fUML Activity Diagrams in *RACR*

– A *RACR* Solution of *The TTC 2015 Model Execution Case* –

Christoff Bürger
Department of Computer Science
Faculty of Engineering, LTH
Lund University
Lund, Sweden
`christoff.burger@cs.lth.se`

This paper summarises a *RACR* solution of *The TTC 2015 Model Execution Case* [6]. *RACR*[1] [1] is a metacompiler library for *Scheme* [3]. Its most distinguished feature is the seamless combination of reference attribute grammars [5] 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 [4, 9] to executable Petri nets [7]. 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 Solution Overview

The solution 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. The second executes generated Petri nets – it is a Petri net interpreter.

Input for the activity diagram language are textual diagram specifications as given by the tool challenge[2] [6]. A hand written recursive-decent parser is used to instantiate attributed abstract syntax trees of the diagram language (the parser is straightforward and not investigated in the following). Reference attribute based analyses extend such abstract syntax trees to abstract syntax graphs that represent the diagram graph. Based on this name analysis the type well-formedness of activity computations and control-flow guards and the validity of the diagram are checked. Finally, further attributes are used to compute a respective Petri net whose execution corresponds to the execution semantics of the given diagram.

The implementation of the Petri net language is similar. Reference attributes are used to extend the abstract syntax tree encoding a net to an abstract syntax graph which can be checked for well-formedness. The actual enabled analyse and transition semantics are implemented by further reference attributes and respective rewrites using them. Execution of a net corresponds to a simple loop which finds an enabled transition using the enabled analysis, deletes its consumed tokens and places the produced ones. Thereby, deleted and produced tokens are refered to by the reference attributes of the name and enabled analyses. In the end, reference attribute analyses are used to guide rewriting and reduce its implementation efforts.

The next two sections present the sketched solution in detail; an evaluation follows.

---

[1] `https://github.com/christoff-buerger/racr`

[2] `https://code.google.com/a/eclipselabs.org/p/moliz/source/browse/?repo=ttc2015`

Submitted to:
TTC 2015

2 Activity Diagram Language: From Activity Diagrams to Petri nets

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

2.1 Abstract Syntax Tree Scheme

The metaclasses and their composit 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-hand sides (following the `->`) denote terminal children – i.e., ordinary properties – whereas names strating uppercase denote non-terminals – i.e., composite relations. Unbounded composites (Klenee 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]).

2.2 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).

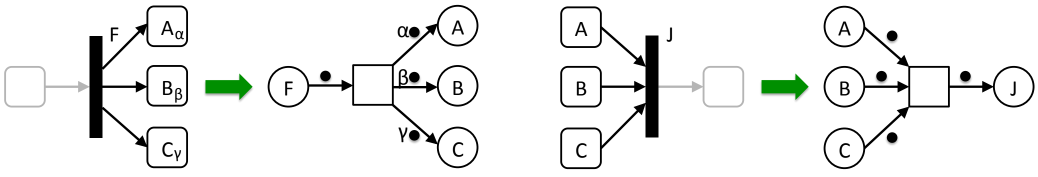
As a name analysis example consider the association of activity edges with nodes (`incoming` and `outgoing` attribute). To do so, hashmaps from node names to their respective incoming and outgoing edges are constructed. Given these maps, each node can just lookup its own name to determine its edges:

```
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)))))
```

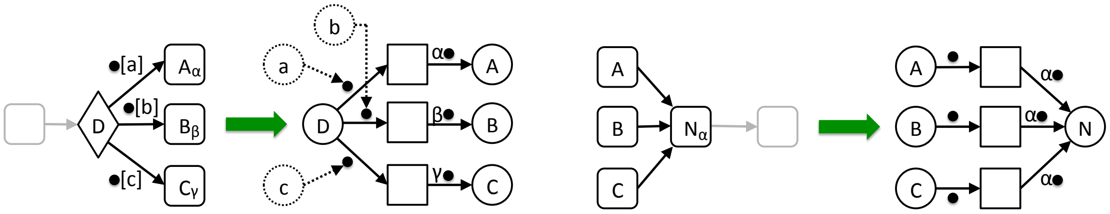
To query an attribute for its value we just write `(=attribute-name n)`; to query an abstract syntax tree child or parent we just write `(->child/terminal-name n)` and `(<- n)` respectively. In both cases, `n` is the context node, i.e., the node the attribute is associated with/which has the child/whose parent is queried respectively. The lookup of incoming edges at an activity node `n` works as follows (Line 4): Get the diagram's hashtable 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 accessor function `->` and list of abstract syntax tree nodes queries all its elements and adds them to a newly constructed hashtable according to their `->` values[3]. 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

---

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



(a) fork and join transformation



(b) decision transformation (c) inital, final, merge & executable node transformation

Figure 1: Activity Diagram to Petri net Transformation Rules

name analysis provides attributes to lookup variables, nodes and edges (`v-lookup`, `n-lookup`, `e-lookup`), the source and target of edges (`source`, `target`) and the initial node (`initial`).

Given the name analysis, type analysis is easy to implement. 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:

```
(DecisionNode     (lambda (n) (and (in n = 1) (out n >= 1) (guarded n #t))))
(ExecutableNode   (lambda (n) (and (in n = 1) (out n = 1) (guarded n #f)
                                   (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 if its outgoing edges can be control-flows. Decisions must have a unique incoming edge, at least one outgoing edge and their outgoing edges must be control-flows. Executable nodes must have a unique incoming and outgoing edge which is not a control-flow. Further, all their expressions must be well-typed.

2.3 Code Generation

2.3.1 Places, Transitions & Arcs

Figure 1 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 1 (c)).

Special means in case of control-flow edges and executable node's expressions have to be taken however. Consider Figure 1 (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 its respective place. Further, before a token is placed by an outgoing arc, all expressions of the node its destination place represents must be executed. In the figure these two actions are represented by dashed arcs from variable places to 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 1 (a)). As an example consider the implementation of the `transitions` attribute of joins:

```
1 (JoinNode
2  (lambda (n)
3   (define incoming (=incoming n))
4   (list
5    (pn::Transition
6     (->name (car incoming))
7     (map >>? incoming)
8     (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[4](Line 7). The incoming and outgoing arcs are constructed by the two support functions `>>?` and `n>>` respectively, which given an activity edge construct a new Petri net arc. Petri net arcs consist of a single symbolic name referencing the source/target place the arc is consuming/producing tokens from/to and a list of functions, each deciding for a given token if it is consumable (i.e., of the expected type)/a single function that given all consumed tokens computes the produced ones[5]. 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 accessor 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 accessor 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

---

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

[5] Thus, coloured, weighted Petri nets are supported (arbitrary many tokens of different types can be consumed from a single incoming arc and arbitrary tokens produced by a single outgoing arc). For the semantics of activity diagrams however, we only need one token type (except tokens encoding variable values) and places always have at most a single 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.

2.3.2 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 challenge scenario – it will construct a new Petri net place only the first time queried; further queries will evaluate to this very place. This caching behavior 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:

```
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 lookedup (Line 3); afterwards, the accessor 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 accessor 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:

```
(ActivityNode
 (lambda (n)
   (define executed (->name n))
   (lambda x (trace executed) (list #t))))
(ExecutableNode
 (lambda (n)
   (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.

3 Petri net Language: Incremental Execution of Nets

Similar to the activity diagram language, also the Petri net language provides attributes to lookup transitions and the source/target places of arcs. The enabled analysis 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. To fire transitions boils down to reuse the enabled analysis to delete consumed tokens and add produced which is accomplished using *RACR's* `rewrite-delete` and `rewrite-add` functions [1]. Because of space considerations we will not investiagte the source code of the Petri net language however. Interested readers can consult its implementation, in particular the `enabled?` attribute and `fire-transition!` function.

Important for the following benchmarks 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 attribute's value 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. For example, when a new value is assigned to a variable via `rewrite-terminal` as shown in the previous section, the enabled status of transitions depending on its value is reevaluated, if, and only if, they either were enabled or, although all tokens they consume are provided, still were disabled. Without special implementation efforts by the developer, *RACR* optimises the execution semantics.

4 Evaluation

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

Source Code File	Solution Part (language task)	LOC	
Activity diagram language (507):		*499*	
`analyses.scm`: *255*	AST specification	18	4%
	ASG accessors (constructors, child & attribute accessors)	65	13%
	Name analysis	32	6%
	Type analysis	23	5%
	Well-formedness	32	6%
	Petri net generation	90	18%
`parser.scm`: *219*	Parsing	214	43%
`user-interface.scm`: *33*	Initialisation & execution	25	5%
Petri net language (255):		*200*	
`analyses.scm`: *102*	AST specification	9	5%
	ASG accessors (constructors, child & attribute accessors)	32	16%
	Name analysis	13	7%
	Well-formedness	10	5%
	Enabled analysis	29	15%
`execution.scm`: *43*	Running and firing semantics	31	16%
`user-interface.scm`: *80*	Initialisation & Petri net syntax	33	17%
	Read-eval-print-loop interpreter	19	10%
	Testing nets (marking & enabled status)	24	12%

Figure 2: Solution Size (lines of code, LOC)

Tasks Performed	Test Cases (`testperformance_variant`)				Time Spend
(later tasks include previous ones)	1	2	3_1	3_2	*(lowest / highest / average)*
Activity diagram parsing	831 / 831	871 / 871	875 / 875	718 / 718	41% / 86% / 50%
Activity diagram well-formedness	926 / 95	1017 / 146	1079 / 204	739 / 21	3% / 11% / 7%
Petri net generation	1042 / 116	1061 / 44	1196 / 117	741 / 2	0% / 6% / 4%
Petri net well-formedness	1220 / 222	1230 / 169	1466 / 270	746 / 5	1% / 14% / 10%
Petri net execution	2026 / 806	1776 / 546	1912 / 446	831 / 85	10% / 40% / 29%

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

mostly one liners to introduce convenient functions for node constructions and child and attribute querying. For example, in the previous listings we wrote `(->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)`. To this end we specify the abstract syntax graph accessor `(define (->target n) (ast-child ’target n))` which is obviously boilerplate. Finally, note that the implementation of user interface functionallity makes up huge parts of the implementation (in case of the activity diagram language 48%; for the Petri net language 39%). To develop language user interfaces is not subject of *RACR* however; input parsing and abstract syntax tree instantiation therefore should also be excluded.

Figure 3 presents the results of benchmarking the performance test cases given by the tool challenge. The becnhmarks have been executed on a *MacBook Air (Mid 2011)* with a 1.7GHz *Intel Core i5* CPU, 4GB 1333MHz DDR3 RAM and *Mac OS 10.10.3*. As *Scheme* system *Larceny 0.98 (General Ripper)*[6] was used. Each test case was performed with increasing numbers of tranlation tasks, such that the actual times spend for parsing, well-formedness checks, Petri net generation and their actual execution can be investigated. For example, `testperformance_variant2.ad` spend 169ms on checking the well-formedness of its

---

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

Petri net making a total of 1230ms with Petri net execution excluded. Of this 1230ms, 44ms where spend to generate the Petri net, 146ms to check well-formedness of the activity diagram and 871ms 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 Petri net languages. The percentage of time spend for a certain task is w.r.t. a test case's total execution time. It is only shown for the test cases with the lowest and highest percentage spend 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 devided by the sum of their total execution times. Again, readers should exclude parsing times when judging *RACR*.

References

[1] Christoff Bürger (2012): *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. Technical Report TUD-Fl12-09, Lehrstuhl Softwaretechnologie, Technische Universitt Dresden.

[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 (2010): *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik, Vieweg+Teubner Verlag.

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

[9] James Rumbaugh, Ivar Jacobson & Grady Booch (1999): *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison-Wesley.