ARTIST 2 Report

AIR File Format Specification

and remarks about CRL2

Revision 1.0.1
defining
File Version 2.1.2

Henrik Theiling

25. August 2006

Inhaltsverzeichnis

Kapitel 1

Introduction

This document defines the file format of *AIR* files, which are used to represent control flow graphs for real-time systems analyses, and their analysis results. Although this is the major design goal, it may also be used for other purposes.

AIR abbreviates '*ARTIST 2* Interchange Format'. *AIR* format is the proposed exchange format of the tools of the groups participating in the *ARTIST 2* project. The format is based on CRL2, which is the successor of CRL. These formats were originally developed in cooperation by Saarland University and AbsInt Angewandte Informatik GmbH over several years of work.

The idea behind *AIR* is that an interface is to be defined on the file format level, in contrast to CRL2, whose interface definition only covers the C++ library interface. Although internally in AbsInt tools, a specification of the C++ library interface is preferred over a file format specification, simply because all tools use the library and thus the storage on disk is secondary, for *ARTIST 2*, different work groups might want to implement own libraries, so there is a demand for a file format specification.

Since CRL2 was not primarily meant to be a file format, much work had to be done before this document could be written. Apart from the mere documentation the file format had to be defined and implemented. In order to get a stable interface on file level CRL2 had to be extended. For example, version numbers and specification IDs had to be added to meet the strict criteria of real-time systems analysis. Thus, this document can be viewed as the first step of the final documentation phase in a larger effort towards an exchange file format.

From the release of this specification on, CRL2's file format interface will be a dialect of the *AIR* file format. CRL2 as well as dialects of other work groups are allowed to feature extensions as long as they are not vital for the operation of the tools. E.g., AbsInt tools will only use plain *AIR* file format during normal operation, the extension of CRL2 mainly implemented for debugging or diagnosis purposes. In the same way, extensions of other dialects shall never be vital to the operation of the corresponding tools.

This document is still ongoing work. The current revision starts by defining the underlying file syntax and morphology. Subsequent revisions will add a precise definition of the control flow structure and its representation in *AIR* structure and its attributes.

Kapitel 2

Control Flow Graphs

There may be several different views on a control flow graph, depending on how the analysis of the corresponding program works.

In general, for static analyses, some abstraction is applied to the real world program. In the same way, the control flow graph is structured in an abstract way even in its basic components, since, e.g. a microprocessor does not necessarily have a low-level concept of routines, although be means of a stack or link register, an abstract routine structure may well be used. For analyses to work, an abstraction must be found that is close enough to the real world, but abstract enough to apply comfortable analysis methods to the program.

Of course, there may be several good abstractions of the control flow graph, maybe different in details, but maybe even in its basic structure.

This chapter presents in detail what a control flow graph specified in *AIR* describes and what the *AIR* abstraction of a control flow graph is. This includes the overall, basic structure as well as constraints about the structure, and the presentation of special constructions found in programs together with its *AIR* representation. In theory, a different structure may well be stored using the overall *AIR* file syntax, but only the abstraction described in this specification is defined to be a valid *AIR* abstraction of the CFG.

This chapter is currently work in progress and will be extended in the future.

2.1 Definitions

2.1.1 Safe CFG Approximation

A *safe* approximation of a control flow graph (CFG) does not miss any feasible paths of the actual program it approximated. I. e., a safe CFG approximation is an upper bound of the set of nodes and edges of the actual CFG.

For brevity, we may use the term *safe CFG*. It should still always be clear to the reader and user that *AIR* represents an *approximation* of the actual CFG.

2.1.2 Skeleton

The Skeleton is *AIR* without any attributes, data, meta information, or declaration. It consists only of the *AIR* CFG items: graph, routines, blocks, edges, instructions and operations.

2.1.3 Core

The Core is the Skeleton extended by be minimal set of attributes needed to represent a safe CFG.

2.2 Principles

This section describes the Principles by which *AIR* works. All structures in *AIR* obey these principles.

Only if it can be proven that upcoming concepts cannot be represented in *AIR* wrt. the Principles, this section shall be changed in future versions of the *AIR* specification.

1. *AIR* describes the control flow at instruction semantics level.
2. *AIR* structure in designed for static analysis.
3. Actual machine instructions and *AIR* instructions can be mapped one-to-one.
4. The Core constitutes a safe CFG approximation.

2.2.1 Static Analysis

Being designed for static analyses means that theoretical structural concepts of the program are enforced. The most important structural concept that is often not a clear concept in machine code is a *routine*. Still, for the sake of analysability, any actual structures in the machine program have to be mapped into conceptual structures.

The following is the principle theoretical structure of a program:

1. a program contains routines,
2. routines contain basic blocks,
3. basic blocks contain instructions,
4. instructions contain operations,
5. there are edges between blocks, which are called *intraprocedural edges*, and there are distinct edges from blocks to routines, which are called *interprocedural edges*.

It is important to note that a call instruction in the machine code might be a good hint for an interprocedural edge and a jump might be a good hint for an intraprocedural edge, but in the conceptual view, a call to a routine may be implemented by both instructions, and even by other means, and in the same way, an intraprocedural edge may be implemented with a call.

Conceptually, a routine is a piece of code that can be reused from different sites in the program: there is a means of returning to the caller of a routine and routine calls can potentially be nested making recursive routines possible. On the other hand, the intuitive view on routines includes code snippets that are only invoked once, or are named in a particular way, or never return. For a formal definition, it is, therefore, easier to impose constraints on intraprocedural edges instead of defining a 'routine'.

For an intraprocedural edge *e* the following condition holds:

1. *e* never returns, i. e., if *e* is traversed twice, no information about the first traversal of the edge is used in the program.

   Intuitionally, this means that in particular no stack structure is used inside routines, but only between routines, to determine control flow (e.g., for return addresses).

Any intraprocedural edge could thus be implemented by an interprocedural edges, since these do not adhere to the above constraint: in a degenerated graph, each routine may consist of only one basic block and the whole control structure is coded in the call graph. This degeneration is far from a good abstraction, however, since analyses typically perform either with less accuracy for interprocedural edges, or have a much higher overhead in handling them, so whenever the constraints are not violated, intraprocedural edges are preferred.

Kapitel 3

AIR File Grammar

The *AIR* format is a format compatible with CRL2, the format used by AbsInt Angewandte Informatik. *AIR* is a subset of CRL2. All AbsInt tools use only this subset for exchange of information. The additional syntax CRL2 supports is/was very seldom used so far, and is either designed for usages currently not implemented by the tools, or for debugging purposes. A the end of this chapter, the remaining syntax will be briefly described.

3.1 File Header

3.1.1 Specification Name

It is planned to keep *AIR* and CRL2 in sync in the future, to ensure interoperability of all tools that use this file format. Changes incompatible with this document will therefore need collaborative work and an update to this document.

Although we plan to keep this document in sync, no-one can guarantee different implementation to keep in sync. Because the file format is used for safety critical analyses, we will have to ensure that incompatible formats are rejected by unaware readers. We therefore introduce a *specification name*, which will indicate which specification a certain implementation is compatible with.

This specification will use the following specification name, which all implementations of this specification **must** use:

```
'f375656e-a41e-4623-aac9-b5dbb261c4bd'
```

The length of the name ensures with a high probability that a randomly generated new name is not equal to any other specification name. New names **should** be generated with the following Unix command or something equivalent in other environments:

```
uuidgen
```

For other OSes, there are UUID generators on the Internet, too. You **should** generate a (good) random UUID when starting a new implementation.

When a syntax incompatible from this specification is used, a different specification name **must** be used. This is to ensure that implementations not aware of the used format reject the input file, instead of possibly accepting it with unspecified behaviour, which is totally unacceptable behaviour in safely-critical environments.

Reader implementations **must** reject files with specification names they are unaware of.

Reader implementations **should** reject files that contain file syntax that is not part of the declared specification, e.g., it **should** not accept private extensions when a pure *AIR* specification was declared.

3.1.2 Implementation Name

Because we programmers are humans, implementations of this specification may be broken or not fully compliant. To enable detection of such implementations, each implementation will have its own library name. As before, the implementation name is an UUID. New implementations **should** be registered in this specification. Currently, there is only one implementation:

Implementation	Name
AbsInt CRL2 Library	`'18399358-21ba-45b1-8339-33592c28f594'`

Readers **may** issue a note about unknown implementation names. However, they **must not** reject a file because of this.

3.1.3 Version

A specification is usually not fixed. When changes occur, reader must be aware of the changes, so the file format must contain a version number.

The version number will be split into four 31-bit unsigned integers:

Generation Code This is a **fixed number** and currently always **2**. Only if a whole new file format is specified, this version digit may be incremented. This code can, therefore, be regarded as a constant.

Reader implementations **must** reject files with generation codes they cannot ensure to handle correctly.

The generation code will never be decremented.

Safety Code Whenever format changes occur that might be read in the wrong way unnoticed by an implementation, the safety code must be incremented. This is not called 'major' version code, since even small changes may cause wrong behaviour of readers, so the important thing is the possibility of **unnoticed misinterpretation** here. The safety code introduces an artificial difference of the file that can clearly be detected.

Reader implementations **must** reject files with safety codes they cannot ensure to handle correctly.

The safety code will never be decremented.

Change Code Any change to the file format not indicated by the safety code will be indicated by incrementing the change code. These are typically changes that either go unnoticed by old implementations without the possibility of misinterpretation, or those that immediately break syntactic compatibility so that an old reader implementation will reject the file implicitly by failure. Syntax extensions typically trigger this type of change.

Reader implementations **should** issue an informational note when encountering an unknown change code, but they **should not** refuse to read the file for this reason.

The change code will never be decremented.

Implementation Version and Sub Version This integer is (part of) the version of the implementation that produced the file. The meaning of this code is implementation specific and can only be handled correctly in conjunction with the implementation name.

Implementors **should** only change the implementation version when significant changes occur and the sub version on each successful library linking. The library sub version **must** change for each public release so that other users can cleanly distinguish library versions.

Reader implementations that are unable to interpret the implementation version correctly (e.g. because they did not recognise the implementation name) **must not** print notices about unknown implementation versions. They **must** read the file without additional messages and assume a correct input file. This rule exists since it is expected to be impossible to keep track of all existing versions.

Reader implementations that are able to interpret the implementation version **may** print notices about old or broken library versions. If they are indeed able to identify broken library versions, reader implementations **must** refuse to read the input file for safety reasons.

The implementation version **must never** be decremented.

The generation code, safety code, change code, and implementation version may be cited in publications, in that order, with periods in between and preferably with a prefixed 'version'. Such a version string always starts with the generation code and may be truncated at any point. E.g. version 2.1.2.1001000 or version 2.1.2 or version 2.1 or version 2.

Please note that the safety code, the change code and the implementation version are unrelated integers. In particular, there is no major version number on whose increment a minor version number is reset to a smaller value (e.g. 2.1.9 is never followed by 2.2.1, but only by 2.2.9 or 2.1.10).

If the generation code changes, anything may happen to the other numbers, since this indicates a whole different file format.

This revision of the specification defines file format **2.1.2**.

3.2 File Grammar

Throughout the file, the amount and nature of white space (but not the presence vs. absence) is insignificant except inside strings (delimited with double quotes) and identifiers (delimited with single quotes). White space is defined to be one of the following ASCII characters given in octal notation: 000, 010, 011, 012, 013, 014, 015, 040.

C and C++ style comments are recognised and ignored. Please refer to a good C/C++ documentation about their syntax.

The following grammar uses different notations for terminal symbol and non-terminal symbols:

<*non-terminal*> A non-terminal whose syntax is given in a separate rule in the below grammar.

[variable terminal] A terminal symbol with a variable value. The morphology of such a terminal will be defined in the next section.

`constant_terminal` A constant terminal symbol that has exactly the given verbatim sequence of characters in the file.

`keyword` This is a keyword, whose overall morphology is an [lc-name].

`special` Non-alphanumeric syntax element to be used exactly as given.

<***parameter***> Parameter of a higher order syntax rule.

The *AIR* syntax is defined in such a way that never an [lc-name] will be concurrent with a keyword, so a reader implementation can always scan a [lc-name] when a keyword is used, and then use an additional *keyword siever* step to computed a keyword token from an [lc-name].

Note to Implementors: Reader implementations **must** except [lc-name] tokens that happen to be a keyword when the grammar requires an [lc-name]. For example, although `crl` is a keyword, is is also an [lc-name], and thus a valid name of an attribute. Implementors using flex and bison are likely to have to pay special attention to this. Typically, a bison rule is needed that accepts all keywords as names and also the generic name to implement [lc-name], because flex implements both the scanner and the keyword siever at once.

3.2.1 Morphology

This section defines the morphology of all variable terminal tokens.

The set of variable tokens is divided into two sets: the simple tokens, and the constrained tokens. Constrained tokens are simple tokens with additional constraints. Therefore, this section first gives the syntax of the simple tokens and then lists the constrained tokens, giving the simple token they are together with their constraints.

To start defining the morphology of tokens, we first define some character classes used later.

Character Classes

Character classes are given either as octal digits, or as singly quoted ASCII characters, or as ranges of those with lower and upper bound separated by a hyphen. Further, classes may be referred to and included in another class by usinge the word 'class' and the class name in single quotes after that.

class 'space'	:=	000, 010, 011, 012, 013, 014, 015, 040
class 'lowcase'	:=	'a' - 'z', 200 - 377
class 'upcase'	:=	'A' - 'Z'
class 'digits1-9'	:=	'1' - '9'
class 'digits0-9'	:=	'0' - '9'
class 'digits0-7'	:=	'0' - '7'
class 'digits0-f'	:=	'0' - '9', 'a' - 'f', 'A' - 'F'
class 'in-name'	:=	class 'upcase', class 'lowcase', class 'digits0-9', 0137

Simple Tokens

Simple tokens are defined by regular expressions using the characters and character classes from the previous section. Further, characters may be given in octal notation or as quoted ASCII characters.

Regular expression operators include parentheses for grouping, the or operator '|', the iteration operators '*' (arbitrary iteration including 0 times) and '+' (arbitrary iteration but at least once), and the marker optionality '?'.

The following list is ordered by priority from top to bottom: whenever a token matches an entry higher up in the list, it is not a token of an entry below that higher entry. E.g. a [potential binary] is not a [potential octal].

[lc-name]	:=	class 'lowcase' (class 'in-name')*
[uc-name]	:=	class 'upcase (class 'in-name')*
[ub-name]	:=	137 (class 'in-name')*
[potential binary]	:=	'0' ('b' | 'B') (class 'in-name')*
[potential hexadecimal]	:=	'0' ('x' | 'X') (class 'in-name')*
[potential octal]	:=	'0' (class 'in-name')*
[potential decimal]	:=	class 'digits1-9' (class 'in-name')*, '08', '09'

Note that '08' and '09', despite their leading 0, are defined to be decimal integers, simply because there is no possibility of misinterpretation. This syntax typically occurs with aligned date/time strings, e.g. `?gmt(2006-08-24-08-16-09)`. Here the month, hours and seconds are valid decimal integers.

Numeric values are parsed ignoring all underbar characters (ASCII 137), thus the [potential decimal] `1_000_000` is indeed an integer denoting one million. Reader implementions **should** avoid printing underbars in numbers, however.

Quoted Character Sequences: *AIR* has doubly quoted C style character sequences denoted by [string]. Please refer to a C standard for their syntax, since we base our definition on the C standard.

Further, there are singly quoted character sequences denoted by [identifier] with a very similar syntax.

[string]s begin with a double quote character (octal 042), and are also terminated with 042. [identifier]s begin with 047 and are terminated with 047. In between, characters occur either literally, or escaped with a backslash (134). Any number of characters, 0 included, is possible.

The following characters may appear literally: 040, 041, 043 - 046, 048 - 0133, 0135 - 0176.

All other characters **must** be escaped. Escape sequences consist of the escape character 134 plus more characters. The sequences `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, and `\\` have the same meaning as in the C standard. The sequence `\'` represents 047, `\"` represents 042, and `\e` represents character 033. Characters escaped in octal notation have the same syntax as in C. Characters escaped in hexadecimal **must** use the prefix `\x`.

All characters in a quoted character sequence **must** be in the range 001 - 0377, whether escaped or literal, i.e., the character code **must** use maximally 8 bits, and the character codes are interpreted as ISO-8859-1 (which is a subset of Unicode). Full Unicode character range is currently not part in this specification, but may be added in future versions if there is need for it.

Note: In contrast to C, we do not allow characters ≥ 0177 in quoted character sequences. We also forbid the character 0, and also quotes in any quoted character sequence, no matter whether doubly or singly quoted. Further, escape sequences `\u` and `\U` are not defined in this version of the specification.

Reader implementations **may** be more generous and allow single quotes in [strings] and double quotes in [identifiers]. They **must not** allow any character ≥ 0177 in quoted character sequences, since the interpretation adds complexity wrt. different character sets (e.g. ISO-8859-1 vs. UTF-8) not currently defined in *AIR*. To allow for future extensions and compatibility, *AIR* files **must** be 7-bit clean.

So writer implementations **must not** print 8-bit characters, but **must** adhere to the definitions made here (of course they do, but we stress it here).

The above paragraphs should be interpreted to explicitly **disallow** abusing the 8-bit strings to store UTF-8 Unicode strings, since this would be misinterpreted by unaware readers. If you need Unicode, an extension of this specification is needed.

Caution: Since the C standard has confusing rules for escaped hexadecimal and octal characters, implementors should pay special attention to them. The confusion typically arises from the following definitions: Inside a quoted string or character, after a backslash, there must be maximally *three octal digits*. Thus, the doubly quoted string `"\0100"` has length two, and consists of the characters 010 and 060.

On the other hand, characters given in hexadecimal notation may consist of *arbitrarily many hexadecimal digits*. Thus the doubly quoted string `"\x000010"` consists of only one character 020.

For this reason, reader implementations **must** implement this behaviour correctly (of course they do, but we stress it here). Further, writer implementations **should** avoid printing octal digits after a character escaped in octal notation in strings and identifiers. Further, writers **should** give exactly three digits when quoting in octal notation, and two when printing in hexadecimal notation (the latter because we only currently allow 8-bit characters).

For example, writers **should** encode a string of characters 010 and 060 as `"\010\060"` when quoting in octal notation, thus quoting both characters with three octal digits.

In the same way, hexadecimal digits after hexadecimally escaped characters **must** be avoided, because that would unintentionally extend the hexadecimal character escape sequence. The encoding of the string consisting of 010 and 060 in hexadecimal quotation **should** be `"\x08\x30"`.

Constrained Tokens

Constrained tokens are basically identical to simple tokens, but add constraints, typically due to conversion from string to a different data type. Sometimes different alternatives are allowed for a constrained token. Constrained tokens may be based on other constrained tokens and add additional constraints.

Constrained Token	Derived from	Constraint
[unsigned]	[potential binary]	The token string can be parsed as a number (see a C manual for details).
or	[potential octal]	
or	[potential decimal]	
or	[potential hexadecimal]	
[*n*-bit unsigned]	[unsigned]	interpreted as an unsigned integer, the result fits into *n* bits. This only specifies the minimum number of bits. Implementations **may** use more bits.
[UUID identifier]	[identifier]	exactly length 36, only hexadecimal digits, all in lower case, except for 9th, 14th, 19th, and 24th characters, which must be dashes (055).

The grammar will be written using parameterized non-terminals in order to keep in small.

3.2.2 Syntax

The syntax is written with higher order rules to keep it small, i.e., with non-terminals accepting parameters used in their right hand sides for definition.

Default values or additional constraints or clarifications are given right after the corresponding rules. This hopefully makes it easier for implementors, since the information is gathered where it is needed.

<file> ::= *<header>* *<body>* `end`

<header> ::= `crl`
`specification` *<specification name>*
`implementation` *<implementation name>*
`version`
<generation code>
<safety code>
<change code>
<implementation version>
<implementation subversion>

<specification name> ::= [UUID identifier]

<implementation name> ::= [UUID identifier]

<generation code> ::= [31-bit unsigned]

<safety code> ::= [31-bit unsigned]

<change code> ::= [31-bit unsigned]

<implementation version> ::= [31-bit unsigned]

<implementation subversion> ::= [31-bit unsigned]

<body> ::= *<body-element>**

<body-element> ::= *<list(<attr-decls>)>*
| `global` *<global>*
| `routine` *<routine>*
| `data` *<data>*
| `meta` *<meta>*

*<list(<**element**>)>* ::= ***<element>***
| *<list(<**element**>)>* ***<element>***

*<comma-list(<**element**>)>* ::= ***<element>***
| *<comma-list(<**element**>)>* `,` ***<element>***

<lc-id-def> ::= [lc-name]

[lc-name]s **must only** be used once in an *<lc-id-def>* in an *AIR* file, i.e., they define **unique** identifiers for items in the file.

<global> ::= *<simple-item(*[lc-name],*<global-special>)>*

<global-special> ::=

<routine> ::= *<lc-id-def>* *<attr-list(<lc-name-ctxt>)>*? `;`
| *<lc-id-def>* *<attr-list(<lc-name-ctxt>)>*?
`{`
*<list(<**context-def**>)>*?
*<list(<**block**>)>*?
`}`

<block> ::= *<lc-id-def>* *<block-special>*
<attr-list(<lc-name-ctxt>)>? `;`
| *<lc-id-def>* *<block-special>*
<attr-list(<lc-name-ctxt>)>?
`{`
*<list(<**edge**>)>*?
*<list(<**instruction**>)>*?
`}`

<block-special> ::= (`(` *<block-type>* `)`)?

This defines the attribute `type`. It is a special attribute and **must not** be used as a normal, generic attribute of blocks. If the block type is missing, it is implicitly defined to be `normal`.

<block-type> ::= `normal`
| `start`
| `end`
| `call`
| `return`
| `external`
| `impasse`

<edge> ::= *<simple-item(<lc-name-ctxt>,<edge-special>)>*

<edge-special> ::= (`(` *<edge-type>* `)`)? `->` [lc-name]

This defines the attributes `type`, `source`, and `target`. These are special attributes and **must not** be used as normal, generic attributes of edges. If the edge type is missing, it is implicitly defined to be `normal`. The [lc-name] **must** be a block identifier of the target of the edge. For `call` edges, it may also be a routine identifier, in which case the edge points to the start block of that routine. The source is implicitly defined by the nesting structure: the edge's source in the block its definition is found in.

<edge-type> ::= `normal`
| `true`
| `false`
| `zero`
| `delay`
| `call`
| `return`
| `local`
| `impasse`

<instr> ::= *<item(<lc-name-ctxt>,<op>,<instr-special>)>*

<instr-special> ::= *<addr-and-width>*?

This defines the attributes `address` and `width`. These are special attributes and **must not** be used as normal, generic attributes of instructions. Both attributes may be undefined, indicated by either a missing *<addr-and-width>* or by question marks (see the corresponding rule).

<addr-and-width> ::= ([64-bit unsigned] | `?`)
`:` ([64-bit unsigned] | `?`)

The first integer defines the address, the second one the width. Both may be `?` to indicate that the corresponding value is undefined.

<op> ::= *<simple-item(<lc-name-ctxt>,<op-special>)>*

<op-special> ::= [string]

This defines the attribute `mnemonic`. This is a special attribute and **must not** be used as normal, generic attribute of operations. The mnemonic is not optional and **must** be given.

<data> ::= *<item(*[lc-name],*<bytes>*,*<data-special>)>*

<data-special> ::=

<bytes> ::= *<simple-item(*[lc-name],*<bytes-special>)>*

<bytes-special> ::= *<addr-and-width>*?

This defines the attributes `address` and `width`. These are special attributes and **must not** be used as normal, generic attributes of bytes. Both attributes may be undefined, indicated by either a missing *<addr-and-width>* or by question marks (see the corresponding rule).

<meta> ::= *<item(*[lc-name],*<info>*,*<meta-special>)>*

<meta-special> ::=

<info> ::= *<simple-item(*[lc-name],*<info-special>)>*

<info-special> ::=

*<simple-item(<**key**>,<**sp**>)>* ::= *<lc-id-def>* *<**sp**>* *<attr-list(<**key**>)>*? `;`

This defining item *<lc-id-def>* **must** be unique in the *AIR* file. The *<**sp**>* parameter parameterises this rule and provides special syntax for certain item attributes. It is basically syntactic sugar to make the *AIR* file easier to read for humans, but also to highlight some important attributes belonging to the CFG core, like targets and sources of edges.

*<item(<**key**>,<**sub**>,<**sp**>)>* ::= *<lc-id-def>* *<**sp**>* *<attr-list(<**key**>)>*? `;`
| *<lc-id-def>* *<**sp**>* *<attr-list(<**key**>)>*? `{` *<list(<**sub**>)>* `}`

This defining item *<lc-id-def>* **must** be unique in the *AIR* file. In contrast to a simple item, the normal item may have sub-structures enclosed in curly braces. The parameter *<**sub**>* defines which sub-structures are contained.

*<attr-list(<**key**>)>* ::= `:` *<comma-list(<attr(<**key**>)>)>*

*<attr(<**key**>)>* ::= *<**key**>* (`=` *<value>*)?

If *<value>* is not given, it defaults to the unsigned integer 1.

<lc-name-ctxt> ::= [lc-name]
| [lc-name] `<` [context-ref] `>`

<context-ref> ::= [lc-name]
| `*`

The [lc-name] in pointed brackets defines the context in which this attribute is defined. It **must** identify a context of the corresponding routine, not one of another routine. (This type of attribute keys is only allowed for routines, blocks, instructions, operations, and edges.) If the context-ref is `*`, the so-marked attribute is the default attribute in contexts not specifically listed.

<context-def> ::= `context` *<lc-id-def>* `:` *<context-seq>*? `;`

This defines a name for a regular expression matching call strings of the program under examination. The name **must** be a unique identifier in the *AIR* file. The non-terminals of the regular expressions are routine calls.

<context-seq> ::= *<comma-list(<context-simple>)>*

<context-simple> ::= *<context-match>*
| *<context-minimum>*
| *<context-repeat>*
| `(` *<context-seq>* `)`

<context-match> ::= ([lc-name] | `?`) `->` ([lc-name] | `?`)

This matches a call in the program. The first [lc-name] **must** be a block identifier or the wildcard `?` (which matches anything). The second [lc-name] **must** be a routine identifier or a wildcard `?`.

<context-minimum> ::= *<context-simple>* `*`
| *<context-simple>* `*-`
| *<context-simple>* `+`
| *<context-simple>* `{` [32-bit unsigned] `,` `}`

`*` and `*-` are equivalent to {`0,`}.
`+` is equivalent to {`1,`}.

<context-repeat> ::= *<context-simple>* `?`
| *<context-simple>* `{` [32-bit unsigned] `,`
[32-bit unsigned] `}`

`?` is equivalent to {`0,1`}.

<value> ::= *<value-numeric>*
| *<value-range>*
| *<value-date>*
| *<value-item>*
| *<value-vector-value>*
| *<value-map-symbol-value>*
| *<value-map-item-value>*
| *<value-functor>*

<value-numeric> ::= *<value-unsigned>*
| *<value-signed>*
| *<value-float>*

<value-unsigned> ::= [64-bit unsigned]

<value-signed> ::= *<64-bit signed>*

*<<**n**>-bit signed>* ::= `+` [(*<**n**>*-1)-bit unsigned]
| `-` [*<**n**>*-bit unsigned]

Minimally *<**n**>*-bits are required to represent the resulting signed integer of this rule; more are not needed (but allowed). The second alternative of the rhs may suggest more bits, but this is only to guarantee the smallest signed integer is parsed correctly.

<value-float> ::= `?float(` [float] `)`

<value-range> ::= `..`
| *<value-numeric>* `..`
| `..` *<value-numeric>*
| *<value-numeric>* `..` *<value-numeric>*

Missing numbers denote that no constraint on the corresponding bound is defined.

<value-date> ::= *<value-date-posix>*
| *<value-date-human>*

<value-date-posix> ::= `?gmt(` [64-bit unsigned] `)`

The integer counts the seconds since January 1st, 1970, 0:00 GMT, also called the POSIX time.

<value-date-humal> ::= `?gmt(`
[unsigned] `-` [unsigned] `-` [unsigned] `-`
[unsigned] `-` [unsigned] `-` [unsigned]
`)`

Time is in 'year-month-day-hour-minute-second' format in the GMT time zone. The year **must not** be abbreviated (e.g. with a two-digit number), but **must** be the year since CE. The year **must not** be less than 1970, the and **should** be below 2037. The unsigned numbers and their minimal bit widths are constrained in such a way that a valid date/time is defined. Note again that only the GMT time zone is allowed.

<value-item> ::= [lc-name]

Item's *<lc-id-def>*s are unique throughout the *AIR* file. This value establishes a reference to an item via such a unique item name.

<value-vector> ::= `[` *<vector-list>*? `]`

<vector-list> ::= *<comma-list(<vector-element>)>*

<vector-element> ::= *<value>*
| [64-bit unsigned] `=` *<value>*

Vectors with gaps (sparse vectors) may be defined by preceding a vector element value with an index. Indexing starts with 0 for the first vector entry. Vectors **must only** be filled in ascending order and **no** element **must** be doubly defined.

<value-map-symbol-value> ::= `{` *<symbol-map>*? `}`

<symbol-map> ::= *<comma-list(<symbol-map-entry>)>*

<symbol-map-entry> ::= *<symbol-map-key>* (`=` *<value>*)?

<symbol-map-key> ::= [lc-name]
| [identifier]
| [string]
| [64-bit unsigned]
| *<64-bit signed>*

If the value is missing, it is implicitly defined to be unsigned 1 (i.e., a boolean true value). Keys syntactically integers are converted to strings by printing them in their shortest decimal form. Thus `'17'` is the same key as `0x11`. Each key **must only** be given once per map. In the current version of *AIR*, only [lc-name] and [unsigned] are ever used as keys and users **should** keep up that tradition for simplicity reasons.

<value-map-item-value> ::= `@{` *<item-map>*? `}`

<item-map> ::= *<comma-list(<item-map-entry>)>*

<item-map-entry> ::= *<item-map-key>* (`=` *<value>*)?

<item-map-key> ::= [lc-name]

If the value is missing, it is implicitly defined to be unsigned 1 (i.e., a boolean 'true' value). Keys refer to one of the uniquely identified items in the *AIR* file. Each key **must only** be given once per map.

<value-functor> ::= *<functor-c>*
| *<functor-mathematica>*
| *<functor-lisp>*
| *<functor-prefix>*
| *<functor-infix>*
| *<functor-circumfix>*
| *<functor-suffix>*

Functors are basically vectors with an additional one or two functor identifiers. They implement a nice syntax for representing function applications or other functor invocations to represent terms in complex expressions. Note that *AIR* syntax poses no restrictions on the number or arguments of a functor depending on the kind (e.g. infix functors may well have more than two arguments).

<functor-c> ::= [identifier] `(` *<vector-list>*? `)`

<functor-mathematica> ::= [identifier] `[` *<vector-list>*? `]`

<functor-lisp> ::= `(` [identifier] `)`
| `(` [identifier] `,` *<vector-list>* `)`

<functor-prefix> ::= `(` [identifier] *<value-vector>* `)`

<functor-infix> ::= `(` *<value-vector>* [identifier] *<value-vector>* `)`

The two lists are appended to form the arguments of the infix functor. The split point of the argument list does not carry any information and writers may choose it arbitrarily.

<functor-circumfix> ::= `(` [identifier] *<value-vector>* [identifier] `)`

<functor-suffix> ::= `(` *<value-vector>* [identifier] `)`

<attr-decls> ::= `attributes` *<item-key>* *<attr-decl-list>*

```
<item-key>          ::= global
                      | routine
                      | block
                      | edge
                      | instruction
                      | operation
                      | data
                      | bytes
                      | meta
                      | info

<attr-decl-list>    ::= <comma-list(<attr-decl>)>

<attr-decl>         ::= [lc-name] : <type>

<type>              ::= <type-simple>
                      | <type-range>
                      | <type-list>
                      | <type-tuple>

<type-simple>       ::= scalar
                      | symbol
                      | identifier
                      | string
                      | enum
                      | numeric
                      | integer
                      | unsigned
                      | gmt
                      | address
                      | bool
                      | signed
                      | float
                      | item
                      | vector
                      | nested
                      | range
                      | any type
                      | any string
                      | simple numeric
                      | simple identifier

<type-range>        ::= <type-numeric> .. <type-numeric>

<type-list>         ::= <type> [ ]
```

<type-tuple> ::= `[` *<comma-list(<type>)>*? `]`

3.3 Extensions in CRL2

This section lists some extensions the CRL2 format has additional to this specification. Note that whenever such an extension is used, the specification code is automatically changed by the CRL2 library. And in accordance with this specification, files that contain such extensions **must not** use the *AIR* specification name, but only the following specification name for CRL2 extended format:

```
d25fd505-3f27-4b2e-9867-65d2e6ca165e
```

3.3.1 Stable Format

This special extension produces syntactically bad files according to this specification since the version stamp is not printed in full detail. Files in stable format **must not** be exchanged, since without the versioning the file format is not considered no be safe. (However, because the file header format is a little different, it is expected to break other reader implementations anyway, so a clean detection of unability to read the file is very likely.)

The purpose of the stable format is obviously not running analyses. In order to provide a means of easy scripted comparison of CRL2 files, mainly in testing and quality assurance processes, the CRL2 library implements this special 'stable' format. Some differences to this specification include:

- Suppression of automatically generated comments containing time or version stamps.
- Suppression of detailed version information. The header is reduced to `crl version 2 stable;`
- All lists are printed line-wise.
- All lists items are terminated with the list separator instead of separating the items with it. This ensures that each line of the list contains the list separator (particularly the last line).
- White space is minimized and changes to indenting and white space usage are minimised between library versions.
- Most other formatting options are ignored in stable mode (e.g. explicit syntax).

3.3.2 Explicit Syntax

This affects the printing of attributes. Instead of printing the attributes in a nested way, each attribute entry is specified by an explicit 'reference'. E.g.:

Instead of:

```
a={ x=1, y=[6,7] }
```

In explicit mode, the library prints:

```
a.x=1, a.y[0]=6, a.y[1]=7
```

This format is often easier to read for humans.

3.3.3 Special Attributes

In contrast to this specification, the CRL2 library reads special attributes in the argument list, too, equivalently to the special syntax defined above. E.g. the `source` and `target` attributes may contain source and target of edges. This is in accordance with the library interface, which also allows access to the special attributes via the standard symbol-based access mechanism.

Special attributes, however, are never written in the normal attribute list, so this feature is an extension of the reader/parser module of CRL2.

3.3.4 Modification

The CRL2 library supports reading modification files additional to a given main file. In the modification file, items can be added or modified.

This feature was never used although we expected it to be used. The CRL2 library will never write this format, but only read it.

3.3.5 Treatment of Separators

The CRL2 library allows separators of lists to be used after the last element, too, and also allows empty list elements. This is to enable reading of stable format files.

3.3.6 Singular vs. Plural

For most English language keywords, the CRL2 library allows both the singular and the plural forms.