Imperial College London

Access-Control Policies via Belnap logic: expressive composition and simple analysis

Michael Huth

imperial.ac.uk/quads/

Abschlussveranstaltung des Graduiertenkollegs
Mathematische Logik und Anwendungen
11-12 September 2008
Freiburg, Germany

Joint work with:

Glenn Bruns
Bell Labs
Naperville, Illinois

Imperial College London

Outline

Motivation

Example access-control policy

Consider partial campus policy, due to (Halpern & Weissman):

$$\texttt{p} = \texttt{p1 merge p2 merge p3}$$

Its sub-policies `pi` each model an aspect of a campus policy:

- `p1` says "faculty has permission to assign grades"
- `p2` says "students must not assign grades"
- `p3` says "non-faculty has permission to enroll in courses"

What does this policy mean? Can we enforce or analyze it?
And if so, how?

Context of work reported here

Access-control policies:

- become more important in many domains, not just security
- may not be explicitly documented
- may be too general, too specific, too ambiguous
- need to be modifiable, comparable to other policies
- need to support different degrees of granularity
- etc.

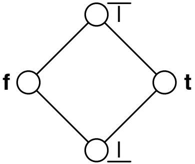
$\Longrightarrow$ Simple but expressive target language for policies, supporting their elaboration, exploration, and analysis should have value.

Some requirements for policy language

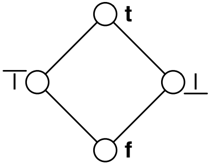
- policies may specify one aspect, may be silent on others
- intuitive yet expressive composition of such policies
- efficient policy analysis (think SAT solvers or BDDs)
- gap and conflict analysis supported
- ability to analyze policy hierarchies and change impact
- ability to specialize or partially evaluate policies
- provide clean interface for specific application domains (e.g. role hierarchies)

Belnap logic

Four: a set with two lattice structures



information ordering



truth ordering

- **f** (Deny), **t** (Grant), $\perp$ (Undefined) or $\top$ (Conflict)
- this four-element set forms a lattice in both
  - the information ordering $\leq_i$ (left)
  - and the truth ordering $\leq_t$ (right)

(Belnap 1976)

Imperial College London

Operations on Belnap Space **Four**

- Negation: $\neg\mathbf{f} = \mathbf{t}$; $\neg\mathbf{t} = \mathbf{f}$; $\top$ and $\bot$ fixed
- Logical meet ($\wedge$), join ($\vee$) for truth ordering $\leq_t$
- Information meet ($\otimes$), join ($\oplus$) for information ordering $\leq_i$
- contradictions are non-catastrophic: $x \wedge \neg x \leq_i y$ is not valid

Recommended reading

- **Four** is the simplest non-trivial bilattice. For bilattices it plays a role similar to that of the Sierpinski space $\{\mathbf{f} < \mathbf{t}\}$ for complete lattices.

Nuel D. Belnap. How a computer should think. In G. Ryle, editor, Contemporary Aspects of Philosophy, pages 30–56. Oriel Press, Stocksfield, 1976.

Melvin Fitting. Bilattices are nice things. In T. Bolander, V. Hendricks, and S. A. Pedersen, editors, Self-reference, pages 53–77. Center for the Study of Language and Information, 2006.

Core policy language

Belnap logic inside

- Policy writers or readers can't be expected to know or understand Belnap logic.
- But that logic is a fine foundation for a core policy language and its analysis.
- Thus we design a core language that has Belnap logic 'inside,' without policy readers or writers realizing this.
- Key will be the ability to
  - demote a policy expression into a set of requests, those requests the policy grants (respecively, denies)
  - promote a set of requests into a policy.
- All of this will be declarative, e.g. sets of requests are propositional constraints.

PolCore: a core policy language

`dec ::=`	(Decision)
`grant`	
`deny`	

`reqs ::=`	(Requests)
`ff`	Falsity
`tt`	Truth
`reqsAtom`	Atom
`!reqs`	Neg.
`reqs & reqs`	Conj.
`reqs | reqs`	Disj.
`pol.dec`	Demote

`pol ::=`	(Policy)
`dec`	Constant
`pol when reqs`	Promote
`pol merge pol`	Merge

Reconsider example policy

Sub-policy

- `p1` says “faculty has permission to assign grades”
- `p2` says “students must not assign grades”
- `p3` says “non-faculty has permission to enroll in courses”

Formalized as `p1 merge p2 merge p3` in *PolCore* where:

- `p1` = `grant when faculty & grades & assign`
- `p2` = `deny when student & grades & assign`
- `p3` = `grant when !faculty & courses & enroll`

What semantics for *PolCore*?

- What does `p1 merge p2 merge p3` mean?
- More simply, what does `grant when faculty & grades & assign` mean?
- Even more simply, what do `faculty`, `grades`, and `assign` mean?
- University administrator's response: It's obvious, give me a person and I will tell you whether they are faculty or not.
- System administrator's response: I need an implementation such as

```
boolean faculty(r: accessRequest){
  r.role == Faculty;
}
```

Request models

- We capture both, the university administrator's and system administrator's response, abstractly.
- A request model $\mathcal{M}$: non-empty set $R^{\mathcal{M}}$ of requests and, for each atom `reqsAtom`, a subset $\mathtt{reqsAtom}^{\mathcal{M}}$ of $R^{\mathcal{M}}$.
- Terms `reqs` that don't invoke clause `pol.dec` are evaluated over $\mathcal{M}$ as a subset $[\![\mathtt{reqs}]\!]_{\mathcal{M}}$ of $R^{\mathcal{M}}$:

$$
\begin{array}{ll}
[\![\mathtt{ff}]\!]_{\mathcal{M}} = \{\} & [\![\mathtt{tt}]\!]_{\mathcal{M}} = R^{\mathcal{M}} \\
[\![\mathtt{reqsAtom}]\!]_{\mathcal{M}} = \mathtt{reqsAtom}^{\mathcal{M}} & [\![\mathtt{!reqs}]\!]_{\mathcal{M}} = R^{\mathcal{M}} - [\![\mathtt{reqs}]\!]_{\mathcal{M}} \\
[\![\mathtt{r1}\ \&\ \mathtt{r2}]\!]_{\mathcal{M}} = [\![\mathtt{r1}]\!]_{\mathcal{M}} \cap [\![\mathtt{r2}]\!]_{\mathcal{M}} & [\![\mathtt{r1} \mid \mathtt{r2}]\!]_{\mathcal{M}} = [\![\mathtt{r1}]\!]_{\mathcal{M}} \cup [\![\mathtt{r2}]\!]_{\mathcal{M}}
\end{array}
$$

The meaning of `pol.dec`

- Expansion Expd(pol.dec) is element of reqs without sub-expressions pol′.dec′. We set

$$[\![\texttt{pol.dec}]\!]_{\mathcal{M}} = [\![\mathrm{Expd}(\texttt{pol.dec})]\!]_{\mathcal{M}}$$

- Expansion first defined over reqs without sub-expressions pol′.dec′, acting as identity:

$$\begin{array}{rcl}
\mathrm{Expd}(\texttt{ff}) & = & \texttt{ff} \\
\mathrm{Expd}(\texttt{tt}) & = & \texttt{tt} \\
\mathrm{Expd}(\texttt{reqsAtom}) & = & \texttt{reqsAtom} \\
\mathrm{Expd}(!\texttt{reqs}) & = & !\mathrm{Expd}(\texttt{reqs}) \\
\mathrm{Expd}(\texttt{r1}\,\&\,\texttt{r2}) & = & \mathrm{Expd}(\texttt{r1})\,\&\,\mathrm{Expd}(\texttt{r2})
\end{array}$$

Expansion function for pol.dec

- We define Expd(pol.dec) by structural induction on pol:

$$
\begin{aligned}
\mathrm{Expd}(\texttt{dec.dec}) &= \texttt{tt} \\
\mathrm{Expd}(\texttt{dec.!dec}) &= \texttt{ff} \\
\mathrm{Expd}((\texttt{p when reqs}).\texttt{dec}) &= \mathrm{Expd}(\texttt{reqs})\,\&\,\mathrm{Expd}(\texttt{p.dec}) \\
\mathrm{Expd}((\texttt{p1 merge p2}).\texttt{dec}) &= \mathrm{Expd}(\texttt{p1.dec}) \mid \mathrm{Expd}(\texttt{p2.dec})
\end{aligned}
$$

where !grant is deny, and !deny is grant.

Thus $[\![\texttt{reqs}]\!]_{\mathcal{M}}$ is defined for all sets of requests reqs.

Imperial College London

The meaning of policies

- Given a request model $\mathcal{M}$ and policy expression $\mathtt{p}$, we define the meaning of $\mathtt{p}$ with respect to $\mathcal{M}$ as a total function from $R^{\mathcal{M}}$ into the Belnap space **Four**, i.e. $[\![\mathtt{p}]\!]_{\mathcal{M}} \colon R^{\mathcal{M}} \to$ **Four**:

$$[\![\mathtt{p}]\!]_{\mathcal{M}}(r) \quad = \quad \begin{cases} \bot & \text{if } r \notin [\![\mathtt{p.grant}]\!]_{\mathcal{M}} \cup [\![\mathtt{p.deny}]\!]_{\mathcal{M}} \\ \top & \text{if } r \in [\![\mathtt{p.grant}]\!]_{\mathcal{M}} \cap [\![\mathtt{p.deny}]\!]_{\mathcal{M}} \\ \mathbf{t} & \text{if } r \in [\![\mathtt{p.grant}]\!]_{\mathcal{M}} \setminus [\![\mathtt{p.deny}]\!]_{\mathcal{M}} \\ \mathbf{f} & \text{if } r \in [\![\mathtt{p.deny}]\!]_{\mathcal{M}} \setminus [\![\mathtt{p.grant}]\!]_{\mathcal{M}} \end{cases}$$

So $\mathtt{p.grant}$ means $\geq_i$ **t** and $\mathtt{p.deny}$ means $\geq_i$ **f**.

Partial evaluation of campus policy

- Expd(`p.grant`) = (`faculty` & `grades` & `assign`) | (!`faculty` & `courses` & `enroll`)
- A query "Can students enroll in courses?" turns into [`student` & `courses` & `enroll`] & Expd(`p.grant`)
- making domain assumption that courses are never grades, assignments are never enrollments, this simplifies to `student` & !`faculty` & `courses` & `enroll`, i.e. to !`faculty` if we leave the query implicit
- so a student can enroll in courses iff she is not faculty
- similarly, using Expd(`p.deny`) we conclude that no student is being denied to enroll in courses: exposes policy gap for students who are also faculty members

Imperial College London

Extending the core language

Adding parameters and methods to *PolCore*

- A standard type system (not shown today) extends *PolCore* with parameters and methods of types `reqs` and `pol`.
- We use methods to define policy composition. We begin with negation:

```
pol negation(P:pol) {
   (grant when P.deny) merge (deny when P.grant)
}
```

- Merge normal form of any policy `p` is

```
(grant when p.grant) merge
(deny when p.deny)
```

  Exercise: What is its meaning?

Some policy combinators

- Predicates for gaps (`undef`) and conflicts (`incon`) written as methods of return type `reqs`:

```
reqs undef(P:pol){
  !P.grant & !P.deny }
reqs incon(P:pol){
  P.grant & P.deny }
```

  Subsequently we use `P.undef` for `undef(P)` etc.
- Priority chaining (use infix $>$ for that subsequently):

```
pol priorityChaining(P1:pol, P2:pol) {
  (grant when P1.grant | (P1.undef & P2.grant))
              merge
  (deny when P1.deny | (P1.undef & P2.deny))
}
```

Some more policy combinators

- Defensive conjunction:

```
pol defensiveConjunction(P1:pol, P2:pol){
  (grant when P1.grant & P2.grant) merge
  (deny  when P1.deny | P2.deny)
}
```

- Deny exception treats denials of policy `P1` as an exception and handles it with policy `P2`:

```
pol denyException(P1:pol, P2:pol){
  (P2 when P1.deny) > P1
}
```

Exercise: Rewrite this for handling only conflictfree denials.

Exercises

- Majority Vote: takes three policies as input, and makes the decisions made by the majority of input policies:

```
pol MajorityOfThree(P1:pol, P2:pol, P3:pol){
   (grant when WHAT GOES IN HERE?) merge
   (deny when WHAT GOES IN HERE?)
}
```

- What is the return type of method `threat`, and what is the possible intent of method `filter2`?

```
pol filter2(P:pol){
   (deny when threat(P)) > P
}
```

Semantis of method invocations

- We expand method invocations into the core language *PolCore* and define the semantics in terms of that of the expanded expression of *PolCore*.
- We only have to extend the expansion function. We do this for method declarations and variables (Pvar and Rvar) as well, as this will enable static method analysis:

$$
\begin{aligned}
\mathrm{Expd}(\mathrm{Pvar}.\mathtt{dec}) &= \mathrm{Pvar}.\mathtt{dec} \\
\mathrm{Expd}(\mathrm{Rvar}) &= \mathrm{Rvar} \\
\mathrm{Expd}(\mathit{namePol}(\vec{E}).\mathtt{dec}) &= \mathrm{Expd}(\mathit{bodyPol}[\vec{V}/\vec{E}].\mathtt{dec}) \\
\mathrm{Expd}(\mathit{nameReqs}(\vec{E})) &= \mathrm{Expd}(\mathit{bodyReqs}[\vec{V}/\vec{E}])
\end{aligned}
$$

Policy analysis

Technical device

- For all expressions `reqs`, the following are equivalent:
  1. `reqs`, interpreted as a formula Expd(`reqs`) of propositional logic, is valid.
  2. `reqs`, interpreted as a propositional expression Expd(`reqs`) over unary predicates `reqsAtom`, holds for all requests in all request models.
- Proof relies on the fact that we can synthesize a request model that captures all 2^n propositional logic models for Expd(`reqs`).
- This result and the subsequent analyses hold for the extended policy language with methods.

Imperial College London

Some policy analyses

- **Gap analysis.** Policy `p` is free of gaps iff Expd(!`p.undef`) is valid as formula of propositional logic.
- **Conflict analysis.** Policy `p` is free of inconsistencies iff Expd(!`p.incon`) is valid as formula of propositional logic.
- **Equality.** Policies `p1` and `p2` are equivalent iff Expd((`p1.deny` $\leftrightarrow$ `p2.deny`)&(`p1.grant` $\leftrightarrow$ `p2.grant`)) is valid as formula of propositional logic.

Imperial College London

Some more policy analyses

- **Policy refinement.** Policy `p2` refines policy `p1` iff Expd(($\texttt{p1.grant} \to \texttt{p2.grant}$) & ($\texttt{p1.deny} \to \texttt{p2.deny}$)) is valid as a formula of propositional logic.
- **Blacklisting.** Policy `p` blacklists a set of requests `reqs` iff Expd($\texttt{reqs} \to \texttt{p.deny}$ & !$\texttt{p.grant}$) is valid as a formula of propositional logic.
- **Shadowing.** Policy `pi` shadows policy `pj` in a priority chain $\texttt{p1} > \ldots > \texttt{pn}$ with $1 \leq i < j \leq n$ iff Expd(!$\texttt{pj.undef}$ $\to$!$\texttt{pi.undef}$) is valid as a formula of propositional logic.

Imperial College London

Analysis of method declarations

- Consider the method declaration

```
pol filter(P:pol, R:reqs) {
  (deny when (P.grant & R)) > P
}
```

- Can we statically extract constraints in terms of input parameters `P` and `R` that precisely capture when an invocation of `filter` is gap free, conflict free, a refinement of some other policy, etc?

Imperial College London

Analysis of method `filter`

```
pol filter(P:pol, R:reqs) {
  (deny when (P.grant & R)) > P
}
```

- We compute Expd(*methodBody*.`dec`), and simplify it today for human consumption.
- For `dec` = `grant` this yields `!R & P.grant`. For `dec` = `deny` this yields `(P.grant & R) | (!(P.grant & R) & P.deny)`.
- Exercise: verify the two claims of the item above, and derive constraints for gap freeness and for conflict freeness.

Imperial College London

Analysis findings as pragmas

- We use pragmas as in Eiffel, JML, and Spec# to denote invariants and pre/post-conditions:

```
//@ gapfree if (P.grant & R)| !P.undef
//@ conflictfree if (P.grant & R) | !P.incon
pol filter(P:pol, R:reqs) {
  (deny when (P.grant & R)) > P
}
```

- One can also imagine pragmas `grants if`, and `denies if`, and `invariant`.

Semantics of pragmas

- Done as in ESC/Java: assume pre-conditions when analyzing method body; require pre-conditions when invoking method body.
- E.g. let `filter` have pragma `gapfree if P.grant`
- To check correctness, that the method is gapfree relative to this pre-condition, we discover the exact condition `(P.grant & R)| !P.undef` for gap freeness and check the validity of $\texttt{P.grant} \rightarrow ((\texttt{P.grant} \,\&\, \texttt{R}) \,|!\texttt{P.undef})$, equivalently, the validity of

$$\texttt{P.grant} \rightarrow ((\texttt{P.grant} \,\&\, \texttt{R}) \mid (\texttt{P.grant} \mid \texttt{P.deny}))$$

if we treat `P.deny`, `P.grant`, and `R` as atomic propositions.

Expressiveness of *PolCore*

- The method extension to *PolCore* does not add real expressiveness, just convenience and reusability.
- For any request model $\mathcal{M}$, let $r_1 \equiv r_2$ denote that r_1 and r_2 have the same Boolean abstraction, i.e. that

$$\forall \mathtt{reqsAtom}\colon (r_1 \in \mathtt{reqsAtom}^{\mathcal{M}} \leftrightarrow r_2 \in \mathtt{reqsAtom}^{\mathcal{M}})$$

- For any request model $\mathcal{M}$, this policy language expresses exactly those total functions $f\colon R^{\mathcal{M}} \to \mathbf{Four}$ that cannot distinguish requests r_1 and r_2 that have the same Boolean abstraction:

$$\forall r_1, r_2 \in R^{\mathcal{M}}\colon\ r_1 \equiv r_2 \text{ implies } f(r_1) = f(r_2)$$

Conclusion

What we did

- We used Belnap logic to design a small, simple, and abstract core policy language.
- We interpreted policies over request models, which capture both real-world domains and implemented IT systems.
- This interpretation used the semantics of propositional logic and an expansion of demoted policies into that logic.
- We extended the core language with methods and gave their invocations meaning in the core language.
- These methods, together with demotions of policies and promotions of requests, allowed us to write powerful policy combinators.
- We showed how to statically analyze method declarations and policy expressions, using either SAT solvers or BDDs.

Imperial College London

Where do we go from here?

- Find a PhD student who will implement and experiment with these ideas.
- Demonstrate that assume-guarantee reasoning can be integrated to this language, and that it can handle hierarchical structures, e.g. in roles.
- Present beta version of tool in Very Controlled Natural Language and conduct field studies with real policy writers and readers. (Think managers not geeks.)
- What else should we do?