NISTIR 7469

Rendering UML Activity Diagrams as Human-Readable Text

David Flater
Philippe A. Martin
Michelle L. Crane

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7469

Rendering UML Activity Diagrams as Human-Readable Text

David Flater
Software Diagnostics and Conformance Testing Division
Information Technology Laboratory

Philippe A. Martin
Griffith University, Australia

Michelle L. Crane
Queen's University, Canada

November 2007

Rendering UML Activity Diagrams as Human-Readable Text

David Flater
National Institute of Standards and Technology, U.S.A.

Philippe A. Martin
Griffith University, Australia

Michelle L. Crane
Queen's University, Canada

November 2007

Abstract

We describe a modification of the Petri Net Linear Form notation to support the rendering of Unified Modeling Language (UML) Activity Diagrams as human-readable text. This new notation, called the Activity Diagram Linear Form, allows UML Activity Diagrams to be expressed in an alternate form with the superior accessibility, compatibility, and simplicity of use of a plain text representation. For some applications, these benefits greatly outweigh the æsthetical and pedagogical advantages of a visual representation.

1 Introduction

An Activity Diagram as defined by the Unified Modeling Language (UML) [1] models procedural actions, the sequencing of actions (control flow), and conditions for coordinating behaviors. Basic Activity Diagrams may be elaborated with UML features to describe the object flow between actions (inputs and outputs) and other relevant information.

A human-readable text form of Activity Diagrams may be used for any number of reasons, such as:

- Graphical representations require special tooling to work with, are time-consuming to create, require conversion for embedding in e-mail or documents, take up a lot of space on a screen, and are often resistant to standard searching or copy/paste operations.
- Graphical representations are not processable by screen readers (assistive technology that converts documents to speech).
- For a programmer who is building new applications, getting a prototype running with a human-readable text format for input or output takes significantly less up-front investment of effort than starting out with a graphical representation or a full-featured interchange format.
- Extensible Markup Language (XML) [2] based languages are cumbersome to use for this purpose because XML syntax rules prevent the definition of specialized punctuation and force graphically adjacent elements to be separated from one another more often than a specialized notation would.

In this document, we describe the Activity Diagram Linear Form (ADLF), a modification of the Petri Net Linear Form (PNLF) [3]. Note that PNLF and ADLF were developed for specific applications. No attempt has been made to handle every possible UML feature that may legally appear in an Activity Diagram. Only the notation pertinent to those features that were required in the original application is defined.

This document is structured as follows: Section 2 discusses related work. Section 3 briefly describes PNLF and provides an example. Section 4 describes the differences between ADLF and PNLF and provides examples of the new notation. Section 5 presents two grammars suitable for recognizing ADLF. Finally, Section 6 gives the conclusion.

Henceforth, familiarity with Activity Diagrams [1] and Petri Nets [4] is assumed.

2 Related Work

PNLF was invented by Philippe Martin to address usability and convenience issues that occurred in dealing with both graphical and XML-based Petri Net notations. Inspired by the Conceptual Graph Linear Form [5], it represents both the graph structure (places, transitions, and arcs) and execution state (explicit tokens) of "plain old" (non-extended) Petri Nets.

There are several relevant XML-based interchange formats that are not intended to be human-readable. For Petri Nets there is the Petri Net Markup Language [6]. For UML there is the XML Metadata Interchange (XMI) Specification [7] and the Diagram Interchange Specification [8].

The UML Human-Usable Textual Notation [9] is designed to conform to human-usability criteria. However, it is defined in terms of the Meta Object Facility (MOF), which only directly supports class modelling. While any UML concept can ultimately be abstracted using classes and expressed using the MOF, for Activity Diagrams this would entail significant obfuscation.

Semantic differences between Petri Nets and UML Activity Diagrams are explored by Störrle and Hausmann [10]. These differences are substantial enough that one cannot define a clean notation that merges concepts from Petri Nets and Activity Diagrams; hence the present work is an adaptation of PNLF rather than an extension or application of it.

3 PNLF Notation

In the following discussion of PNLF and plain Petri Nets, the term *node* refers to either a place or a transition.

The mathematical formalism of plain Petri Nets does not include node names, so node names have no formal meaning in plain Petri Nets. However, node names are commonly used in the graphical representations of Petri Nets to associate nodes informally with types or concepts relevant to the states and processes being modelled. These external types or concepts are separate from the Petri Nets *per se* and should not be confused with place or transition types that are defined formally within extended Petri Nets such as High-Level Petri Nets [11].

As described by Martin [3], PNLF defines the following:

- The entire Petri Net is represented as one or more statements separated by semicolons. The last statement ends with a period.
- Places are represented by the place name enclosed within parenthesis `()`.
- Transitions are represented by the transition name enclosed within square brackets `[]`.
- Arcs are represented by ASCII arrows (`->` and `<-`).
- Each token is represented by the `@` character.

- Multiple branches to or from a node are grouped together by curly brackets `{}` and separated by commas.
- Coreference variables are used for multiple references to the same place or transition, which unavoidably result from the linearization of the graph. They are represented by an identifier preceded by an asterisk (`*`).
- Identifiers may be enclosed within matching single or double quote characters or may be left unquoted. An unquoted identifier can normally contain only alphanumeric characters, hyphens, and underscores, and may not begin or end with a hyphen. However, the backslash-escape convention may be used to embed any character at any position within a quoted or unquoted identifier.
- Informal notes and commentary extend from the first occurrence of a double slash (`//`) to the end of the line, as in the C++ programming language, or between `/*` and `*/` delimiters, as in the C programming language.

3.1 Example 1

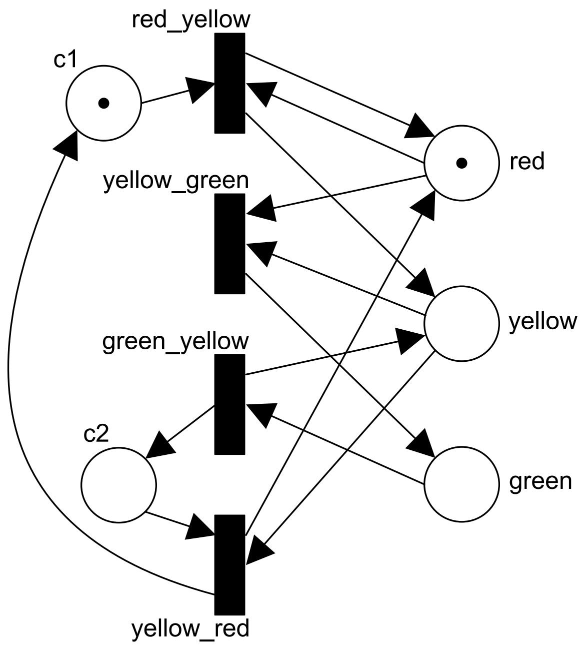

Figure 1: Petri Net for Example 1 (used by permission from Wil van der Aalst [12])

The Petri Net in Figure 1 is translated to the following:

```
[red_yellow *ry]
  { <-(c1 *c1 @),
    ->(red *r @) { ->[*ry], ->[yellow_green *yg] },
    ->(yellow *y)
      { ->[yellow_red *yr] { ->(*c1), ->(*r) },
        ->[*yg]->(green)->[green_yellow] { ->(*y), ->(c2)->[*yr] }
      }
  }.
```

This example demonstrates several elements of PNLF style:

- For best readability, all the outputs of a node should be presented together. Some inputs may also be given.
- All the outputs of an implicit OR-split (where arcs lead from a given place to two or more transitions, resulting in nondeterministic execution) should be presented together.
- When a coreference variable is referenced after its initial appearance, the node name need not be repeated. Different nodes can have the same name but a coreference variable is unique to a node.
- Paths should be explored/presented as soon as readability permits it. (The arcs to and from `*yr` could have been presented later, but this would have made the understanding of the graph more difficult.)

More examples using PNLF can be found at the cited location [3].

4 ADLF Notation

While PNLF is suitable for describing Petri Nets, ADLF is used for describing UML Activity Diagrams. To create ADLF, PNLF was adapted in the following ways:

- Since they do not appear in Activity Diagrams, explicit tokens (`@`) are not supported.
- Action nodes are represented by the node name in parenthesis `()`, which imitates the rounded shape of action nodes in Activity Diagrams.
- Object nodes are represented by the node name in square brackets `[]`, which imitates the rectangular shape of object nodes in Activity Diagrams.
- Both sequential control and object flows are indicated with ASCII arrows (`->` and `<-`).[1]
  - A guard condition qualifying a flow is inserted immediately before the affected flow (adjacent to the `-`) following the syntax rules for an identifier (i.e., either with no enclosing symbols or delimited by quotes).
  - An input parameter multiplicity (such as 0..1) is inserted immediately after the affected flow (adjacent to the `<` or `>`) with no enclosing symbols.
- The following UML control nodes appear with their standard names: `<InitialNode>`, `<Fork Node>`, `<JoinNode>`, `<DecisionNode>`, `<MergeNode>`, `<ActivityFinal>`, and `<FlowFinal>`. They are distinguished from object and action nodes by being enclosed between a less-than sign and a greater-than sign `<>`, which imitates the diamond shape of decision and merge nodes in Activity Diagrams. (The other control nodes have different shapes that are not easily imitated in plain text, so the same notation is used consistently for all control nodes.)
- Coreference variables, curly braces, commas, semicolons, and the period are used in a manner analogous to how they are used in PNLF.

[1]The option to use leftward arrows preserves, to the extent possible, a degree of freedom that is enjoyed by the graphical representation, where arrows may point in any direction. Nevertheless, leftward arrows are neither strictly necessary in theory nor heavily used in practice, and only one of the two grammars provided later in this report supports them.

- Any action, object, or control node may be annotated with name-value pairs or annotation strings included inside of the delimiting parenthesis, square brackets, or less-than and greater-than signs.
  - The preferred method of specifying that the state of an object is x is with a name-value pair such as "state=x."
  - Annotation strings (which unlike comments are not discarded at parsing time but stored in the abstract model with the node they are associated to) may be specified like identifiers or delimited by `(^` and `^)` (similar to C-style comments).
- Informal notes, UML partitions, and any other Activity Diagram features that are not otherwise supported are expressed using comments or annotations.

Object nodes in ADLF have the same expressiveness limitations as the "standalone style" notation in UML Activity Diagrams, where the node name typically indicates only the type of the object node. Using the more expressive "pin style" defined by UML [1, 12.3.44], it is possible to handle complications such as when an action inputs or outputs two different parameters of the same type, or when the types of the input and output parameters are different.

Two examples using conceptual Activity Diagrams follow. Although the content and conventions of the diagrams are not important to their suitability as examples, it may aid understanding to know that some simplifications were made:

- The expansion regions around actions that are performed for every instance of some class are not shown.
- When a particular object may or may not exist depending on predeterminative facts that are external to the process being modelled—a static choice, not a "run-time" decision within the process itself—that object is modelled as an optional parameter to an action. This does not capture the constraint that subsequent actions must wait on this object in those cases where it exists; i.e., if it exists then it is required.

4.1 Example 2

The Activity Diagram in Figure 2 is translated to the following:

```
<InitialNode>-><MergeNode *merge>->("Prepare for election")
  ->["Equipment, voter lists, ballot styles and/or ballots"]-><ForkNode>
    { ->("Prepare for voting (precinct)")-><ForkNode>
        { ->("Gather in-person vote") // Includes early voting.
            ->["Ballots and/or ballot images"]->(Collect *c),
          "Precinct count"->("Count (precinct count)")
            ->["Machine totals"]->0..1(*c)
        },
      ->("Gather absentee / remote votes")->["Ballots and/or ballot images"]
        ->(*c),
      ->("Prepare for voting (central)")->("Wrap up voting (central)" *w)
    };

(*c)->["Ballots, ballot images and/or machine totals"]
  ->("Wrap up voting (precinct)")
    ->["Ballots, ballot images and/or precinct totals"]->(*w)
      ->["Counts" state=certified]->("Wrap up election")-><*merge>.
```

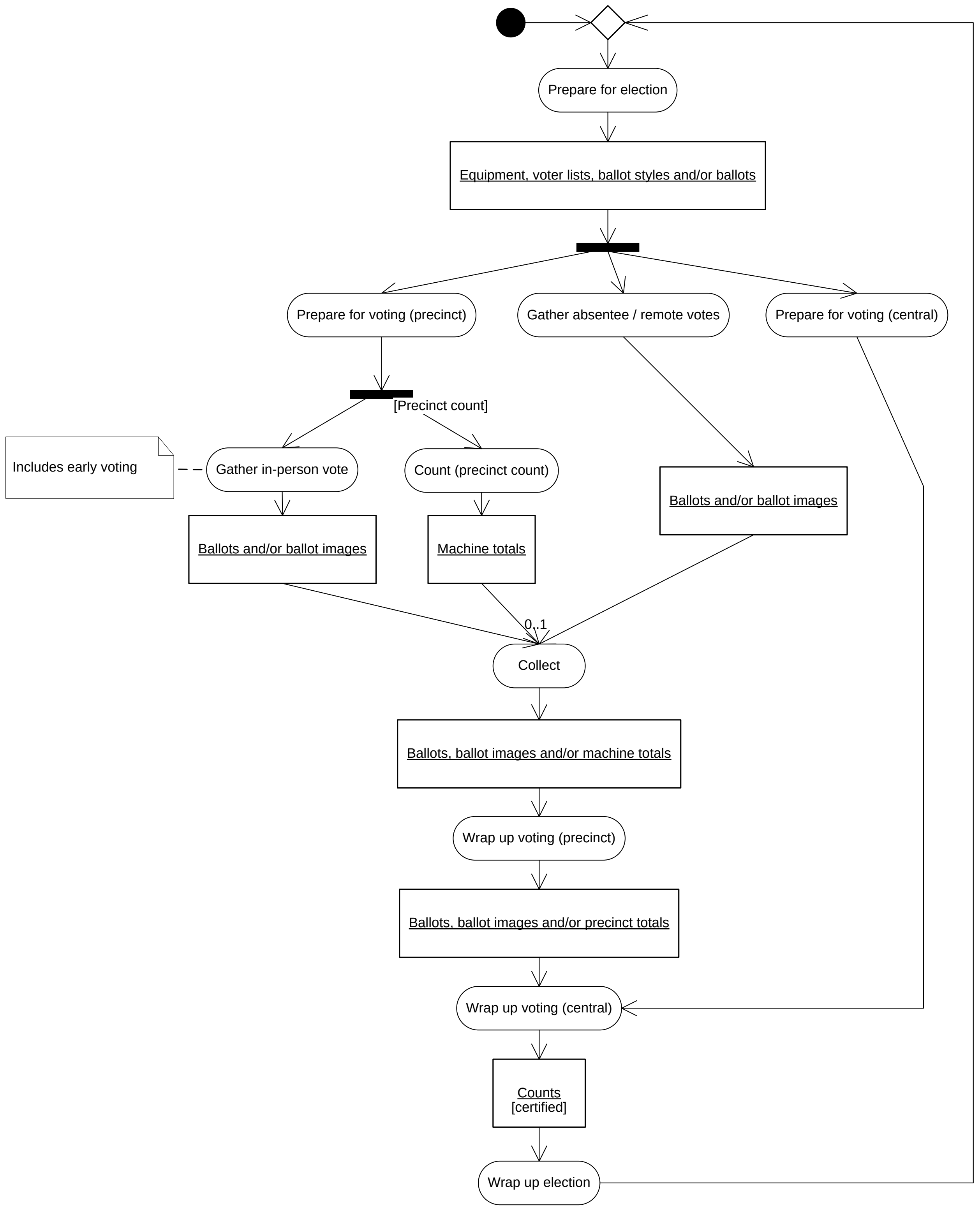
Prepare for election
Equipment, voter lists, ballot styles and/or ballots
Prepare for voting (precinct)
Gather absentee / remote votes
Prepare for voting (central)
[Precinct count]
Includes early voting
Gather in-person vote
Count (precinct count)
Ballots and/or ballot images
Ballots and/or ballot images
Machine totals
0..1
Collect
Ballots, ballot images and/or machine totals
Wrap up voting (precinct)
Ballots, ballot images and/or precinct totals
Wrap up voting (central)
Counts
[certified]
Wrap up election

Figure 2: Activity Diagram for Example 2

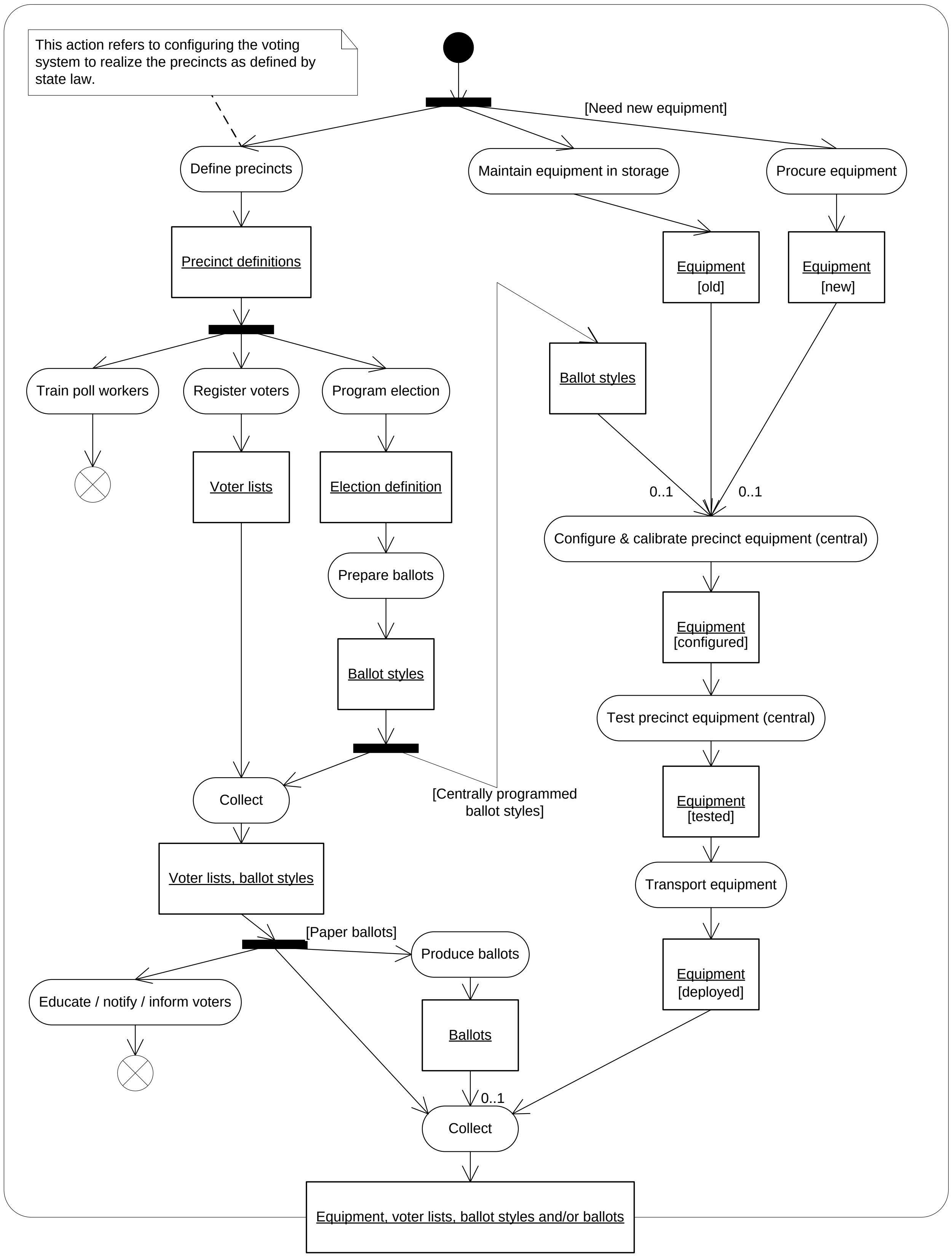

Figure 3: Activity Diagram for Example 3

4.2 Example 3

The Activity Diagram in Figure 3 is translated to the following:

```
<InitialNode>-><ForkNode>
  { ->("Define precincts") // This action refers to configuring the voting system
                           // to realize the precincts as defined by state law.
      ->["Precinct definitions"]-><ForkNode>
        { ->("Train poll workers")-><FlowFinal>,
          ->("Register voters")->["Voter lists"]->(Collect *c1),
          ->("Program election")->["Election definition"]->("Prepare ballots")
            ->["Ballot styles"]-><ForkNode>
              { ->(*c1),
                "Centrally programmed ballot styles"->["Ballot styles"]
                  ->0..1("Configure & calibrate precinct equipment (central)" *cc)
              }
        },
    ->("Maintain equipment in storage")->[Equipment state=old]->(*cc),
    "Need new equipment"->("Procure equipment")->[Equipment state=new]->0..1(*cc)
  };

(*c1)->["Voter lists, ballot styles"]-><ForkNode>
  { ->("Educate / notify / inform voters")-><FlowFinal>,
    ->(Collect *c2),
    "Paper ballots"->("Produce ballots")->[Ballots]->0..1(*c2)
  };

(*cc)->[Equipment state=configured]->("Test precinct equipment (central)")
  ->[Equipment state=tested]->("Transport equipment")->[Equipment state=deployed]
    ->(*c2)->["Equipment, voter lists, ballot styles and/or ballots"].
```

5 ADLF Grammars

A complete syntactic parser developed using Flex [13] and Bison [14] is available for download [15].[2] The semantic analysis/validation has not yet been implemented. A different parser for a subset of ADLF has been developed using JavaCC [16].

Flex and Bison (based on Lex [17] and Yacc [18]) generate an LALR(1) parser while JavaCC generates an LL(1) parser [19]. LR parsers are described as working in a "bottom-up" fashion while LL parsers are described as working in a "top-down" fashion. Additionally, JavaCC supports Extended Backus-Naur Form (EBNF) [20], with optional terms and other features, while Flex/Bison only supports "plain" Backus-Naur Form (BNF) [21]. These differences force one to use different grammatical idioms in order to achieve an elegant implementation. The goal here is to provide near-enough equivalent expressions of ADLF using the idioms that are natural in each environment, thereby enabling elegant implementations using both kinds of parser generators.

[2]Specific software is identified in this paper to support reproducibility of results. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the software identified is necessarily the best available for the purpose.

The grammars from the two parsers are detailed in the following subsections. Please note that these grammars accept input whose interpretation as an Activity Diagram may violate well-formedness constraints on Activity Diagrams, e.g., the constraint that the edges coming into and out of a decision node must be either all object flows or all control flows [1, Section 12.3.22]. While many such constraints could possibly be enforced in a purely grammatical fashion, validating Activity Diagrams expressed in ADLF is separable from defining ADLF *per se* and has been deferred to future work.

5.1 Flex/Bison

5.1.1 Tokens

The regular expressions below are written using Lex syntax, wherein the interpretation of special characters depends on context and may be counterintuitive to some readers. For example, `[...]` gives a set of acceptable characters, but if the first character after the left square bracket is a caret, it signifies all characters *except* that set. Hence `[^^]` means any character except caret. The right square bracket is used to close the set and the hyphen is used to specify a range of characters; however, if either of these appears as the first character after the left square bracket, it is instead interpreted as a plain character. So `[][=*()<>;,.{}]` means any of the characters between the leftmost and rightmost square brackets, including both the left and right square brackets, and `[-a-zA-Z0-9_]` means any alphanumeric character, hyphen, or underscore.

```
[ \f\n\r\t\v]+                       (Whitespace ignored)
"//".*                               (C++ style comments ignored)
"/*"((\*[^/])|[^*])*"*/"             (C style comments ignored)
"(^"((\^[^)])|[^^])*"^)"             ANNOTATIONSTRING
".."                                 DOTDOT
"<-"                                 LEFTARROW
"->"                                 RIGHTARROW
[][=*()<>;,.{}]                      (Characters tokenized as themselves)
"InitialNode"                        INITIALNODE
"ForkNode"                           FORKNODE
"JoinNode"                           JOINNODE
"DecisionNode"                       DECISIONNODE
"MergeNode"                          MERGENODE
"ActivityFinal"                      ACTIVITYFINAL
"FlowFinal"                          FLOWFINAL
[0-9]+                               NUMBER
(((([a-zA-Z0-9_])|(\\(.|\n)))+)|
 ((([a-zA-Z0-9_])|(\\(.|\n)))
  (([-a-zA-Z0-9_])|(\\(.|\n)))*
  (([a-zA-Z0-9_])|(\\(.|\n)))))      IDENTIFIER
\"(([^"\\])|(\\(.|\n)))*\"           IDENTIFIER
'(([^'\\])|(\\(.|\n)))*'             IDENTIFIER
```

5.1.2 Grammar

The following grammar is expressed using Yacc syntax. The start symbol is ADLF.

```
action:                 '(' nodeGuts ')'

ADLF:                   statementVector '.'

annotation:             IDENTIFIER '=' IDENTIFIER
                        | IDENTIFIER '=' NUMBER
                        | ANNOTATIONSTRING
                        | IDENTIFIER

annotationVector:       annotation
                        | annotationVector annotation

arrow:                  leftArrow | rightArrow

branch:                 arrow chain

branchVector:           branch
                        | branchVector ',' branch

chain:                  noTreeChain
                        | noTreeChain tree

controlNode:            '<' controlNodeGuts '>'

controlNodeGuts:        controlNodeIdentifier
                        | reference
                        | controlNodeIdentifier reference
                        | controlNodeIdentifier annotationVector
                        | reference annotationVector
                        | controlNodeIdentifier reference annotationVector

controlNodeIdentifier:  INITIALNODE | FORKNODE | JOINNODE | DECISIONNODE
                        | MERGENODE | ACTIVITYFINAL | FLOWFINAL

leftArrow:              LEFTARROW
                        | multiplicity LEFTARROW
                        | LEFTARROW IDENTIFIER
                        | multiplicity LEFTARROW IDENTIFIER

multiplicity:           NUMBER
                        | NUMBER DOTDOT NUMBER
                        | NUMBER DOTDOT '*'
                        | '*'

node:                   controlNode | action | object
```

```
nodeGuts:           IDENTIFIER
                    | reference
                    | IDENTIFIER reference
                    | IDENTIFIER annotationVector
                    | reference annotationVector
                    | IDENTIFIER reference annotationVector

noTreeChain:        node
                    | noTreeChain arrow node

object:             '[' nodeGuts ']'

reference:          '*' IDENTIFIER | '*' NUMBER

rightArrow:         RIGHTARROW
                    | RIGHTARROW multiplicity
                    | IDENTIFIER RIGHTARROW
                    | IDENTIFIER RIGHTARROW multiplicity

statementVector:    chain
                    | statementVector ';' chain

tree:               '{' branchVector '}'
```

5.2 JavaCC

The JavaCC grammar differs slightly from the Flex/Bison grammar defined previously; the language recognized by the former is a proper subset of the language recognized by the latter. Specifically, the JavaCC grammar has the following limitations:

- The left arrow (`<-`) is unsupported.
- Certain structural constraints are enforced by the grammar; for instance:
  - An initial node cannot be the target of any flow.
  - A path through the Activity Diagram cannot start with a control node unless it is an initial node or a control referral node, e.g., `<*join>`. (The term "referral node" is used for a node that contains a coreference variable but not an identifier, and is thus merely a reference to another node.)
  - Decision and fork nodes must be followed by at least one target branch surrounded by curly brackets, e.g., `<DECISION> {"guard" -> (Action)}`.

These limitations were deemed acceptable as they do not significantly limit the Activity Diagrams that can be recognized. An activity making use of the left arrow can be rewritten to make use of only the right arrow. Activity Diagrams that require violating the other limitations are ill-formed.

5.2.1 Tokens

The regular expressions below are written using JavaCC syntax. In contrast with Lex, JavaCC requires all character data to be quoted, so character data cannot be confused with operators.

```
" " | "\t" | "\n" | "\r" | "\f" | "\013"        (Whitespace ignored)
"//..."                                         (Java single-line comments ignored)[3]
"/*...*/"                                       (Java multi-line comments ignored)[3]
"InitialNode"                                   <INITIAL>
"ForkNode"                                      <FORK>
"JoinNode"                                      <JOIN>
"DecisionNode"                                  <DECISION>
"MergeNode"                                     <MERGE>
"ActivityFinal"                                 <ACTIVITYFINAL>
"FlowFinal"                                     <FLOWFINAL>
";"                                             <SPLIT>
"->"                                            <SEQ>
"."                                             <END>
"{"                                             <PARBEGIN>
"}"                                             <PAREND>
","                                             <PARSPLIT>
(["0"-"9"])+                                    <NUMBER>
".."                                            <DOTDOT>
"*"                                             <STAR>
((["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])+ |
 (["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])
 (["a"-"z","A"-"Z","0"-"9","_","-"]|"\\"~[])*
 (["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[]))       <IDENTIFIER>
"\"" ( ~["\"","\\"] | "\\"~[] )* "\""           <DOUBLEQUOTED_STRING>
"'" ( ~["'","\\"] | "\\"~[] )* "'"              <SINGLEQUOTED_STRING>
"(^" ( "^"~[")"] | ~["^"] )* "^)"               <ANNOTATION_STRING>
<EOF>                                           (End of file)
```

5.2.2 Grammar

The Extended Backus-Naur Form (EBNF) specification below describes ADLF, subject to the limitations described in Section 5.2. The following conventions are used:

- Tokens are enclosed in angle brackets, e.g., `<END>`.
- Non-terminals are not decorated, e.g., ActionNode.
- Optional items are enclosed in parenthesis, followed by a question mark, e.g., (Label)?.
- Items repeating zero or more times are enclosed in parentheses, followed by an asterisk, e.g., (`<IDENTIFIER>`)*.

[3]The description of these tokens requires several lines in the JavaCC `jj` file. Essentially, single-line comments terminate with a carriage return; multi-line comments terminate with a closing `*/`. Any symbol is permissible within either type of comment.

- Items repeating one or more times are enclosed in parentheses, followed by a plus sign, e.g., (FullStatement)+.
- Alternate choices are separated by a pipe, e.g, (Identifier | <NUMBER>).

The start symbol is Activity.

```
ActionName ::=              Identifier

ActionNode ::=              "(" ( ActionNodeInitial | ActionNodeReferral )

ActionNodeInitial ::=       ActionName (Label)? (AnnotationSeq)? ")"

ActionNodeReferral ::=      Label (AnnotationSeq)? ")"

Activity ::=                (FullStatement)+ <END> <EOF>

ActivityFinalNode ::=       <ACTIVITYFINAL> (Label)? (AnnotationSeq)?

Annotation ::=              ( Identifier ( "=" ( Identifier | <NUMBER> ) )?
                            | <ANNOTATION_STRING> )

AnnotationSeq ::=           (Annotation)+

ControlNode ::=             ( ControlNodeInitial | ControlNodeReferral )

ControlNodeInitial ::=      ( JoinNode | MergeNode | ActivityFinalNode | FlowFinalNode )
                            ">"

ControlNodeReferral ::=     Label (AnnotationSeq)? ">"

DecisionNode ::=            <DECISION> (Label)? (AnnotationSeq)?

DecisionStatement ::=       DecisionNode ">"
                            <PARBEGIN>
                            StatementSeq ( <PARSPLIT> StatementSeq )*
                            <PAREND>

FlowFinalNode ::=           <FLOWFINAL> (Label)? (AnnotationSeq)?

ForkNode ::=                <FORK> (Label)? (AnnotationSeq)?

ForkStatement ::=           ForkNode ">"
                            <PARBEGIN>
                            StatementSeq ( <PARSPLIT> StatementSeq )*
                            <PAREND>

FullStatement ::=           ( "<" ( InitialNode ">" (MultipleOutgoing)?
                                    | ControlNodeReferral )
                            | ObjectNode
```

```
                              | ActionNode )
                              ( StatementSeq (<SPLIT>)? )?

Guard ::=                     Identifier

Identifier ::=                ( <IDENTIFIER> | <SINGLEQUOTED_STRING>
                              | <DOUBLEQUOTED_STRING> )

InitialNode ::=               <INITIAL> (Label)? (AnnotationSeq)?

JoinNode ::=                  <JOIN> (Label)? (AnnotationSeq)?

Label ::=                     "*" ( Identifier | <NUMBER> )

Lower ::=                     <NUMBER>

MergeNode ::=                 <MERGE> (Label)? (AnnotationSeq)?

MultipleOutgoing :: =         <PARBEGIN>
                              StatementSeq ( <PARSPLIT> StatementSeq )*
                              <PAREND>

Multiplicity ::=              ( Lower <DOTDOT> )? UpperOrOnly

ObjectNode ::=                "[" Identifier (Label)? (AnnotationSeq)? "]"

Statement ::=                 ( "<" ( ForkStatement
                                    | DecisionStatement
                                    | ControlNode )
                              | ActionNode (MultipleOutgoing)?
                              | ObjectNode (MultipleOutgoing)? )

StatementSeq ::=              ( (Guard)? <SEQ> (Multiplicity)? Statement )+

UpperOrOnly ::=               ( <STAR> | <NUMBER> )
```

6 Conclusion

We have described a modification of the Petri Net Linear Form (PNLF) notation to support the rendering of UML Activity Diagrams as human-readable text. The Activity Diagram Linear Form (ADLF) can be processed by screen readers and is more usable than XML-based syntaxes. As a textual notation, it is suitable for embedding in text-only documents and it accommodates standard search and copy/paste operations. Finally, ADLF, along with its attendant parsers, is useful during the prototyping of new applications. Instead of interfacing with graphical representations or full-featured interchange formats, one can use the textual format for input/output, allowing the majority of effort to be spent on the core application.

7 Acknowledgment

The authors thank Paul Black and Conrad Bock for their helpful reviews and suggestions.

References

[1] OMG Unified Modeling Language Superstructure Specification, version 2.1.1. Document formal/2007-02-05, Object Management Group, February 2007. http://www.omg.org/cgi-bin/doc?formal/2007-02-05.

[2] World Wide Web Consortium. Extensible Markup Language (XML), 2007. http://www.w3.org/XML/.

[3] Philippe A. Martin. The Petri Net Linear Form, 2007. http://www.phmartin.info/wf/pnlf/.

[4] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[5] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, 1984.

[6] Michael Weber. Petri Net Markup Language, 2006. http://www2.informatik.hu-berlin.de/top/pnml/about.html.

[7] OMG Meta Object Facility (MOF) 2.0 / XML Metadata Interchange (XMI) Mapping Specification, version 2.1. Document formal/2005-09-01, Object Management Group, September 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[8] OMG Unified Modeling Language Diagram Interchange Specification, version 1.0. Document formal/2006-04-04, Object Management Group, April 2006. http://www.omg.org/cgi-bin/doc?formal/2006-04-04.

[9] OMG Human-Usable Textual Notation (HUTN) Specification, version 1.0. Document formal/2004-08-01, Object Management Group, August 2004. http://www.omg.org/cgi-bin/doc?formal/2004-08-01.

[10] Harald Störrle and Jan Hendrik Hausmann. Towards a formal semantics of UML 2.0 activities. In *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 64 of *LNI*, pages 117–128. GI, 2005. http://wwwcs.uni-paderborn.de/cs/ag-engels/Papers/2005/SE2005-Stoerrle-Hausmann-ActivityDiagrams.pdf.

[11] Software and system engineering—High-level Petri nets—Part 1: Concepts, definitions and graphical notation. ISO/IEC 15909-1, International Organization for Standardization, December 2004. http://www.iso.org/.

[12] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, March 2004. See also http://www.workflowcourse.com/.

[13] Flex: the Fast Lexical Analyzer, 2006. http://flex.sourceforge.net/.

[14] Bison: GNU parser generator, 2006. http://www.gnu.org/software/bison/.

[15] NIST. Human-readable text form for UML activity diagrams (download page for parser), 2007. http://purl.org/net/dflater/org/nist/adlf.

[16] JavaCC, the Java Compiler Compiler, 2006. https://javacc.dev.java.net/.

[17] M. E. Lesk. Lex—a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, October 1975. See also http://dinosaur.compilertools.net/lex/.

[18] Stephen C. Johnson. Yacc—yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, July 1975. See also http://dinosaur.compilertools.net/yacc/.

[19] Theodore S. Norvell. The JavaCC FAQ, 2007. http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm.

[20] Information technology—Syntactic metalanguage—Extended BNF. ISO/IEC 14977, International Organization for Standardization, 1996. http://standards.iso.org/ittf/PubliclyAvaliableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[21] Peter Naur (editor), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.