fUML Activity Diagrams with RAG-controlled Rewriting
– A *RACR* Solution of *The TTC 2015 Model Execution Case* –

Christoff Bürger
christoff.buerger@gmail.com

This paper summarises a *RACR* solution of *The TTC 2015 Model Execution Case*. *RACR* is a metacompiler library for *Scheme*. Its most distinguished feature is the seamless combination of reference attribute grammars and graph rewriting combined with incremental evaluation semantics. The presented solution sketches how these integrated analyses and rewriting facilities are used to transform *fUML Activity Diagrams* to executable Petri nets. Of particular interest are (1) the exploitation of reference attribute grammar analyses for Petri net generation and (2) the efficient execution of generated nets based on the incremental evaluation semantics of *RACR*.

1 Prerequisites and Contents

The following document describes a *RACR*-based [1] solution of the *Model Execution Case* [6] of the *8th Transformation Tool Contest* which was part of the *Software Technologies: Applications and Foundations* (*STAF*) conference 2015. It assumes readers are familiar with the contest task (cf. [6]); no further previous knowledge is required, although a basic understanding of reference attribute grammars [5] and familiarity with the *Scheme* programming language [3] will be helpful. The presented solution is part of *RACR's* source code repository at `https://github.com/christoff-buerger/racr`; a deployed *SHARE* [8] demonstrator is provided at `https://is.ieis.tue.nl/staff/pvgorp/share/`.

The structure of this document is as follows: Section 2 gives a short overview of the solution. It first presents the implemented analyses in Section 2.1, concluding in a sketch of the intended abstract syntax graphs used to execute *fUML Activity Diagrams* [4]. Afterwards, Sections 2.2 sketches the implementation of execution semantics by means of rewrites reusing the implemented analyses. An evaluation follows in Section 3. The actual source code is investigated in the appendix; readers are highly encouraged to closely follow it and consult *RACR's* reference manual [1] as required.

2 Solution Overview

The activity diagram interpreter presented in the following is realised in the form of two language processors. The first analyses the actual activity diagram and its inputs and translates them to a Petri net [7]. The second executes generated Petri nets – it is a Petri net interpreter.

2.1 RAG-based Analyses: From Activity Diagrams to Petri Nets

Figure 1 sketches the abstract syntax graph of an exemplary activity diagram. Our interpreter is implemented in terms of such graphs; they represent the original input diagram, its current execution state and analysis results, including static *and* dynamic analyses like diagram well-formedness or whether activities are ready for execution. The graph consists of two abstract syntax trees (black and purple nodes and edges). The black one on the left encodes the actual activity diagram; it is the original input of the

Submitted to:
TTC 2015

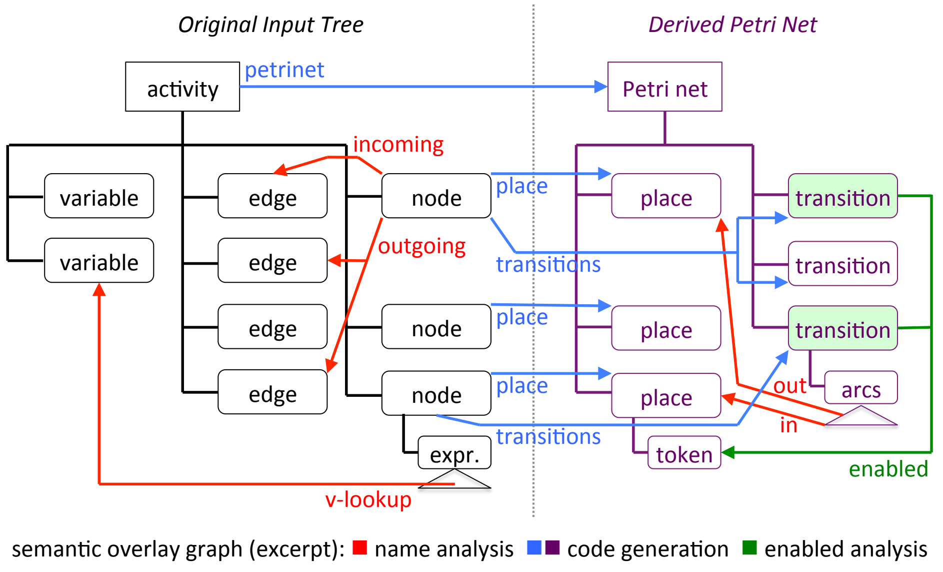


Figure 1: Example abstract syntax graph of the activity diagram interpreter ©Christoff Bürger

interpreter[1] and constructed by a hand written recursive-decent parser (the parser is straightforward and not investigated in the following). The purple right tree encodes the Petri net used to execute the diagram; code generation derives it from the original input tree (blue edges). Name analysis extends both trees to the actual diagram each represents. In case of the original input tree, it resolves the symbolic names of activity edges to the target and source activity nodes they refer to, such that each node knows all its in- and outgoing edges (*Original Input Tree*, red edges). In case of the derived Petri net, name analysis resolves the symbolic names of the arcs of transitions to the actual places they refer to (*Derived Petri Net*, red edges). Enabled analysis finally associates transitions with the tokens they consume if fired or no token if disabled (green edges). It just is a special kind of name analysis, searching for consumable tokens and returning the tokens consumed if a transition is enabled and *false* if it is disabled.

Everything in Figure 1, except the original input tree encoding the activity diagram, is derived by the interpreter. The interpreter computes an semantic overlay graph that extends its input tree to a graph well-suited for digram execution. The required analyses are implemented using *RACR's* reference attribute grammar facilities, each by a set of attributes. Figure 1 shows only an excerpt of the actually implemented analyses and the resulting abstract syntax graph. The names of reference attributes are labeled next to the edges they induce; for example, the `v-lookup` attribute finds the variables the assignee and operands of expressions refer to. Not shown are non-reference attributes like type and well-formedness analysis, parts of the code generation, for example for expressions of executable nodes, and minor query-support analyses like lookup of activity nodes and edges by name.

Important for the development-effort-benchmarks in Section 3 is, that analyses can be interdependent, fostering reuse and modularisation. *RACR's* demand-driven evaluation strategy automatically deduces correct evaluation orders, easing the implementation of complex or mutually dependent analyses. For example, code generation can reuse name analysis, execution reuses code generation and the runtime lookup of tokens (i.e., the name and enabled analyses of Petri nets) reuses the places code generation generated. From the perspective of a user – whether interpreter or Petri net developer – a common

[1]Input for the interpreter are textual diagram specifications as given by the tool contest [6]. Parsing such specifications yields abstract syntax trees like the one labeled *Original Input Tree* in Figure 1; they satisfy the scheme in Appendix A.1.

interface for querying analyse results is provided: abstract syntax graphs as shown in Figure 1. Moreover, analyses are automatically memoized; deduced abstract syntax tree parts are only re-evaluated if required.

2.2 Rewriting-based Transformations: Incremental Execution of Petri Nets

Given abstract syntax graphs as in Figure 1 and all the deduced analyse results they encode, the specification of execution semantics boils down to simple transformations manipulating their tokens. After all, convenient means to find enabled activity nodes considering the state of execution are already provided (enabled analysis reasons about the current marking of the generated Petri net). Execution therefore can be realised by a simple loop that reuses the enabled analysis to find an enabled transition and deletes its consumed and adds its produced tokens using *RACR's* primitive rewrite functions `rewrite-delete` and `rewrite-add` [1]; if no transition is enabled, execution terminates.

Important for the performance-benchmarks in Section 3 are the automatic incremental evaluation semantics of *RACR*. When an abstract syntax graph information is queried throughout attribute evaluation, *RACR* maintains a dependency to remember that the value of the attribute depends on the queried information. If an abstract syntax graph information changes, *RACR* invalidates all attributes transitively depending on it. The enabled analysis of the Petri net language is no exception since it is implemented using attributes. It depends on tokens that would be consumed or are missing, including the special case of tokens encoding variable values. All these dependencies are automatically tracked by *RACR*, such that the enabled status of incoming arcs is only re-evaluated if it could be changed by a fired transition, otherwise the cached is used. Likewise, the enabled status of transitions is only re-evaluated if the enabled status of any of their incoming arcs was invalidated. For example, if any of the two enabled transitions of Figure 1 (highlighted green) is fired, the enabled attributes of both are invalidated since each depends on the token deleted according to firing semantics. Similarly, when a new value is assigned to a variable via `rewrite-terminal` (cf. Appendix A.3.2), the enabled status of transitions depending on its value is re-evaluated, if either, they were enabled or, although all tokens they consume are provided, still were disabled. Without special implementation efforts, *RACR* optimises the implemented execution semantics.

The activity diagrams of the tool contest result in very simple and restricted Petri nets with just a single token type (except tokens encoding variable values; cf. Appendix A.3.2) and at most one token per place. The developed Petri net language is much more expressive however, supporting coloured, weighted Petri nets with arbitrary input arc conditions and output computations; it was developed before the tool contest for more general applications. In case of the tool contest, the restricted type and number of tokens, and therefore simple enabled decisions, preclude major performance benefits from *incremental* enabled analysis. If there are only few tokens and conditions to check, caching the results of such checks does not pay-off as much as in more complex cases. Of course, the execution semantics could be optimised for such less expressive nets. For example, the transitions of the Petri nets generated for most activity diagrams never compete for tokens (this holds for example for all test cases given by the tool contest). In this case, all enabled transitions can be fired in one pass (enabled pass); only thereafter, for the next iteration of the execution loop, enabled analysis has to be repeated[2]. In general however, Petri net transitions can compete for tokens. For example, in Figure 1 the two enabled transitions highlighted green compete for the same token; their enabled attributes point to the same token to consume if fired. To fire one of the two enabled transitions disables the other one.

[2]Enabled passes still sequentially execute parallel fork branches; they perform no multi threaded execution. They execute one activity *of each* active branch in each iteration step instead of *a single* activity of some active branch.

Source code file	Solution part (language task)	LOC	
Activity Diagram interpreter (584):		*548*	
`analyses.scm:` *308*	AST specification	16	3%
	ASG accessors (constructors, child & attribute accessors)	89	16%
	Name analysis	36	7%
	Type analysis	21	4%
	Well-formedness	30	5%
	Petri net generation	94	17%
`parser.scm:` *234*	Parsing	229	42%
`user-interface.scm:` *42*	Initialisation & execution	33	6%
Petri net interpreter (243):		*222*	
`analyses.scm:` *134*	AST scheme	6	3%
	ASG accessors (constructors, child & attribute accessors)	34	15%
	Query support	12	5%
	Name analysis	19	9%
	Well-formedness	12	5%
	Enabled analysis	38	17%
`user-interface.scm:` *109*	Initialisation and Petri net syntax	32	15%
	Running and firing interface	14	6%
	Read-eval-print-loop interpreter	23	10%
	Testing (marking & enabled status)	32	15%

Figure 2: Solution size (in lines of code, LOC)

3 Evaluation

Development-effort-benchmarks Figure 2 summarises the size of the implementation in terms of lines of code, excluding empty lines and pure comments. The difference between the size of the solution parts and their source code files is due to boilerplate code for library imports and exports not being accountable to any certain task. Also, the abstract syntax graph accessors are boilerplate code that could be generated and should not be counted. They are mostly one liners to introduce convenient functions for node constructions and child and attribute querying. For example, in the listings of Appendix A we will write `(->target n)` to query the target of an activity edge. *RACR* provides generic query functions however, such that the query would be `(ast-child 'target n)` (cf. reference manual [1, Chapter: *Abstract Syntax Trees*]). To this end we specify the abstract syntax graph access function `(define (->target n) (ast-child 'target n))` which is obviously boilerplate. Finally, note that the implementation of user interface functionality makes up huge parts of the implementation (in case of the activity diagram language 48%; for the Petri net language 46%). To develop language user interfaces is not subject of *RACR* however; input parsing and abstract syntax tree instantiation therefore should also be excluded.

Performance-benchmarks Figure 3 presents the results of benchmarking the performance test cases given by the tool contest. The benchmarks have been executed on a *MacBook Air (Mid 2011)* with a 1.7GHz *Intel Core i5* CPU, 4GB 1333MHz DDR3 RAM and *Mac OS 10.11.3*. As *Scheme* system *Larceny 0.98 (General Ripper)*[3] was used. Times were measured using the `time` command of *UNIX* without warming up the *Larceny* virtual machine just by execution from *Bash*. Each test case was performed with increasing numbers of translation tasks, such that the actual times spent for parsing, well-formedness

[3] `http://www.larcenists.org` and `https://github.com/larcenists/larceny`

Tasks performed	Test case				Time spent
(later include previous)	1	2	3_1	3_2	*(low / high / average)*
Activity diagram parsing	762 / 762	763 / 763	797 / 797	641 / 641	45% / 92% / 53%
Activity diagram well-formedness	859 / 97	869 / 106	983 / 186	643 / 2	0% / 11% / 7%
Petri net generation	973 / 114	989 / 120	1125 / 142	647 / 4	1% / 8% / 7%
Petri net well-formedness	1141 / 168	1158 / 169	1296 / 171	655 / 8	1% / 11% / 9%
Petri net enabled	1167 / 26	1185 / 27	1376 / 80	656 / 1	0% / 5% / 2%
Petri net execution...	1617 / 450	1555 / 370	1768 / 392	699 / 43	6% / 28% / 22%
...using enabled passes	2274 / 1107	1229 / 44	1462 / 86	718 / 62	4% / 49% / 23%
Incremental savings (enabled analyses not cached)					*(low / high / average)*
Petri net execution...	9894 / 8727	8171 / 6986	8707 / 7331	916 / 260	83% / 95% / 95%
...using enabled passes	18889 / 17722	1536 / 351	1818 / 442	1057 / 401	81% / 94% / 93%

Figure 3: Time measurements (times in ms: total / task-only)

checks, Petri net generation, the first enabled analysis and actual execution can be investigated. For example, test case 2 spent 27ms for the very first evaluation of the enabled status of all transitions of its generated Petri net making a total of 1185ms with further Petri net execution excluded. Of this 1185ms, 120ms and 169ms where spent to generate the Petri net and check its well-formedness, 106ms to check well-formedness of the activity diagram and 763ms to parse the test file and construct an abstract syntax tree. The activity diagram parsing time includes loading the *Larceny* virtual machine, *RACR* and the activity diagram and non-hierarchical Petri net languages. The percentage of time spent for a certain task (last column of the first six rows in Figure 3) is w.r.t. a test case's total execution time if no enabled passes are used. It is only shown for the test cases with the lowest and highest percentage spent for each task, highlighted by colouring the time of the respective test case. The average percentage is the sum of all test cases to perform a certain task divided by the sum of their total execution times. Again, readers should exclude parsing times when judging *RACR*. The implementation of a variant with enabled passes requires three more lines of code. As described in Section 2.2, it just fires all enabled transitions each execution loop iteration instead of a single. Of course, if there are no forks the enabled pass variant wastes time to filter all enabled transitions. If there are parallel branches however, enabled passes improve execution performance a lot. Thanks to the incremental enabled analysis, the execution without enabled passes nevertheless performs surprisingly well.

Regardless if enabled passes are used or not, the benefits of *RACR's* incremental evaluation are profound (second part of Figure 3). The incremental savings, compared to a non-cached enabled analysis, vary between 83–95% without and 81–94% with enabled passes. On average 95% and 93% of actual Petri net execution time are saved respectively. These measurements are not disturbed by dynamic attribute dependency graph maintenance, since no further attributes used for Petri net execution depend on the non-cached enabled analysis. The incremental savings would be dramatic if caching is deactivated for further attributes (for example Petri net name analysis or well-formedness). The benefits of *RACR's* incremental evaluation are also confirmed by the ratio of parsing to interpretation time. At least 45% of time is spent just to load the *Larceny* virtual machine and parse the activity diagram; on average 53%! Considering, that the textual diagram language given by the tool contest is simple – a hand-written recursive decent parser with a single look ahead is used – the amount of time required for all other activities is very low.

References

[1] Christoff Bürger (2012): *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. Technical Report TUD-FI12-09, Lehrstuhl Softwaretechnologie, Technische Universität Dresden. Updated version distributed with *RACR* at `https://github.com/christoff-buerger/racr`.

[2] Christoff Bürger, Sven Karol, Christian Wende & Uwe Aßmann (2011): *Reference Attribute Grammars for Metamodel Semantics*. In Brian Malloy, Steffen Staab & Mark van den Brand, editors: *Software Language Engineering: Third International Conference*, *Lecture Notes in Computer Science* 6563, Springer, pp. 22–41.

[3] R. Kent Dybvig (2009): *The Scheme Programming Language*, 4 edition. MIT Press.

[4] Object Management Group (2013): *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Technical Report, Object Management Group. Version 1.1.

[5] Görel Hedin (2000): *Reference Attributed Grammars*. *Informatica (Slovenia)* 24(3), pp. 301–317.

[6] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. Technical Report, Business Informatics Group, Vienna University of Technology.

[7] Wolfgang Reisig (2013): *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer. English translation of *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*.

[8] Pieter Van Gorp & Steffen Mazanek (2011): *SHARE: a web portal for creating and sharing executable research papers*. *Procedia Computer Science* 4, pp. 589–597.

A Activity Diagram Language

The abstract syntax graph of the activity diagram language corresponds to the metamodel given in the task description [6, Figure 1].

A.1 Abstract Syntax Tree Scheme

The metaclasses and their composite relations determine the solution's abstract syntax tree scheme. For example, the following excerpt of the abstract syntax tree scheme specifies the metaconcepts `Activity`, `Variable`, `ActivityEdge` and `ControlFlow`:

```
(ast-rule 'Activity->name-Variable*-ActivityNode*-ActivityEdge*)
(ast-rule 'Variable->name-type-initial)
(ast-rule 'ActivityEdge->name-source-target)
(ast-rule 'ControlFlow:ActivityEdge->guard)
```

Note, that names starting lowercase on right-hands (following the `->`) denote terminal children – i.e., ordinary properties – whereas names starting uppercase denote non-terminals – i.e., composite relations. Unbounded composites (Kleene closures/unbounded repetitions) are denoted by a `*` following the respective non-terminal. Analogous to the task description's metamodel, `ControlFlow` inherits from `ActivityEdge` denoted by `:ActivityEdge`. By doing so control-flow edges not only inherit the name, source and target properties of activity edges, but also their attributes and therefore semantic analyses (in terms of metamodelling the attributes of a reference attribute grammar are derived properties and methods [2]).

A.2 Name, Type and Well-formedness Analyses

The main purpose of the attribute-based semantic analyses of the activity diagram language is, besides the actual generation of Petri nets, the provision of information convenient for such code generation. This comprises the construction of a graph structure encoding all information required for code generation (name analysis) and checks that ensure diagrams are also valid such that the generated Petri nets do not misbehave (type and well-formedness analyses).

As a name analysis example consider the association of activity edges with nodes (`incoming` and `outgoing` attribute). To do so, hash maps from node names to their respective incoming and outgoing edges are constructed. Given these maps, each node can just lookup its own name to determine its edges:

```
1 (ag-rule
2  incoming ; List of incoming edges of a node.
3  (Activity        (lambda (n) (make-connection-table ->target (=edges n))))
4  (ActivityNode    (lambda (n) (hashtable-ref (=incoming (<- n)) (->name n) (list)))))
```

To query an attribute for its value we just write `(=attribute-name n)`; to query an abstract syntax tree child or parent we just write `(->child/terminal-name n)` and `(<- n)` respectively. In all three cases, `n` is the context node, i.e., the node the attribute is associated with/the node which has the child/the node whose parent is queried respectively. The lookup of incoming edges at an activity node `n` works as follows (Line 4): get the diagram's hash table via `(=incoming (<- n))` and query it with the activity node's name; if it has no entry, return the empty list (the last `(list)` on Line 4). To construct the actual table (Line 3), we just call a support function which given an access function `->` and list of abstract syntax tree nodes queries all its elements and adds them to a newly constructed hash table according to their `->` values[4]. In our case the arguments are just all edges of the diagram (supported by the `=edges` attribute) and the target query function `->target`. Likewise, the name analysis provides attributes to lookup variables, nodes and edges (`v-lookup`, `n-lookup` and `e-lookup` attribute), the source and target of edges (`source` and `target` attribute) and the initial node (`initial` attribute).

Given the name analysis, type analysis is easy to implement (`well-typed?` attribute). Consider for example unary expressions, which, according to the metamodel, must be negations:

```
1 (UnaryExpression
2  (lambda (n)
3    (define ass (=v-lookup n (->assignee n)))
4    (define op (=v-lookup n (->operand1 n)))
5    (and ass op (eq? (->type op) Boolean) (eq? (->type ass) Boolean)))))
```

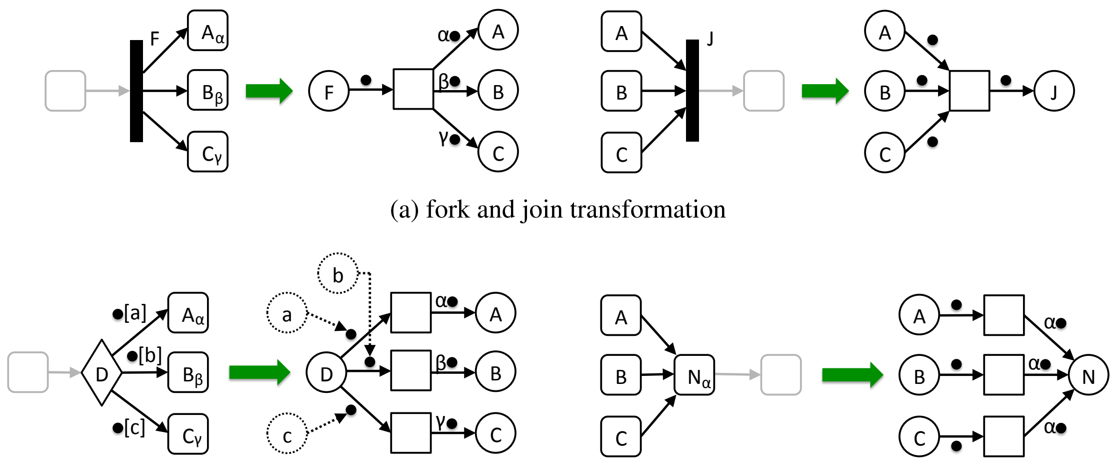
First we lookup the variable to write the result to and the negated operand (Lines 3 & 4). Afterwards we ensure both exist and are indeed of type Boolean (Line 5).

Based on type and name analyses we can check well-formedness. As an example consider decisions and executable nodes:

```
1 (DecisionNode     (lambda (n) (and (in n = 1) (out n >= 1) (guarded n #t))))
2 (ExecutableNode   (lambda (n) (and (in n = 1) (out n = 1) (guarded n #f)
3                                 (for-all =well-typed? (=expressions n)))))
```

In both cases we use three support functions. The `in` and `out` functions ensure the node has a certain number of incoming and outgoing edges. The `guarded` function asserts, depending on its boolean argument, whether outgoing edges must be control-flows (in case of *true* they must be, otherwise not). Decisions must have a unique incoming edge, at least one outgoing edge and their outgoing edges must be control-flows (Line 1). Executable nodes must have a unique incoming and outgoing edge which is not a control-flow (Line 2). Furthermore, all their expressions must be type correct (Line 3).

[4]The implementation is straightforward and based on `hashtable-update!` provided by *Scheme* [3].



(a) fork and join transformation

(b) decision transformation

(c) initial, final, merge & executable node transformation

Figure 4: Activity Diagram to Petri net transformation rules ©Christoff Bürger

A.3 Code Generation

A.3.1 Places, Transitions & Arcs

Figure 4 summarises the code generation rules. For each activity node and variable a Petri net place is constructed (`places` attribute). In case of variables, the place contains their respective initial value as token. Otherwise, only the place of the initial node has a token. The general rule for generating transitions (`transitions` attribute) is, that given an activity node, a transition is constructed for each of its predecessor nodes. The transition just consumes a token from the predecessor's place and puts it into the node's place (Figure 4 (c)).

Special means in case of control-flow edges and executable node's expressions have to be taken however. Consider Figure 4 (b). In case of control-flow edges, the respective guard must be checked before any token is consumed. To do so, it is sufficient to lookup the value encoded in the token of the place which encodes the variable the guard refers to. Further, before a token is placed by an outgoing arc, all expressions of the node its destination place represents must be executed. In Figure 4, these two actions are represented by dashed arcs from variable places to guarded input arcs and by Greek letters representing the expressions to execute.

Forks and joins are exceptions form these default rules however, because of their parallelising and synchronising semantics. In case of a fork, all its outgoing edges yield a single transition. Likewise, all incoming edges of a join are translated to a single transition (Figure 4 (a)). As an example consider the implementation of the `transitions` attribute of joins:

```
(JoinNode
 (lambda (n)
   (define incoming (=incoming n))
   (list
    (pn::Transition
     (->name (car incoming))
     (map >>? incoming)
     (list (n>> (car incoming)))))))
```

Based on the join's incoming edges (Line 3) a new transition named like the "first" incoming edge is constructed (Lines 5 & 6). The transition has a single outgoing arc (Line 8) and for each incoming edge of the join one incoming arc[5](Line 7). These arcs are constructed by the `>>?` and `n>>` support functions which given an activity edge construct a new incoming or outgoing arc respectively. Incoming arcs consist of a single symbolic name referencing the source place the arc is consuming tokens from and a list of functions, each selecting a token to consume. Outgoing arcs consist of a single symbolic name referencing the target place the arc is producing tokens to and a single function that given all consumed tokens computes the produced ones. Consider the construction of incoming arcs via `>>?`:

```
1 (define (>>? n) ; Construct incoming Petri net arc for activity edge.
2   (if (ast-subtype? n 'ControlFlow)
3       (pn::Arc (->source n) (list (=v-accessor (=v-lookup n (->guard n)))))
4       (pn::Arc (->source n) (list (lambda (t) #t)))))
```

First, it is checked if the given activity edge is a control-flow (Line 2). If it is, the consumption function has to query the value of its guard, i.e., given a consumable token the arc is enabled if, and only if, the guard's value is *true*. To enable the querying of variable values at runtime (i.e., during Petri net execution), we construct special access functions that return the value of the token of the variable's place (`v-accessor` attribute). In case of a control-flow, `>>?` therefore finds the guard variable in the activity diagram via `=v-lookup` and defines its access function to be the consumption function of the arc (Line 3). If the argument of `>>?` is not a control-flow, the consumption function just returns *true*, i.e., whenever a consumable token is given the arc is enabled (Line 4). In both cases, the place to consume a token from is the given activity edge's source, i.e., `(->source n)`. All of this happens before runtime. When the generated Petri net is executed the consumption function and source are already settled by the code generation; no runtime lookup is required.

A.3.2 Variables, Expressions & The Execution of Executable Nodes

As already explained, each variable is translated to a place containing a single token encoding its value. The `v-token` attribute refers for each variable to the respective token encoding its runtime value. Its implementation queries the place representing the variable (`places` attribute), its list of tokens and finally the list's first and only child:

```
(ag-rule
 v-token ; The Petri net token encoding the runtime value of the variable.
 (Variable     (lambda (n) (ast-child 1 (pn:->Token* (=places n))))))
```

Remember, that *RACR* is incremental and caches all attributes. As long as information `places` depends on is not changed – like in the given tool contest scenario – it will construct a new Petri net place only the first time queried; further queries will evaluate to this very place. This caching behaviour holds for all attributes of the activity diagram language. Based on `v-token`, implementing `v-accessor` is straightforward:

```
(ag-rule
 v-accessor ; Function returning the runtime value of the variable.
 (Variable     (lambda (n) (define token (=v-token n)) (lambda x (pn:->value token)))))
```

First, lookup the token representing the variable's value using `v-token`. Afterwards, return a function in whose closure the token is and which uses the Petri net language to query its value via `pn:->value`.

After investigating how runtime values of variables are encoded and can be accessed, it remains to show how they are changed by expressions. The `computation` attribute generates for each expression a function assigning its left-hand the value of its right-hand. For example, consider unary expressions:

[5]Incoming and outgoing arcs are consuming and producing tokens when a transition is fired respectively.

```
1 (UnaryExpression
2  (lambda (n)
3    (define assignee (=v-token (=v-lookup n (->assignee n))))
4    (define op1 (=v-accessor (=v-lookup n (->operand1 n))))
5    (define op (->operator n))
6    (lambda () (rewrite-terminal 'value assignee (op (op1))))))
```

First, the token representing the assignee is looked up (Line 3); afterwards, the access function of the operand variable and the operation to perform (Lines 4 & 5). These information are the closure of the function to construct. The function itself uses *RACR's* `rewrite-terminal` function to change the value of the assignee to the one computed by applying the operator on the value the operand's value access function returns (Line 6). Again, all lookups are at generation time of the Petri net and not runtime.

The `computation` attribute is defined for every activity node. It generates a function whose execution represents the execution of the respective activity node at runtime. This comprises three runtime actions: (1) tracing the node's execution, (2) computing its expressions if any (i.e., if the node is an executable node) and (3) establishing its offers for successor nodes:

```
1 (ActivityNode
2  (lambda (n)
3    (define executed (->name n))
4    (lambda x (trace executed) (list #t))))
5 (ExecutableNode
6  (lambda (n)
7    (define executed (->name n))
8    (define computations (map =computation (=expressions n)))
9    (lambda x (trace executed) (for-each (lambda (f) (f)) computations) (list #t))))
```

Note, that the computation functions generated by the `computation` attribute accept arbitrary many arguments and always return a singleton list with element *true*. Their tracing and expression execution is obvious (Lines 4 & 9); how token offers are established we still have to clarify however.

As already explained, for each activity node a place is generated. A token in such a place indicates that the activity node provides an offer to its successors. According to the semantics of activity diagrams, the offers of an activity edge are provided immediately *after* executing its expressions. The computation function of an activity node therefore has to be executed immediately *before* a token is put into its respective place, i.e., whenever an outgoing arc of a transition places a token in its place. Thus, outgoing arcs must apply the computation function of their target. The implementation of `>>n` therefore is:

```
(define (n>> n) ; Construct outgoing Petri net arc for activity edge.
  (pn::Arc (->target n) (=computation (=target n))))
```

As explained before, an outgoing arc consists of a symbolic name referencing the target place and a production function that given the consumed tokens computes the ones placed in its target place. The functions generated by the `computation` attribute are valid production functions; they accept arbitrary many consumed tokens and place a single *true* token.