Easier Rules and Constraints for Programming*

Yanhong A. Liu Scott D. Stoller

We discuss how rules and constraints might be made easier for more conventional programming. We use a language that extends DistAlgo, which extends Python, and we use the RBAC programming challenge plus distributed RBAC as examples.

Python. Python is a high-level programming language with an easy-to-read syntax. It supports conventional imperative programming and object-oriented programming. It also supports database-style programming with sets and queries (comprehension and generator expressions) and functional programming with recursive functions and even a syntax for lambda. However, it does not support rules and constraints.

DistAlgo. DistAlgo [LSL17, LL17] is a language that extends Python to support distributed programming with processes and message passing. It also extends Python to support more powerful queries with constraints and tuple patterns, including logic quantifications with witnesses. These query constructs were created to better express high-level synchronization conditions over messages and processes but also high-level queries in general, while integrating seamlessly with imperative programming.

For example, consider a set `UR` of user-role pairs and a particular user `user`. The set of roles that `user` has can be expressed using a set comprehension with a tuple pattern as follows.

```
setof(r, (_user,r) in UR)
```

The membership condition is exactly a constraint, and in general any number of constraints can be used. `(_user,r)` is a tuple pattern, where the underscore indicates a variable on the left side of a membership clause whose value is bound before the query. Note that we have also implemented a more ideal syntax for the same query, shown below, but here we use Python accepted syntax, shown above, so that the Python parser can be used.

```
{r: (=user,r) in UR}
```

Similarly one may compute aggregation (e.g., `countof` and `minof`) over sets, and universal and existential quantifications (`each(x in s, has= p(x)` or `some(x in s, has= p(x))`.

Extension with constraint optimization. With the more powerful set queries as above, it is easy to write an additional constraint to filter out only those that minimize some objective function, e.g., the constraint `f(r) == minof(f(x), (_user,x) in UR)` can be inserted in the set comprehension shown above. It is even easier to simply add the constraint as follows,

```
minimize= exp
```

where `exp` expresses the objective function, e.g., `minimize= f(r)` can be inserted in the set comprehension shown above. This is just as in mathematical programming tools.

*Authors' contact: Computer Science Department, Stony Brook University, Stony Brook, New York. Email: {liu,stoller}@cs.stonybrook.edu. This work was supported in part by NSF under grants CCF-1414078, IIS-1447549, and CNS-1421893, and ONR under grant N000141512208.

Extension with rules. Just as declaring a named function or method, one should be able to easily declare a named set of rules, e.g.,

```
def rules (name='Trans_rules'):
  if edge(x,y): path(x,y)
  if edge(x,z) and path(z,y): path(x,y)
```

and call an inference function to infer values using the rule set, e.g., the following returns the set of pairs for which predicate `path` holds using rule set `Trans_rules` given a set of edges `RH`.

```
infer(path, edge=RH, rules=Trans_rules)
```

One can also use `path(1,v)` in place of `path` to return the set of values of `v` for which `path(1,v)` holds. Note that predicates `edge` and `path` are simply set-valued variables, without needing high-order logic.

Extension with backtracking under choices. While planning problems can be expressed as constraint solving and optimization, it is more direct if actions in the program can be expressed with choices, with actions sequenced, with backtracking in an allowed scope, until sequences of actions satisfying a condition are found and returned or all choices are enumerated. This is easily expressed with a pair of `assume` and `achieve` statements that surround statements with choices.

In particular, given a set of actions `acts` that are allowed operations, e.g., method definitions, let `instances(acts)` generate all instances of calls to those methods, and let `do(a)` execute method call `a`. The following code finds any sequence that satisfies `condition`, where `some` makes a choice.

```
assume(True)
seq = []
while not condition:
  if some(a in instances(acts)):
    do(a)
    seq.append(a)
achieve(anyof(seq))
```

Also, a cost function can be computed along the sequence, and solutions that minimize the cost may be returned.

Implementation. The extensions are being implemented by extending DistAlgo. The implementation is currently incomplete. The main challenge will be efficient implementation to provide competitive performance compared with lower-level or more complex manually programmed solutions.

RBAC programming challenge solution. The Appendix shows how to express all components and functions of the RBAC programming challenge, plus a component for distributed RBAC, in the extended language. It is aimed to express everything in the clearest and most direct way possible.

`process` in class header is needed only for distributed execution for the distributed RBAC component; for others, it is included only to allow use of more powerful set queries with constraints and tuple patterns. `pre` for preconditions could be implemented simply as `assert` in Python but we plan to support it directly in the extensions to DistAlgo.

These components can run with DistAlgo: `CoreRBAC`, `HierarchicalRBAC_set`, `HierarchicalRBAC`, `CoreRBACwithSSD`, `HierarchicalRBACwithSSD`, and `DistRBAC`. Their less powerful variants in Python without `DistRBAC` were run and optimized to run efficiently previously [LWG+06, GLSR12].

These do not currently run: `HierarchicalRBAC_rules` and `AdminRBAC`.

References

[GLSR12] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, and Tom Rothamel. Composing transformations for instrumentation and optimization. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, pages 53–62, 2012.

[LL17] Bo Lin and Yanhong A. Liu. DistAlgo: A language for distributed algorithms. `http://github.com/DistAlgo`, 2017. Beta release September 27, 2014, latest release November 23, 2017.

[LSL17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, 39(3):12:1–12:41, May 2017.

[LWG+06] Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*, pages 112–120, 2006.

Appendix: RBAC challenge in a language that extends Python

```
"""
We consider Role-Based Access Control (RBAC) with 6 components:

  Core RBAC,
  Hierarchical RBAC,
  Core RBAC with Static Separation of Duty constraint (also called Constrained RBAC),
  Hierarchical RBAC with Static Separation of Duty constraint,
  Administrative RBAC, and
  Distributed RBAC
"""

class CoreRBAC(process):
  """
  Core RBAC keeps several sets including the following:

    USERS: set of users
    ROLES: set of roles
    PERMS: set of permissions
    UR: set of user-role pairs
    PR: set of permission-role pairs

  with constraints:

    UR subset USERS * ROLES
    PR subset PERMS * ROLES

  update functions for each set, subject to the constraints above:

    AddUser, DeleteUser, AddRole, DeleteRole, AddPerm, DeletePerm
    AddUR, DeleteUR, AddPR, DeletePR
    each Add has pre-conditions:
      the element is not yet in the set and the constraints will not be violated
    each Delete has the pre-condition that the element is in the set,
      and maintains the constraints

```

```
query functions including the following:

  AssignedUsers(role): the set of users assigned to role in UR
  AssignedRoles(user): the set of roles assigned to user in UR
  UserPermissions(user):
    the set of permissions assigned to the roles assigned to user
  CheckAccess(user, perm):
    whether some role is assigned to user and is granted perm
"""

def setup():
  self.USERS = set()
  self.ROLES = set()
  self.PERMS = set()
  self.UR = set()             # UR subset USERS * ROLES
  self.PR = set()             # PR subset PERMS * ROLES

def AddUser(user):            # pre: user not in USERS
  USERS.add(user)

def DeleteUser(user):         # pre: user in USERS
  UR -= setof((user,r), r in ROLES) # maintain UR
  USERS.remove(user)

def AddRole(role):            # pre: role not in ROLES
  ROLES.add(role)

def DeleteRole(role):         # pre: role in ROLES
  UR -= setof((u,role), u in USERS) # maintain UR
  PR -= setof((p,role), p in PERMS) # maintain PR
  ROLES.remove(role)

def AddPerm(perm):            # pre: perm not in PERMS
  PERMS.add(perm)

def DeletePerm(perm):         # pre: perm in PERMS
  PR -= setof((perm,r), r in ROLES) # maintain PR
  PERMS.remove(perm)

def AddUR(user, role):
  # pre: user in USERS, role in ROLES, (user,role) not in UR
  UR.add((user,role))

def DeleteUR(user, role):     # pre: (user,role) in UR
  UR.remove((user,role))

def AddPR(perm, role):
  # pre: perm in PERMS, role in ROLES, (perm,role) not in PR
  PR.add((perm,role))

def DeletePR(perm, role):     # pre: (perm,role) in PR
  PR.remove((perm,role))

def AssignedUsers(role):      # pre: role in ROLES
  return setof(u, (u,_role) in UR)

def AssignedRoles(user):      # pre: user in USERS
  return setof(r, (_user,r) in UR)

def UserPermissions(user):    # pre: user in USERS
  return setof(p, (_user,r) in UR, (p,r) in PR)

def CheckAccess(user, perm): # pre: user in USERS, perm in PPRMS
  return some(r in ROLES, has= (user,r) in UR and (perm,r) in PR)
```

```
class HierarchicalRBAC_set(CoreRBAC,process):   # using while for Trans

  def Trans(E):
    T = E
    while some((x,y) in T, (y,z) in E, has= (x,z) not in T):
      T.add((x,z))
    return T | setof((r,r), r in ROLES)

class HierarchicalRBAC_rules(CoreRBAC,process): # using rules for Trans

  def rules(name= 'Trans_rules'):
    if edge(x,y): path(x,y)
    if edge(x,z) and path(z,y): path(x,y)

  def Trans(E):
    return infer(path, edge=E, rules=Trans_rules) | setof((r,r), r in ROLES)

class HierarchicalRBAC(HierarchicalRBAC_set,process):
  """
  Hierarchical RBAC keeps also a role hierarchy:

    RH: set of pairs of roles, called ascendant and descendant roles,
    where an ascendant role inherits permissions from a descendant role

  with constraints:

    RH subset ROLES * ROLES, and RH is acyclic

  update functions for RH, subject to the constraints above:

    AddInheritance(asc, desc)
    DeleteInheritance(asc, desc)
    with the same kinds of pre-conditions as updates in CoreRBAC

  query functions including the following:

    Trans:
      the transitive closure of role hierarchy union reflexive role pairs
    AuthorizedUsers(role):
      the set of users of role or ascendant roles of role
    AuthorizedRoles(user):
      the set of roles of user or descendant roles of the roles
  """

  def setup():
    self.RH = set()            # RH subset ROLES * ROLES, where asc inh desc

  def AddInheritance(a, d):
    # pre: a in ROLES,d in ROLES, (a,d) notin RH, a!=d, (d,a) notin Trans(RH)
    RH.add((a,d))

  def DeleteInheritance(a, d): # pre: (a,d) in RH
    RH.remove((a,d))

  def AuthorizedUsers(role):
    return setof(u, (u,asc) in UR, (asc,_role) in Trans(RH))

  def AuthorizedRoles(user):
    return setof(r, (_user,asc) in UR, (asc,r) in Trans(RH))


class CoreRBACwithSSD(CoreRBAC,process):
  """
  Core RBAC with SSD keeps also a set of SSD items, where each item has:
```

```

    a name,
    a set of roles, and
    a cardinality

  with constraints:

    all roles in all SSD items  subset ROLES
    for each SSD item, its cardinality is > 0 and < the number of its roles
    for each user, for each SSD item,
      the number of assigned roles (AssignedRoles) of the user
      that are in the item's set of roles is at most the item's cardinality

  update functions, subject to the constraints above:

    CreateSsdSet(name, roles, c): add SSD item having name, roles, c
    DeleteSsdSet(name): delete SSD item having name
    AddSsdRoleMember(name, role): add role to roles of SSD item having name
    DeleteSsdRoleMember(name, role): del role fr roles of SSD item having name
    SetSsdSetCardinality(name, c): set c to be card. of SSD item having name
    with the same kinds of pre-conditions as updates in CoreRBAC, except that
    all updates have also pre-conditions that no constraints will be violated

  query functions including the following:

    SsdRoleSets(): the set of names of SSD items
    SsdRoleSetRoles(name): the set of roles in SSD item having name
    SsdRoleSetCardinality(name): the cardinality of SSD item having name
  """

  def setup():
    self.SsdNAMES = set()  # set of names of constraints
    self.SsdNR = set()     # set of pairs of name and role
                           # SsdNR subset SsdNAMES * ROLES
    self.SsdNC = set()     # set of pairs of name and cardinality
                           # SsdNC: SsdNAMES -> int

  # constraint named SSD, as post condition for all updates
  def constraint(name= 'SSD'):
    return each(u in USERS, (name,c) in SsdNC, has=
                countof(r, r in AssignedRoles(u), (_name,r) in SsdNR) <= c)

  def CreateSsdSet(name, roles, c):
    # pre: name not in SsdNAMES, roles subset ROLES, 1 <= c < count(roles)
    SsdNAMES.add(name)
    SsdNR |= setof((name,r), r in roles)
    SsdNC.add((name,c))

  def DeleteSsdSet(name):  # pre: name in SsdNAMES  #don't need post SSD
    SsdNR -= setof((name,r), r in SsdRoleSetRoles(name))
    SsdNC.remove((name,SsdRoleSetCardinality(name)))
    SsdNAMES.remove(name)         # delete ssd name last

  def AddSsdRoleMember(name, role):
    # pre: name in SsdNAMES, role in ROLES
    # pre: role not in SsdRoleSetRoles(name)
    SsdNR.add((name,role))

  def DeleteSsdRoleMember(name, role):
    # pre: name in SsdNAMES, role in SsdRoleSetRoles(name)
    # pre: c < SsdRoleSetCardinality(name)-1
    SsdNR.remove((name,role))

  def SetSsdSetCardinality(name, c):
    # pre: name in SsdNAMES, SsdRoleSetCardinality(name) != c
```

```
    SsdNC.remove((name,SsdRoleSetCardinality(name)))
    SsdNC.add((name,c))

  def SsdRoleSets():
    return SsdNAMES

  def SsdRoleSetRoles(name):        # pre: name in SsdNAMES
    return setof(r, (_name,r) in SsdNR)

  def SsdRoleSetCardinality(name):  # pre: name in SsdNAMES
    return anyof(c, (_name,c) in SsdNC)


class HierarchicalRBACwithSSD(HierarchicalRBAC,CoreRBACwithSSD,process):
  """
  Hierarchical RBAC with SSD combines all from
  Hierarchical RBAC and Core RBAC with SSD, except that
  the SSD constraint uses AuthorizedRoles in place of AssignedRoles.
  """

  def constraint (name= 'SSD'):
    return each(u in USERS, (name,c) in SsdNC, has=
                countof(r, r in AuthorizedRoles(u), (_name,r) in SsdNR) <=c)


class AdminRBAC(HierarchicalRBACwithSSD):
  """
  Administrative RBAC for HierarchicalRBACwithSSD
  has optimization and planning functions:

    MineMinRoles:
      find a smallest set of roles with UR' and PR' assignments
      such that UR' * PR' = UR * PR

    MineMinRoleAssignments:
      find a smallest set of UR' and PR' assignments
      such that UR' * PR' = UR * PR = UP

    GetRolesPlan(user, roles, acts):
      find a sequence of actions, i.e., updates, in acts that
      allows user to get roles

    GetRolesShortestPlan(user, roles, acts):
      find a shortest sequence of actions, i.e., updates, in acts that
      allows user to get roles

  Any subset of updates can be used as acts.
  All constraints must hold after each action.

  The first two can have a version that includes finding RH'.

  Administrative RBAC could also be for
    CoreRBAC, HierarchicalRBAC, or CoreRBACwithSSD.
  """

  def MineMinRoles():
    return anyof((R, UR2, PR2), R in subset(ran(UR)&ran(PR)),
                 UR2 in subset(dom(UR)*R), PR2 in subset(dom(PR)*R),
                 UR2 * PR2 == UR * PR, minimize= count(R))

  def MineMinRoleAssignments():
    return anyof((R, UR2, PR2), R in subset(ran(UR)&ran(PR)),
                 UR2 in subset(dom(UR)*R), PR2 in subset(dom(PR)*R),
                 UR2 * PR2 == UR * PR, minimize= count(UR2+PR2))
```

```
  def GetRolesPlan(user, roles, acts):
    assume(True)
    seq = []
    while not each(r in roles, has= (_user,r) in UR):
      if some(a in instances(acts)):
        do(a)
        seq.append(a)
    achieve(anyof(seq))

  def GetRolesShortestPlan(user, roles, acts):
    assume(True)
    seq = []
    cost = 0
    while not each(r in roles, has= (_user,r) in UR):
      if some(a in instances(acts)):
        do(a)
        seq.append(a)
        cost += 1
    achieve(anyof((seq, cost), minimize= cost))


class DistRBAC(HierarchicalRBACwithSSD,process):
  """
  A Distributed RBAC process keeps also the following sets:

    OTHERS: set of other RBAC processes
    GuestR: set of pairs of a rbac-role pair and a guest role

  with constraints:

    domain(domain(GuestR)) subset OTHERS
    range(GuestR) subset ROLES

  update functions for each set subject to the constraints above:

    AddGuestRole, DeleteGuestRole
    AssignGuestRole:
      assign to user of role in rbac the corresponding guest roles
    DeassignGuestRole
      deassign from user of role in rbac the corresponding guest roles

  query functions:

    GuestRoles (rbac,role): the set of guest roles for role of rbac
    OthersRoles(guest): the set of rbac-role pairs for role guest

  Distributed RBAC can also be for only
    CoreRBAC, HierarchicalRBAC, or CoreRBACwithSSD,
    or Administrative RBAC for any of these.
  """

  def setup(OTHERS):
    self.GuestR = set()

  def AddGuestRole(rbac, role, guest):     # pre: rbac in OTHERS,guest in ROLES
    GuestR.add(((rbac,role),guest))

  def DeleteGuestRole(rbac, role, guest):  # pre: ((rbac,role),guest) in GuestR
    GuestR.remove(((rbac,role),guest))

  def GuestRoles(rbac, role):
    return setof(guest, ((_rbac,_role),guest) in GuestR)

  def OthersRoles(guest):
    return setof((rbac,role), ((rbac,role),_guest) in GuestR)
```

```
def AddGuestUR(user, rbac, role):        # pre: rbac in OTHERS
  send(('credential', user, role), to= rbac)
  if await(received(('accept', user, role), from_= rbac)):
    for r in GuestRoles(rbac, role):
      AddUR(user, r)

def DeleteGuestUR(user, rbac, role):
  for r in GuestRoles(rbac, role):
    DeleteUR(user, r)

def receive(msg=('credential', user, role), from_= rbac):
  if (user,role) in UR:
    send(('accept', user, role), to= rbac)
  else:
    send(('reject', user, role), to= rbac)

def receive(msg=('AddGuestUR', user, rbac, role)):
  AddGuestUR(user, rbac, role)

def receive(msg=('DeleteGuestUR', user, rbac, role)):
  DeleteGuestUR(user, rbac, role)
```