programs. At the same time, users from fields such as bioinformatics or statistics are given a tool that enables them to exploit existing parallel computing resources without the need to manually adapt their programs.

The framework enables its users to apply different parallelization analysis methods to R programs and execute these parallelized programs on suitable hardware.

R has been chosen as the base language of this thesis because it appears to be well-suited for automatic parallelization:

- R programs tend to live on a higher semantic level than e.g. C programs. Hence, it is easier to deduce programmers' intentions from the code.
- R programs have no pointers or references.
- R has built-in data-parallel functions operating on container types, e.g. `sin()` function on vectors, lists, and matrices.
- R programs tend to be short.

While transparently speeding up existing programs seems to be a goal sufficiently ambitious and worthwhile for itself, ALCHEMY shall also become a laboratory that makes it easy to learn what parallelism is and where it hides in scripting languages like

R. Ideas for parallelization may materialize when we look at a problem from different perspectives such as the problem domain itself, its algorithmic realization, its concrete mapping to a programming language, or the instruction scheduling on a concrete machine. ALCHEMY shall help getting better insight into where these perspectives differ, how they depend on the executing machine, and what parallelizations can be found where.

Results

A software system like ALCHEMY is never completely finished. However, with the submission of this thesis the following results can already be documented:

- *The intermediate language AIR has been created.* Although AIR must still prove its effectiveness as a language for analyzing programs, the creation of first parallelization modules has left the impression that the basic mechanisms might indeed be useful. AIR can flexibly express parallel programming patterns (skeletons) and provides a flexible method for querying program elements.
- *The Java-based software system ALCHEMY has been created.* ALCHEMY controls the analysis and transformation of AIR programs by delegating to specific software components, so called AIR Transmutators. The flexibility of the “transmutation” process, in that program analysis and transformation take place, has been a major design goal.
- *The R interpreter has been extended to transform R programs to AIR, interact with ALCHEMY, and transform AIR back to R programs.*
- *Three simple AIR Transmutators have been created.* These Transmutators process AIR programs or program fragments with different goals, e.g. finding “embarrassingly” obvious opportunities for parallelization.

2 Approach

This section presents a high-level overview of the operations of the ALCHEMY framework. Figure 2.1 shows the typical processing steps when ALCHEMY is used to perform parallelization analysis and parallel execution of an R program. The following sections describe these steps by help of a simple example program.

It should be noted that ALCHEMY does not strictly impose this workflow. On the contrary, ALCHEMYs configuration language allows for many different kinds of execution patterns, including e.g. loops between different analysis modules that may be useful for e.g. dynamic or semi-dynamic program analysis.

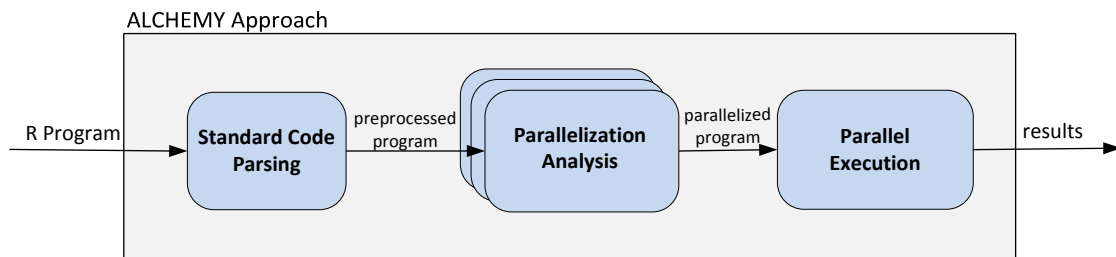


Figure 2.1: ALCHEMY Overview

2.1 Interfacing with the user

When the R environment is started up, the interpreter begins evaluating R language expressions that are either read from a user-provided program file or that are entered in an interactive console session. If R has been built with ALCHEMY support, R users can control ALCHEMY processing e.g. by executing ALCHEMY specific R commands. Figure 2.2 shows the screenshot of the startup of a typical interactive R session that uses ALCHEMY.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> alchemy.loglevel(3)
setting loglevel to 3
> alchemy.enable()
ALC: Trying to initialize Alchemy session with server tcp://localhost:1984
ALC: Initialization succeeded (session ID = 1227083285)
ALC: Registering service 'ValSvc.getVal'
ALC: Registering service 'EnvSvc.lookup'
ALC: Registering service 'AIRtoR.convert'
>
```

Figure 2.2: Startup of R interpreter with integrated ALCHEMY

After ALCHEMY has been activated, an R-ALCHEMY adapter intercepts the parsed R expression before the R interpreter has a chance to perform evaluation. The first task of the adapter is to transform the expression into a representation that can be processed by ALCHEMY.

2.2 Preprocessing the R Program for Parallelization Analysis

ALCHEMY uses specialized software components called “Transmutators” that are responsible for the transformation of R code. In order to simplify the access of those components to structural program information, ALCHEMY provides means to transform the R source program into an analogous Abstract Syntax Tree (AST) of a language called “AIR” (Analysis Intermediate Representation) whose design and realization is a result of this thesis.

AIR design has been inspired by other data-parallel intermediate languages such as VCODE (see [BC90]) in its explicit integration of primitives that represent Parallel Programming Patterns (skeletons). The concrete AIR implementation has the following goals:

- has vectors of arbitrary size as first-class data type
- can express a (non-trivial) subset of the R language
- contains an extensible set of language elements that represent parallel programming patterns (skeletons)
- can represent programs in different parallel computation models, e.g. PRAM, CSP, etc.
- has an interpreter or compiler that facilitates executing AIR programs on different backends
- provides means to annotate language constructs, e.g. resource requirements of a skeleton expression
- facilitates the easy replacement of parts of a program

Not all of these goals have been reached in the first version of AIR. In particular, the applicability for different computation models has not been a major design goal.

To facilitate the adaption of an AIR program to the needs of specific parallelization modules, ALCHEMY provides

- an interface that allows querying the AIR program and navigating through the set of result nodes.
- the possibility to back-transform result nodes (and subtrees) to a corresponding R expression.

Figure 2.1 shows the XML representation of the AIR program that corresponds to the R expression `sin(c(1,2,3))`, which computes the sine of the numbers 1 to 3.

After the R expression has been converted to AIR, ALCHEMY is able to perform parallelization analysis.

Listing 2.1: AIR Example

```

1 <AIR environment-proxy="tcp://127.0.0.1:1985"
2     value-proxy="tcp://127.0.0.1:1985" environment-id="
3     169965928">
4     <Program>
5         <FuncCall>
6             <funcexpr>
7                 <SymbolExpr name="sin"/>
8             </funcexpr>
9             <params>
10                <ParamExpr>
11                    <FuncCall>
12                        <funcexpr>
13                            <SymbolExpr name="c"/>
14                        </funcexpr>
15                        <params>
16                            <ParamExpr>
17                                <ConstantExpr type="real">
18                                    <RealValue data="1.0"/>
19                                </ConstantExpr>
20                            </ParamExpr>
21                            <ParamExpr>
22                                <ConstantExpr type="real">
23                                    <RealValue data="2.0"/>
24                                </ConstantExpr>
25                            </ParamExpr>
26                            <ParamExpr>
27                                <ConstantExpr type="real">
28                                    <RealValue data="3.0"/>
29                                </ConstantExpr>
30                            </ParamExpr>
31                        </params>
32                    </FuncCall>
33                </ParamExpr>
34            </params>
35        </FuncCall>
36    </Program>
37 </AIR>

```

2.3 Parallelization Analysis

As already stated above, in ALCHEMY parallelization analysis and all other AIR transformations are conducted by software components called “Transmutators”. ALCHEMY makes no assumptions about how these Transmutators operate on AIR programs. In the context of the TRANSPAR project, preparations have begun to realize various Transmutators such as

- *MATSU*, a parallelization technique that provides efficient parallelizations for certain kinds of dynamic programming problems for vector machines (see [KMM⁺05]),

- *SURE*, which aims at finding opportunities for parallelization in loops that access array data in specific ways based on [Dar97], or
- *EMBA*, a module for parallelizing apparent data-parallelity in R code.

It is an interesting question if and how a collaboration of multiple, different Transmutators can yield good results. *ALCHEMY* does not try to automatically find such collaborations but rather provides a suitable data abstraction and a flexible configuration language that allow researchers and regular R users to specify how single Transmutators should work together. Figure 2.3 shows a complex example of how multiple Transmutators might be composed to form a transformation graph.

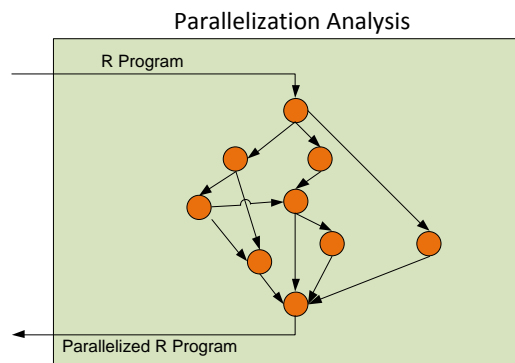


Figure 2.3: Example of Complex Transmutation Configuration

Figure 2.4 shows a simple Transmutation configuration that makes *ALCHEMY* sequentially operate on the following Transmutators

1. *FuncDefFilter*: filters out programs that consist only of a function definition
2. *EMBA*: finds “embarassingly parallel” expressions and converts them to the data-parallel “MAP” skeleton
3. *RMulticoreBackend*: identifies occurrences of the MAP skeleton¹ and transforms them to calls to the R “multicore” library (see section 3.2)

Among these three Transmutators, *EMBA* is the only one that fully deserves to be called a “parallelization analyzer” with *FuncDefFilter* acting like an input validator and *RMulticoreBackend* being a (pseudo) parallel backend adapter. This may serve as an indication that the “Transmutator” concept allows the realization of very different usage scenarios.

The *EMBA* Transmutator, however, parallelizes its input in the following way. It checks whether its input contains special cases of “embarassingly parallel” program elements, i.e. elements that are parallelizable “by construction”. For instance, in R, many built-in and 3rd party functions accept arguments of type “vector” or “list” and perform operations on every element of these compound values without depending on other value elements. The *EMBA* transmutator replaces every occurrence of such a function (for a predefined subset of functions) by an instance of the *MAP* skeleton.

¹In future releases of this Transmutator, other skeletons will be included.

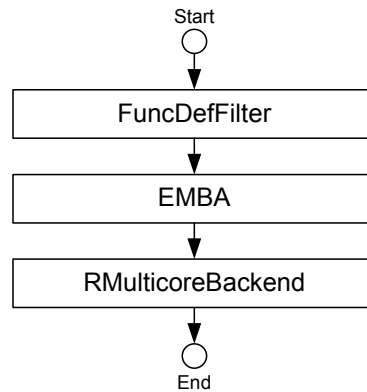


Figure 2.4: Simple Configuration as Used in Example

After these AIR modifications have been finished, EMBA output is passed on to the *RMulticoreBackend* Transmutator.

Figure 2.2 shows the output AIR of EMBA after processing example 2.1. For better readability, ALCHEMY allows the conversion of AIR into R or “pseudo R”². For instance, the corresponding conversion of program 2.2 yields

```
alchemy.applySkeleton(name = "MAP", collection = c(1,2,3), kernel = sin)
```

Listing 2.2: AIR Example

```

1 <AIR environment-proxy="tcp://127.0.0.1:1985"
2   value-proxy="tcp://127.0.0.1:1985" environment-id="
3     169965928">
4   <Program>
5     <SkeletonExpr name="MAP">
6       <params>
7         <param name="collection">
8           <AIRVector basetype="real">
9             <Data data="1.0,2.0,3.0" length="3" />
10            </AIRVector>
11          </param>
12          <param name="kernel">
13            <SymbolExpr name="sin"/>
14          </param>
15        </params>
16      </SkeletonExpr>
17    </Program>
18  </AIR>
  
```

²The “pseudo” does not mean that the generated R is syntactically invalid but that ALCHEMY may have introduced function names that are unknown to R

2.4 Parallel Execution

The most “radical” types of transmutation are the ones that replace a program entirely or partially by corresponding computation results. Those Transmutators are called “Executors” within ALCHEMY. Executors are often adapters to concrete (parallel) computations backends using e.g. OpenMP ([[Ope](#)]) or MPI.

As AIR programs may contain parallel programming skeletons, i.e. language elements that bear information about parallelizability without referring to the specifics of a concrete machine, Transmutators may often be able to directly transform these skeletons to backend code.

RMulticoreBackend is an example of an Executor. It

1. identifies certain skeleton expressions in the AIR,
2. transforms them to R code that employs the R “multicore” library (see [3.2](#)),
3. submits that code to an instance of the R interpreter that has the “multicore” library installed,
4. and replaces the original skeleton expressions in the AIR program with the results of the backend computation.

It should be noted that in the current release of ALCHEMY, *RMulticoreBackend* does *not* use a specific R instance for computation but effectively only executes steps 1. and 2. By doing this, the transmutation result that is returned back from ALCHEMY to R contains the R multicore code that must eventually be interpreted by the client R instance.

The following listing shows the R code that *RMulticoreBackend* creates for the *EMBA* output from the previous section ([2.2](#)):

Listing 2.3: R “multicore” Code

```

1 {
2   library(multicore)
3   mclapply(c(1, 2, 3), FUN = sin)
4 }
```

The `mclapply` function is part of the R “multicore” library. It parallelizes the evaluation of a function on a compound data structure, i.e. vector or a list, by splitting the data and distributing the computation to the available CPU cores.

If *RMulticoreBackend* used a specific “backend R”, the resulting AIR output of the Transmutator would be as follows:

Listing 2.4: Final AIR program

```

1 <AIR environment -proxy="tcp://127.0.0.1:1985"
2   value-proxy="tcp://127.0.0.1:1985" environment-id="
3     169965928">
4   <Program>
5     <ConstantExpr>
6       <AIRVector basetype="real">
```

```

6           <Data data="0.8414710,0.9092974,0.1411200" length
           = "3"/>
7           </AIRVector>
8           </ConstantExpr>
9           </Program>
10 </AIR>

```

After that, ALCHEMY decides that Transmutation has been finished and returns the overall result AIR to the client R instance, which outputs the result in the interactive R session:

```

> sin(c(1,2,3))
[1] 0.8414710 0.9092974 0.1411200
>

```

Figure 2.5: Final output of the interactive R session

Figure 2.6 illustrates the communication relationships between high-level ALCHEMY components of the previous example. The individual elements of the diagram are explained in section 5.

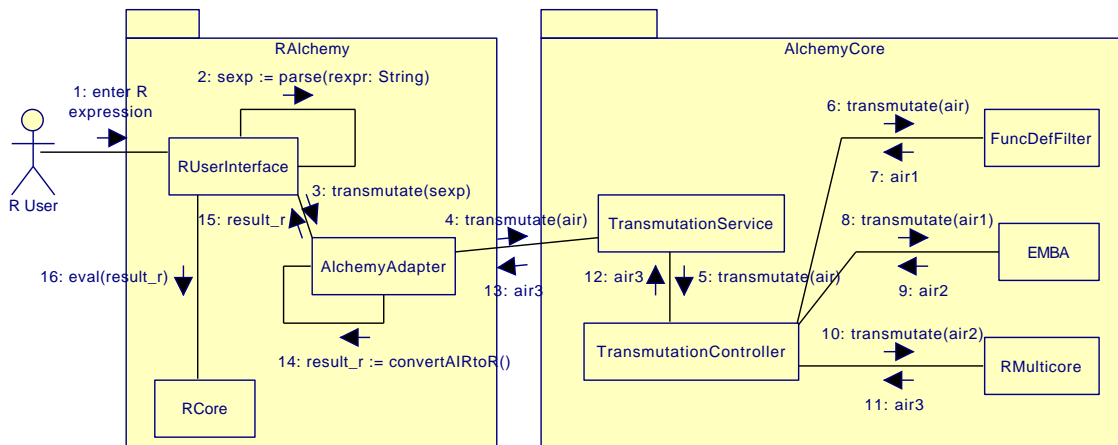


Figure 2.6: ALCHEMY collaborations for the introductory example

2.5 Scope of this Thesis

The semantics of an interpreted language such as R is eventually defined by the execution of its interpreter. The ability to isomorphically map the entire language onto another language, e.g. AIR, without losing or altering semantics would require to mimic or respect interpreter behavior in all detail. AIR implements certain “mismatch” strategies for handling those cases, when an R language construct cannot be properly mapped to AIR.

Additionally, R is a semantically very rich language with more than 20 basic language types called “S expressions”. Many of these types are never exposed to R users but used internally, e.g. bytecode types, weak references, etc. Currently, ALCHEMY shows no

defined (or at least useful) behavior when explicitly be confronted with some of these types.

3 Related Work

3.1 R

The R project [R Db] provides a programming environment for numeric computing. R consists of an interpreter core, an extensible set of functional libraries, and mechanisms to visualize data. It is a free implementation of the S language which has wide spread success among researchers for more than 30 years.

By default, users interact with the R environment via a Read-Eval-Print-Loop (REPL) interface that repeatedly performs the following steps:

1. The user enters an R language expression.
2. The interpreter checks
 - if the expression is not a valid R construct, in which case the interpreter prompts an error message, and returns to step 1,
 - if the expression is a valid, but incomplete R construct, in which case the interpreter shows a continuation prompt, and returns to step 1,
 - if the expression is a valid R construct, in which case the interpreter proceeds with step 3.
3. The interpreter evaluates the entered expression. This evaluation might create side-effects like I/O. An important side effect is the modification of the “R environment”, i.e. the global symbol table that is used for interpretation.
4. The evaluation result is printed to the screen.

Additionally, R offers a batch interface that allows users to provide a series of R expressions in a text file.

3.1.1 S Expressions

Parsed R programs and values are internally represented as *S expressions* (called SEXP within R), a special type of Abstract Syntax Tree (AST). Among the 24 different node types that are possible in an SEXP tree, the following ones are particularly important for the understanding of ALCHEMY (see [R Da] for a detailed description of all SEXP types):

SYMSXP Represents “symbols” in R, i.e. primarily function and variable names. The class of “function names” in R does also encompass operators like [, :, or A SYMSXP is associated with a CHARSEX that represents a character string.

LANGSEX Represents functions (named and anonymous ones) in R. A LANGSEX is usually associated with a SYMSXP denoting the name of the function to be called and a list of other SEXPS representing the function parameters.

CHARSXP Represents a character string.

LGLSXP, INTSXP, REALSXP Represent vectors of boolean, integer, or real values, respectively.

CLOSXP Represents a “closure”, i.e. an anonymous function with a persistent private environment.

The evaluation of an R expression can be regarded as a transformation of an SEXPR where parts of the tree are recursively replaced by its evaluation results until there are no further evaluations possible. Figure 3.2 shows a representation of the SEXPR that is created by the R parser for the language expression `sin(42)`.

Figure 3.1 shows the SEXP graph that corresponds to the following program:

```

1 u <- function(a,b,c) {
2     return (sin(a) + cos (b+c) )
3 }
```

Syntax and semantics of the R language are well described in [VR00] and [Adl10]. [R Da] gives a good introduction into how the R interpreter internally works.

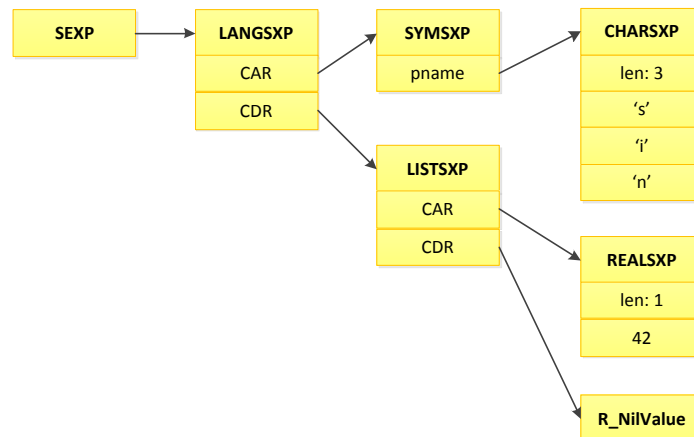


Figure 3.2: SEXP tree of `sin(42)`

3.2 Approaches to Parallelizing R

There are several packages available that enable R users to benefit from parallel hardware and cluster environments. [SME⁺09] provides an overview over the different approaches that are currently used.

In brief, existing techniques can be coarsely assigned to two major categories: those that try to provide means to exploit the inherent data parallelity in R data types and those that adapt to third-party clustering environments like MPI or PVM. Virtually all packages provide special functions that must be explicitly used by R users to parallelize their programs. There are only a few approaches that try to parallelize user programs in a transparent and automatic way.

3.2.1 Exploit Inherent Data-Parallelism in R

This class of techniques can more or less accurately be characterized as the “papply” solution class, as its members provide replacements for the R `apply()` function family. In R, the `apply()` function family takes different R data types, such as `list` or `vector` and a unary function object as input and applies that function object to all elements of the given data structure, yielding a data structure that contains the results of these function applications.

This task belongs to the class of “embarrassingly parallel” problems and can be parallelized without taking data dependencies into consideration. Hence, most of these parallelization packages are intended to be simple wrappers around parallel execution frameworks like MPI or PVM. In order to make use of those libraries, users must modify their programs.

3.2.2 Exploit Task-Parallelism Hidden in Sequential R Programs

Currently, I know only of one project that tries to tackle the problem of automatically finding inherent concurrency. The `pR` package “parallelizes sequential R code without requiring any source code modification” [MLS07]. It works as follows:

1. automatically R statements are selected that shall be parallelized (also in simple `for` statements),
2. “Tomasulo’s algorithm” [Wike] is applied to find program instructions that have no prohibited data-dependencies and may be parallelized safely,
3. parallel tasks are distributed via MPI

Additionally, `pR` provides various wrapper and replacement functions for ordinary R functions.

Unfortunately, the “full” `pR` package is not publicly available for testing but only the somewhat less sophisticated `taskpR` package that requires users to manually annotate the R source code with parallelization hints.

3.3 Parallel Programming Patterns (Skeletons)

The term “skeleton” was coined 1989 by Murray Cole ([Col89]). Skeletons are abstracted solutions to commonly occurring problems in parallel programming. Analogously to the more widely known (general) *Design Patterns* (see [GHJV94]) in software engineering, skeletons try to help with following goals:

- have a common vocabulary that simplifies communication about parallel programming problems with other developers
- hide the complexities of a concrete solution behind a common concept or idea of the solution
- make a solution to a common parallelization problem reusable by providing parametrizable skeleton libraries

While the terms “skeleton” and “parallel programming pattern” are often used synonymously, there exist parallel programming patterns that live on a more conceptual level than the usual programming- or design-level skeletons. [MSM04] provides a detailed categorization of parallel programming patterns

At present, there are more than 20 skeleton libraries or frameworks available that are implemented in different programming languages and provide different sets of skeletons. The following list shows some examples of skeletons that can usually be found in those frameworks:

MAP Applies a computation to every element of a container keeping the structural properties of the container.

REDUCE Combine all elements of a container using an binary-associative operator. Figure 3.3 illustrates a specific approach to implementing REDUCE that minimizes the data dependencies of single operations.

SCAN Iteratively combine the elements of a linear container from start to end, storing the intermediate results to an array.

PIPE Subdivide a computation into linear, dependent, and elementary steps that, for independent input, can work independently.

See [Wika] or [Col] for a more comprehensive list of skeletons and skeleton libraries.

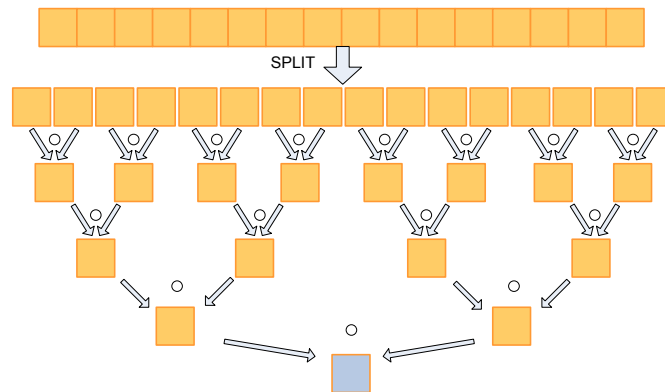


Figure 3.3: Illustration of REDUCE skeleton operation

3.4 Parallel Intermediate Representations

[MHD09] has come to the conclusion that there are currently only a few programming languages that are suitable as intermediate languages and offer more than basic elements to express concurrency. These languages are often very specific with regards to their model of concurrency.

VCODE [BC90] is an intermediate language specifically devised for the requirements of highly data-parallel problems in environments that allow for the efficient computation of vector operations. VCODE is the basis of the data-parallel NESL language [Ble95] that, in turn, has prepared the ground for recent developments in parallel computing such as Intel Ct [Intb] or Intel Array Building Blocks (ARBB) [Inta].

3.5 Approaches to Automatic Parallelization

Modern CPUs usually employ techniques that analyze the interdependence of consecutive elements of the instruction stream in a program. These techniques make it possible to dynamically schedule instructions to processing pipelines in a way that reduces the number of wait states and avoids dependency conflicts (data hazards). Two well known mechanisms for this kind of parallelization are e.g. “Scoreboarding” [Wikb] and “Tomasulo’s algorithm” [Wike]. Tomasulo’s algorithm is also used with the `pR` package (see section 3.2).

On a compiler level, static mechanisms to identify candidates for parallelization often aim at finding the dependencies of scalar variables (e.g. [Dar97]) or identifying parts of arrays that may be processed in parallel. There are also other approaches such as analyzing the commutativity of code sections (e.g. [DD97]).

More high-level techniques try to find algorithmic or domain specific evidence for parallelism. For instance, [KMM⁺05] is able to recognize specific kinds of dynamic-programming problems and transform them into a form that can be efficiently computed on vector-machines, [KBR07] is able to parallelize certain kinds of stencil computations.

[CWK93] presents a more general approach that applies pattern matching to a representation of C programs. The patterns that are used form an extensible parallelization knowledge base and are not restricted with regards to a single parallelization idea.

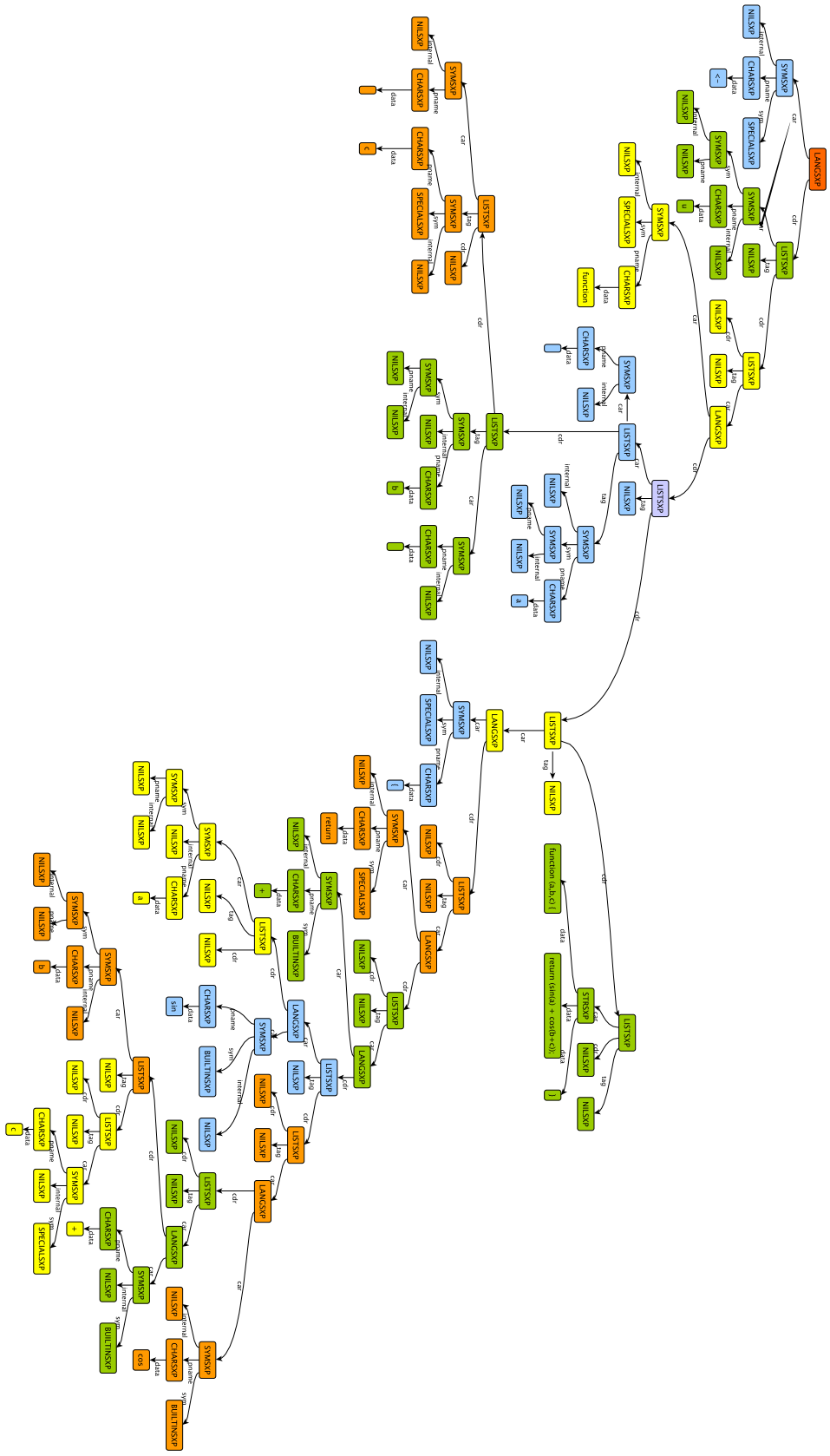


Figure 3.1: SFXP tree

4 Requirements Analysis

Chapter 2 has presented a high level illustration of ALCHEMY's basic workflow, which implies several requirements of the ALCHEMY software architecture. In this chapter, requirements are refined by detailing the architecturally relevant use-cases by help of an object-oriented analysis.

Figure 4.1 summarizes what use cases are realized by the ALCHEMY laboratory.

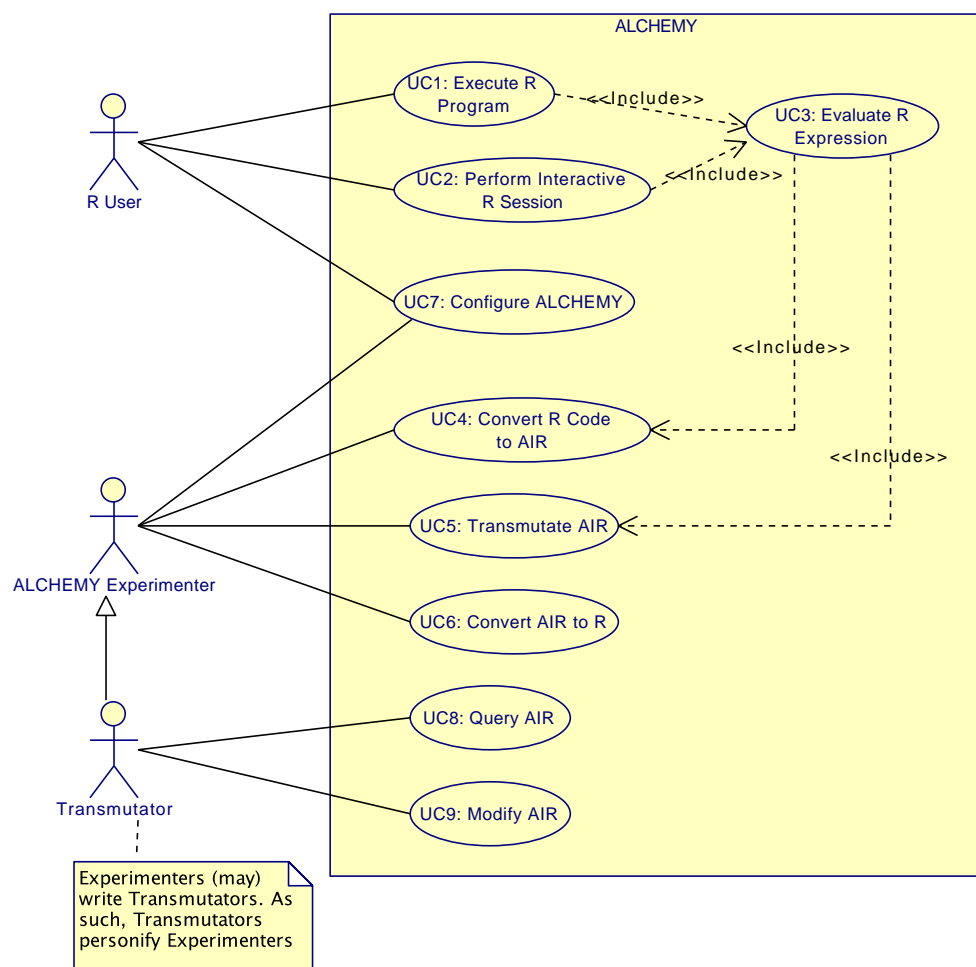





Figure 4.1: ALCHEMY Use Cases

The following use cases are described in a casual way. For every use case, a corresponding analysis-level use case realization has been worked out along with its participating analysis classes. Analysis classes are shown with “Robustness icons”, see [JBR99].

Table 4.1: Meaning of Robustness Icons used in analysis class diagrams

Control class		Represents a class that coordinates or controls other objects.
Entity class		Represents information that is long-lived, often persistent, and conceptually important for the system.
Boundary class		Represents an actor interface to the system.

4.1 UC 1: Execute R Program

An R user starts an ALCHEMY-enabled R interpreter in batch mode providing a text file containing an R program. The user may specify additional ALCHEMY-related command-line parameters that influence the way ALCHEMY is executed (logging level, debugging behavior, etc). The interpreter shall also accept all parameters that are recognized by a regular, i.e. ALCHEMY-unaware, R interpreter (as for release 2.13.1).

The interpreter executes the given R program in a way that is indistinguishable from a regular R interpreter execution given the same input, i.e. input and output is the same and arrives in the same order.

This use case includes *UC 3: Evaluate R Expression* for any full R expression that is included in the input file.

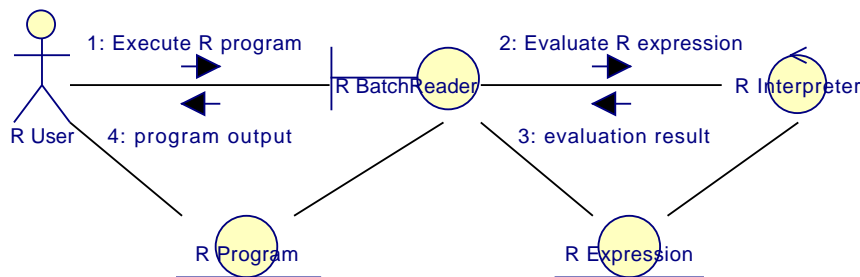


Figure 4.2: Usecase Realization “UC 1: Execute R Program”

The following analysis classes participate in the realization of this use case:

R BatchReader Reads the text from a user supplied file, splits it to individual (full) R expressions, calls sequentially the R Interpreter (see *UC 3*). After the last expression has been evaluated, “R BatchReader” outputs the final result to the user.

R Program Text file that contains a sequence of R expressions.

R Interpreter Controls overall R evaluation. Uses the ALCHEMY Transmutation Controller to parallelize/execute its input.

R Expression A valid element of the R language that may be parsed and evaluated by an R interpreter.

4.2 UC 2: Perform Interactive R Session

An R user may execute an ALCHEMY-enabled R console that is indistinguishable from an interactive R session. The session is an example of a REPL, i.e. a “Read-Eval-Print Loop”. In an ALCHEMY-R REPL the following sequence is repeated until the interpreter terminates:

1. “Read”: The user enters an R expression that is parsed by the interpreter. If the expression contains an error or is incomplete, the console prints an error message or prompts for continuation of the expression, respectively. The user may also enter commands that control the behavior of ALCHEMY.
2. “Eval”: If the expression could be parsed correctly, use case *UC 3: Evaluate R Expression* is triggered.
3. “Print”: After *UC 3: Evaluate R Expression* has finished either an error message, or, if evaluation has failed, the evaluation result is printed to the console.

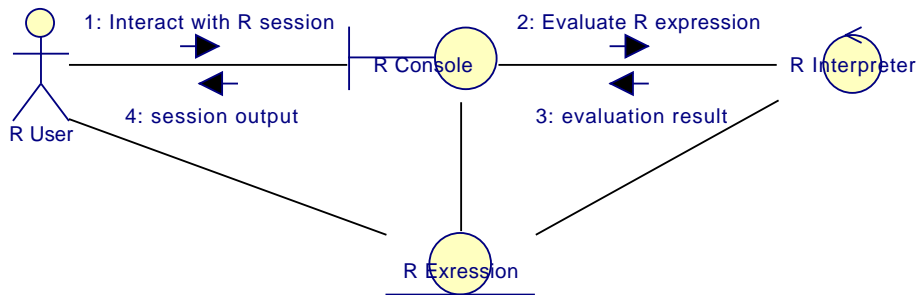


Figure 4.3: Usecase Realization “UC 2: Perform Interactive R Session”

The following new analysis classes participate in the realization of this use case:

R Console Reads user input from standard input and prints user directed output. Calls the *R Interpreter* whenever the user has finished a line (has pressed Enter), waits for the interpretation results and outputs them to the user. If the *R Interpreter* signals that the input was incomplete, a continuation prompt is printed and the input of the following line is appended to that of the former line. The look-and-feel of the R console should be indistinguishable from that of the original R console except when using special ALCHEMY features such as debugging. The externally observable behavior that must be implemented by the R Console is described in [Adl10].

4.3 UC 3: Evaluate R Expression

This use case is triggered whenever a full expression from an interactive session or batch processing shall be evaluated. The evaluation result is an arbitrary R expression [Adl10].

Figure 4.4 shows a use case realization for the first case, when a user has entered a full R expression at the R console and R evaluation is started. Evaluation for batch processing is no different from this case.

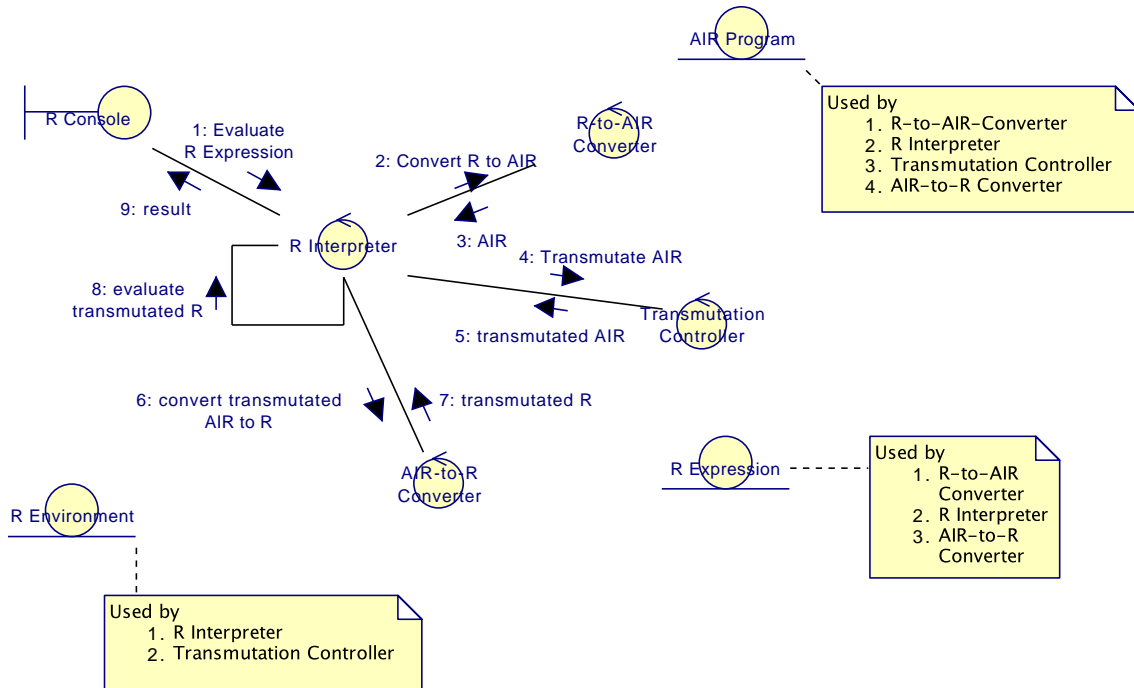


Figure 4.4: usecase realization “UC 3: Evaluate R Expression”

In UC 3 the following steps are performed:

1. After the *R Interpreter* has received the *R Expression* from the *R Console*, it uses the *R-to-AIR Converter* to generate an *AIR Program* from the given *R expression* and the current *R Environment* (see *UC 4: Convert R Code to AIR*).
2. The *R Interpreter* now sends the resulting *AIR expression* to the *Transmutation Controller*, which manages all *AIR* processing in *ALCHEMY* (see *UC 5: Transmute AIR* for details).
3. After the *R Interpreter* receives the fully transmutated *AIR Expression*, it converts it back to an *R expression*. (see *UC 7: Convert AIR to R*).
4. With a regular *R Expression* at hand, the *R Interpreter* is now able to perform normal *R* evaluation, which includes e.g. computing the *R* functions of the “base package”, an *R* library that contains hundreds of mostly mathematical functions.
5. After evaluation has finished, the result is returned to the *R Console*.

The following new analysis classes are participating in this use case:

R Environment An assignment of *R* language symbols, i.e. strings with optional namespaces, to *R Expressions*

AIR Expression see 4.3.1

AIR Environment Associative hash mapping symbols, i.e. strings, to *AIR Expressions*

AIR Program Pair of:

- *AIR Expression*: the root element, i.e. expression, of the given program that may recursively contain other *AIR Expressions*.
- *AIR Environment*: interpreter symbol table at the beginning of program execution

R-to-AIR Converter Converts an *R Expression* to an equivalent *AIR Expression*.

AIR-to-R Converter Converts an *AIR Expression* to an equivalent *R Expression*.

4.3.1 AIR Expression

An *AIR Expression* represents a valid expression of the AIR (Analysis Intermediate Representation) language that, in turn, represents an R language expression. See section 2 for a high-level description of AIR properties.

The AIR language elements are shown as subclasses of the analysis class `AIRExpr` in diagram 4.5.

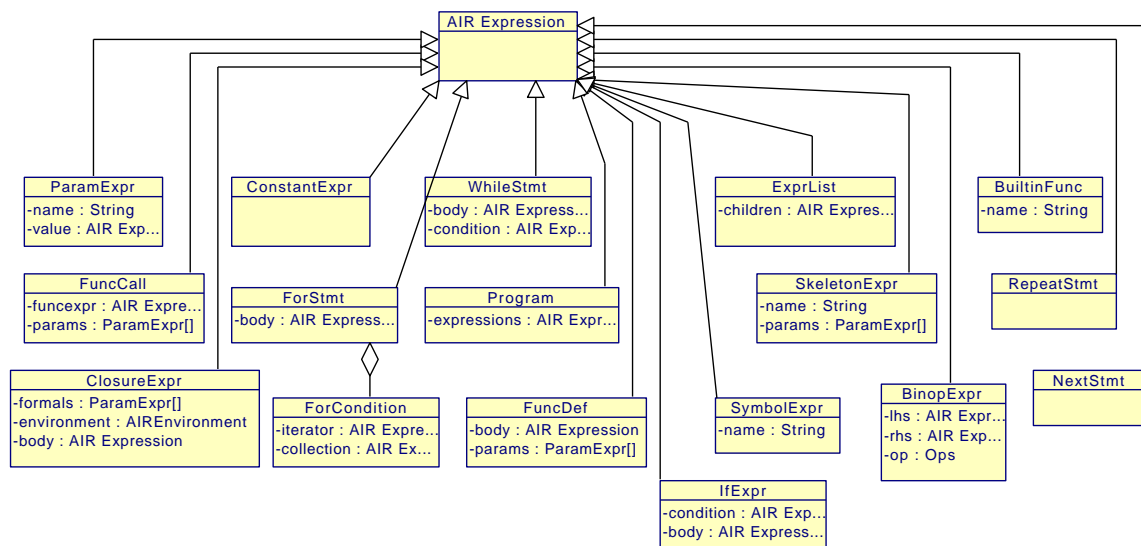


Figure 4.5: AIR Class Hierarchy

AIR consists of the following language elements, i.e. AIR expressions,:

BinopExpr A binary operation with

- an operator from a predefined set of operations, such as “plus”, “minus”, etc
- a left-hand side operand and a right-hand side operand that both must be AIR expressions

BreakStmt If used within a loop construction, such as `ForStmt` or `WhileStmt`, an evaluation of the `BreakStmt` shall lead to the immediate abortion of loop processing.

BuiltinFunc An expression whose semantics is unknown to `ALCHEMY` and that must not be interpreted

ClosureExpr An “anonymous function” that bears a body to be executed and an environment that shall be used for execution.

ComponentExpr Access to components of data frame or individual “named” list items

ConstantExpr A language constant, such as a string or an integer value. For a more detailed discussion about ALCHEMY’s value types, see [4.3.1.1](#).

ExprList A list of AIR expressions as represented by e.g. an R code block.

ForStmt An R alike “for” statement that consists of

- a collection that is iterated on
- a body that is executed for every element of the collection

FuncCall A function call that consists of

- a function expression, i.e. either a closure (**ClosureExpr**) for a call to an anonymous function or a (**SymbolExpr**) for a call to a named function
- a mapping to parameter names to parameter values that are AIR language expressions
- a function body

FuncDef A function definition with

- a list of formal parameters that may have default values
- a function body

IfExpr An “if” expression, representing the conditional execution of a body element. Optionally, the **IfExpr** may contain an alternate “else” expression.

NextStmt If used within a loop construction, such as **ForStmt** or **WhileStmt**, an evaluation of the **NextStmt** shall lead to the immediate execution of the next loop iteration ignoring the remainder of the loop body.

ParamExpr Represents a parameter, e.g., of a function call. Consist of the parameter name and the parameter value, which may be any AIR expression.

Program List of AIR expressions

SkeletonExpr Represents a “skeleton”, i.e. a parallel programming pattern (see [3.3](#)). Skeletons are modeled as functions with named parameters. A **SkeletonExpr** has a name, e.g., “MAP” and a list of **ParamExpr** constituting the skeleton parameters.

SymbolExpr Represents a language symbol, i.e. a variable or function name.

WhileExpr A “while” loop construct with a condition expression and a loop body that is repeatedly executed as long as the condition is evaluated true.

4.3.1.1 AIR Types and Values

AIR provides values of the following types, which may appear as the contents of variables and language literals:

IntegerValue, RealValue, LogicValue, StringValue One of element of “integer”, “real”, “logic”, or “string”, respectively.

AIRVector Contains an arbitrary number of homogeneous elements of a BaseType.

AIRMatrix Contains an arbitrary number of elements of a BaseType. Contains additionally information about the dimensionality of the matrix.¹

AIRList Contains an arbitrary number of heterogenous, optionally named elements.²

4.3.2 XML Representation

A conceptual *AIR Expression* has a concrete “canonical” representation as an XML document. This XML representation is part of the ALCHEMY domain model, as Use Case 8 (see 4.8) depends on it. Section 2 shows some examples of AIR XML documents. See also appendix D.2 for a more complex XML representation for program listing D.1.

4.4 UC 4: Convert R Code to AIR

In this use case, which is triggered by the R Interpreter, the *R-to-AIR Converter* creates an *AIR Expression* from an *R Expression*. AIR is designed to be able to express a large subset of R, so the output of this conversion results in an AIR object that has very similar semantics to that of the original expression.

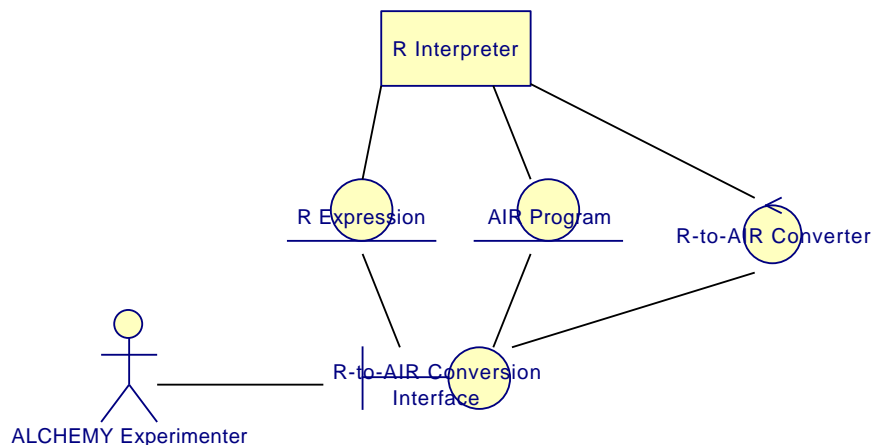


Figure 4.6: Usecase Realization “UC 4: Convert R to AIR”

The conversion is realized by mapping R language elements to AIR language elements. The mapping rules can be deduced from the element descriptions in the previous section.

¹AIRMatrix is not fully implemented, yet.

²AIRList is not fully implemented, yet.

Figure 4.7 represents an *AIR Program* that has been converted from the R program shown in listing D.1³.

Listing 4.1: Example R Program

```

1 a <- 3
2 while (a < 10) {
3     x <- c(3,1,4,1,5);
4     for (i in x) {
5         a <- a + i;
6     }
7     for (j in x) {
8         a <- a * j;
9     }
10 }
```

4.5 UC 5: Transmutate AIR

Possible actors for this use case are the *R Interpreter* that aims at transmutating an *AIR expression* or an *ALCHEMY experimenter* that uses a specific interface for performing transmutations.

Transmutation consists of the following steps:

1. *R Interpreter* sends an *AIR Expression* to the *Transmutation Controller* for transmutation.
2. *Transmutation Controller* evaluates its *Transmutation Configuration* that may, depending on its *Configuration Rules* and the state of the *Transmutators*, modify the state of the *Transmutation Queue*.
3. *Transmutation Controller* takes the first *Transmutator* out of the *Transmutation Queue* and delegates transmutation of an *AIR Set* (initially containing the input *AIR Expression*) to it.
4. *Transmutator* analyzes the elements of the *AIR Set* and decides if and where to perform modifications.
5. *Transmutator* replaces parts of the *AIR Expression*.
6. After a *Transmutator* has finished transmutation, the *Transmutation Controller* checks if the termination condition is fulfilled and if so, finishes the transmutation, handing control back to the *R Interpreter*.
7. Otherwise, if the termination condition has not been reached, yet, the *Transmutation Controller* repeats this procedure starting from evaluating its configuration.

³The meaning of the highlighted subtree of diagram 4.7 is described in section 4.8

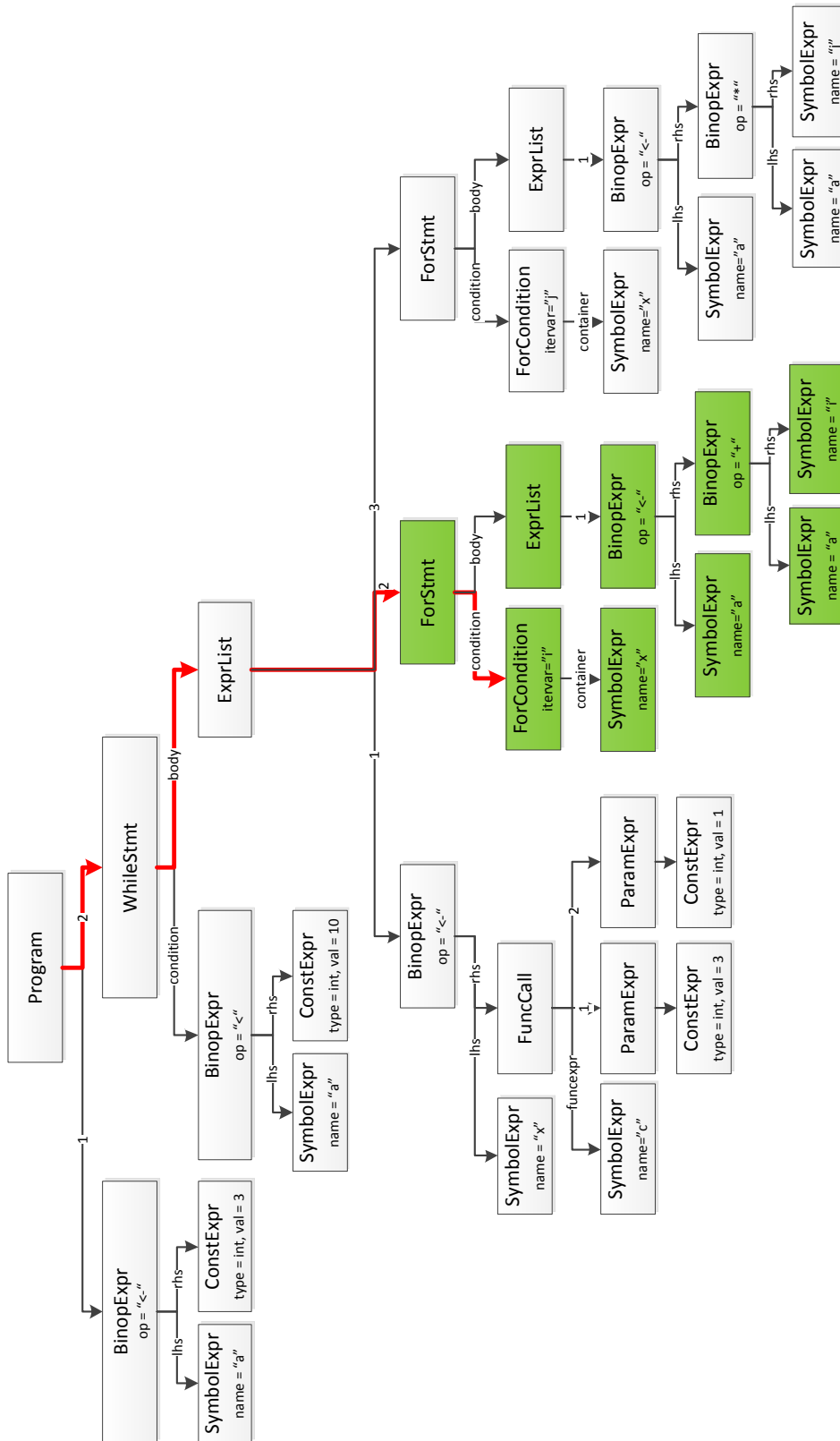


Figure 4.7: Example AIR tree

The capability to set breakpoints, do debugging during transmutation, and send trace messages to the R session is a future extension that must not necessarily be included in the first ALCHEMY release, but should be considered in its architecture.

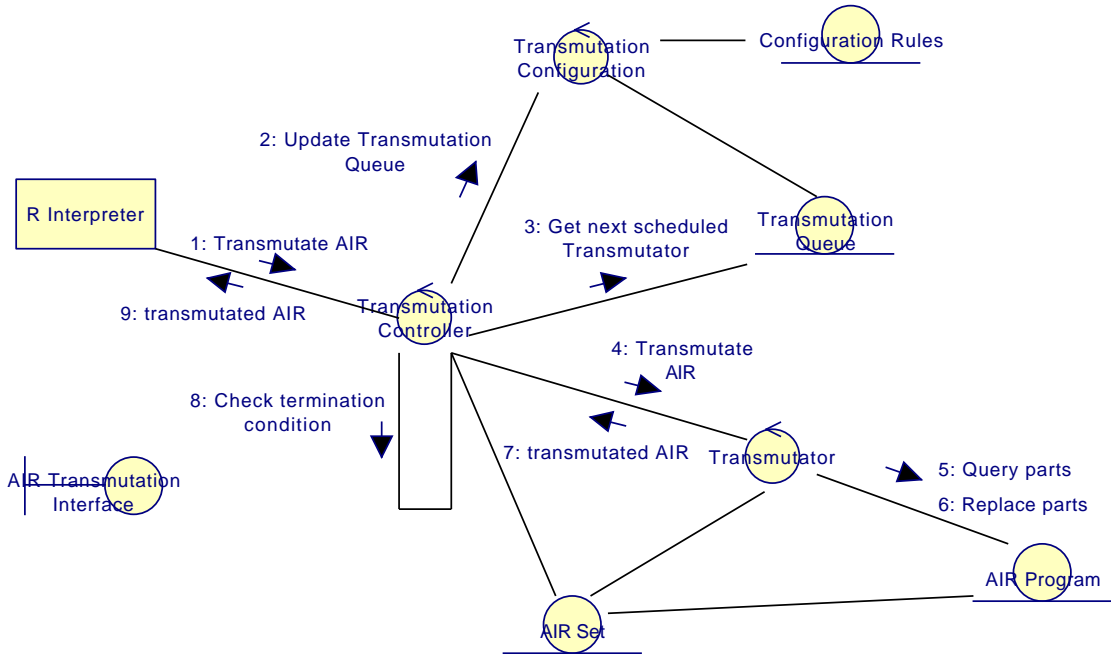


Figure 4.8: Usecase Realization “UC 5: Transmutate AIR”

The following new analysis classes participate in the realization of this use case:

AIR Set A set or list of *AIR Programs*

Transmutation Controller See 4.5.2

Transmutation Configuration See 4.5.2

4.5.1 Transmutator

An *AIR Transmutator* takes an *AIR Set* as input, processes its enclosed *AIR Programs*, and returns the resulting *AIR Set*. Transmutators may serve different purposes, e.g.

- Act as a *filter* to identify *AIR Programs* that should not be processed by ALCHEMY.
- *Transform* an *AIR Program*. This is, how Transmutators are used most often.
- Simplify an *AIR Program* by computing the value of the entire program or fragments thereof.

Each ALCHEMY transmutator has one *Input Port*. After a transmutator has been scheduled, its *Input Port* contains the *AIR Set*, i.e. the set of *AIR programs*, that it should process. Likewise, each ALCHEMY transmutator has one *Output Port*. After the transmutation has been finished, the result is put into the Transmutator’s *Output Port*. Figure 4.9 shows the structure of the transmutator analysis class.

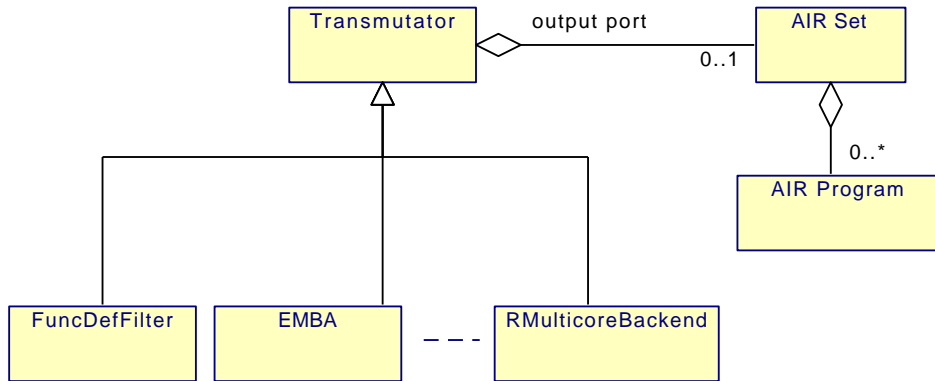


Figure 4.9: Transmutator analysis class

Transmutator execution is scheduled by the *Transmutation Controller* (see 4.5.2).

At present, ALCHEMY provides the following *Transmutators* (see chapter 7 for a detailed description): FuncDefFilter, EMBA, RMulticoreBackend.

4.5.2 Transmutation Controller

As described in section 4.5.1, a *Transmutator* transforms an input *AIR Set*, i.e. a list of *AIR Programs* into an output *AIR Set*. Thus, as input and output are of the same type, it is possible to apply a “Pipes and Filters” pattern [BMR⁺96] to execute multiple *Transmutators* in sequence.

It is the responsibility of the *Transmutation Controller* to decide what this sequence looks like and when it terminates. The combination of a *Transmutator* and that input *AIR Set*, which it shall process, is called a *Transmutation*. The *Transmutation Controller* delegates the decision what *Transmutation* to schedule next to a *Transmutation Configuration*, which, following some strategy, modifies the so called *Transmutation Queue*.

The *Transmutation Controller* simply relies on the order of *Transmutations* in this *Transmutation Queue* to decide what *Transmutator* to call next using what input. Transmutation terminates when the *Transmutation Queue* is empty after the *Transmutation Configuration* has been evaluated. When transmutation terminates, the transmutation result is the first element of the resulting *AIR Set* of the last *Transmutation* that has taken place.

There are many *Transmutation Configurations* conceivable, e.g. ones that strictly hard-code the order of *Transmutations* or ones that dynamically or even randomly choose the next *Transmutation*. By default, ALCHEMY uses a configuration based *Transmutation Configuration* that is described in more detail in the following section.

As the *Transmutation Controller* transforms an input *AIR Set* into an output *AIR Set*, it can be regarded a *Transmutator* itself. Hence it might be (recursively) used in higher-level transmutation strategies, making complex scenarios with different, nested, and isolated *Transmutation Configurations* possible.

Figure 4.10 shows an imaginary complex configuration that is built from a main configuration with focus on parallelization analysis and a sub-configuration with focus on AIR execution.

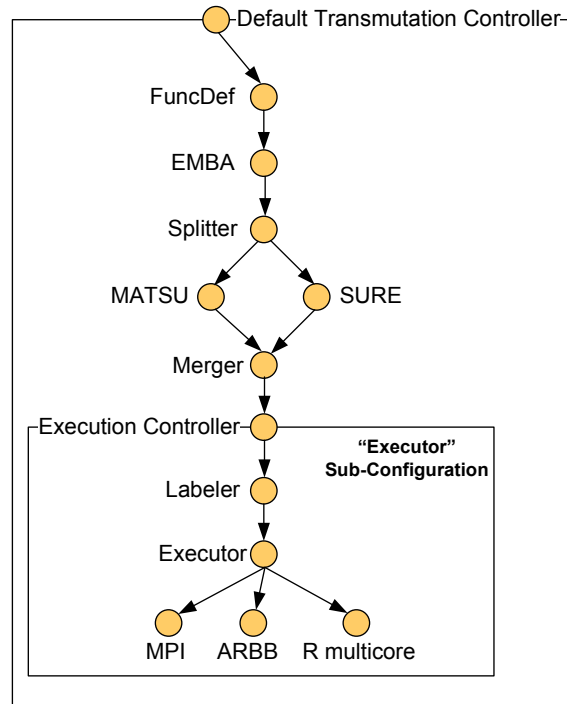


Figure 4.10: Transmutation Configuration Example

In the following sections, I describe a rule-based Transmutation Configuration.

4.5.3 Rule-Based Transmutation Configuration

The Rule-based Transmutation Configuration is composed of a list of *Rules*. Every rule consists of

- a *Condition* part that defines when a rule “matches”
- an *Input* part that forms the Transmutation input
- an *Action* part that is used to, e.g. schedule a Transmutation, i.e. put a Transmutation onto the *Transmutation Queue*. There are also other Actions possible.

Additionally, the Transmutation Configuration defines a set of Transmutator instances that can be used in rule specifications.

Figure 4.11 shows a schematic overview of an Transmutation Configuration instance.

When the configuration is evaluated, the conditions of all rules are evaluated and the actions of matching rules put on the *Transmutation Queue*. Rules are processed in order of occurrence in the rules list, so their order has an impact on the action schedule.

An important rule is that *one Transmutator may consume one input AIR Set only exactly once*. This is necessary because the Transmutation Configuration is evaluated anew after each round and multiple repeated “firings” of rule instances is prohibitive. Without this rule, a repeated scheduling of rules would happen naturally, as *Transmutator* output ports are not automatically cleared of *AIR Sets* to make deferred *Transmutator* execution possible.

Transmutation Configuration

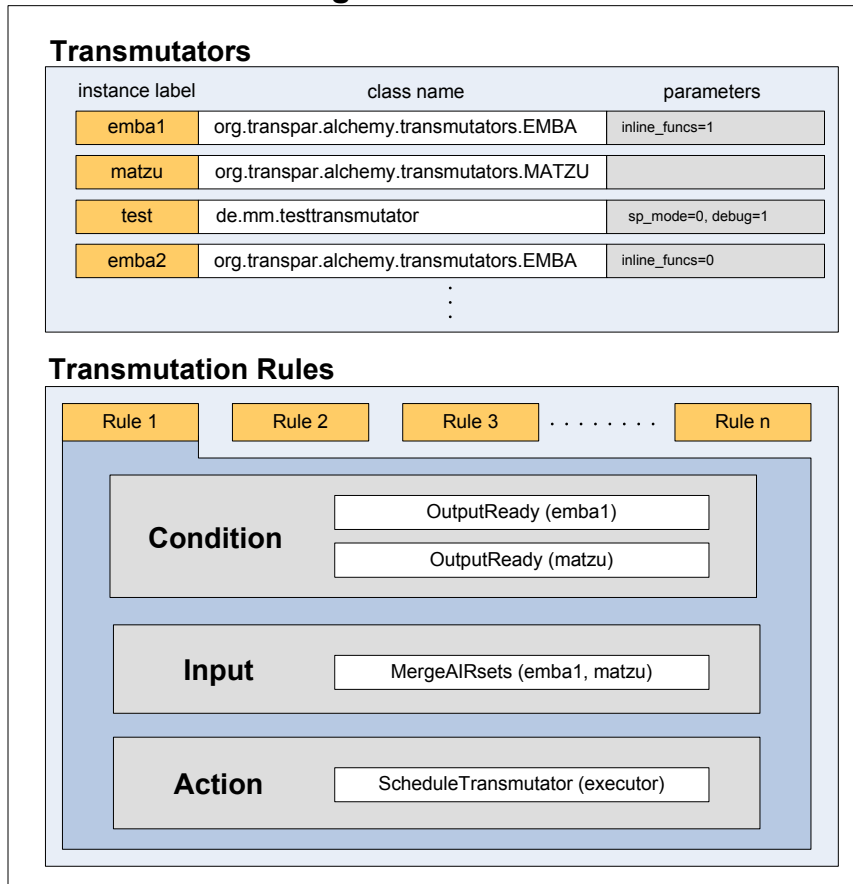


Figure 4.11: Transmutation Configuration Example

Figure 4.12 shows the sequence of actions that happen during the evaluation of a configuration rule.

4.5.3.1 Transmutator Instances

Throughout the configuration, Rules refer to specific instances of Transmutators that are created within a special section of the configuration that contains instance specifications. An instance specification creates a Transmutator instance from its type, a symbolic name using that rules can refer to the instance, and an optional set of configuration options⁴.

4.5.3.2 Conditions

A *Condition* consists of a list of *Predicates* and matches if all Predicates match, i.e. are evaluated true. Other matching strategies are possible, e.g. matching if the first or any Predicate matches.

⁴In the current implementation of ALCHEMY, it is not possible to provide configuration options via the configuration file.

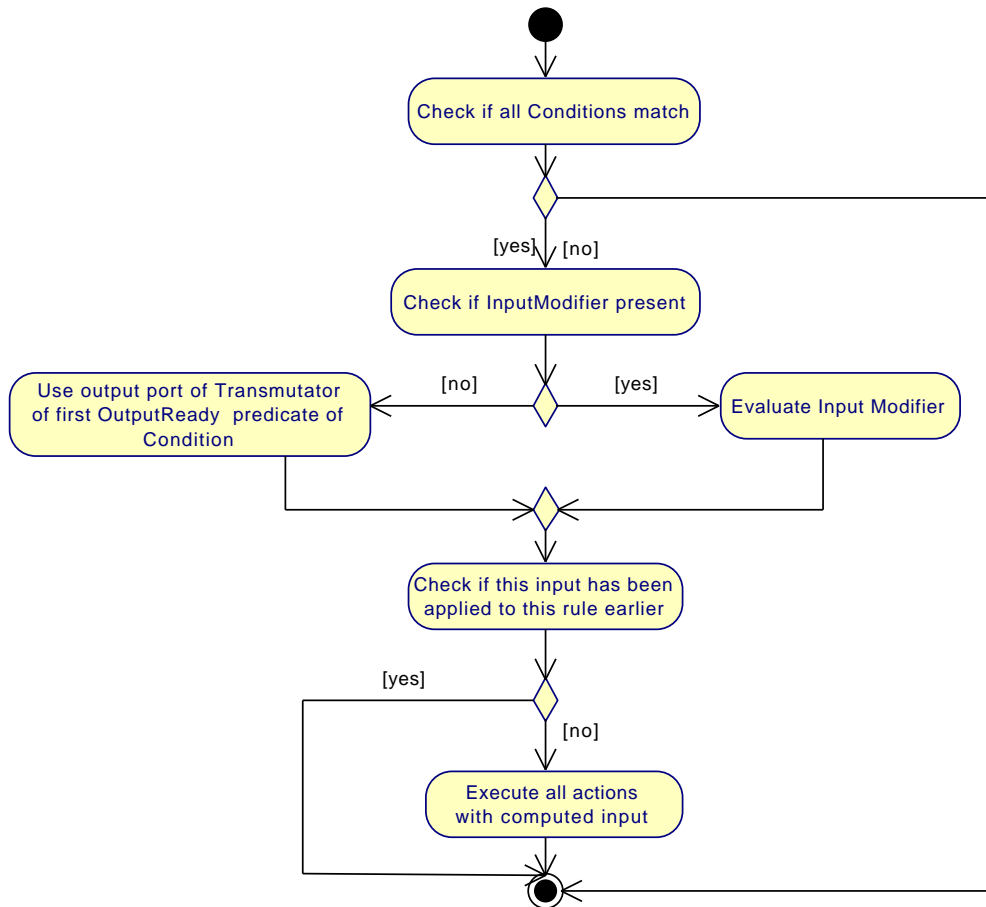


Figure 4.12: Analysis Activity Diagram “Rule Processing”

A *Predicate* has a set of predicate-specific named arguments that are used when the *Predicate* is evaluated. ALCHEMY currently provides the following Predicates:

OutputReady The OutputReady Predicate matches if the *Transmutator* that is referenced by the argument “at” has a non-empty output port.

OutputLabelSet This Predicate matches if the label that is specified by the argument “label” is set at the output port of the *Transmutator* that is referenced by the argument “at”.

OutputLabelEquals This Predicate matches if the the label that is specified by the argument “label” at the output port of the *Transmutator* that is referenced by the argument “at” is equal (character-wise) to the string that is specified in the argument “text”.

The dependencies that are imposed by the rule conditions may be represented as a directed graph. Figure 4.13 shows a graph that results from an example configuration that is printed in section C.1.

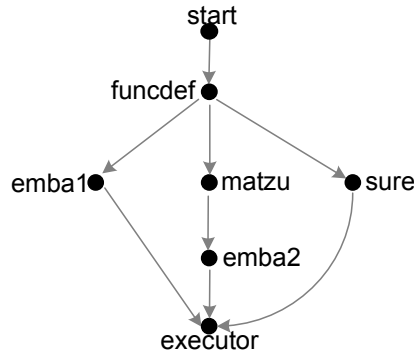


Figure 4.13: Rule dependencies in transmutation configuration

4.5.3.3 Input Modifiers

By default, a *Rule* provides the content of the output port of the *Transmutator* that is specified in the first *OutputReady* predicate of the *Condition* to the *Rule Action*.

If an *Input Modifier* is specified for a rule, this *Input Modifier* is executed at scheduling time to create the input for the *Action* part. The *Input Modifier* may create the input freely, e.g. from the output port of a *Transmutator* that is referenced in the *Condition* part or anything else.

However, *ALCHEMY* only guarantees that the assertions that are made in the *Condition* part are actually met, i.e. if an *Input Modifier* depends on the output of a *Transmutator* that was not checked in the corresponding *condition* part, there is a possibility of failure.

An *Input Modifier* results in exactly one *AIR Set* that is provided to the *Rule Action*.

ALCHEMY knows the following *Input Modifiers*:

MergeAIRSets Creates a new *AIR Set* from the output ports of a given set of *Transmutator* instances and provides that as input.

SimpleInput Provides the output port of a given *Transmutator* instance as input.

4.5.3.4 Rule Action

A *Rule Action* consists of a list of *Actions* that are all applied when the rule matches.

There is no semantic difference between one *Rule Action* with multiple *Actions* and multiple *Rules* with same *Condition* and an individual *Rule Action* for each *Action*.

Currently, only the following *Rule Action* is provided by *ALCHEMY*

ScheduleTransmutator Add the *transmutator* that is referenced by the “*transmutator*” attribute to the scheduling queue.

4.5.4 Static and Dynamic Configuration

The static configuration is provided by the user when *ALCHEMY* starts. A concrete XML-based example of a static configuration is shown in C.1. Alternatively, *Transmutators* are allowed to modify the *Transmutation Configuration* dynamically, by e.g. adding or removing *Rules*.

4.6 UC 6: Configure ALCHEMY

General R users and ALCHEMY experimenters must be able to configure ALCHEMY behavior and specific technical parameters of the system. “Configuration points” are restarts of the ALCHEMY system, i.e. it is not necessary to detect dynamically that the configuration has changed.

4.7 UC 7: Convert AIR to R

As ALCHEMY does not enforce the full evaluation of *AIR Programs* by *Transmutators*, after AIR processing has finished the *AIR Program* might still consist of a complex *AIR Expression*. For that reason, the *R Interpreter* must be able to perform “normal” R evaluation on the ALCHEMY results. At the same time, an ALCHEMY experimenter might sometimes be interested in seeing the R equivalent of an *AIR Expression*.

This use case converts a given *AIR Expression* or *AIR Program* into a corresponding *R Expression*. Ideally, that conversion is the inverse operation to the R-to-AIR conversion use case in 4.4. However, some special cases have to be taken care of, such as when the R-to-AIR mapping was not completely injective or when ALCHEMY has introduced *AIR Expressions* that are unknown to R, e.g., `SkeletonExpr`.

A general mechanism to resolve the first case does not exist because, depending on the concrete problem instance, the *AIR-to-R Converter*

- may sometimes be able to safely map an *AIR Expression* to a R construct that is different from the original one without changing semantics. For example, the valid, yet rarely used, assignment operator `=` may be mapped to `<-` without danger.
- might have to take the context of an *AIR Expression* into account, when mapping it to an *R Expression*. Until now, it has not been necessary to employ this resolution strategy.

For the second case, i.e., to convert *AIR Expressions* that have no obvious counterpart in R, there is also no general strategy. For instance, the `SkeletonExpr` mentioned above can be expressed as an R function expression with named parameters. However, if the chosen function is not known to R, this will result in an R runtime error. Hence, ALCHEMY *Transmutation Configuration* must ensure that no such *AIR Expressions* “leak” to R.

4.8 UC 8: Query AIR

The main task of a *Transmutator* is to analyze an (AIR) program for certain characteristics and modify the program based on these findings. Hence, it is important that *Transmutators* are able to conveniently inspect every detail of a program. For a given *AIR Expression*, a *Transmutator* must have easy access to all attributes and possible sub-expressions, e.g. to the left-hand side of a binary operation, i.e. a `BinopExpr`.

To further simplify element access, *Transmutators* may query a given expression to receive a list of matching descendants. The query is represented as an “XPath”

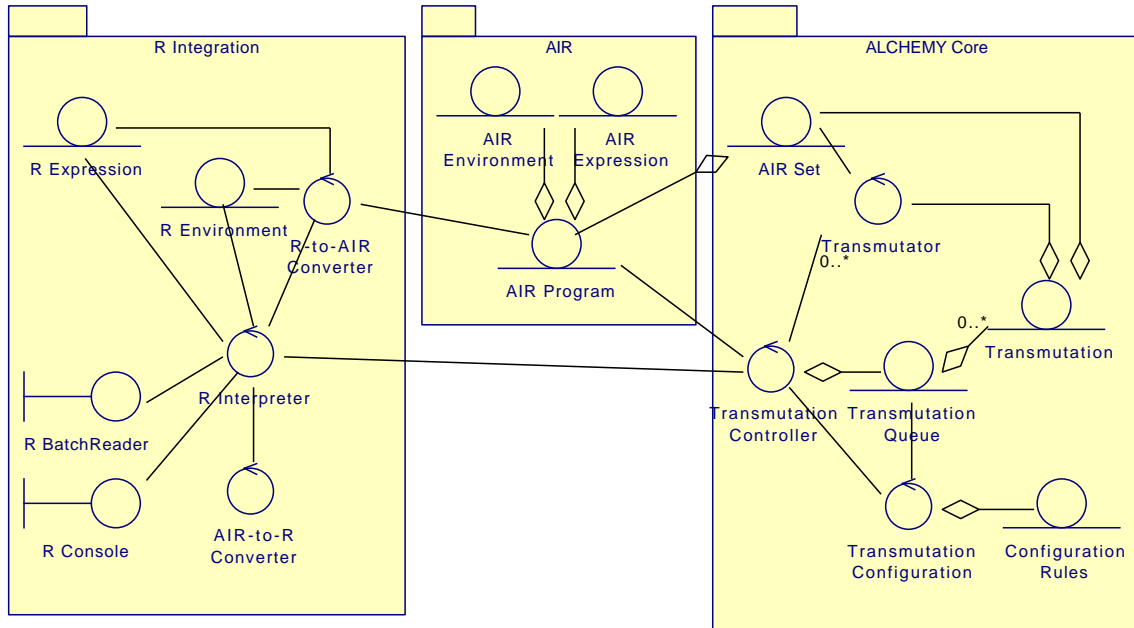


Figure 4.14: Overall analysis class diagram

based query string ([W3C07]). XPath is a language for selecting nodes from an XML document. In the context of AIR, it uses the AIR XML representation (see 4.3.2).

As an example, the highlighted subtree of figure 4.7 corresponds to the following XPath query:

```
/Program/WhileStmt/body/*/ForStmt[./ForCondition/@itervar="i"]
```

4.9 UC 9: Modify AIR

A *Transmutator* may replace *AIR Expressions* by other *AIR Expressions*. ALCHEMY provides programmatic aids for creating new expressions.

4.10 Analysis Overview

Figure 4.14 shows all analysis classes that participate in the identified use cases. Classes have been assigned to three analysis packages: *R Integration*, *AIR*, and *ALCHEMY Core*.

This partitioning in analysis packages is based on the principles of high cohesion, i.e., the containing classes are highly related, and loose coupling, i.e., dependencies between packages are minimized. Among the three packages, two contain the main functionality of the system (“R Integration” and “ALCHEMY Core”). “AIR” has been factored out as a package of its own, as it is heavily used by either of the other packages and putting it in one of them would increase coupling considerably.

5 ALCHEMY Software Design and Implementation

This section describes the software design of ALCHEMY, i.e., how the object-oriented analysis of chapter 4 is realized in software, taking all functional and nonfunctional requirements into consideration.

Section 5.1 depicts the overall system-architecture of ALCHEMY, i.e., the major building-blocks or components this software is constructed from. The following sections focus on specific architectural relevant, complex, or otherwise important elements of ALCHEMY’s software design. The description always starts with a refinement of the original requirements and goes as much into detail as it seems necessary to explain the topic.

5.1 System Architecture

An architectural sound design model answers the question of how to fulfill the functional requirements given all other constraints in a way that makes system’s future evolution and variation as easy as possible. This section tries to develop such an architecture based on the requirements identified in section 4. The description follows roughly a structure that is known as “Software Architect Document” (or SAD document) from the *4+1 Architectural View* model (see [Kru95]). In contrast to the common organization of an SAD document, the description of the individual architectural views, e.g. logical or physical view, is interwoven in the illustration of a design-level use-case realization.

5.1.1 Architectural Factors

The analysis-level partitioning of ALCHEMY classes in chapter 4 has resulted in the identification of three major packages (see diagram 4.14), with two of them, i.e., *R Integration* and *ALCHEMY Core*, responsible for the main functionality of the system, and the third one, *AIR* being the most important data or entity class.

It makes sense to take the analysis-level break-up of these three conceptual packages as a blueprint of ALCHEMY’s software design. It is instructing to reiterate the major requirements of each of these packages individually with special focus on their supplementary requirements:

Package “R Integration” The main responsibility of “R Integration” is to provide a user interface that behaves like the original R infrastructure, most of the time. In addition, it intercepts an R expression before evaluation, converts it to AIR, sends the AIR to the ALCHEMY Core *Transmutation Controller*, waits for its response, evaluates the response, and reports the result back to the user.

The most important nonfunctional requirement for this package is that its behavior must differ from that of R as little as possible, i.e. it must adopt its user interface and all its nonfunctional properties, such as interaction latency. If that requirement is not satisfied, ALCHEMY is at risk of losing user acceptance.

Additionally, as R is an active project that publishes about two releases every year, “R Integration” must be able to adapt easily to changes in R that are visible to the user.

Package “AIR” AIR represents an abstract access to the AIR language and provides means to search for *AIR Expressions* and replace them by others.

Although the R language itself and accordingly the R related subset of AIR can be considered stable, AIR must nonetheless be prepared for change as other skeletons or data types might turn out useful in the future.

Whether AIR is an abstract representation of a concrete language or there exists a concrete “verbalization” of AIR, which is abstract in nature: it is important to have a good representation that can be used to present AIR instances to users or exchange AIR instances between different software processes.

Package “ALCHEMY Core” This package is responsible for driving the transmutation of *AIR Expressions*, i.e. selecting and running *Transmutators*.

As ALCHEMY puts much emphasis on being an experimentation laboratory for parallelization, it has a few natural points of evolution and variation. First, the set of available *Transmutators* will change frequently. Experimenters will create *Transmutators* or modify old ones on an ad-hoc basis. These changes must be easy to perform.

Secondly, the sequence in that *Transmutators* are executed will vary. This requirement has already been captured during analysis with the introduction of the *Transmutation Configuration*. However, it must even be possible to integrate a completely different transmutation strategy.

As a third requirement, the *Transmutation Configuration*, as described in section 4, uses various *Predicates*, *Input Modifiers*, and *Actions*. It must be easy to extend each of these sets.

Additionally, transmutation performance might not be the most important requirement but must still not be neglected. Moreover, the design must make preparations to implement the optional requirements to be able to set break-points and let *ALCHEMY Core* send informational messages to the R session.

5.1.2 Architectural Decisions

In order to fulfill the requirements stated above, the following architectural decisions have been made:

1. The “R Integration” package is realized as an extension to the original R infrastructure, which is named `RAlchemy` from now on. By doing this, the danger of R users missing features or experiencing incompatibilities is completely removed.

2. The “*ALCHEMY Core*” package is realized as a stand-alone Java application with plugin mechanisms for Transmutators and other object types, which is called `AlchemyCore` as of now. With R being implemented in C and C being considered not adequate for experimenters, Java seems like a natural choice, because it is often taught as a first object-oriented language at universities. Additionally, Java’s reflection capabilities simplify the dynamic or configuration-based integration of plugins.
3. *AIR Environment objects* are realized as *Proxies*. The complete, i.e., global R environment (including the “base environment” that contains the R base package) contains hundreds of objects. Even without the base environment, a serialization of the environment would be very expensive and, depending on the actions the *Transmutators*, might be necessary several times per transmutation.
For that reason, `RAIchemy` does only communicate *proxy objects* (see [GHJV94]) to `AlchemyCore`. See section 5.3 for details.
4. *Value objects* may be realized as *Proxies*. R values may become huge, in particular in scenarios that *ALCHEMY* tries to be useful for, i.e. computations on large data sets. Even a single serialization of such a value may be prohibitive. Therefore, `RAIchemy` transmits only Proxy objects for vectors or lists whose size exceeds a certain threshold. See 5.3 for details.
5. Stateless operation of `AlchemyCore` will be prepared. Any architecture but in particular a distributed system like *ALCHEMY* should avoid unnecessary complexity. The requirement to set a breakpoint and making transmutator execution “wait” for an R user to resume the transmutation would, in a naive implementation, suspend server execution and leave the running `AlchemyCore` process in a client dependent state. This is error-prone, e.g. as the server won’t be able to react to client failures. For that reason, it is good practice to implement server software in a “stateless” way.

These decisions have consequences on the design, the implementation, and the deployment of *ALCHEMY*:

- `RAIchemy` and `AlchemyCore` must be deployed as distinct operating system processes, i.e., active components that don’t (automatically) share memory. As there exists a “one-way” dependency from `RAIchemy` to `AlchemyCore`, this effectively leads to a client-server architecture with `RAIchemy` being in the client role and `AlchemyCore` being in the server role.
- `RAIchemy` must offer services to allow clients of a Value or Environment Proxy access to the corresponding full object. This creates a (logical) dependency from the server, i.e. `AlchemyCore`, to the client, i.e., `RAIchemy`. It is important to localize these dependencies in classes that are local to `AlchemyCore`.

5.1.3 Description of *ALCHEMY*’s Architecture

Package diagram 5.1 shows a high-level view of *ALCHEMY*’s architecture, diagram 5.2 a typical deployment scenario with component dependencies. In the following sections,

the depicted packages are described in more detail. The presentation of the architecture does encompass elements of all levels of abstraction and kinds of representation, i.e. analysis, design, implementation, dynamic, or static, as long as the presented information helps to understand the system.

The next few sections will explain the main architectural classes, as they participate in the use case *UC 2: Perform Interactive R Session* as described in section 4.2.

Figure 5.3 shows the high-level communication between ALCHEMY components for a simple transmutation scenario that does not involve parallel backends.

As explained in the previous section, the user interacts with an “ALCHEMY enabled” version of the original R runtime environment, `RAIchemy`, that is further discussed in the next section.

5.1.4 Package: `RAIchemy`

The illustration of `RAIchemy` in diagram 5.1 reveals a loose multi-layer architecture with layers that contain R interfaces, services, its domain model, and commonly used infrastructure classes. This structure may appear somewhat artificial on the side of the original R environment, as there is no known R design model and this illustration assigns the C modules to their corresponding design classes based on their “assumed intention”.

`RAIchemy` realizes the analysis package “R Integration” with the following design classes and packages:

- `RUserInterface` realizes *R Console* and *R BatchReader*
- `R Core` realizes *R Interpreter*
- `AIRtoRConverter` realizes *AIR-to-R Converter*
- `RtoAIRConverter` realized *R-to-AIR Converter*

Additionally, `RAIchemy` contains the following artifacts that are necessary for satisfying the architectural decisions from section 5.1.2:

`AlchemyAdapter` a *proxy* or *business delegate* (see [BHS07]) for `AlchemyCore`

`RServer` provides services that are implemented in `RServices` to clients

`RServices` services that are required to implement the proxy mechanisms for environment lookup, value access, and transformation from AIR to R

`Logging` Helper functions for logging and pretty-printing R data structures

`Communication` Low-level classes for inter-process communication

5.1.4.1 Implementation

`RAIchemy` bases on the existing R runtime environment and enriches it with functionality to interact with ALCHEMY. R is implemented in C. So some design patterns, e.g. *Visitor* (see [GHJV94]), that heavily rely on object-oriented features such as runtime-polymorphism cannot be implemented in a completely clean way.

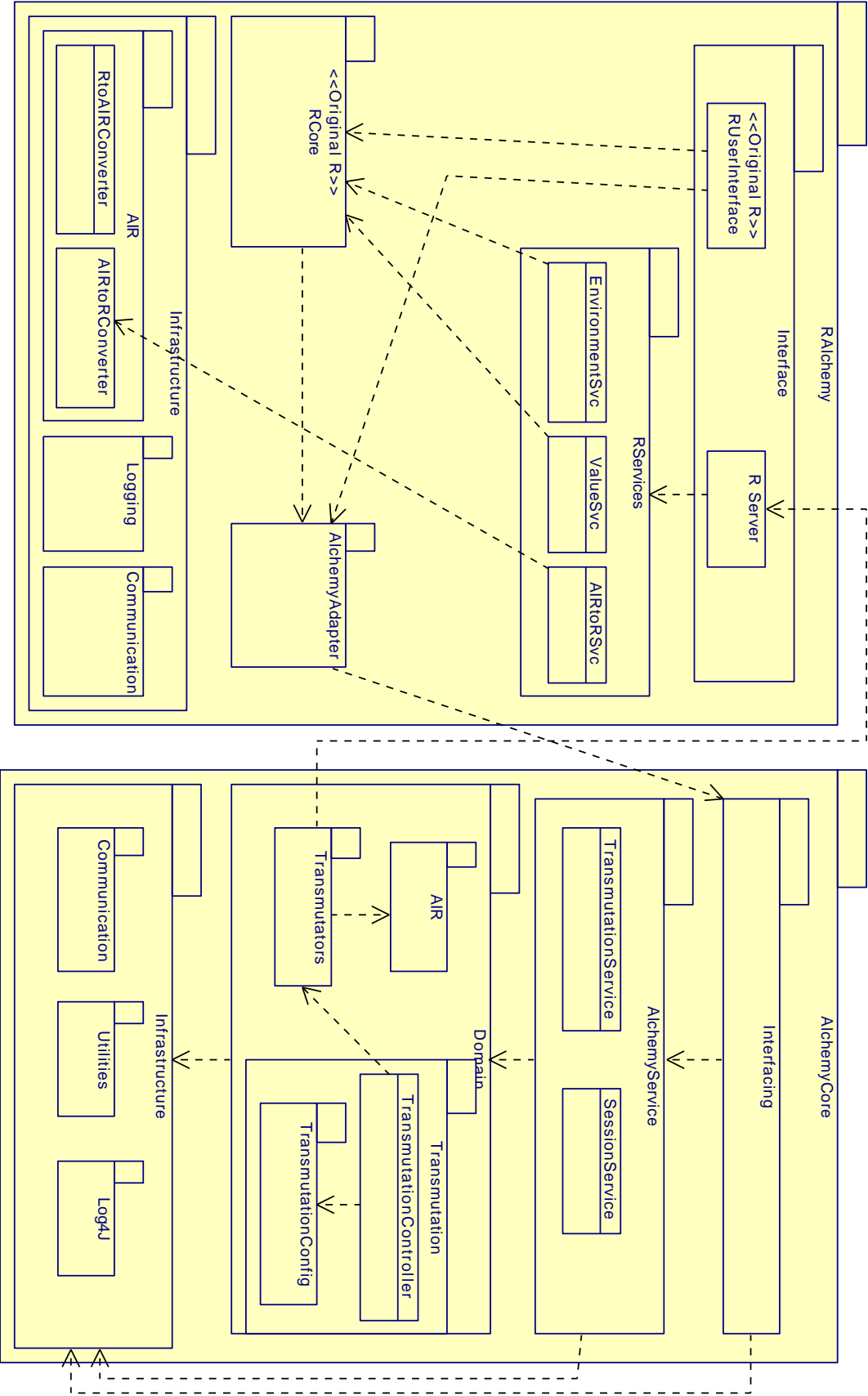


Figure 5.1: Overall logical design

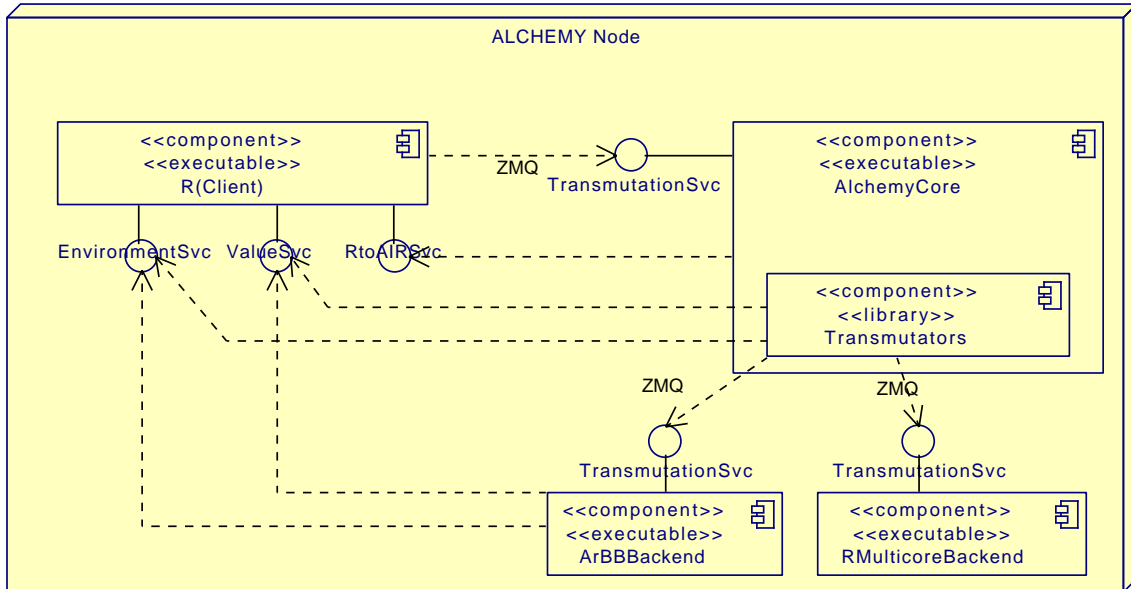


Figure 5.2: Alchemy Deployment

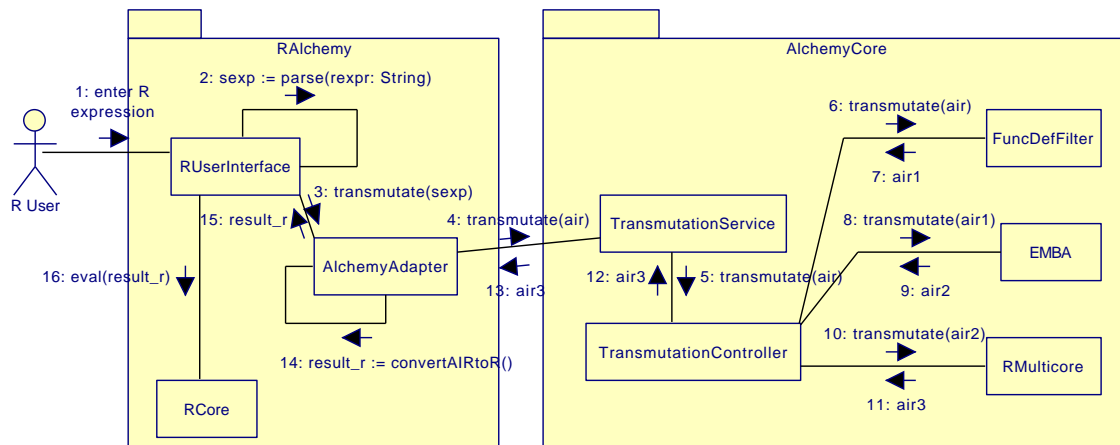


Figure 5.3: Alchemy High-Level Collaboration Diagram

In general, `RAIchemy`'s code style tries to mimic the structure of object-oriented or object-based languages, where possible. In particular, in correspondence to e.g. C++, most design classes are implemented having an individual header file and translation unit. The names of C functions that belong to such "classes" have a prefix that is related to the design class name. The "method name" is appended to the prefix. The instance object of a method call, i.e., the `textttthis` in a JAVA or C++ analogy, is provided as the first argument of the call. For example, the corresponding C code to the Java snippet

```
1 AlchemyAdapter adapter = new AlchemyAdapter();
2 adapter.transmutate();
```

would be

```
1 AlchemyAdapter* adapter = AlchemyAdapter_new();
2 AlchemyAdapter_transmutate (adapter);
```

In order to be prepared for new releases of R, one of the main implementory constraints is to integrate `ALCHEMY` in a non-invasive way, i.e., without making too many modification to the R code base. To achieve that goal, `ALCHEMY` code has been extracted to unique modules that are referenced from R code only at a few places.

Figure 5.4 shows the first activities for the realization of an instance of use case 2 on the part of `RAIchemy` as a UML communication diagram. This example is continued in the following sections as the program flow reaches other design elements.

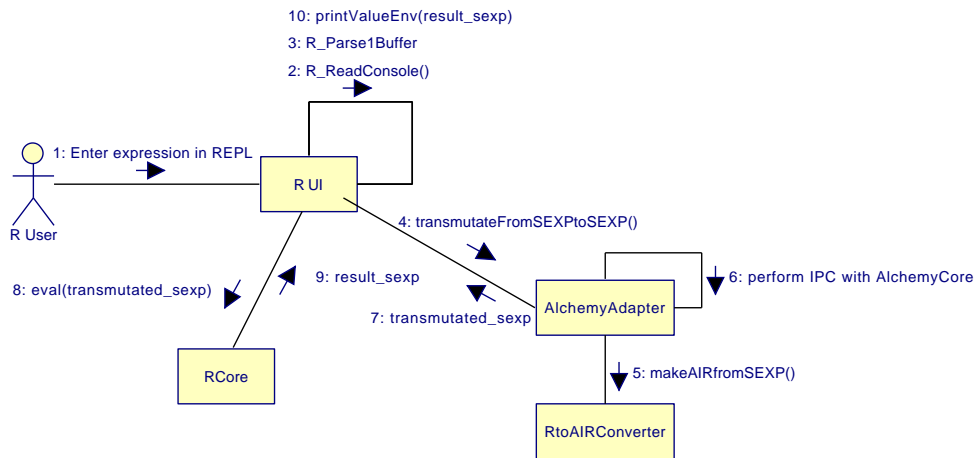


Figure 5.4: 1st Part of Use Case 3 Realization

On startup, the R runtime environment performs various initialization tasks such as preparing the base (symbol) environment, i.e. the *R Environment* known from analysis, preparing Input/Output channels, initializing the R memory management¹, and setting up the R error handling subsystem. These actions are performed in `main/main.c:setup_Rmainloop()`.²

¹which is out of the scope of this thesis, but to summarize: R employs a custom generational garbage collector

²As of now, this means that the corresponding implementation is located in the function `setup_Rmainloop()` that is defined in the R source file `main.c` in the `src/main` subdirectory of an R source installation.

After startup is completed, the R interpreter enters the REPL (Read-Eval-Printloop), which is located in the design class `RUserInterface`.

ALCHEMY is not automatically enabled after system startup. The user must initialize the subsystem by executing the R function `alchemy.enable()` that is added to the list of R built-in functions at startup, along with `alchemy.disable` and `alchemy.loglevel`. The assignment of R built-in function names to C functions takes place in `main/names.c`.

At initialization, the function `main/alchemy.c:do_alchemy_enable()` that is associated with `alchemy.enable()` performs the following steps:

1. A new `AlchemyAdapter` instance is created. (see 5.1.6)
2. An IPC strategy is instantiated and associated with the adapter instance. At present, this is always `ZMQConnection`. (see 5.2)
3. A new instance of `AlchemySession` is created. `AlchemySession` encapsulates or references all data that belongs to a specific R session. In the current release, `AlchemySession` is a pure design element in `RAlchemy` and `AlchemyCore` and has no domain- or analysis-level counterpart. However, in future releases, it is planned to add transmutation management to ALCHEMY, as described in 9.
4. A new `RServer` instance is created that is used to accept incoming requests for services like the environment service or the value service. (see 5.2)
5. A new `ValueRepository` object is created and associated with the session.
6. A new `ValueService` object is created and initialized with the `ValueRepository`. The `ValueService` is registered at the `RServer`.
7. A new `EnvironmentService` is created and registered with the `RServer`.
8. A new `AIRtoRService` is created and registered with the `RServer`.

5.1.5 Class: `RUserInterface` (or R UI)

The `RUserInterface` design class realizes the analysis classes *R BatchReader*, *R Console*, and implements some responsibilities of the *R Interpreter* class, i.e. parsing the input and delegating control to transmutation and R evaluation. In fact, from an architectural point of view, that might be too many responsibilities for an interface class. However, as the existing R realization does not allow a strict separation of interface, service, and domain logic, this ugliness is tolerated in this context.

As `RUserInterface` is responsible for parsing the user input, thereby converting it to an “S Expression” (R’s internal language representation, see 3.1.1). S Expressions or “SEXP”s, as they are called within the R source code, realize the analysis class *R Expression*.

`RUserInterface` is realized as a set of functions that are contained in `src/main/main.c`: `main/main.c:R_ReplConsole()` and `main/main.c:Rf_ReplIteration()`. `Rf_ReplIteration()` is responsible for the REPL. It controls the user interaction, R parsing, and evaluation by delegating to specific functions: first, it calls `main/main.c:R_ReadConsole()`, which implements the “Read” part of the R REPL by reading user input from

the console. After that `Rf_ReplIteration()` uses the R parser by calling `main/main.c:R_Parse1Buffer()` until a complete R expression could be parsed or an error is detected. The parser routine returns a “parse SEXP” of the entered expression.

In an original, i.e. unmodified R, or if the R ALCHEMY integration is not activated the parse SEXP is directly evaluated by calling `main/eval.c:eval()`. However, when ALCHEMY is integrated and enabled, before evaluation takes place, the parsed expression is sent to the the `AlchemyAdapter` design class that is implemented by

`main/AlchemyProxy.c:AlchemyAdapter_transmutate_fromSEXP_toSEXP()`.

Diagram 5.5 shows the sequence of calls that follows user input.

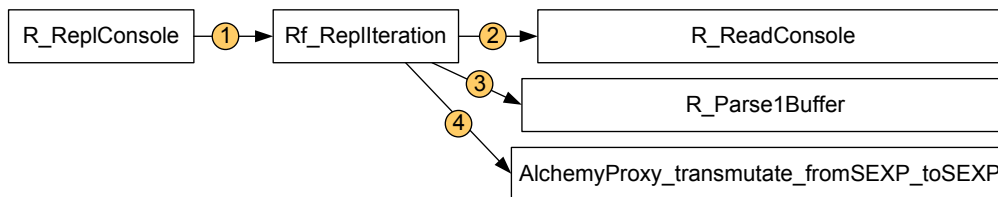


Figure 5.5: Call sequence in R with enabled ALCHEMY

After the call to `AlchemyAdapter_transmutate_fromSEXP_toSEXP()` returns a (possibly transmuted) SEXP, `RUserInterface` checks if transmutation was successful, i.e. that `AlchemyCore` has not reported an error or declined transmutation. Depending on whether an error has been detected, the original, untransmuted or the transmuted SEXP is chosen for further processing. At this point, ALCHEMY’s interception finishes and R continues as usual by evaluating the SEXP and printing the result to the user.

Every access to `AlchemyCore` happens in the context of an `AlchemySession` that encapsulates R session specific state within `AlchemyCore` and hence will facilitate asynchronous operation.

5.1.6 Package: AlchemyAdapter

`AlchemyAdapter` offers the services of `AlchemyCore`, delegates their execution to the `AlchemyCore` Java process, and transparently manages the necessary inter-process communication (IPC) with this process. In this sense, it matches the description of a Proxy or Business Delegate. However, `AlchemyAdapter` also acts as an Adapter and enriches the `AlchemyCore` interface with operations that are specific to the needs of R, e.g. is allows the transmutation of an SEXP (instead of an AIR expression).

From an analysis level perspective, most important operation of `AlchemyAdapter` is `transmutate_fromSEXP_toSEXP()` that performs a transmutation of an SEXP and waits for the result. For this call, there is also a non-blocking variant in preparation called `transmutate_fromSEXP_toSEXP_async()` that returns a `AlchemyTransmutateFuture` (see [BHS07] for a description of the Future synchronization pattern) instead of an SEXP. The asynchronous call will become necessary as soon as ALCHEMY’s debugging facilities such as breakpointing will be integrated.

In order to be able to trigger transmutation at `AlchemyCore`, `AlchemyAdapter` has to convert the given R SEXP into a corresponding AIR expression. `AlchemyAdapter` delegates this conversion to `RtoAIRConverter`.

After `RtoAIRConverter` has returned an XML representation of the AIR expression, this result must be communicated to `AlchemyCore` via inter-process communication. See section 5.2 for an in-depth explanation of the proxying mechanism and how the communication is realized.

The AIR object returned by `AlchemyCore` is converted back into an SEXP by `AIRtoRConverter`. This SEXP is returned to `RUserInterface`.

5.1.7 Class: `RtoAIRConverter`

`RtoAIRConverter` converts an R SEXP into its AIR equivalent. The design class is implemented in `main/RtoAIRConverter.c` and applies a Visitor pattern (see [GHJV94] for a detailed explanation). The class is used by `AlchemyAdapter` to prepare transmutation, by `ValueService` to generate the AIR representation of an R value, and by `EnvironmentService` to create the AIR expression associated with a symbol table entry.

While a Strategy pattern may be used to decouple a class from a specific algorithm, i.e. a specific way to perform an action, a Visitor is a tool to encapsulate (often related) behavior for a set of classes with a common superclass. In most cases (but not necessarily) this is used to process object hierarchies in an extensible way that does not pollute the interfaces of the classes used in the hierarchy.

For a concrete realization of this pattern all objects of an object hierarchy implement a method called `accept()` that takes a Visitor object as an argument. The target object usually “calls the Visitor back” by executing a method on the Visitor that is specific to the type of the caller. For instance, a common scenario is that an object of type `SpecificNodeType` has a method with a signature like `accept(GeneralNodeVisitor vis)`. When this method is called, it directly executes `vis.visitSpecificNodeType(this)` and returns afterwards.

As Visitor operation in most cases involves traversing the object hierarchy, there is a degree of freedom in choosing where tree traversal is performed: in the visitor, the visited object, or in an iterator.

Class diagram 5.6 shows the design model for so-called `SEXPVisitors` that `ALCHEMY` has introduced to process S expressions.³ As can be seen, every visitor provides an operation for every possible SEXP type.

Sequence diagram 5.7 shows the call sequence for the conversion of a simple R expression. As `RtoAIRConverter` is realized in C, the sequence diagram shows a “pseudo-objectified” illustration of the call sequence.

Every `visit()` method of `RtoAIRConverter` is responsible for converting one specific SEXP type to the XML representation of its corresponding AIR node. `RtoAIRConverter` uses `libxml` (available at [Gno]) and creates the final XML document recursively as it traverses the SEXP tree.

The following conversions are currently performed by `RtoAIRConverter`:

`CLOSSXP` used for closures, i.e. anonymous functions and function definitions in the environment

³The visitor `PrettyprintVisitor` can be used to generate a beautified text representation of SEXPs, which has proven invaluable while debugging. The visitor `GraphmlVisitor` creates a “GraphML” document from an SEXP. GraphML can be processed by common graph drawing utilities such as `yEd` (see [ywo]). Diagram 3.1 has been created with this visitor.

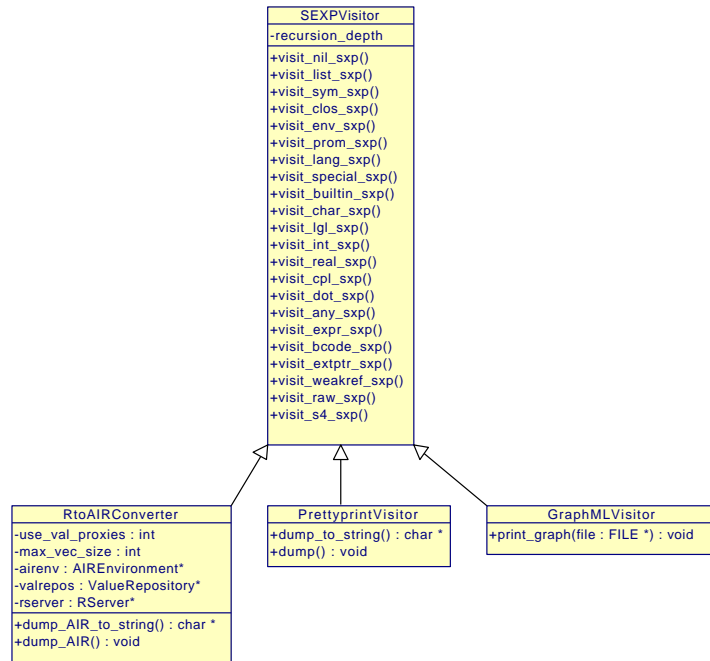
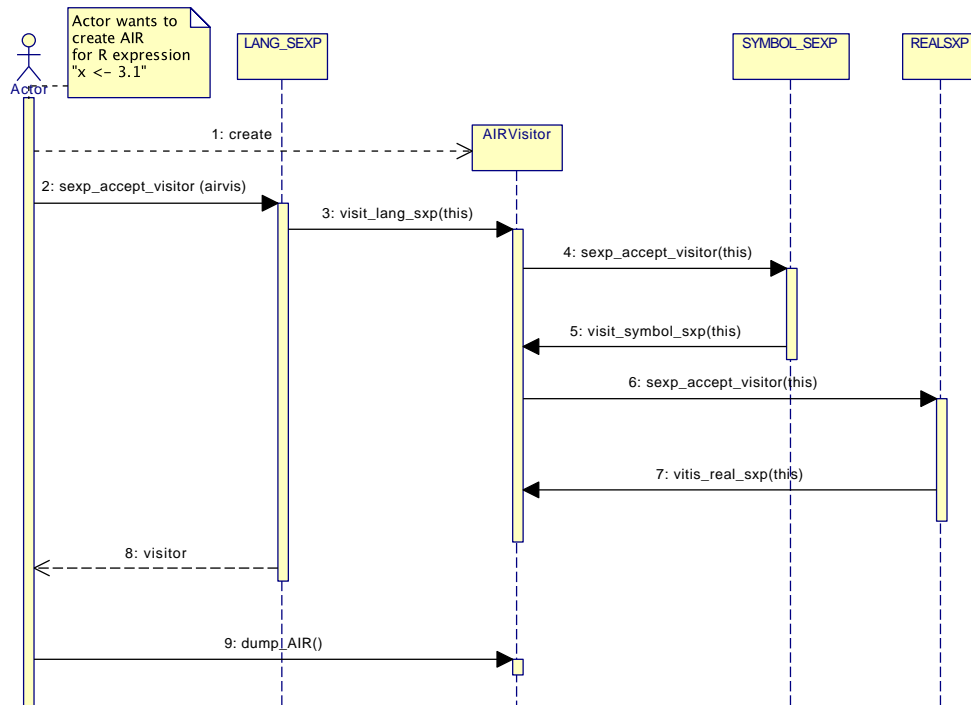


Figure 5.6: (Design) Class hierarchy of SEXP visitors

Figure 5.7: (Pseudo-)sequence diagram of RtoAIRVisitor processing of `x <- 3.1`

- LANGSXP** used for function calls of all kinds. `RtoAIRConverter` maps SEXPS of this type to a variety of AIR expressions such as `WhileStmt`, `BinopExpr`, etc.
- SPECIALSXP** used for internal R functions. Mapped to `BuiltinFunc`
- BUILTINSXP** used for internal R functions. Mapped to `BuiltinFunc`
- CHARSXP** used for representing strings. Mapped to a `ConstantExpr` with AIRType “string”.
- LGLSXP** used for representing boolean vectors. Currently mapped to a `ConstantExpr` with a value of AIRType “logic” or a corresponding `AIRVector` type.
- INTSXP** used for representing pure integer vectors. Currently mapped to a `ConstantExpr` with a value of AIRType “integer” or a corresponding `AIRVector` type.
- CPLSXP** used for representing vectors of complex numbers. Not implemented, yet.
- REALSXP** used for representing vectors of real numbers. Currently mapped to a `ConstantExpr` with a value of AIRType “real” or a corresponding `AIRVector` type.
- STRSXP** used for representing vectors of strings. Currently mapped to a `ConstantExpr` with a value of AIRType “string”. Vectors with length greater than one are not implemented, yet.
- DOTSXP** ellipsis operator, i.e. “...”, is not fully implemented, yet.
- ANYSXP** not used in R.
- VECSXP** used for lists, factors, etc. Not fully implemented, yet (can’t be received via `ValueProxy`)
- EXPRSXP** “language vectors” that may have been returned by e.g. R’s `parse()` function. Not implemented.
- BCODESXP** Object of R Bytecode compiler. Not implemented.
- WEAKREFSXP** Not implemented.
- EXTPTRSCP** Not implemented.
- RAWSXP** Not implemented.
- S4SXP** S4 object. Not implemented, yet.

It should be noted, however, that most of these SEXP types are not of interest to an ALCHEMY transmutator until it tries to e.g. lookup a variable containing one of them. In this case, there is no defined behavior in the current ALCHEMY release.

5.1.8 Class: AIRtoRConverter

`AIRtoRConverter` converts an AIR expression in XML representation into a corresponding R expression. This class is used by the `AIRtoRService` and by `AlchemyAdapter` to transform a transmutation result back to R. It is implemented in `main/AIRtoRConverter.c:AIRtoRConverter_convert_AIRXML()`.

The operation `AIRtoRConverter` uses the libxml XML parser to create a DOM representation of the AIR XML document and generates the resulting SEXP by traversing that DOM tree.

5.1.9 Package: RCore

The `RCore` design package realizes the *R Interpreter* analysis package. In particular, it has the responsibility to evaluate parsed S expressions and implement the R base library. In principle, one could contend that the whole remaining R source code realizes `RCore` with most of the existing C and Fortran files being in charge of realizing the library functions of the R language.

5.1.10 Package: RServices

The package `RServices` contains various services that `RAlchemy` provides to external clients via `RServer`. See 5.3 for a detailed description.

5.1.11 Package: Communication

This infrastructure package contains currently only `AlchemyZMQConnection` which is described in 5.2

5.1.12 Package: Logging

Logging provides basic logging facilities that is ubiquitously used in the `ALCHEMY` R code.

5.1.13 Package: RServer

`RServer` is used to provide services to external clients via an `AlchemyConnection`. See 5.3 for details.

5.1.14 Package: AlchemyCore

`AlchemyCore` realizes the *ALCHEMY Core* analysis package. Hence, its main responsibility is the transmutation of AIR expressions. Depending on `AlchemyCore`'s configuration, its behavior may differ significantly. Figures 5.8 and 5.9 show two possible workflows resulting different `AlchemyCore` configurations.

The high-level design of `AlchemyCore` matches a layered architecture quite well. From top to bottom, the following layers can be identified:

Interfacing Responsible for providing `AlchemyCore` services its clients, e.g. `RAlchemy`. Interfacing is described in section 5.2.

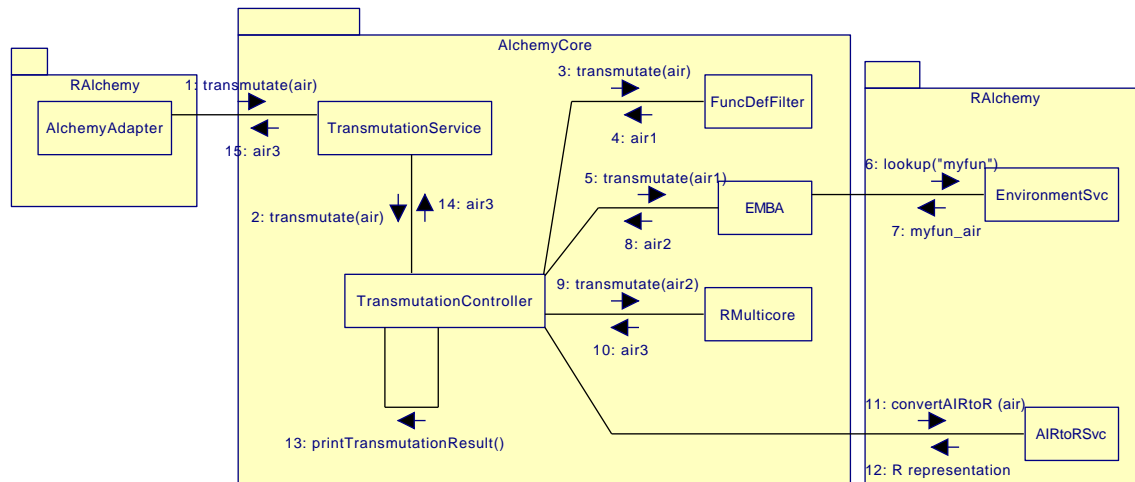


Figure 5.8: Collaboration example for transmutation

Service Responsible for encapsulating use case or domain object specific functions that are offered to clients by the **Interfacing** layer. This layer contains currently the classes `TransmutationService` (see `transmsvc` and `SessionService` (see `sesssvc`).

Domain Responsible for implementing the domain logic that is currently mostly related with the transmutation of AIR. See 4.5.2 for a detailed explanation of the design of ALCHEMY's transmutation subsystem. Another responsibility of the **Domain** layer is the handling of AIR expressions.

In future releases (with asynchronous operations), **Domain** will also manage the relationships between sessions, transmutations and other entity classes.

Infrastructure `AlchemyCore` provides various infrastructural services to its components. Among others, in every class `Log4J` can be used for logging, there are several utilities to work with XML documents or AIR expressions, and IPC is encapsulated in multiple classes.

5.1.15 Class: `TransmutationService`

The `TransmutationService` class provides the `transmutate()` operation and is used by interface classes. It realizes `transmutate()` by instantiating a concrete `TransmutationController` and delegating transmutation to this controller.

5.1.16 Class: `SessionService`

The `SessionService` provides session relates services to its clients. In particular, it is responsible for creating new sessions, removing sessions but also for managing session parameters. The `SessionService` is currently not fully implemented, as a full "session management" will not be necessary until ALCHEMY is ready for asynchronous operation. However, `RAlchemy` already uses the `initSession()` operation to generate a new session ID.

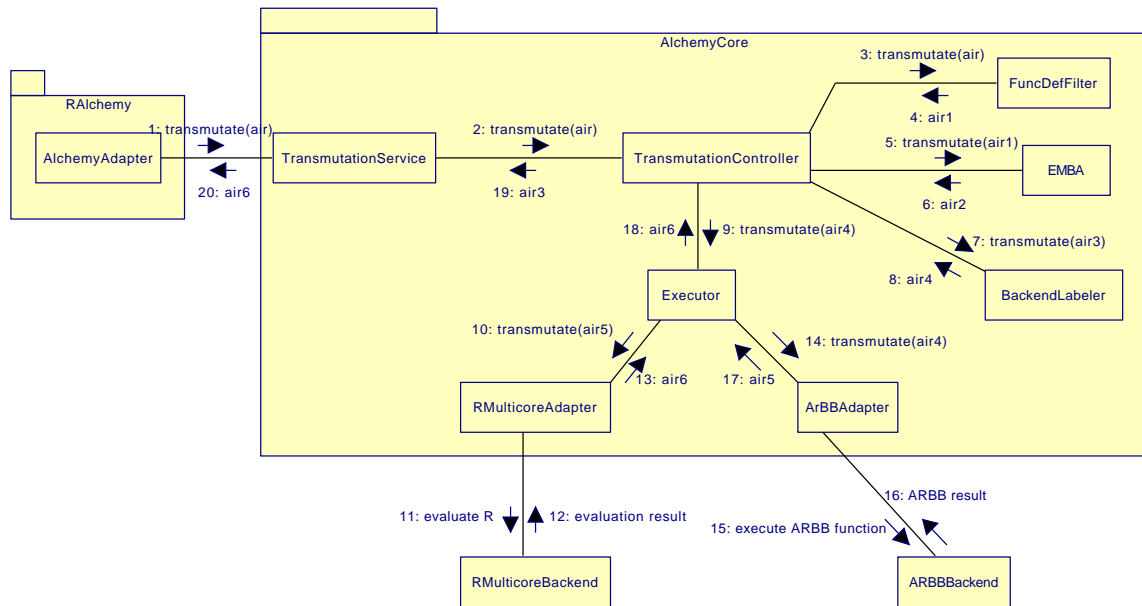


Figure 5.9: Collaboration example for transmutation

5.2 Inter-Process Communication

In RAlchemy `AlchemyAdapter` encapsulates the logic for executing operations between the R process and the running ALCHEMY Core Java application. `AlchemyAdapter` is instantiated with an `AlchemyConnection` object (see class diagram 5.10).

An `RServer` is used to provide services to external clients (such as `AlchemyCore`). The server is also associated with an `AlchemyConnection` that is used to communicate with the client.

`AlchemyConnection` provides a generic abstraction for the communication between two peers. It can be used for establishing a client connection to a server or (passively) accepting a server connection from a client. Concrete connection classes must implement operations for setting a local or remote connection endpoint (`setServerURI()`), where the “URI” is interpreted by the concrete connection technology. For instance, by an HTTP connection type, “URI” might be interpreted as the HTTP URL to send the request to or to await a request at. To connect to the server designated by the “server URL”, a connection user calls (`open()`).

All data that is sent over a connection is structured as a list of strings. The members of that list are called “message parts”, as of now. User may send messages (consisting of multiple parts) to a peer and receive messages from a peer.

A concrete connection type must realize the following functions (the function signatures are represented in a Java notation. The RAlchemy C implementation uses analogous C types, instead):

`void setServerURI(String)` Set the “address” of the connection’s server peer. The address is interpreted by the underlying transport protocol, as described above.

`void open()` The calling peer adopts the “client” role in this connection and tries to establish a connection to the peer that is denoted by the “server URI” of the

connection.

`void bind()` The calling peer adopts the “server” role in this connection and awaits connection requests on the address that is defined in the “server URI”.

`void send(String[])` The caller sends a list of strings to its peer. The call is considered successful, if all list members have been received by the peer in the same order as they have been sent.

`String[] receive()` The caller blocks while waiting for data from its peer. The call does not return until the full list of strings has been received.

Currently, the only connection type that is implemented in `ALCHEMY` uses the ZeroMQ networking API (see [Zer]). ZeroMQ has been chosen over e.g. HTTP or pure socket based TCP or UDP communication for the following reasons:

- ZeroMQ has atomic message semantics. Data is sent in messages with “deliver atomically or failure” semantics.
- ZeroMQ naturally allows dividing messages into “parts”, which makes it easy to realize “list of strings” semantics.
- ZeroMQ is very performant.
- ZeroMQ has a very simple API.
- ZeroMQ has a portable API over many operating systems and programming languages.
- ZeroMQ provides internal message buffering to reduce client latencies.

In `RAlchemy` the state of a connection is kept in the type `AlchemyZMQConnection`. `AlchemyZMQConnection` realized the operations as described above.

In `AlchemyCore` the analogous Java class is called `ZMQConnection`.

Figure 5.10 shows the relationships of the R classes, i.e. modules, that are involved in integrating `RAlchemy` and `AlchemyCore`.

On a higher level `ALCHEMY` communication relies on a minimal protocol on top of the message abstraction that bases on the following conventions:

- A message from a “client” to a “server” is called a “request”.
- A message from a “server” to a “client” is called a “response”.
- The first part of a request does always contain the service name that is requested.
- The following parts of a request contain the (positional) service parameters.
- The first part of a response always contains the response status. The response status may be one of “OK” or “ERROR”.
- If the response status is “ERROR”, the next part contains a description of the error.

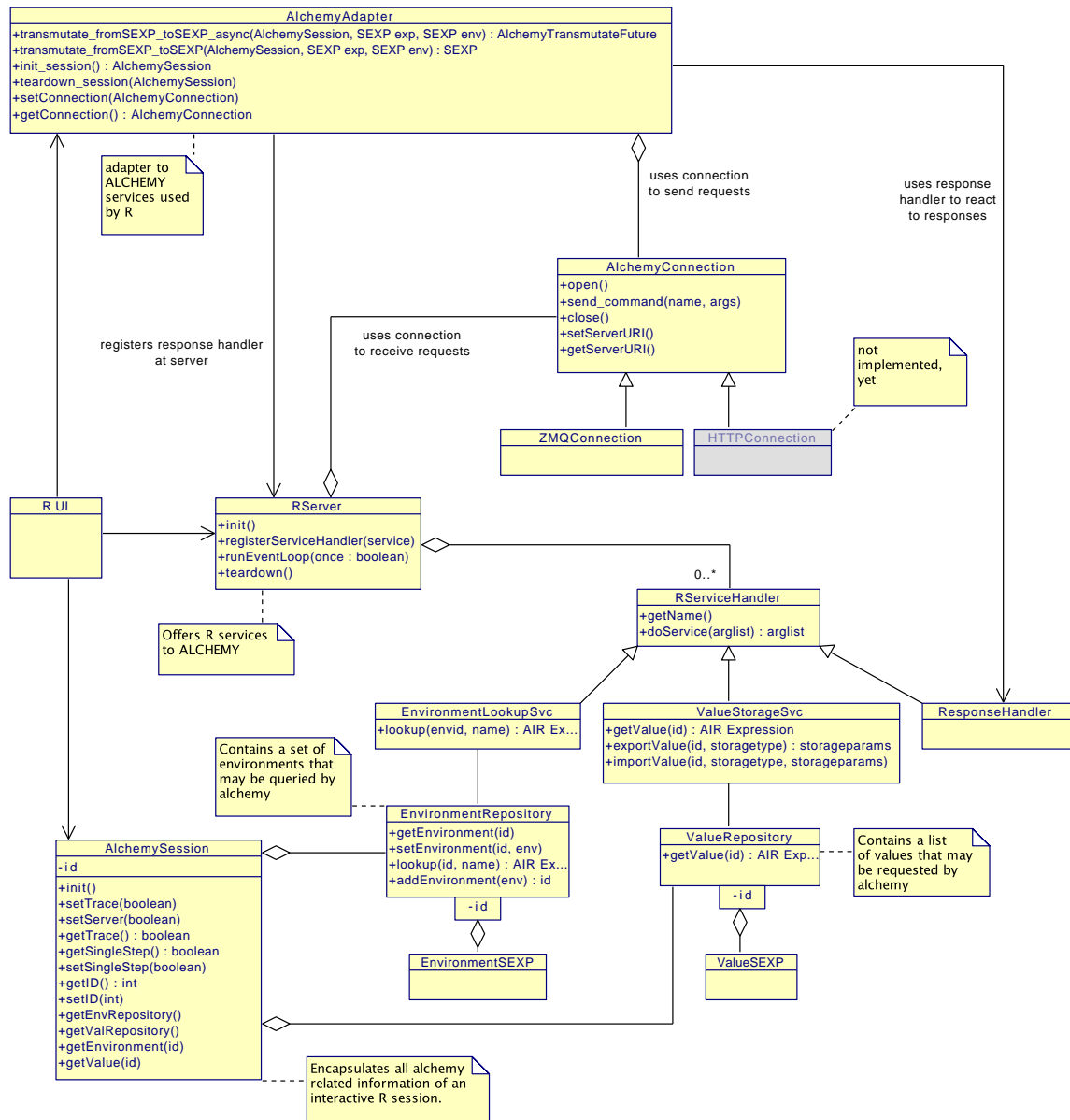


Figure 5.10: R design classes participating in R - Alchemy integration

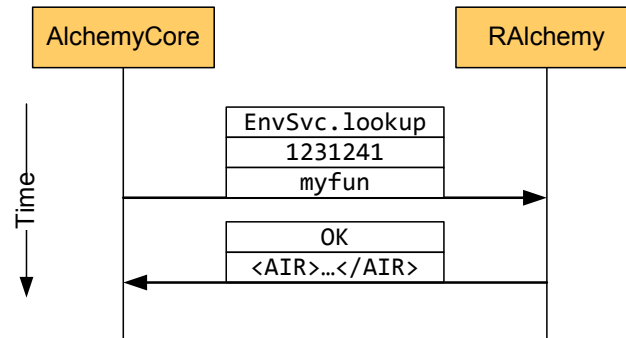


Figure 5.11: Example request/response cycle for the minimal ALCHEMY protocol

- If the response status is “OK”, the following parts contain a list of response data items.

Figure 5.11 shows an example, in which `AlchemyCore` uses the `EnvironmentService` of `RAlchemy` to look up the symbol “myfun” in the environment with the id 1231241. Section 5.3 describes the existing R services and their serializations.

5.3 R Services

R provides several services to `AlchemyCore`, transmutators, and parallel execution back-ends that are necessary to realize the *virtual proxies* that provide lazy access to the environment and values. Additionally, R provides a service that converts an AIR expression that is given in its XML representation into a corresponding R expression string.

R services are provided by an `RServer` (see figure 5.10) that is composed of an `AlchemyConnection`⁴, which is used to communicate with the client, and a list of `RServiceHandlers` that realize R’s services. The following sections describe what services are currently implemented.

In principle, R Services can be used by any client. Their main purpose, however, is providing the value proxies and `SymbolExpr`, i.e. the “environment proxies” in `AlchemyCore` the means to work as intended.

5.3.1 Value Service

The Value Service provides access to the data of a value, e.g. a large vector that is processed within an R program or stored within R memory. The main motivation behind the Value Service is that values that are physically large, i.e. that have a large physical representation, must not be serialized and transported between components unnecessarily.

Instead of large values, proxy documents are introduced to AIR programs that refer to a Value Service instance. Whenever a value is needed, e.g. because a parallelization

⁴The visualization in 5.10 is idealized because C does not support inheritance as known from object-oriented languages. Currently, in `RAlchemy` inheritance is “by intention” in the sense that “sub-classes” guarantee to realize methods as intended.

strategy requires knowledge of the structure of data, clients can use the Value Service to obtain the actual data.

Within `RAlchemy` the Value Service uses a `ValueRepository` (see 5.10) to keep track of the proxied values. The `ValueRepository` is, in principle, a list of pairs of “value ids” and SEXPs. Whenever a part of `RAlchemy` (i.e. usually the `RtoAIRConverter`) wants to introduce a value proxy, this can be achieved using the function `ValRepository_addVal()` that requires an SEXP as argument and that returns the id that can be used in the proxy.

The Value Service provides the following operations:

`ValSvc.getVal` This operation returns the AIR representation of a value. Warning: the value is serialized, regardless of its size. The first and only argument of this operation is the value id that is associated with a value. On success, the XML serialized AIR expression is contained in the first response part.

`ValSvc.setVal` (PLANNED) This is the inverse operation to `ValSvc.getVal`. By using this operation, a client can set the value that is associated with a value id to an AIR expression that is provided as argument. The first argument of this operation is the id of the value to be imported or “-1” if a new value shall be created. The second argument is the XML representation of an AIR expression that shall be stored as the value. On success, the operation provides the id of the stored value in the first response part.

`ValSvc.publish` (PLANNED) This operation makes the Value Service “publish” a value avoiding its serialization. “Publishing” in this sense means to make it available to a client in some way, e.g. using Shared Memory. The first argument of this operation is the id of the value to be published. The second argument designates the publishing channel, e.g. “shared_mem”. The following arguments are specific to the publishing channel and documented accordingly. On success, message parts contain information on the published value that is specific to the publishing channel.

`ValSvc.slurp` (PLANNED) This is the inverse operation to `ValSvc.publish`, meaning that a client wants `RAlchemy` to (synchronously) import a value over a channel. The first argument of this operation is the id of the value to be imported or “-1” if a new value shall be created. The second argument specifies the import channel, e.g. “shared_mem”. The following arguments are specific to the publishing channel and documented accordingly. On success, the operation provides the id of the stored value in the first response part.

5.3.2 Environment Service

The Environment Service provides access to R’s symbol environment(s). Within `ALCHEMY`, the Environment Service is realized by the `AlchemyEnvironmentSvc` that uses an `EnvironmentRepository` (see 5.10). The `EnvironmentRepository` is composed of a list of `EnvironmentSEXP` instances, i.e. of SEXPs of type `ENVSEXP`. With the different environments it is possible to represent and make accessible, e.g. the local environment of a closure along with the global interpreter environment.

The Environment Service provides the following operations:

`EnvSvc.lookup` This operation returns an AIR expression that is associated with a symbol. The first argument of this call is the environment id of the environment to look the symbol up in. The second argument is the name of the symbol to look up. On success, the XML serialized AIR expression is contained in the first response part. If the “length” of the value exceeds a certain limit, i.e. 256 in the current implementation, the value is not serialized but returned as a proxy, giving clients a chance to access the value by other means than serialization. The meaning of “length” depends on the value type, e.g. the number of elements for `AIRVector`.

`EnvSvc.install` (PLANNED) This operation installs an AIR expression with a given symbol in an environment. The first argument is the environment to install the symbol in. The second argument is the XML representation of the AIR expression to be installed. If a Value Proxy is given as the second argument, the proxy is dereferenced and the actual value is stored in the environment.

5.3.3 AIRtoR Service

The AIRtoR Service provides the R representation to a given AIR expression. The service applies the `AIRtoRConverter` of `RAIchemy` followed by a call to R’s built-in `deparse()` function that converts an SEXP into an R string.

AIRtoR Service provides the following operation:

`AIRtoR.convert` Converts an AIR expression string into an R expression string. The first argument is the XML representation of the AIR to convert. On success, the operation returns the resulting R expression in the first response part.

5.3.4 R Service Client: `EnvServiceProxy` class in `AlchemyCore`

`SymbolExpr` instances in AIR consist of a name and an optional namespace. To lookup an AIR expression that is associated with a name, `AlchemyCore` provides one or more `AIREnvironments`. For instance, every instance of `AIRprogram` has one main `AIREnvironment` that represents the R symbol table at the instant when transmutation has been triggered. This `AIREnvironment` instance is created when the `AIRprogram` is generated from its XML representation. The environment id is encoded as an XML attribute, as shown in the following AIR fragment:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <AIR environment-proxy="tcp://127.0.0.1:1985" value-proxy="tcp:
   //127.0.0.1:1985" environment-id="154445984">
3   <Program>
4   [...]
```

The decision to instantiate a concrete `ZMQProxyEnvironment` is currently hard-coded in the factory of the `AIRprogram`. If there will be more communication types in the future, an optional attribute, e.g. “envservertype”, could be added to the AIR XML representation or the proxy type could be (heuristically) deduced from the value of the “environment-proxy” attribute.

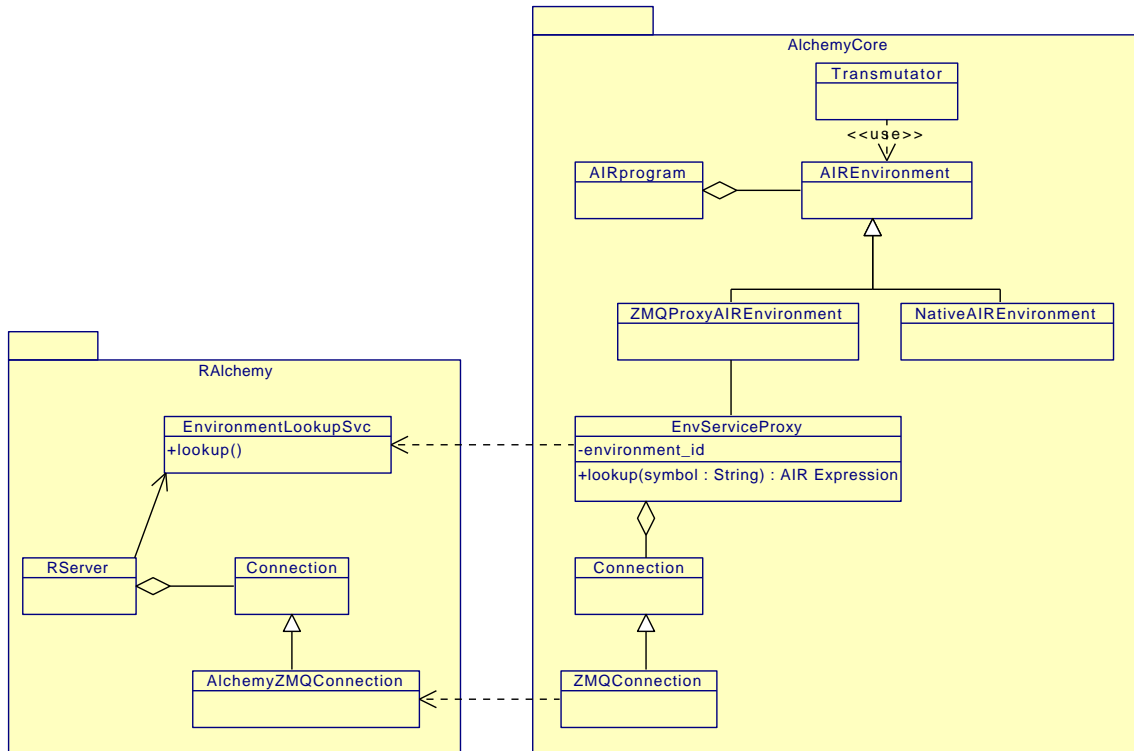


Figure 5.12: Classes involved in environment lookup

Class diagram 5.12 shows the main classes from `AlchemyCore` and `RAlchemy` that participate in the lookup of a symbol. While `AIREnvironment` provides the interface `EnvServiceProxy` represents the communication *strategy* (see [GHJV94]).

5.3.5 R Service Client: ValueServiceProxy in AlchemyCore

Compound `AIRValues` such as `AIRVector` contain a *storage strategy* that determines how and where the actual data of the value may be accessed. One possible storage strategy is the `ZMQAIRValueStorage`. The XML representation of an `AIRValue` differs with regard to its storage strategy.

Listing 5.1 shows an excerpt of the resulting AIR XML for a vector with 245 elements that was generated by executing the R command `seq(1,123,0.5)`. Listing 5.2 shows the resulting AIR XML for the command `seq(1,129,0.5)` holding 257 elements (the limit, when to use a proxy, is currently hard-coded to 256 in `RAlchemy`).

As can be seen in 5.2, the complete address to the Value Service is not specified in the proxy but only the value id corresponding to the `AIRValue`.

An `AIRVector` instance delegates to its storage strategy whenever a concrete data access is performed via the object interface. For instance, a call to `AIRVector.asList()` leads to the execution sequence as depicted in figure 5.13. The class relationships for `AIRValues` are shown in class diagram 5.15

Listing 5.1: AIRVector without proxy

```

1 <ConstantExpr>
2   <AIRVector basetype="real">
3     <Data data="
4       1.000000,1.500000,2.000000,2.500000,3.000000,3.500000,
5       4.000000,4.500000,5.000000,5.500000,6.000000,6.500000,7.000000,
6       7.500000,8.000000,11.000000,11.500000,12.000000,12.500000,
7       13.000000,13.500000,14.000000,14.500000,15.000000,15.500000,
8       [...]
9       116.500000,117.000000,117.500000,118.000000,118.500000,
10      119.000000,119.500000,120.000000,120.500000,121.000000,
11      121.500000,122.000000,122.500000,123.000000" length="245"/>
12   </AIRVector>
13 </ConstantExpr>

```

Listing 5.2: AIRVector with proxy

```

1 <ConstantExpr type="Vector">
2   <AIRVector basetype="real">
3     <ZMQValueProxy id="851319081"/>
4   </AIRVector>
5 </ConstantExpr>

```

5.4 AIR Design

Chapter 2 has already introduced into the high-level requirements of the AIR language. An instance of an AIR tree represents an abstract syntax tree of the source, i.e. R, program under analysis.

Class diagram 5.14 shows the most relevant classes for AIR.

AIRset A list (not a set⁵) of AIRprograms. Realized the *AIR Set* analysis class. An AIRset contains a (possibly empty) list of string-based key-value pairs that are called “labels” and that may be used, e.g. in transmutation configuration.

AIRprogram Realizes the *AIR Program* analysis class. Its instance is composed of a root AIRExpr that represents the root of the AST, a list of labels, an AIREnvironment, and a QueryModel that has no domain-specific meaning but is used to realize the query() method.

AIRNode All elements of an AIR tree are of type AIRNode. Every AIRNode knows its parent in the AIR tree and the AIRprogram it is part of. AIRNode offers many methods to facilitate working with AIR expressions, e.g. query() or replaceBy().

AIRExpr AIRExpr are AIRNodes that can be evaluated. Currently, it is not clear, whether for ALCHEMY the distinction between AIRNode or AIRExpr offers any

⁵The name is misleading and will be changed in refactoring

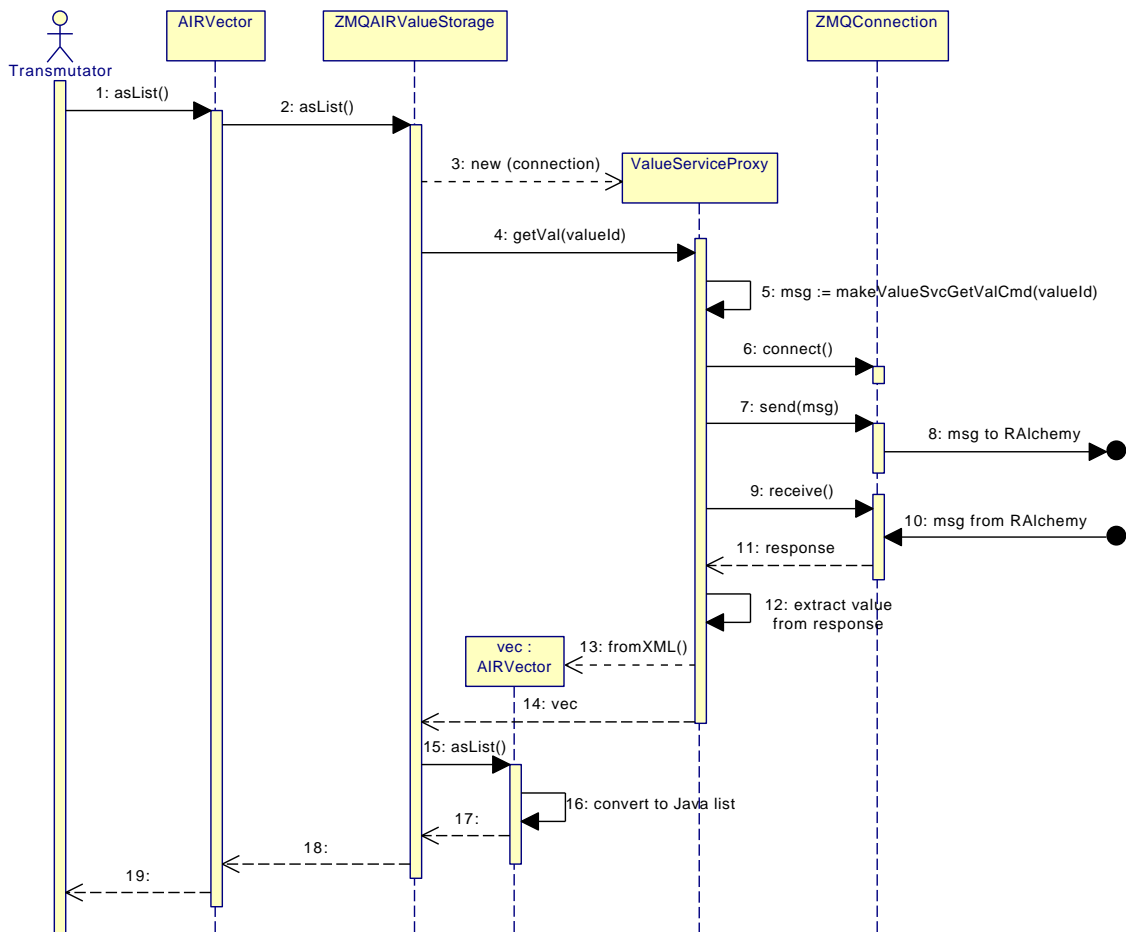


Figure 5.13: Execution sequence for a call to an AIRVector with proxy storage strategy

benefit. In future releases, `AIRExpr` or `AIRNode` may be omitted. The many subclasses of `AIRExpr` that represent the AIR language elements have been introduced earlier in 4.3.1.

AIRType The type of an `AIRExpr`. For compound types such as `AIRVector`, the `AIRType` designates the base type of the individual elements.

AIREnvironment Represents the symbol table of R at the instant the given R expression has been converted to AIR. See 5.12 for a more detailed explanation.

QueryModel A representation of the `AIRprogram` that makes querying the AIR tree possible. See 5.5 for a description.

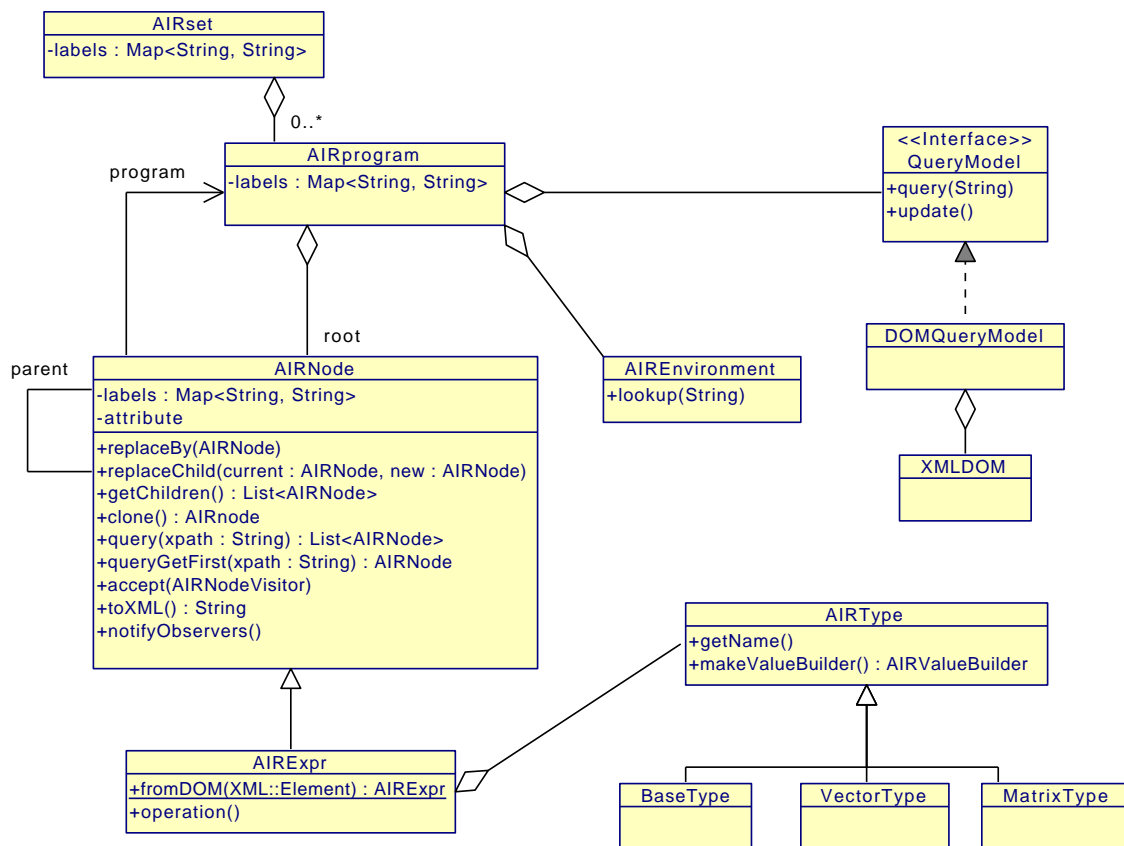


Figure 5.14: AIR Programs

5.4.1 Types

AIR provides the following primitive types:

- Basic types for integers, floating point data, strings, boolean and complex values
- Vector types for all basic types.
- Matrix types for all basic types.

- Functional types representing AIR expressions.

5.4.2 Values and Storages

AIR programs must be easily transferred between different processes that may live on different systems. As values in AIR programs tend to contain lots of data, unnecessary copying of value data between processes must be avoided. For this reason, AIR values make heavy use of *virtual proxies*, called “storages”. Storage proxies are described in section 5.3.5.

The following Storage types are created for this thesis or planned for future releases:

ZMQAIRValueStorage The actual value can be retrieved from the given proxy server using a ZMQ connection. The value is identified by an ID.

SharedMemoryStorage (PLANNED) The actual value can be retrieved by accessing a shared memory region.

The design of AIRValue and its subclasses, as shown in class diagram 5.15, makes it easy to introduce new proxy types.

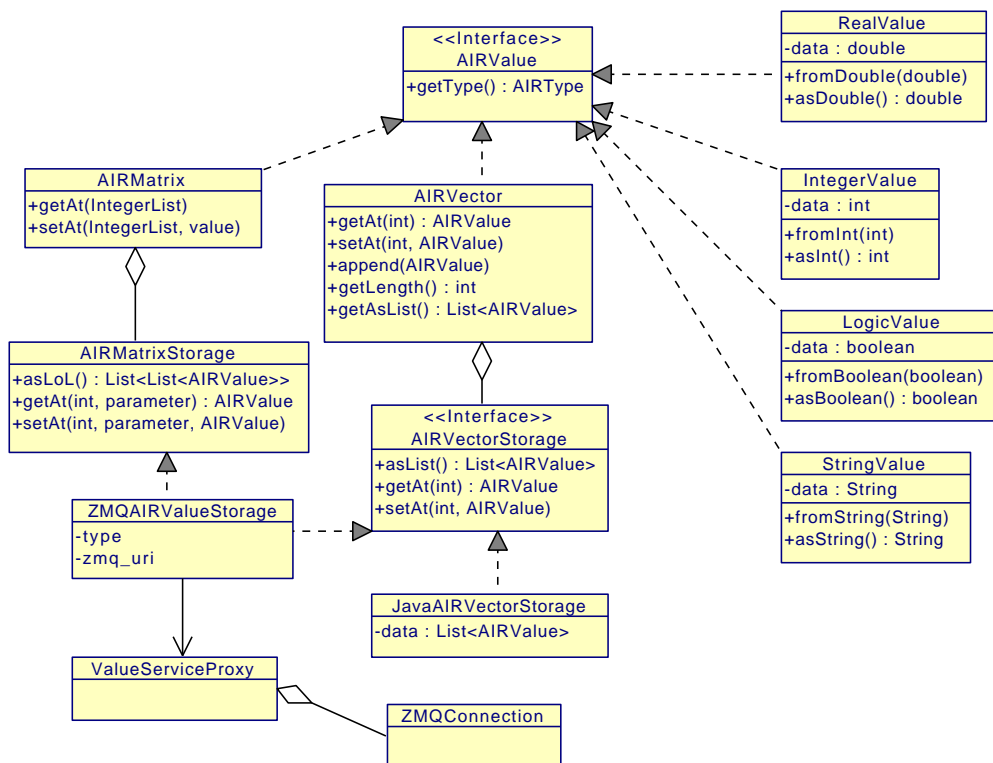


Figure 5.15: Value class hierarchy

5.5 The AIR Interface

One of the reasons for creating AIR is that it should be easy for experimenters to programmatically process AIR programs and program fragments. For that reason, AIR

offers an interface that helps programmers to find and modify AIR trees.

In addition to the accessors of the individual `AIRExpr` objects that allow access to its direct children, every `AIRNode` offers the following operations:

`List<AIRNode> query(String xpath)` The call returns all `AIRNodes` of the document that match the XPath expression (see [W3C07]).

`void replaceBy(AIRNode)` The call replaces the current `AIRNode` in the AIR tree and performs any actions that are necessary to bring the AIR tree back into a sane state, e.g. in particular it updates the `QueryModel`.

`void replaceChild(AIRNode curchild, AIRNode newchild)` That call replaces a child of an `AIRNode` by a new one and brings AIR management back in order.

The `query()` operation bases on the “canonical XML representation” of an AIR expression as described in section 4.3.2. All xpath strings that conform to the XPath standard can be used to selecting AIR nodes in a tree.

The `DOMQueryModel` is the only `QueryModel` currently implemented in ALCHEMY. It uses the XML library `libxml2` (see [Gno]) and works by holding a parallel representation of the current AIRprogram as DOM Document in memory. The DOM document must be updated, whenever changes to the AIR tree have been performed.

Sequence diagram 5.16 illustrates the execution of a `query()` call. As can be seen in step 4, the AIR model is updated at query time (and only then), as I assume that modifying an AIR node might be a more frequent operation than querying the AIR. In `updateAIRModel()` the expensive update operation is not executed if the model has not been marked “dirty” by an earlier modification operation.

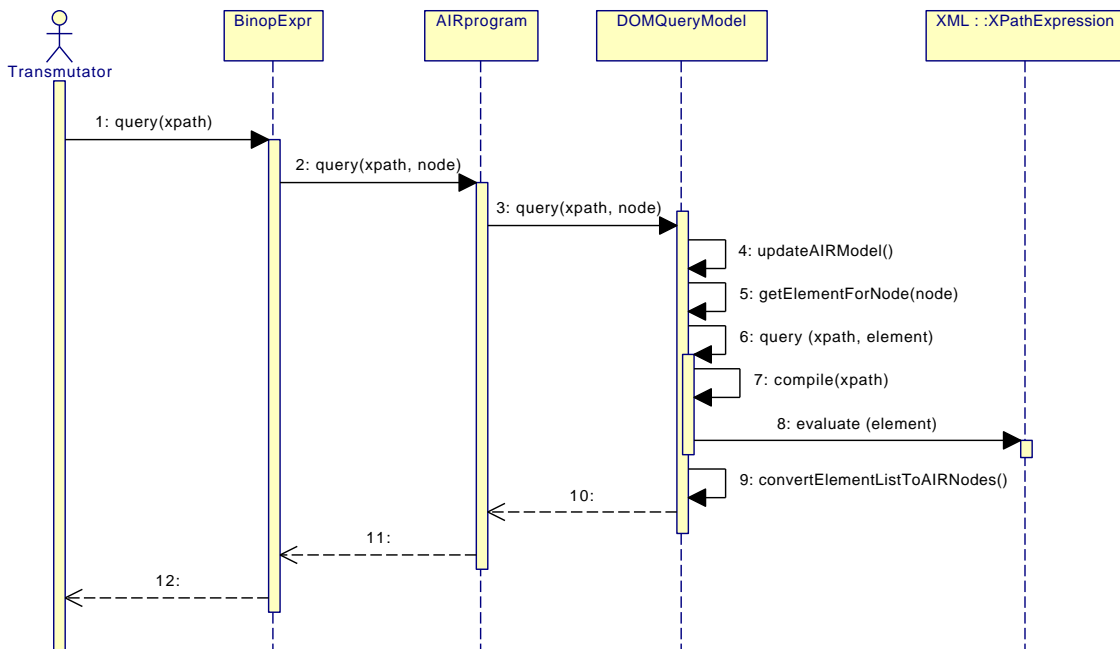


Figure 5.16: Call sequence for `query()`

5.5.1 Conversion to/from AIR XML

Conversion from a given AIR expression or AIR program is performed by the class `AIRNodeDOMVisitor`. `AIRNodeDOMVisitor` implements “visit” methods for all subclasses of `AIRExpr` that contain code to maps the given subclass instance to an XML DOM Element.

Listing 5.3: Example of visit method in `AIRNodeDOMVisitor.java`

```

1     public void visit (SymbolExpr node) {
2         Element symbol_elem = (Element) _doc.createElement("
           SymbolExpr");
3         symbol_elem.setAttribute("name", node.getName());
4
5         getNodeElementMap().put(node, symbol_elem);
6         getElementNodeMap().put(symbol_elem, node);
7
8         setCurElem(symbol_elem);
9     }
```

Conversion from XML elements to AIR objects is carried out by a factory method called `fromDOM()` that is provided by all subclasses of `AIRExpr`.

5.6 Transmutation

The ALCHEMY transmutation logic is realized by `DefaultTransmutationController` that is used by the `TransmutationService` class⁶ for serving `transmutate()` requests by `RAIchemy`.

See section 4.5 for a high-level explanation of the transmutation workflow and basic concepts.

When a new `transmutate()` request arrives, the `DefaultTransmutationController` first initializes a new instance of the `Config` class using a static XML-based configuration file (see C.1) that has the hard-coded name “`txmut_control.xml`” and is loaded as a Java `Resource`. Next, a pseudo Transmutator called `StartTransmutator` is instantiated that serves as the (artificial) entry point into the transmutation process.

Class diagram 5.18 shows the design classes that participate in the transmutation process activity diagram 5.17 visualizes the steps that are performed during transmutation.

What makes that simple workflow flexible enough for the requirements of an experimentation laboratory is the realization of the `Config` class. Class diagram 5.19 shows the classes that are associated with the transmutation config. `Config` provides an `evaluate()` method that processes all `Rules` that are associated with it. See section 4.5.2 for a general description of rule processing in the transmutation configuration.

One of the most important design criteria for the `Config` class was facilitating extension and variation. For that reason, transmutation configuration can be dynamically extended with new functionality. Appendix C.2 shows an example of the “type environment” configuration file that determines what `Predicates`, `Input Modifiers`, and `Actions` are available in the configuration.

⁶Currently, this is not configurable.

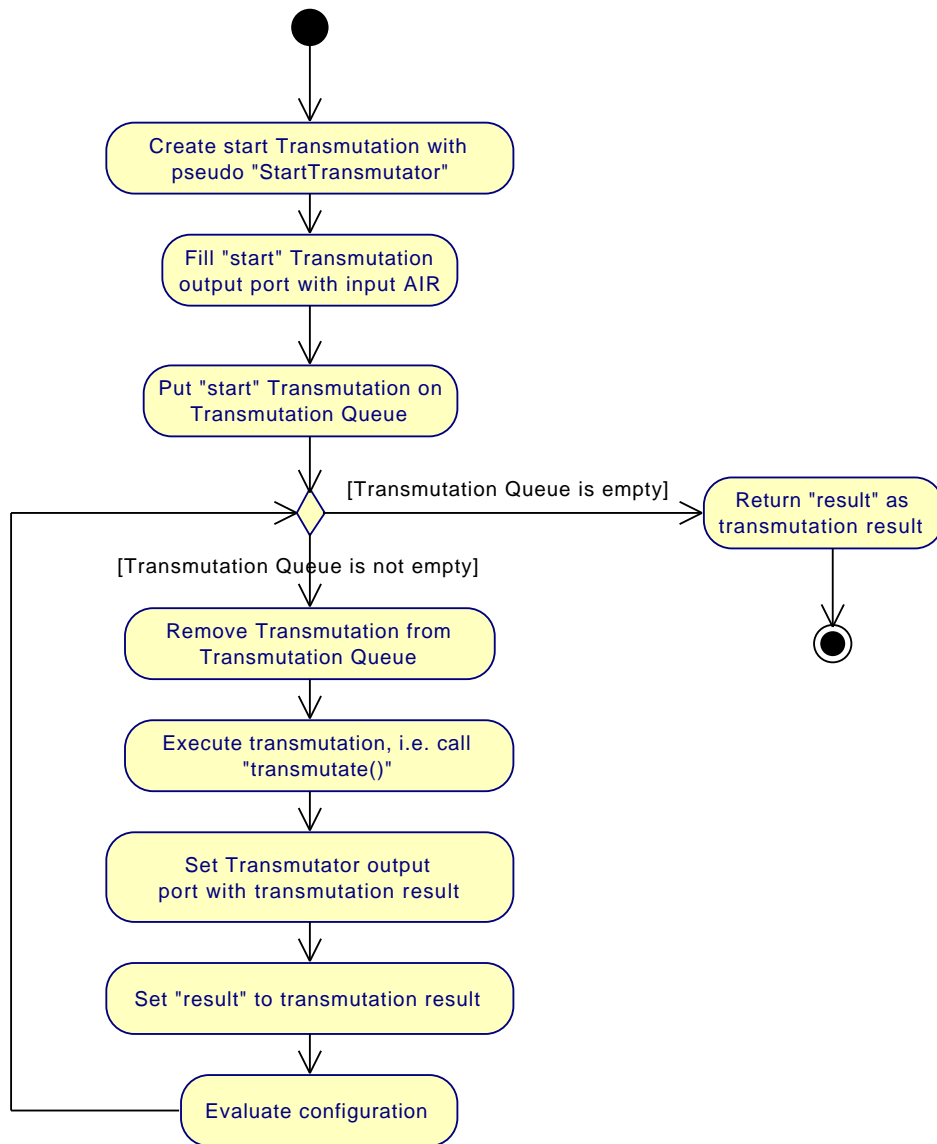


Figure 5.17: Activities involved in transmutation

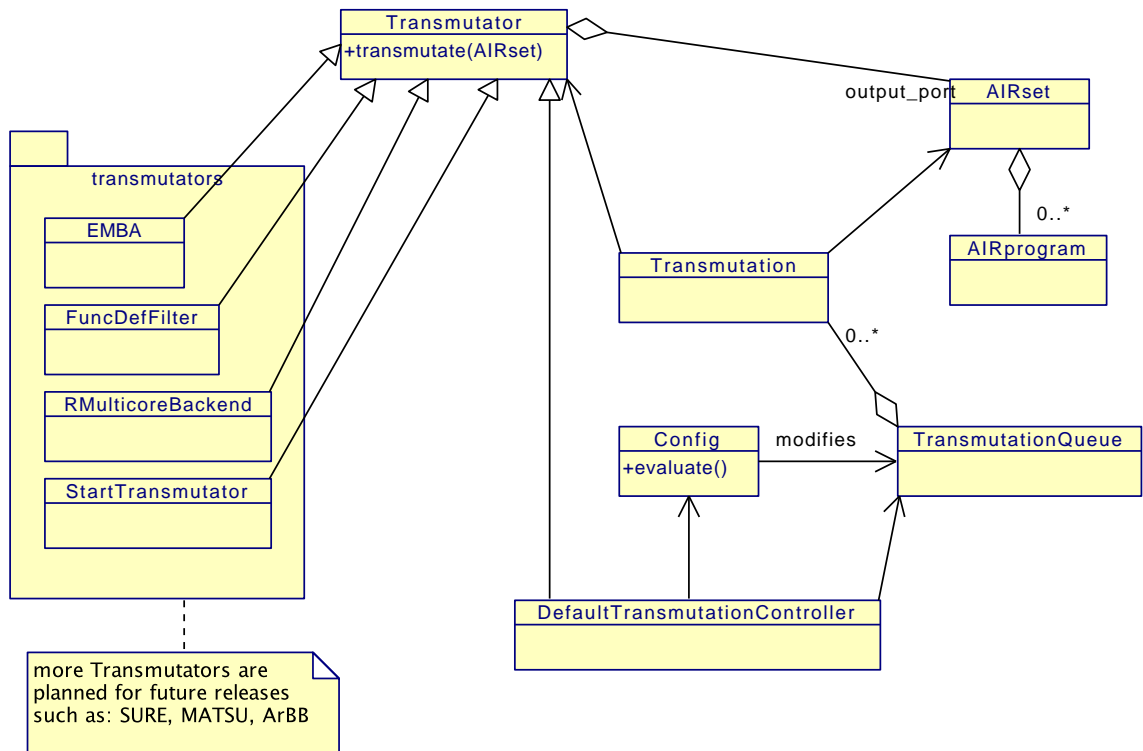


Figure 5.18: Important design classes related to transmutation

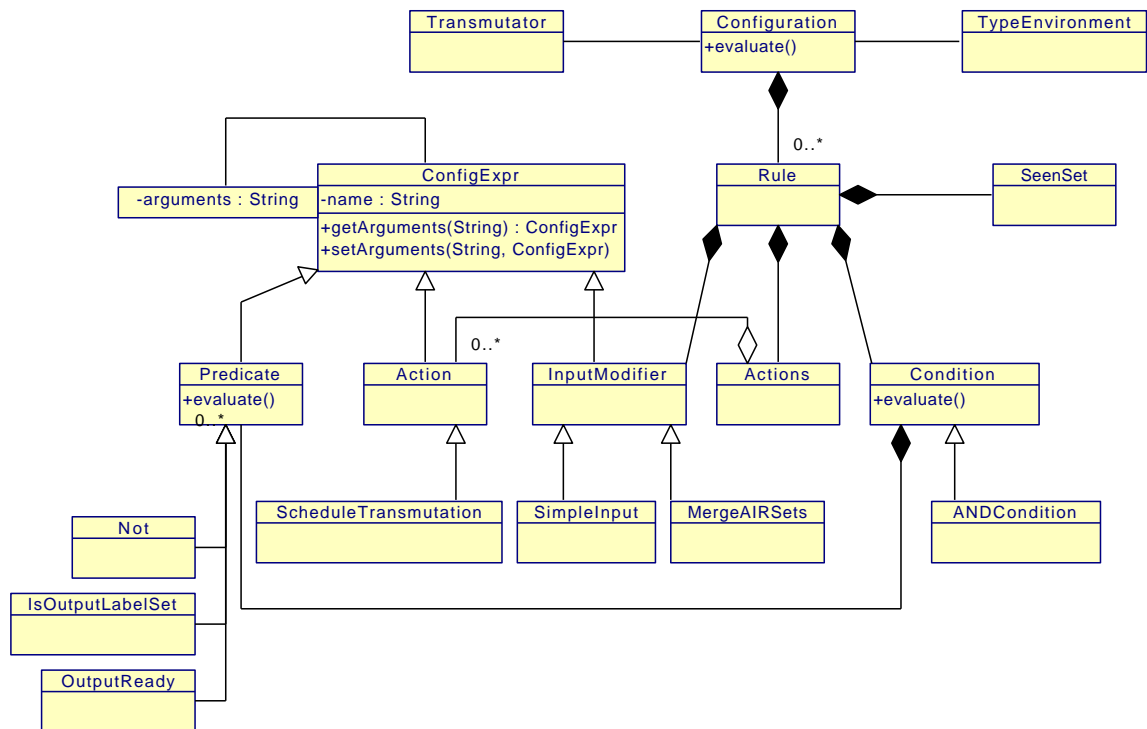


Figure 5.19: Design classes participating in transmutation configuration

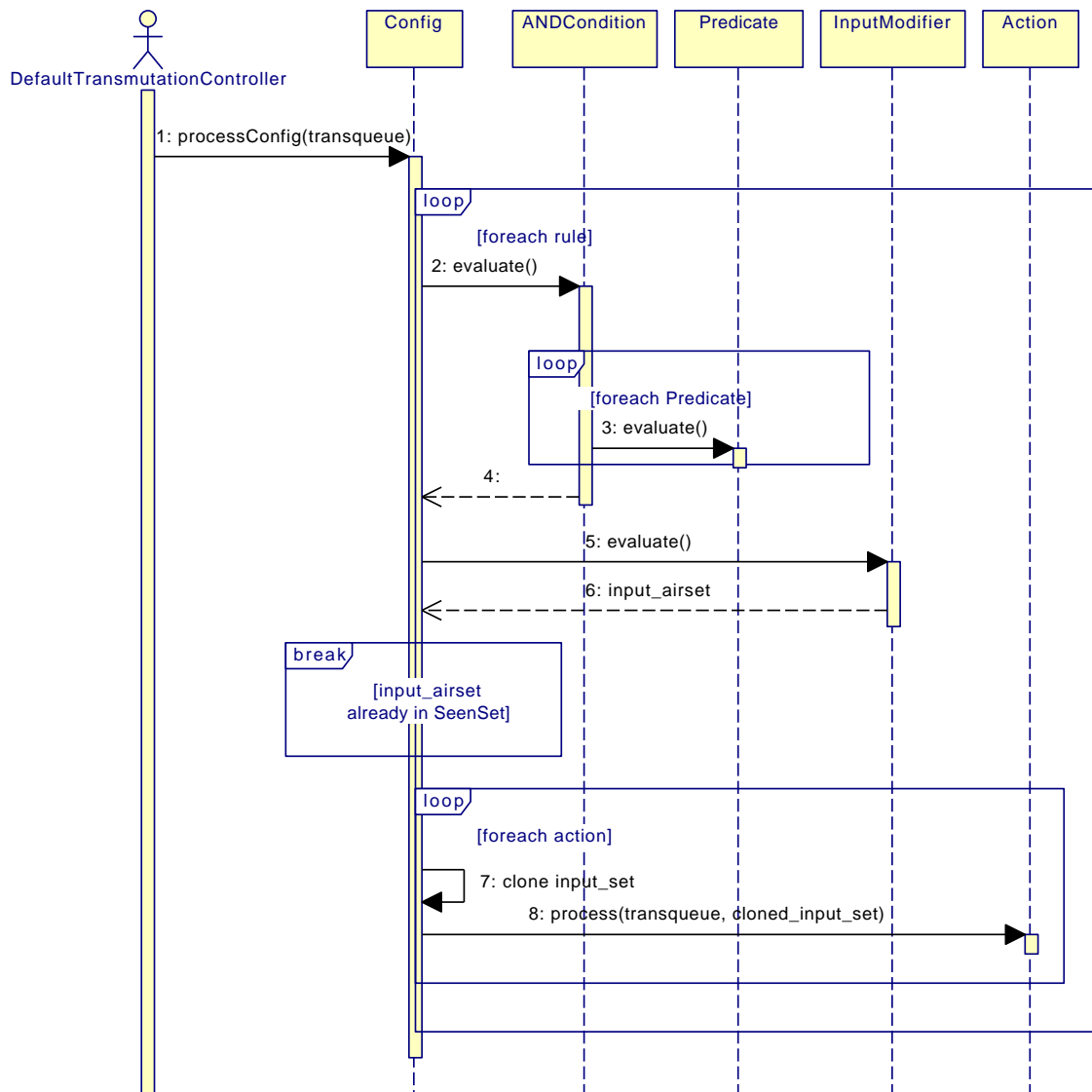


Figure 5.20: Execution sequence for configuration evaluation

All these classes that represent configuration elements inherit from the common superclass `ConfigExpr`, which provides an infrastructure that can be used to access attributes and child elements of the configuration file.

Sequence diagram 5.20 shows the chain of activities that take place when the transmutation configuration is evaluated.

5.7 Planned: Parallelization Backends

Parallelization backends, i.e. (active) components the compute an AIR program entirely or in parts, might be integrated into ALCHEMY by providing backend-specific “adapter transmutators” as sketched in figure 5.21.

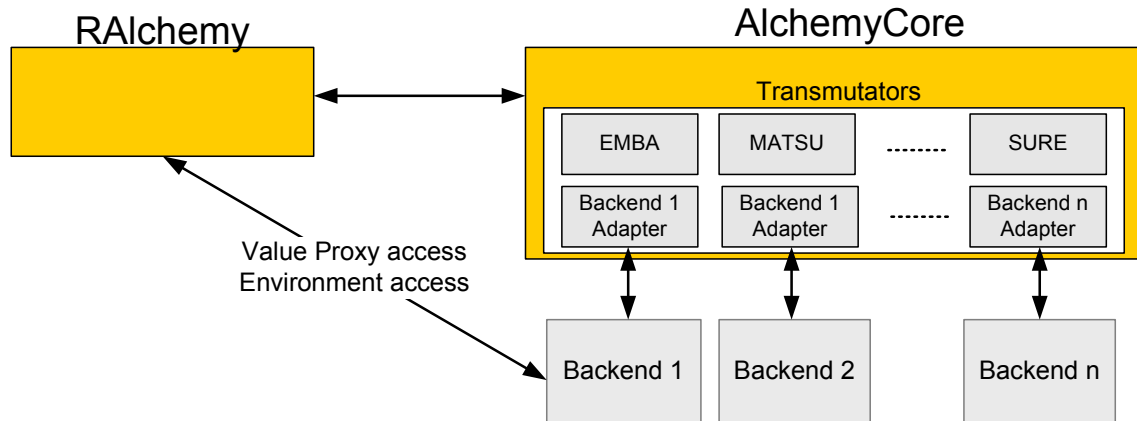


Figure 5.21: Possible backend integration using adapter transmutators

5.8 Planned: Tracing, Single-Step, and Breakpoints

The capability to observe transmutation behavior is paramount to ALCHEMY's functionality. In the current release, these features have been implemented only in a very rudimentary way as debugging messages that can be displayed in R and in the AlchemyCore server console output.

In future releases, this functionality must be completed. There are various technical possibilities how to achieve this goal. First, extending `RServer` by a `ServiceHandler` that accepts logging output and displays it in the R console session is very straightforward.

Giving users the possibility to interrupt transmutation processing at specific points is more difficult. For one, bringing `AlchemyCore` into a suspended state for an indefinite amount of time increases the overall system complexity considerably, as situations such as a crashed client process must be handled, properly. From an architectural point of view this point becomes even more demanding when ALCHEMY must handle multiple user sessions concurrently as proposed in chapter 9. In this case, the only viable and scalable solution would involve persisting ALCHEMY's transmutation state during breakpoints. This requires major design efforts. The second challenge is to convey transmutation state to the breakpointing R session in a way that allows for close inspection of transmutation behavior.

6 Implementation

As a consequence of ALCHEMY’s design, the implementation has resulted in two major artifacts. The first one being the realization of of the C based `RAlchemy` component and the second one being the Java based `AlchemyCore`.

`RAlchemy` consists of about 15 C modules that contain the code for the design classes that have been identified in chapter 5. The ALCHEMY related parts have 5.000 lines of code. Most of them contribute to the `RtoAIRConverter` and `AIRtoRConverter` and the helper visitors `GraphMLVisitor` and `PrettyprintVisitor`. While implementing `RAlchemy` it has been an important principle to modify the original R code in as few locations as possible to facilitate maintenance of future releases.

`AlchemyCore` is a Maven 2 based “multi-project” project with the following physical sub-projects or packages:

`alchemy-commons` Responsible for functionality that may be used in various location of `AlchemyCore`. For instance, `alchemy-commons` contains the classes of the `AIRNode` hierarchy and the realization of classes of the `Infrastructure` design package, e.g. `Connection`. `alchemy-commons` does not depend on any other ALCHEMY package.

`alchemy-core` Responsible for the realization of the transmutation code and the `AlchemyCore Service` package. `alchemy-core` depends on the ALCHEMY packages `alchemy-commons` and `alchemy-core.transcommon`.

`alchemy-core.transcommon` This package contains classes that are important for the realization of Transmutators.

`alchemy-core.applications` This package contains the active classes that can be run by an ALCHEMY user. In particular, this package contains the `ZMQServer` and a few command line clients that have been used during development. The package depends on `alchemy-core`, `alchemy-commons`, and `alchemy-core.transcommon`.

Figure 6.1 shows the dependencies between these physical packages.

The packages have been divided in two Subversion projects:

- `commons` containing `alchemy-commons` and
- `core` containing `alchemy-core`, `alchemy-core.transcommon`, and `alchemy-core.applications`

to simplify parallel development on those packages. All subversion projects follow the “standard” subversion convention of having a separate subtree for

`trunk` that contains the current development head,

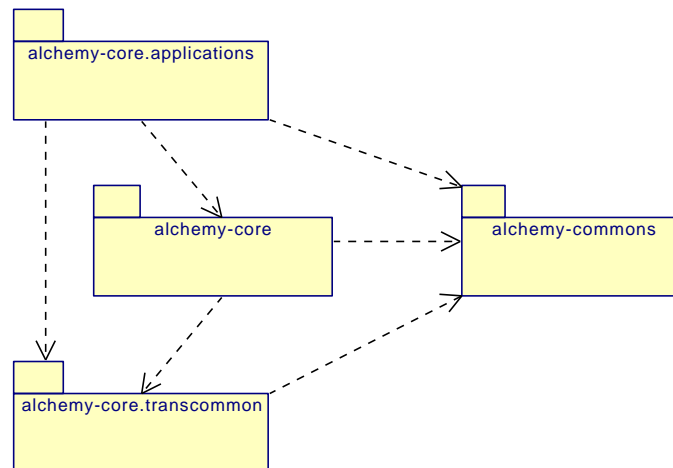


Figure 6.1: Physical AlchemyCore packages

tags that contain development snapshots, e.g. for specific software releases.

branches that contain development branches for feature development or performing bug fixes on specific releases.

AlchemyCore consists of about 130 Java classes that contain about 10.000 lines of code.

7 Transmutators

Every transmutator can roughly be assigned to one or more functional classes:

- *Filter* transmutators identify `AIRprograms` that shall not be modified by `AlchemyCore`. As they decline transmutation globally, they are effectively a dynamic mechanism for transmutation control. `FuncDefFilter` belongs to this class.
- *Utility* transmutators modify the AIR as a service for other transmutators. For instance, they can label `AIRNodes` in a way that can be exploited later or dereference symbolic `FuncCall`.
- *Parallelization Analysis Modules (PAMs)* analyze an AIR and replace parts by parallel skeletons, effectively also acting as a helper for later *Backend Adapters*.
- *Executors* compute parts of an `AIRprogram` and replace the corresponding expressions by the computation results.
- *Backend adapters* act much like *Executors* in that they replace AIR expressions by computed results. However, they do not perform these computations themselves but delegate them to computation backends. Depending on the implementation, these actual backends may be considered *Executor* transmutators, i.e. when they directly act on the AIR, or communicate with the *Backend adapter* in a way that is completely opaque to `ALCHEMY`, e.g. by constructing a CUDA program and using the operating system for their compilation and execution.
- *Transmutation control* transmutators actively influence the transmutation configuration or the labeling of `AIRNodes`, `AIRprograms`, or `AIRsets`.
- *Delegation* transmutators do not perform AIR modifications themselves but use other transmutators to “do the work”. In principle, they can also use and evaluate a local instance of a transmutation `Config`.

A number of transmutators have already been implemented in `ALCHEMY` or are planned for the near future. These transmutators are described in the following sections.

7.1 FuncDefFilter

The `FuncDefFilter` filter transmutator checks whether an `AIRprogram` consists only of an function definition, i.e. an `FuncDef` or an assignment whose right-hand side is a function definition. The transmutator globally declines transmutation in these cases.

`FuncDefFilter` is an optimization in cases where function analysis is performed when they are used instead of when they are defined, e.g. because analyzers want to take the environment or data into consideration. Depending on the overall configuration, this optimization may be useful or not.

7.2 EMBA

EMBA represents a Parallelization Analysis Module (PAM). EMBA analyzes a given AIRprogram for the existence of “embarassingly parallel constructs” such as `apply()`-like functions that uniformly operate on all elements of a data structure.

After EMBA has identified such an expression, it replaces it by the corresponding parallel skeleton “MAP”. The transmutator does not replace subtrees of the AIR that have already been processed by EMBA in order to avoid (possible) inefficiencies with nested parallel skeletons. Future versions of EMBA may make this behavior configurable. The set of AIR expressions that is identified as “embarassingly parallel” is currently hard-coded into the transmutator.

Furthermore, EMBA performs “constant folding” on the `c()` function that is used in R to create vectors from single elements.

Activity diagram 7.1 shows the individual steps that are performed by EMBA. Figure 7.2 shows an example of how EMBA modifies an AIR expression.

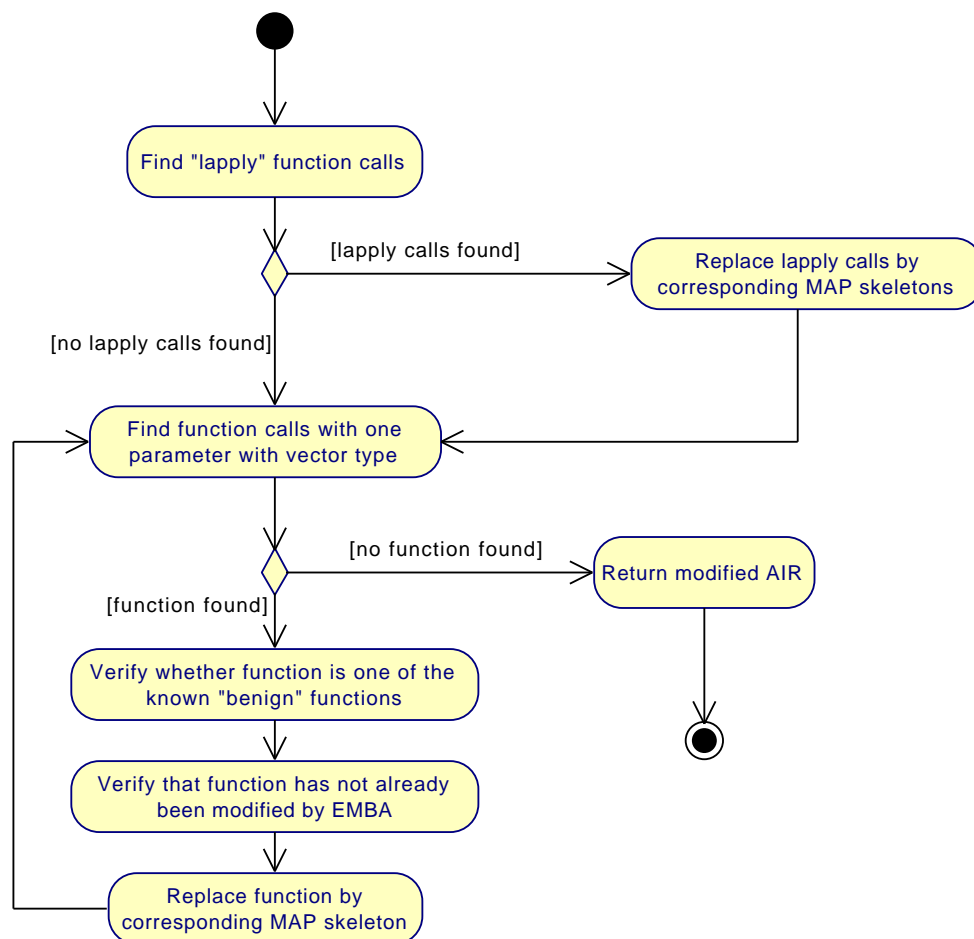


Figure 7.1: EMBA Workflow

7.3 RMulticoreBackend

The `RMulticoreBackend` transmutator identifies MAP skeleton expressions in an AIR and converts them to analogous function calls from the R “multicore” package. The transmutator does also introduce an R call at the beginning of the code to load the “multicore” library.

`RMulticoreBackend` does not fully belong to one of the functional classes listed above.

7.4 Planned Transmutators

The following transmutators are planned or currently worked at:

BackendLabeler A utility transmutator that labels `AIRNodes` in a way that can later be used to assign execution backends to specific subtrees of the AIR. The `BackendLabeler` uses a static configuration file that matches the desired AIR subtrees using XPath expressions.

ExecutionController A delegation transmutator that distributes AIR computation to other Backends depending on the labeling of `AIRNodes`.

MATSU A PAM that is able to transform certain classes of Dynamic Programming problems (“maximum marking problems”) into equivalent programs that use data parallel skeletons. The basic principle behind MATSU is an assignment of input data items to “generations” such that the computation on input items in one generation only depends on items in the same or earlier generations. [KMM⁺05] shows that a good assignment strategy leads to a reformulation of the problem that allows the efficient application of the data-parallel skeletons ZIP, REDUCE, MAP, and SCAN.

SURE A PAM that identifies opportunities to reorder nested loops in a way that partially allows their parallel computation. SURE (as described in [Dar97] and [KMW67]) analyzes the data dependencies of program statements that are used in nested “for” loops and results in a reformulation of the program that contains loops that can safely be computed in parallel.

InteractiveTransmutator A transmutator that allows experimenters to interactively control the transmutation process, query the AIR, and perform modifications at the AIR.

ArBBAdapter A backend adapter that converts a given AIR program that contains skeleton expressions into an equivalent C++ program using the Intel ArBB parallelization library.

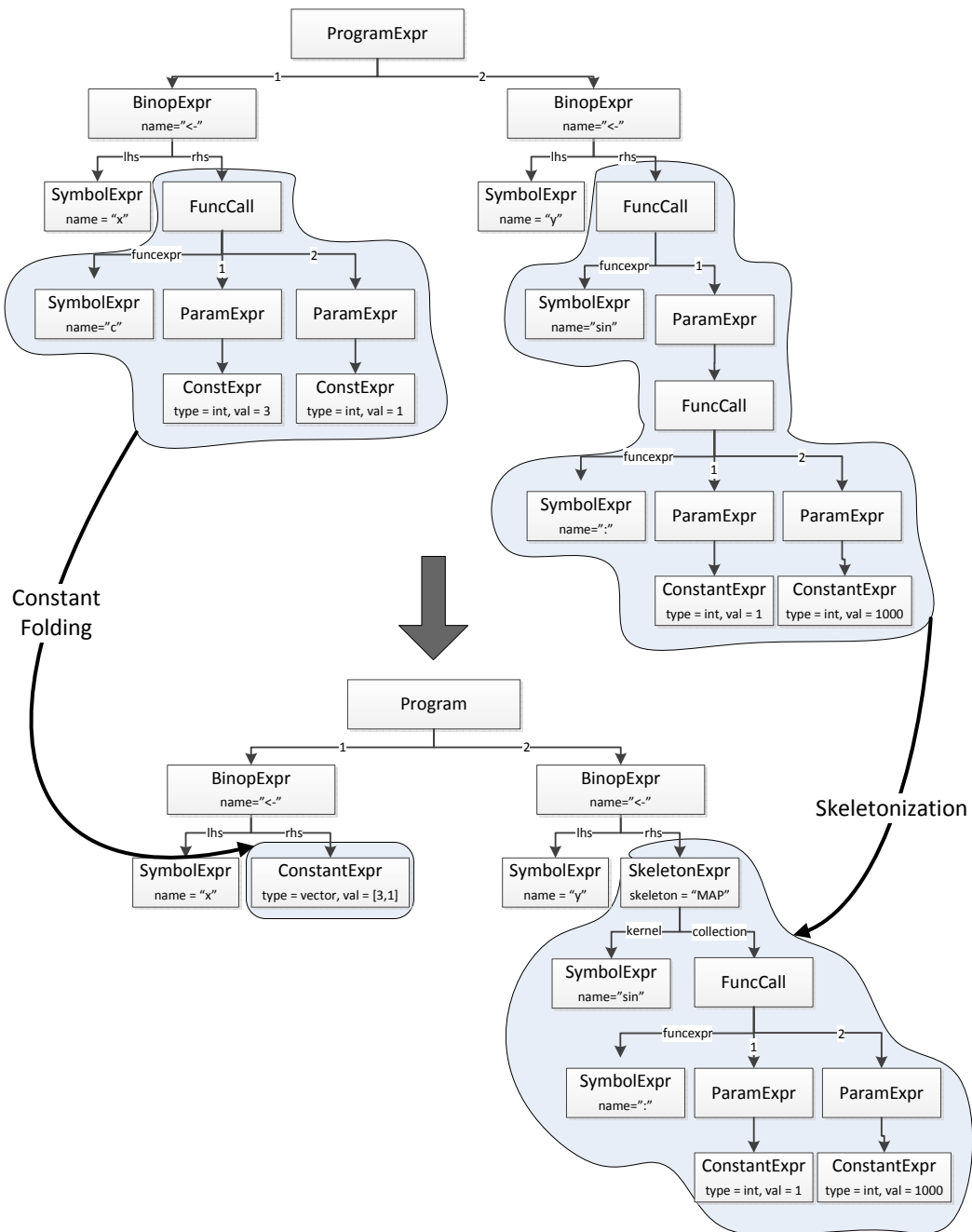


Figure 7.2: Example of an EMBA AIR modification

8 Experimental Evaluation

The following experiments shall reveal if ALCHEMY’s parallelization analysis yields reasonable results, i.e. if analysis results match human expectations with regards to the selection and integration of parallel skeletons. Furthermore, the performance of the resulting, parallelized programs is evaluated.

As ALCHEMY is currently not optimized towards optimizing transmutation speed, no experiments have been performed that measure that behavior. Subjectively, at least for a simple transmutator such as EMBA, transmutation speed is satisfactory, with overall, i.e. ALCHEMY wide, transmutation time well below one second.

8.1 Test Environment

All tests were conducted on a standard PC with the following characteristics:

- Processor: AMD FX-120 with eight CPU cores (64bit)
- 8 GB RAM (2 x 4GB DDR3-1333 MHz)
- Operating System: Ubuntu 11.10 64bit

During the tests, no other programs were running except of system processes.

8.2 End-to-End Parallelization with EMBA and RMulticoreBackend

This experiment verifies the correctness of ALCHEMY parallelization for one example and measures the performance of the parallelized program.

8.2.1 Setup

Listing 8.1 shows a template for the input program that is used for this test. The program is designed to be embarrassingly parallel, i.e. it is easily possible to separate the the program into a number of parallel tasks. In the example program, this separable task is the `helper()` subfunction. Furthermore, each single task is computationally more or less expensive. This assumption has been made to make the results of this experiment stick out more visibly. This decision obviously has an impact on the interpretation of the results and is discussed later. The program computes `helper()` for all integer numbers in the range `[1, ..., < numelements >]`.

AlchemyCore is prepared by configuring a transmutation sequence of `FuncDefFilter` \rightsquigarrow `EMBA` \rightsquigarrow `RMulticoreBackend`. `RMulticoreBackend` has one parameter that represents the number of cores that shall be use by the generated R “multicore” code. Every measurement has been repeated using 1, 2, 4, 6, and 8 cores.

Listing 8.1: Experiment 1 input R program

```

1 myfun <- function (vec) {
2   helper <- function (x) {
3     sum <- 0
4     for (i in 1:100) {
5       sum <- sum + 1/(sin(x) + i*cos(x))^2
6     }
7     return(sum)
8   }
9
10  lapply (vec, helper)
11 }
12
13 alchemy.loglevel(6)
14 alchemy.enable()
15
16 system.time(myfun (1:<numelements>))

```

The program template must be preprocessed before it is runnable. For this, the `<numelements>` parameter must be replaced by a concrete numeric value. In preparation of the experiment, 10 R programs have been instantiated from the template with equidistant values for `<numelements>` ranging from 100,000 to 1,000,000.

Thus, overall 10 program instances have been tested with 5 different Alchemy configurations. To create the same conditions for all test instances, AlchemyCore has been restarted before each run. AlchemyCore was started on the command line with `alchemy -1.0/core/trunk/bin/startserver.sh`. 5 seconds after the restart has been triggered, an R session has been started in batch mode with the shell command `R -q --vanilla <$scriptname with $scriptname being the path to one of the 50 prepared R scripts.`

8.2.2 Execution Analysis

When R is started up, the R interpreter first reads and parses the `myfun()` function definition. It does *not* send the function definition to AlchemyCore because evaluation interception has not been activated at this point. Instead, the assignment to `myfun()` is evaluated by R resulting in the function being added to the environment.

The next expressions, `alchemy.loglevel(6)` and `alchemy.enable()` activate configure ALCHEMY related logging in R and activate the interception logic. From now on, all expressions that are parsed on the command line are sent to RAlchemy via AlchemyAdapter.

Hence, the expression `system.time(myfun (1:<numelements>))` is sent to ALCHEMY.

The first Transmutator that analyzed this expression is `FuncDefFilter`. It ignores the expression, as its filter criterion is not matched, i.e. the expression does not represent a function definition.

The next Transmutator, EMBA, starts with the following input¹:

¹Note, that to improve readability not the actual input to EMBA, which would be an AIR expression, is shown but its corresponding R representation, that has been created using `AIRtoRSvc`

```

1 {
2   system.time(myfun(1:1e+05))
3 }

```

As described in chapter 7, before analysis takes place EMBA substitutes named function calls, i.e. function calls that are designated by their function name, for equivalent anonymous function calls if they can be retrieved via `EnvironmentSvc`. After this transformation, the R representation of EMBA looks like this:

```

1   system.time(function (vec)
2     {
3       helper <- function(x) {
4         sum <- 0
5         for (i in 1:100) {
6           sum <- sum + 1/(sin(x) + i * cos(x))^2
7         }
8         return(sum)
9       }
10      lapply(vec, helper)
11    }(1:1e+05))
12 }

```

EMBA converts the `lapply()` call into an analogous `SkeletonExpr`. Due to EMBA's “only one parallelization per subtree” policy, the possible parallelization of the anonymous function does not take place. EMBA output is:

```

1   system.time(function (vec)
2     {
3       helper <- function(x) {
4         sum <- 0
5         for (i in 1:100) {
6           sum <- sum + 1/(sin(x) + i * cos(x))^2
7         }
8         return(sum)
9       }
10      alchemy.applySkeleton(name = "MAP", collection = vec,
11        kernel = helper)
12    }(1:1e+05))
13 }

```

The next configured transmutator, `RMulticoreBackend`, emulates a complete backend by introducing function calls for the “R multicore” package to be evaluated by the client R interpreter. In this case, it replaces the “MAP” skeleton as suggested by EMBA by a call to `mclapply()`. `mclapply` is given the following arguments: the collection to operate on, the function to execute on collection elements, and the number of multicore CPU cores to use. This last parameter is defined in `RMulticoreBackends` configuration. The following listing shows the result of `RMulticoreBackend` that is also the final output of `AlchemyCore`:

```

1 {
2   library(multicore)
3   system.time(function (vec)
4     {
5       helper <- function(x) {

```

```

6         sum <- 0
7         for (i in 1:100) {
8             sum <- sum + 1/(sin(x) + i * cos(x))^2
9         }
10        return(sum)
11    }
12    mclapply(vec, FUN = helper, mc.cores = 8)
13 } (1:1e+05)
14 }

```

This result is returned to `RAlchemy` that passes it back to `RUserInterface` which proceeds with calling the standard R evaluator on that R expression.

8.2.3 Results

Figure 8.1 shows the running, i.e. wall-clock, times for executing the test programs for the different numbers of cores and problem instances. Please note that data items that are labeled as “1 Core” represent the execution of the normal, i.e. unparallelized, version of the program and not a parallelized execution that is restricted to one core.

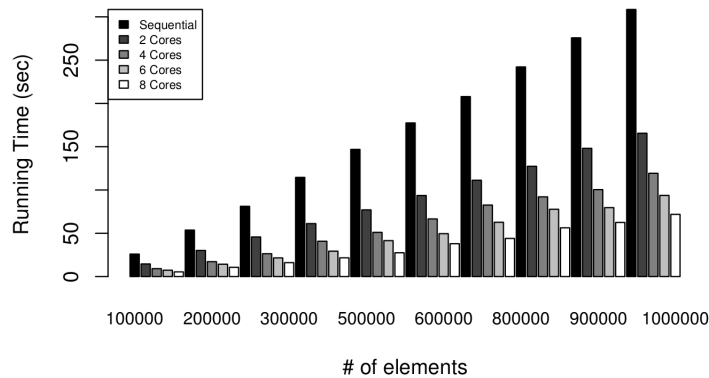


Figure 8.1: Execution times on a multicore machine

Figure 8.2 puts the parallelized execution into relation to a completely sequential execution, i.e. the “Speedup” for a given problem instance is computed as

$$Speedup = \frac{ExecutionTime(P_{serial})}{ExecutionTime(P_n)} \quad (8.1)$$

with $ExecutionTime(P_n)$ being the measured running time of the parallelized R program for the problem instance and $ExecutionTime(P_{serial})$ being the corresponding running time of the sequential program.

The numerical results are listed in appendix E.

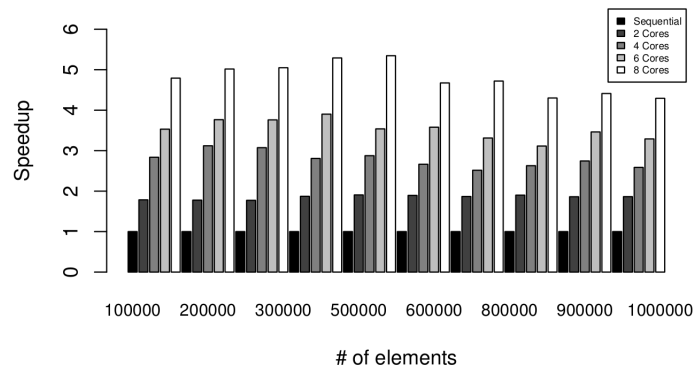


Figure 8.2: Parallel speedups for different numbers of cores for the measured problem instances

9 Conclusion and Outlook

This thesis describes the concepts, design, and realization of the parallelization laboratory `ALCHEMY` and its integration into the R runtime environment. As proof of concept, the transmutators `EMBA` and `RMulticoreBackend` have been created that show that `ALCHEMY` is able to modify existing R code in a way that eventually makes a parallelization of R programs possible.

As the potential of `ALCHEMY` is considered to grow with the number and quality of its Transmutator plugins, a final proof that the project will be able to reach its ambitious goals is still to be presented.

There are multiple ways in that `ALCHEMY` may be improved or extended in the future.

First, the current implementation is still lacking some important features:

- debugging facilities are insufficient for regular users
- not all R types and operators are fully supported
- many operations of the R Services are not implemented like `ValueSvc.publish`
- there is currently only one skeleton node available in the AIR, important types are missing

Additionally, as it has already been pointed out, `ALCHEMY`'s capabilities grow with the availability of transmutators. For the majority of existing automatic parallelization methods, it should be possible to write a corresponding PAM transmutator. Likewise, there exist many powerful parallel middlewares and libraries such as Intel ArBB, OpenMP, CUDA, MPI, etc. that can be used as parallel backends with corresponding backend adapters.

During the implementation of `ALCHEMY` no effort has been put into enhancing the usability of the system. In particular, it would be worthwhile to have a graphical user interface for working with the transmutation configuration or inspecting AIR trees.

Although the transmutation configuration provides the capabilities to be modified dynamically, i.e. at runtime, the configuration is currently static in nature. After some experience has been acquired with the capabilities of static transmutator graphs, it could be interesting to play with the automatic generation of transmutation configurations or more advanced *transmutation control* transmutators that automatically learn from earlier experiences.

Finally, the knowledge of how to parallelize a given program in the best possible way is a valuable good. With `ALCHEMY` it could be possible to build a library of transmutators and transmutation strategies. This library could even be provided as an internet service that is regularly updated by contributors.

Bibliography

- [Adl10] Joseph Adler. *R in a Nutshell - A Desktop Quick Reference*. O'Reilly, 2010.
- [BC90] Guy Blelloch and Siddhartha Chatterjee. Vcode: A data-parallel intermediate language. In *In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.
- [BH93] Richard S. Barr and Betty L. Hickman. Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions. *ORSA Journal On Computing*, 5:2–18, Winter 1993.
- [BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, UK, 2007.
- [Ble95] Guy Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon, 1995.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.
- [Col] Murray Cole. Skeletal parallelism. <http://homepages.inf.ed.ac.uk/mic/Skeletons/>.
- [Col89] Murray Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. MIT Press, Research Monographs in Parallel and Distributed Computing edition, 1989.
- [CWK93] Wolfgang J. Paul Christoph W. Keßler. Automatic parallelization by pattern-matching. *Lecture Notes in Computer Science*, 734(978-3-540-57314-2):166–181, 1993.
- [Dar97] Alain Darté. Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model. Technical Report RR97-26, ENS-Lyon, 1997.
- [DD97] M. Diniz and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19:942–991, 1997.
- [DMI96] B. Di Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proc. 4th IEEE Workshop on Program Comprehension*. Los Alamitos: IEEE Computer Society Press. Citeseer, 1996.

- [Edd09] Dirk Eddelbuettel. Presentation: R HPC Tutorial USER Conference. <http://dirk.eddelbuettel.com/papers/useR2009hpcTutorial.pdf>, 2009.
- [GB98] Sergei Gorlatch and Holger Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.
- [GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [GMS99] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. In *In Proceedings of the 1999 Conference on Parallel Algorithms and Compilation Techniques (PACT) '99*, 1999.
- [Gno] Gnome Project. The XML C parser and toolkit for Gnome. <http://xmlsoft.org>.
- [GP94] Zvi Galil and Kunsoo Park. Parallel algorithms for dynamic programming recurrences with more than $o(1)$ dependency. *J. Parallel Distrib. Comput.*, 21(2):213–222, 1994.
- [Hag95] S.B.J.R. Hagemester. A Pattern-matching Approach for Reusing Software Libraries in Parallel Systems. *First International Workshop on Knowledgebased Systems for the ReUse of Program Libraries*, 1995.
- [HT01] K. Hornik and Luke Tierney. Compiling r: A preliminary report. In *DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, 2001.
- [Inta] Intel. Intel Array Building Blocks. <http://software.intel.com/en-us/articles/intel-array-building-blocks/>.
- [Intb] Intel. Intel Ct Wikipedia. http://en.wikipedia.org/wiki/Intel_Ct.
- [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, Mass., 1999.
- [KBR07] Sriram Krishnamoorthy, Muthu Baskaran, and Uday Bondhugulaj Ramanujam. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN PLDI 2007*, 2007.
- [Keß96] Christoph W. Keßler. Pattern-driven automatic parallelization. *SCIENTIFIC PROGRAMMING*, 5:251–274, 1996.
- [KMM⁺05] Kazuhiko Takehi, Kiminori Matsuzaki, Akimasa Morihata, Kento Emoto, and Zhenjiang Hu. Parallel dynamic programming using data-parallel skeletons. *Proceedings of the 22nd JSSST Conference*, Sep 2005.
- [KMW67] Richard Karp, Raymond Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM*, 14:563–590, 1967.

- [Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.
- [LOMPM05] Rita Loogen, Yolanda Ortega-Mallen, and Ricardo Pena-Mari. Parallel functional programming in eden. *Journal of Functional Programming*, 15:431–475, May 2005.
- [LP10] Mario Leyton and Jose Piquer. Skandium: Multi-core programming with algorithmic skeletons. *Euro-micro PDP 2010*, 2010.
- [Mar96] Robert C. Martin. The dependency inversion principle. *C++ Report*, May 1996.
- [MAS05] MASPLAS05: Mid-Atlantic Student Workshop on Programming Languages and Systems, University of Delaware. *Comparative Survey of Approaches to Automatic Parallelization*, 2005.
- [Mat] Timothy Mattson. Blog: Parallel programming environments: less is more. http://blogs.intel.com/research/2007/10/parallel_programming_environment.php.
- [MH08] M.Poldner and H.Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, Vol. 18(No. 1):117–131, March 2008.
- [MHD09] Stefan Marr, Michael Haupt, and Theo D’Hondt. Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support. In *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages*, pages 3:1–3:2, New York, NY, USA, October 2009. ACM. (extended abstract).
- [MLS07] Xiaosong Ma, Jiangtian Li, and Nagiza F. Samatova. Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing. *IPDPS*, pages 1–6, 2007.
- [MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2004.
- [Ope] OpenMP Architecture Review Board. OpenMP. <http://www.openmp.org>.
- [PF01] Christoph Paum and Robert D. Falgout. Automatic parallelization with expression templates. Technical Report UCRL-JC-146179, Lawrence Livermore National Laboratory, 2001.
- [R Da] R Development Core Team. R Internals. <http://cran.r-project.org/doc/manuals/R-ints.pdf>.
- [R Db] R Development Core Team. The R Project for Statistical Computing. <http://www.r-project.org>.
- [R Dc] R Development Core Team. Writing r extensions. <http://cran.r-project.org/doc/manuals/R-exts.pdf>.

- [SME⁺09] Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. State-of-the-art in Parallel Computing with R. Technical Report 47, Department of Statistics University of Munich, 2009.
- [SPT10] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Engineering parallel applications with tunable architectures. *ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 1, 2010.
- [SW81] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.
- [Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 1967.
- [Uni] Uniprot. Uniprot protein database. <http://www.uniprot.org>.
- [VR00] William Venables and B.D. Ripley. *S Programming*. Springer, 2000.
- [W3C07] W3C XSL/XML Query Working Groups. The XPath 2.0 Standard, 2007.
- [Wika] Wikipedia. Algorithmic skeletons. http://en.wikipedia.org/wiki/Algorithmic_skeleton.
- [Wikb] Wikipedia. Scoreboarding wikipedia. <http://en.wikipedia.org/wiki/Scoreboarding>.
- [Wike] Wikipedia. Tomasulo Wikipedia entry. http://en.wikipedia.org/wiki/Tomasulo_algorithm.
- [ywo] yworks. yEd - Graph Editor. <http://www.yworks.com/en/products%5Fyed%5Fabout.html>.
- [Zer] ZeroMQ Project. Zeromq. <http://www.zeromq.org>.

A Installing ALCHEMY

A.1 Prerequisites

Installation has been tested in the following environment:

- Operating system: Ubuntu 10.04.
- Required Ubuntu packages:
 - libzmq (> 2.1)
 - libx11-dev
 - libxt-dev
 - libxml2-dev
 - maven2
 - git
- Additional requirements:
 - JDK (> 1.6)
 - ZeroMQ Java Binding

A.2 Installation

Installation of ALCHEMY includes the installation of a modified R environment and the Java based ALCHEMY Core subsystem. A source based installation of ALCHEMY is possible using the archive `alchemy_1.0.tgz`. First, extract the archive `alchemy-1.0.tgz` to a directory of your choice. Perform the following steps:

A.2.1 R Installation

- Change to the directory `alchemy-1.0/R-src/R-2.13.1`.
- Execute the command `./configure`. Check output for errors and e.g. install missing items. If ALCHEMY shall be installed locally, use the option `--prefix`. Consult the documentation for further information.
- Execute the command `make`. Check output for errors.
- Execute the command `sudo make install` to perform a system-wide installation (or `make install` if a local installation has been chosen, earlier.)

A.2.2 ZeroMQ Java Binding

First, ensure that `libzmq-dev` is installed on the system. Then download the ZMQ Java Binding using `git`:

```
git clone https://github.com/zeromq/jzmq.git
```

Next, change to the directory where `jzmq` has been cloned to and call `./configure`, `make`, and `sudo make install` in sequence. Now, the ZMQ jar should be installed at `/usr/local/share/java/zmq.jar`.

A.2.3 ALCHEMY Core Installation

First, change to the directory `alchemy-1.0`. Then, execute the following command:

```
1 for i in commons/trunk/alchemy-commons commons/trunk/alchemy-  
   commons.parent/ core/trunk/alchemy-core.parent/ core/trunk/  
   alchemy-core core/trunk/alchemy-core.transcommon/ core/trunk/  
   alchemy-core.applications/;  
2 do  
3     oldpwd=$(pwd); cd $i; mvn install; cd $oldpwd;  
4 done
```

To install the Alchemy ZMQServer, change into the directory `alchemy-1.0/core/trunk/alchemy-core.applications`

Now execute the command `mvn package` that bundles all required libraries and resources into the jar file

```
target/alchemy-core.applications-1.0-SNAPSHOT-jar-with-dependencies.jar
```

B Using ALCHEMY

After ALCHEMY has been installed (see A), the ALCHEMY ZMQServer can be started by executing the script

```
alchemy-1.0/core/trunk/bin/startserver.sh
```

Now start an R that has been compiled with ALCHEMY support. After the interpreter appears on the screen, the user can enable ALCHEMY by executing the command `alchemy.enable()`. After that all R commands are intercepted and sent to the ALCHEMY server.

To disable ALCHEMY, type `alchemy.disable()`. The amount of logging can be adjusted by executing the command `alchemy.loglevel(<n>)` where `<n>` is a number between 1 and 6. The lower the number, the more verbose the ALCHEMY logging.

B.1 ALCHEMY Configuration

All `AlchemyCore` configuration files are located in the directory `alchemy-1.0/core/trunk/conf/`. The following configuration files are available:

`core.xml` Global settings for `AlchemyCore`

`core_environment.xml` Defines what predicates, input modifiers, and actions are available in the transmutation configuration. See section [?] for an example.

`tx_control.xml` Static configuration for `DefaultTransmutationController`

All configuration files are described inline.

C Sample Core Configurations

C.1 Transmutation Controller Configuration

Listing C.1: Example Transmutation Controller Configuration

```
1 <TransmutationConfig>
2   <Transmutators>
3     <Transmutator id="funcdef" class="org.transpar.alchemy.
4       transmutators.FuncDefFilter" />
5     <Transmutator id="emba" class="org.transpar.alchemy.
6       transmutators.EMBA2" />
7     <Transmutator id="matzu" class="org.transpar.alchemy.
8       transmutators.MATZU" />
9     <Transmutator id="sure" class="org.transpar.alchemy.
10      transmutators.SURE" />
11    <Transmutator id="emba2" class="org.transpar.alchemy.
12      transmutators.EMBA2" />
13    <Transmutator id="executor" class="org.transpar.alchemy.
14      transmutators.Executor" />
15  </Transmutators>
16  <Rules>
17    <Rule>
18      <Condition>
19        <!-- list of conditions. implicitly "ANDED" -->
20        <OutputReady at="start" />
21      </Condition>
22      <Action>
23        <!-- by default, every action is called with
24          exactly one argument (named "arg1"),
25          i.e. the output of the first OutputReady
26          condition -->
27        <ScheduleTransmutation transmutator="funcdef" />
28      </Action>
29    </Rule>
30
31    <Rule>
32      <Condition>
33        <OutputReady at="funcdef" />
34      </Condition>
35      <Action>
36        <ScheduleTransmutation transmutator="emba1" />
37        <ScheduleTransmutation transmutator="matzu" />
38        <ScheduleTransmutation transmutator="sure" />
39      </Action>
40    </Rule>
```

```

33
34     <Rule>
35         <Condition>
36             <OutputReady at="matzu" />
37         </Condition>
38         <Action>
39             <ScheduleTransmutation transmutator="emba2" />
40         </Action>
41     </Rule>
42
43     <Rule>
44         <Condition>
45             <OutputReady at="emba1" />
46             <OutputReady at="emba2" />
47             <OutputReady at="sure" />
48         </Condition>
49         <Input>
50             <MergeAIRsets strategy="unify_by_id">
51                 <Output at="emba1" />
52                 <Output at="emba2" />
53                 <Output at="sure" />
54             </MergeAIRsets>
55         </Input>
56         <Action>
57             <ScheduleTransmutation transmutator="executor" />
58         </Action>
59     </Rule>
60
61     <Rule>
62         <Condition>
63             <OutputReady at="executor" />
64             <Not><OutputLabelSet at="executor" label="done" />
65             </Not>
66         </Condition>
67         <Action>
68             <ScheduleTransmutation transmutator="executor" />
69         </Action>
70     </Rule>
71 </Rules>
</TransmutationConfig>

```

C.2 Type Environment Configuration

Listing C.2: Example R Program

```

1 <Environment>
2     <!-- Predicates -->
3     <Entry name="IsOutputLabelSet" class="org.transpar.alchemy.
4         core.transmutationcfg.predicates.IsOutputLabelSet" />
5     <Entry name="OutputReady" class="org.transpar.alchemy.core.
6         transmutationcfg.predicates.OutputReady" />

```

```
5     <Entry name="Not" class="org.transpar.alchemy.core.
      transmutationcfg.predicates.Not" />
6
7     <!-- Input Modifiers -->
8     <Entry name="SimpleInput" class="org.transpar.alchemy.core.
      transmutationcfg.inputmod.SimpleInput" />
9     <Entry name="MergeAIRsets" class="org.transpar.alchemy.core.
      transmutationcfg.inputmod.MergeAIRsets" />
10
11    <!-- Actions -->
12    <Entry name="ScheduleTransmutation" class="org.transpar.
      alchemy.core.transmutationcfg.actions.
      ScheduleTransmutation" />
13 </Environment>
```

D Example AIR XML Representation

Original R program:

Listing D.1: Example R Program

```
1 a <- 3
2 while (a < 10) {
3     x <- c(3,1,4,1,5);
4     for (i in x) {
5         a <- a + i;
6     }
7     for (j in x) {
8         a <- a * j;
9     }
10 }
```

XML representation of corresponding AIR (a visualization of this AIR can be found at 4.7).

Listing D.2: Example AIR XML representation for D.1

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <AIR environment-proxy="tcp://127.0.0.1:1985" value-proxy="tcp:
  //127.0.0.1:1985" environment-id="154445984">
3   <Program>
4     <ClosureExpr>
5       <formals/>
6       <body>
7         <ExprList>
8           <BinopExpr op="&lt;->"
9             <lhs>
10              <SymbolExpr name="a"/>
11            </lhs>
12            <rhs>
13              <ConstantExpr type="real">
14                <RealValue data="3.000000"/>
15              </ConstantExpr>
16            </rhs>
17          </BinopExpr>
18        <WhileStmt>
19          <condition>
20            <FuncCall>
21              <funcexpr>
22                <SymbolExpr name="&lt;"/>
23              </funcexpr>
24            <params>
25              <ParamExpr>
```

```
26         <SymbolExpr name="a"/>
27     </ParamExpr>
28     <ParamExpr>
29         <ConstantExpr type="real">
30             <RealValue data="10.000000"/>
31         </ConstantExpr>
32     </ParamExpr>
33 </params>
34 </FuncCall>
35 </condition>
36 <body>
37     <ExprList>
38         <BinopExpr op="&lt;->"
39             <lhs>
40                 <SymbolExpr name="x"/>
41             </lhs>
42             <rhs>
43                 <FuncCall>
44                     <funcexpr>
45                         <SymbolExpr name="c"/>
46                     </funcexpr>
47                     <params>
48                         <ParamExpr>
49                             <ConstantExpr type="real">
50                                 <RealValue data="3.000000"/>
51                             </ConstantExpr>
52                         </ParamExpr>
53                         <ParamExpr>
54                             <ConstantExpr type="real">
55                                 <RealValue data="1.000000"/>
56                             </ConstantExpr>
57                         </ParamExpr>
58                         <ParamExpr>
59                             <ConstantExpr type="real">
60                                 <RealValue data="4.000000"/>
61                             </ConstantExpr>
62                         </ParamExpr>
63                         <ParamExpr>
64                             <ConstantExpr type="real">
65                                 <RealValue data="1.000000"/>
66                             </ConstantExpr>
67                         </ParamExpr>
68                         <ParamExpr>
69                             <ConstantExpr type="real">
70                                 <RealValue data="5.000000"/>
71                             </ConstantExpr>
72                         </ParamExpr>
73                     </params>
74                 </FuncCall>
75             </rhs>
76         </BinopExpr>
77     </ForStmt>
78 </condition>
```

```
79         <IteratorExpr>
80         <itervar>
81             <SymbolExpr name="i"/>
82         </itervar>
83         <collection>
84             <SymbolExpr name="x"/>
85         </collection>
86     </IteratorExpr>
87 </condition>
88 <body>
89     <ExprList>
90         <BinopExpr op="&lt;->"
91         <lhs>
92             <SymbolExpr name="a"/>
93         </lhs>
94         <rhs>
95             <FuncCall>
96                 <funcexpr>
97                     <SymbolExpr name="+"/>
98                 </funcexpr>
99                 <params>
100                     <ParamExpr>
101                         <SymbolExpr name="a"/>
102                     </ParamExpr>
103                     <ParamExpr>
104                         <SymbolExpr name="i"/>
105                     </ParamExpr>
106                 </params>
107             </FuncCall>
108         </rhs>
109         </BinopExpr>
110     </ExprList>
111 </body>
112 </ForStmt>
113 <ForStmt>
114     <condition>
115         <IteratorExpr>
116         <itervar>
117             <SymbolExpr name="j"/>
118         </itervar>
119         <collection>
120             <SymbolExpr name="x"/>
121         </collection>
122     </IteratorExpr>
123 </condition>
124 <body>
125     <ExprList>
126         <BinopExpr op="&lt;->"
127         <lhs>
128             <SymbolExpr name="a"/>
129         </lhs>
130         <rhs>
131             <FuncCall>
```

```
132         <funcexpr>
133           <SymbolExpr name="*" />
134         </funcexpr>
135         <params>
136           <ParamExpr>
137             <SymbolExpr name="a" />
138           </ParamExpr>
139           <ParamExpr>
140             <SymbolExpr name="j" />
141           </ParamExpr>
142         </params>
143       </FuncCall>
144     </rhs>
145   </BinopExpr>
146 </ExprList>
147 </body>
148 </ForStmt>
149 </ExprList>
150 </body>
151 </WhileStmt>
152 </ExprList>
153 </body>
154 <env envuri="tcp://127.0.0.1:1985" envid="296463126" />
155 </ClosureExpr>
156 </Program>
157 </AIR>
```

Listing D.3: Example AIR XML response from RMulticore Transmutator

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <AIR>
3   <Program>
4     <FuncCall>
5       <funcexpr>
6         <SymbolExpr name="library" />
7       </funcexpr>
8       <params>
9         <ParamExpr>
10          <SymbolExpr name="multicore" />
11        </ParamExpr>
12      </params>
13    </FuncCall>
14    <FuncCall>
15      <funcexpr>
16        <SymbolExpr name="pvec" />
17      </funcexpr>
18      <params>
19        <ParamExpr>
20          <FuncCall>
21            <funcexpr>
22              <SymbolExpr name="c" />
23            </funcexpr>
24            <params>
25              <ParamExpr>
26                <ConstantExpr type="real">
27                  <RealValue data="1.0" />
28                </ConstantExpr>
29              </ParamExpr>
30              <ParamExpr>
31                <ConstantExpr type="real">
32                  <RealValue data="2.0" />
33                </ConstantExpr>
34              </ParamExpr>
35              <ParamExpr>
36                <ConstantExpr type="real">
37                  <RealValue data="3.0" />
38                </ConstantExpr>
39              </ParamExpr>
40            </params>
41          </FuncCall>
42        </ParamExpr>
43        <ParamExpr name="FUN">
44          <SymbolExpr name="sin" />
45        </ParamExpr>
46      </params>
47    </FuncCall>
48  </Program>
49 </AIR>
```

E Numerical Results of Evaluation Chapter

Table E.1: Numerical results

# of elements	# of cores	running time (sec)	speedup (to sequential)
100000	2	14.557	1.78491447413615
100000	4	9.158	2.83719152653418
100000	6	7.358	3.53125849415602
100000	8	5.422	4.7921431206197
200000	2	30.202	1.77746506853851
200000	4	17.201	3.12092320213941
200000	6	14.255	3.76590669940372
200000	8	10.700	5.01710280373832
300000	2	45.767	1.77311163065091
300000	4	26.402	3.07363078554655
300000	6	21.579	3.76060058390101
300000	8	16.073	5.04883966901014
400000	2	61.210	1.8714752491423
400000	4	40.805	2.80732753339052
400000	6	29.356	3.90220057228505
400000	8	21.651	5.29088725693963
500000	2	77.042	1.90686898055606
500000	4	51.093	2.87532538703932
500000	6	41.519	3.53835593342807
500000	8	27.482	5.34564442180336
600000	2	93.655	1.89463456302386
600000	4	66.599	2.6643342993138
600000	6	49.568	3.57976920593932
600000	8	37.970	4.67321569660258
700000	2	111.301	1.86778196062928
700000	4	82.606	2.51659685737114
700000	6	62.757	3.31255477476616
700000	8	44.046	4.71974753666621
800000	2	127.357	1.90125395541666
800000	4	92.107	2.62887728402836
800000	6	77.751	3.11427505755553
800000	8	56.296	4.30115816399034
900000	2	148.129	1.86162736533697
900000	4	100.453	2.74517436014853
900000	6	79.639	3.46263765240648
900000	8	62.503	4.41196422571717
1000000	2	165.548	1.8634595404354
1000000	4	119.279	2.58630605555043
1000000	6	93.761	3.29019528375337
1000000	8	71.847	4.29373529862068

F Description of AIR XML Representation

The following sections describe for every AIR expression its corresponding XML representation. The description is not formal, e.g. following XML Schema or something similar, but shall give a quick intuition about how an AIR instance is constructed.

F.1 BinopExpr

```
1 <BinopExpr op="#opname#">
2   <lhs>
3     #AIRExpr#
4   </lhs>
5   <rhs>
6     #AIRExpr#
7   </rhs>
8 </BinopExpr>
```

#opname# is the name of the operator. Currently, only the <- operator can appear here. On the left-hand side (<lhs>) as well as on the right-hand side (<rhs>) can appear any AIR expression.

F.2 BreakStmt

```
1 <BreakStmt />
```

F.3 BuiltinFunc

```
1 <BuiltinFunc name="#name#" />
```

#name# is the name of an internal function.

F.4 ClosureExpr

```
1 <ClosureExpr>
2   <body>
3     #AIRExpr#
4   </body>
5   <formals>
6     #ParamExpr#
7     #ParamExpr#
8     ...
9     #ParamExpr#
10  </formals>
11  <env envid="#envid#" envuri="#envuri#" />
12 </ClosureExpr>
```

The `<body>` element contains the closure body that can be any AIR expression. The `<formals>` element contains the formal parameters of the closure. If an `<env>` element is present, it represents the environment that was active when the closure has been created.

F.5 ComponentExpr

Not implemented, yet. Appears as `FuncExpr` with function name `$`, first argument equal to the data structure that is accessed and second parameter a `SymbolExpr` that refers to the component name.

F.6 ConstantExpr

Listing F.1: AIR XML for `ConstantExpr`

```
1 <ConstantExpr type="#type#" value="#optvalue#">
2   #AIRValue#
3 </ConstantExpr>
```

`#type#` is the string representation of the `AIRType` that this constant holds. If the value attribute is present, `#optvalue#` is the string, i.e. not XML, representation of a value.

F.7 ExprList

Listing F.2: AIR XML for `ExprList`

```
1 <ExprList>
2   #AIRExpr#
3   #AIRExpr#
4   ...
5   #AIRExpr#
6 </ExprList>
```

Contains a list of AIR expressions.

F.8 ForStmt

```
1 <ForStmt>
2   <condition>
3     #IteratorExpr#
4   </condition>
5   <body>
6     #AIRExpr#
7   </body>
8 </ForStmt>
```


The `<condition>` element contains an `IteratorExpr`, the `<body>` an arbitrary AIR expression that is repeatedly evaluated.

F.9 FuncCall

```
1 <FuncCall>
2   <funcexpr>
3     #AIRExpr#
4   </funcexpr>
5   <params>
6     #ParamExpr#
7     #ParamExpr#
8     ...
9     #ParamExpr#
10  </params>
11 </FuncCall>
```

A function call consists of a `<funcexpr>` that either contains a `<SymbolExpr>` or a `<ClosureExpr>`. The `<params>` children are a sequence of `<ParamExpr>` elements.

F.10 FuncDef

```
1 <FuncDef>
2   <params>
3     #ParamExpr#
4     #ParamExpr#
5     ...
6     #ParamExpr#
7   </params>
8   <body>
9     #AIRExpr#
10  </body>
11 </FuncDef>
```

A function definition consists of a list of function parameters that must be `ParamExpr` elements and a body that may be of any type.

F.11 IfExpr

```
1 <IfExpr>
2   <condition>
3     #AIRExpr#
4   </condition>
5   <body>
6     #AIRExpr#
7   </body>
8 </IfExpr>
```

The `<condition>` element contains an arbitrary AIR expression that determines if the `<body>`, which can also be any AIR expression, is evaluated.

F.12 IteratorExpr

```

1 <IteratorExpr>
2   <itervar>
3     #AIRExpr#
4   </itervar>
5   <collection>
6     #AIRExpr#
7   </collection>
8 </IteratorExpr>
```

`IteratorExprs` are used in `ForStmt` elements. They specify an iteration variable `<itervar>` and a collection that is iterated over. There are no restrictions on the types of these parameters.

F.13 NextStmt

Not implemented, yet.

F.14 ParamExpr

```

1 <ParamExpr name="#name#">
2   #AIRExpr#
3 </ParamExpr>
```

A `ParamExpr` that represents an AIR parameter has a name and a value that may be an arbitrary AIR expression.

F.15 Program

```

1 <Program>
2   #AIRExpr#
3   #AIRExpr#
4   ...
5   #AIRExpr#
6 </ParamExpr>
```

A `Program` is a sequence of AIR expressions.

F.16 RepeatStmt

Not implemented, yet.

F.17 SkeletonExpr

```

1 <SkeletonExpr name="#name#">
2   <params>
3     <param name="#paramname#">
4       #AIRExpr#
5     </param>
6     <param name="#paramname#">
7       #AIRExpr#
8     </param>
9     ...
10    <param name="#paramname#">
11      #AIRExpr#
12    </param>
13  </params>
14 </SkeletonExpr>

```

A `SkeletonExpr` has a `#name#`, e.g. “MAP”, and a list of named parameters that may refer AIR expressions.

F.18 SubscriptExpr

```

1 <SubscriptExpr>
2   <collection>
3     #AIRExpr#
4   </collection>
5   <subscripts>
6     #AIRExpr#
7     #AIRExpr#
8     ...
9     #AIRExpr#
10  </subscripts>
11 </SubscriptExpr>

```

The child AIR expression of `<collection>` is the argument, i.e. having type list or array, that the subscript is applied to. The subscript may be an arbitrarily long list of AIR expressions.

F.19 SymbolExpr

```

1 <SymbolExpr name="#name#" />

```

`#name#` is the name of the Symbol expression.

F.20 UnaryExpr

```

1 <UnaryExpr op="#opname#">
2   #AIRExpr#
3 </UnaryExpr>

```

`#opname#` determines the kind of the unary expression, e.g. `!`. The `#AIRExpr#` is an arbitrary AIR expression that is used as argument.

F.21 WhileStmt

```
1 <WhileStmt>
2   <condition>
3     #AIRExpr#
4   </condition>
5   <body>
6     #AIRExpr#
7   </body>
8 </WhileStmt>
```

The `<condition>` element contains an arbitrary AIR expression that determines if the `<body>`, which can also be any AIR expression, is evaluated.