Model Checking Legal Documents

Daniel Gorín[1] Sergio Mera[1] Fernando Schapachnik[1]

[1]FormaLex Lab,
Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires,
Buenos Aires, Argentina

JURIX 2010

Model Checking Legal Documents

Aim

Find incoherences in (formalizations of) every-day regulations

Model Checking Legal Documents

Aim

Find incoherences in (formalizations of) every-day regulations

Problems there rarely go to court, yet affect “users”.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.
    - That's specially true if we focus not on high-level, rights-giving, general Laws, but on everyday, operational regulations.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.
    - That's specially true if we focus not on high-level, rights-giving, general Laws, but on everyday, operational regulations.

Normative Propositions

What someone should/shouldn't, can/can't do.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.
    - That's specially true if we focus not on high-level, rights-giving, general Laws, but on everyday, operational regulations.

Normative Propositions

What someone should/shouldn't, can/can't do.

Software Specs

What the software should/shouldn't, can/can't do.

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.
  - That's specially true if we focus not on high-level, rights-giving, general Laws, but on everyday, operational regulations.

Normative Propositions

What someone should/shouldn't, can/can't do.

Software Specs

What the software should/shouldn't, can/can't do.

- They seem to be somehow similar. Are they?

Model Checking Legal Documents

- Model Checking is usually conceived as the mathematical analysis of models. These models tend to represent software artifacts, and in particular, software specs.
- Legal documents, we know: a collection of normative propositions.
  - That's specially true if we focus not on high-level, rights-giving, general Laws, but on everyday, operational regulations.

Normative Propositions

What someone should/shouldn't, can/can't do.

Software Specs

What the software should/shouldn't, can/can't do.

- They seem to be somehow similar. Are they?
- Why is that important?

Why is it so important?

- Because we know how to analyze software specs!

Why is it so important?

- Because we know how to analyze software specs!
    - Many successful tools: model checkers, theorem provers (SAT solvers, SMT solvers, etc.)

Why is it so important?

- Because we know how to analyze software specs!
    - Many successful tools: model checkers, theorem provers (SAT solvers, SMT solvers, etc.)
    - Scalability and expressivity are still an issue, but lots of effort put in that direction.

Why is it so important?

- Because we know how to analyze software specs!
  - Many successful tools: model checkers, theorem provers (SAT solvers, SMT solvers, etc.)
  - Scalability and expressivity are still an issue, but lots of effort put in that direction.
- So, if regulations were similar to specs we could use existing technology.

Why is it so important?

- Because we know how to analyze software specs!
  - Many successful tools: model checkers, theorem provers (SAT solvers, SMT solvers, etc.)
  - Scalability and expressivity are still an issue, but lots of effort put in that direction.
- So, if regulations were similar to specs we could use existing technology.
- If not, ad-hoc techniques will have to be developed, and that takes decades!

Specs vs software

Contrary-To-Duty Obligations

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \land O_Y(X) = O_Y(X)$
- Regulations: $O(X) \land O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Permissions

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Permissions

- Difficult topic in DL literature, yet present in regulations.

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Permissions

- Difficult topic in DL literature, yet present in regulations.
- In specs, are they much more than non-determinism?

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Permissions

- Difficult topic in DL literature, yet present in regulations.
- In specs, are they much more than non-determinism?
- Beware! "*The user may print the displayed listing*" sounds like permission but it is an obligation (to developers) to provide the printing option to the user.

Specs vs software

Contrary-To-Duty Obligations

- Specs: $O(X) \wedge O_Y(X) = O_Y(X)$
- Regulations: $O(X) \wedge O_Y(X) =$ incoherence?

Amendments to Deal with Contradictions

- Regulations: F(kill) + P(kill in self-defense) =
  F(kill unless self-defense)
- Specs: F(kill) + P(kill in self-defense) = incoherence

Permissions

- Difficult topic in DL literature, yet present in regulations.
- In specs, are they much more than non-determinism?
- Beware! "*The user may print the displayed listing*" sounds like permission but it is an obligation (to developers) to provide the printing option to the user.
- More on permissions later on today.

Specs vs software (2)

Hierarchies

Specs vs software (2)

Hierarchies

- Regional rule: tax = $10

Specs vs software (2)

Hierarchies

- Regional rule: tax = \$10
- National rule: tax = \$20

Specs vs software (2)

Hierarchies

- Regional rule: tax = \$10
- National rule: tax = \$20
- So, current tax = \$20

Specs vs software (2)

Hierarchies

- Regional rule: tax = \$10
- National rule: tax = \$20
- So, current tax = \$20
- National rule derogated afterwars.

Specs vs software (2)

Hierarchies

- Regional rule: tax $= \$10$
- National rule: tax $= \$20$
- So, current tax $= \$20$
- National rule derogated afterwars.
- Specs: probably, tax $= \$0$

Specs vs software (2)

Hierarchies

- Regional rule: tax = \$10
- National rule: tax = \$20
- So, current tax = \$20
- National rule derogated afterwars.
- Specs: probably, tax = \$0
- Regulations: tax = \$10

Specs vs software (2)

Hierarchies

- Regional rule: tax $=$ \$10
- National rule: tax $=$ \$20
- So, current tax $=$ \$20
- National rule derogated afterwars.
- Specs: probably, tax $=$ \$0
- Regulations: tax $=$ \$10

Specs vs software (2)

Hierarchies

- Regional rule: tax $= \$10$
- National rule: tax $= \$20$
- So, current tax $= \$20$
- National rule derogated afterwars.
- Specs: probably, tax $= \$0$
- Regulations: tax $= \$10$

Ontologies

Specs vs software (2)

Hierarchies

- Regional rule: tax $= \$10$
- National rule: tax $= \$20$
- So, current tax $= \$20$
- National rule derogated afterwars.
- Specs: probably, tax $= \$0$
- Regulations: tax $= \$10$

Ontologies

- Very common in regulations.

Specs vs software (2)

Hierarchies

- Regional rule: tax = \$10
- National rule: tax = \$20
- So, current tax = \$20
- National rule derogated afterwars.
- Specs: probably, tax = \$0
- Regulations: tax = \$10

Ontologies

- Very common in regulations.
- Common in code (inheritance, subclassing), not so common in specs.

Specs vs software (3)

Nesting of Deontic Operators

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
    - The judge may issue the order to the citizen, and the citizen fail to comply.

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
    - The judge may issue the order to the citizen, and the citizen fail to comply.
- Do not confuse with "*The system is obligated to obligate users to do Y*", with Y being "*not access each other's private files*".

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
  - The judge may issue the order to the citizen, and the citizen fail to comply.
- Do not confuse with "*The system is obligated to obligate users to do Y*", with Y being "*not access each other's private files*".
  - There's not such thing as the system issuing the order and the users deciding not to follow it.

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
    - The judge may issue the order to the citizen, and the citizen fail to comply.
- Do not confuse with "*The system is obligated to obligate users to do Y*", with Y being "*not access each other's private files*".
    - There's not such thing as the system issuing the order and the users deciding not to follow it.
    - If the system doesn't enforce Y, then the system is at fault (its developers are).

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
  - The judge may issue the order to the citizen, and the citizen fail to comply.
- Do not confuse with "*The system is obligated to obligate users to do Y*", with Y being "*not access each other's private files*".
  - There's not such thing as the system issuing the order and the users deciding not to follow it.
  - If the system doesn't enforce Y, then the system is at fault (its developers are).
  - Also, if users fail to do Y, then the same system is also responsible.

Specs vs software (3)

Nesting of Deontic Operators

- Eg, obligations to obligate.
- "*The judge is obligated to obligate the citizen to do X*": two obligations, two responsible parties.
  - The judge may issue the order to the citizen, and the citizen fail to comply.
- Do not confuse with "*The system is obligated to obligate users to do Y*", with Y being "*not access each other's private files*".
  - There's not such thing as the system issuing the order and the users deciding not to follow it.
  - If the system doesn't enforce Y, then the system is at fault (its developers are).
  - Also, if users fail to do Y, then the same system is also responsible.
- This type of predicates seem to be just a complex wording for only one obligation.

Specs vs software (4)

Self-Referencing Modifications

Specs vs software (4)

Self-Referencing Modifications

- Eg, "*Let Article X of Bill Y be modified to mandate that from now on such and such*".

Specs vs software (4)

Self-Referencing Modifications

- Eg, "*Let Article X of Bill Y be modified to mandate that from now on such and such*".
- Very different to run-time or compile-time configurations.

Specs vs software (4)

Self-Referencing Modifications

- Eg, "*Let Article X of Bill Y be modified to mandate that from now on such and such*".
- Very different to run-time or compile-time configurations.
- Specs do not self-modify themselves...

Specs vs software (4)

Self-Referencing Modifications

- Eg, "*Let Article X of Bill Y be modified to mandate that from now on such and such*".
- Very different to run-time or compile-time configurations.
- Specs do not self-modify themselves...
- ...except for some prototype dynamic specification languages with self-referencing capabilities...

Specs vs software (4)

Self-Referencing Modifications

- Eg, "*Let Article X of Bill Y be modified to mandate that from now on such and such*".
- Very different to run-time or compile-time configurations.
- Specs do not self-modify themselves...
- ...except for some prototype dynamic specification languages with self-referencing capabilities...
- ...but they are still far from being used in the current state of the practice.

Specs vs software (5)

Deontic-Conditional Validity

Specs vs software (5)

Deontic-Conditional Validity

- Eg, "*if at the time of the execution the agent were obligated to … then she …*"

Specs vs software (5)

Deontic-Conditional Validity

- Eg, "*if at the time of the execution the agent were obligated to ... then she ...*"
- Software specs may impose behaviours based on run-time conditions.

Specs vs software (5)

Deontic-Conditional Validity

- Eg, "*if at the time of the execution the agent were obligated to ... then she ...*"
- Software specs may impose behaviours based on run-time conditions.
- But they don't specify behaviour that is conditional to the *runtime requirements*...

Specs vs software (5)

Deontic-Conditional Validity

- Eg, "*if at the time of the execution the agent were obligated to ... then she ...*"
- Software specs may impose behaviours based on run-time conditions.
- But they don't specify behaviour that is conditional to the *runtime requirements*...
- ...if the term makes any sense at all.

Specs vs software (recap)

- Last three's common denominator: considering deontic operators as first-class operators.

Specs vs software (recap)

- Last three's common denominator: considering deontic operators as first-class operators.
- They don't seem to occur in specs.

Specs vs software (recap)

- Last three's common denominator: considering deontic operators as first-class operators.
- They don't seem to occur in specs.
- If we only consider regulations that do not need them:

Specs vs software (recap)

- Last three's common denominator: considering deontic operators as first-class operators.
- They don't seem to occur in specs.
- If we only consider regulations that do not need them:
  - + Still able cover an important and varied amount of regulations that are common in the real world.

Specs vs software (recap)

- Last three's common denominator: considering deontic operators as first-class operators.
- They don't seem to occur in specs.
- If we only consider regulations that do not need them:
  + Still able cover an important and varied amount of regulations that are common in the real world.
  + Can resort to tools and technologies meant to analyze software specs.

Which existing tools?

- Miriad of possible approaches with success stories in specs.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.
- General approach:

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.
- General approach:
    - Automata network describing system behaviour.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.
- General approach:
  - Automata network describing system behaviour.
  - Formula to assert requirements.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.
- General approach:
  - Automata network describing system behaviour.
  - Formula to assert requirements.
  - If model checker answers *YES*, formula holds.

Which existing tools?

- Miriad of possible approaches with success stories in specs.
- We chose LTL.
- Many available, well-established model checkers: SPIN, DiVinE, NuSMV, etc.
- General approach:
  - Automata network describing system behaviour.
  - Formula to assert requirements.
  - If model checker answers *YES*, formula holds.
  - If it answers *NO*, then it outputs a counterexample trace.

Introducing the FL language

- We devised a wrapper language: FL

Introducing the FL language

- We devised a wrapper language: FL
- Background theory:

Introducing the `FL` language

- We devised a wrapper language: `FL`
- Background theory:
    - Aim: reflect the “real world”.

Introducing the `FL` language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:

Introducing the `FL` language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.

Introducing the `FL` language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the "real world".
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$
  - $F(\varphi) = \Box\neg\varphi$

Introducing the FL language

- We devised a wrapper language: FL
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$
  - $F(\varphi) = \Box\neg\varphi$
  - Also, repaired versions:

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$
  - $F(\varphi) = \Box\neg\varphi$
  - Also, repaired versions:
    - $O_\psi(\varphi) = \Box(\neg\varphi \rightarrow \psi)$

Introducing the FL language

- We devised a wrapper language: `FL`
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$
  - $F(\varphi) = \Box\neg\varphi$
  - Also, repaired versions:
    - $O_{\psi}(\varphi) = \Box(\neg\varphi \rightarrow \psi)$
    - $F_{\psi}(\varphi) = \Box(\varphi \rightarrow \psi)$

Introducing the FL language

- We devised a wrapper language: FL
- Background theory:
  - Aim: reflect the “real world”.
  - Simple language, translates into automata.
- Formulae:
  - To state normative propositions.
  - Deontic operators: O and F.
  - They affect the set of (legal) traces.
  - $O(\varphi) = \Box\varphi$
  - $F(\varphi) = \Box\neg\varphi$
  - Also, repaired versions:
    - $O_\psi(\varphi) = \Box(\neg\varphi \to \psi)$
    - $F_\psi(\varphi) = \Box(\varphi \to \psi)$
  - Permission is contemplated, but behaves differently (later on).

Example

- Background theory:

actions SemBegins**,** SemEnds
interval Semester **defined by actions** SemBegins-SemEnds
        only occurs in scope AcademicYear **occurrences** 2
action TakeExam **outputValues** {PassWithHonors, Pass, Fail}
                only occurs in scope Semester

Example

- Background theory:

  actions SemBegins**,** SemEnds
  interval Semester **defined by actions** SemBegins-SemEnds
  only occurs in scope AcademicYear **occurrences** 2
  action TakeExam **outputValues** {PassWithHonors, Pass, Fail}
  only occurs in scope Semester

- The students are obligated to take at least one exam per academic year
  $O(\Diamond_{\text{AcademicYear}}\text{TakeExam})$

Example

- Background theory:

  actions SemBegins, SemEnds
  interval Semester **defined by actions** SemBegins-SemEnds
  only occurs in scope AcademicYear **occurrences** 2
  action TakeExam **outputValues** {PassWithHonors, Pass, Fail}
  only occurs in scope Semester

- The students are obligated to take at least one exam per academic year
  $O(\Diamond_{\text{AcademicYear}} \text{TakeExam})$

- It is forbidden to fail two or more exams during a semester. If that happens, situation can be fixed by passing with honors some exam in that same semester

  $F_\rho(\Diamond_{\text{Semester}}(\text{TakeExam.Fail} \wedge X \Diamond_{\text{Semester}} \text{TakeExam.Fail}))$
  where $\rho = \Diamond_{\text{Semester}} \text{TakeExam.PassWithHonors}$

Example

- Background theory:
  actions SemBegins, SemEnds
  interval Semester **defined by actions** SemBegins-SemEnds
  only occurs in scope AcademicYear **occurrences** 2
  action TakeExam **outputValues** {PassWithHonors, Pass, Fail}
  only occurs in scope Semester
- The students are obligated to take at least one exam per academic year
  $O(\Diamond_{\text{AcademicYear}} \text{TakeExam})$
- It is forbidden to fail two or more exams during a semester. If that happens, situation can be fixed by passing with honors some exam in that same semester

  $F_\rho(\Diamond_{\text{Semester}}(\text{TakeExam.Fail} \wedge X \Diamond_{\text{Semester}} \text{TakeExam.Fail}))$
  where $\rho = \Diamond_{\text{Semester}} \text{TakeExam.PassWithHonors}$
- Complex property: it is permitted to fail up to *n* exams
  counter failed **increments with action** TakeExam.Fail
  $\text{P}(\text{failed} \leq n)$

Permissions

- Permissions do not affect the set of traces.

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as “checks” that the rest of the rules must fulfill.

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists\tau \in \text{system traces}, \tau \models \Diamond\varphi$

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond\varphi$
- Satisfies usual desiderata:

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond \varphi$
- Satisfies usual desiderata:
  1. *Obligatory $\rightarrow$ should be regarded as permitted.*

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond\varphi$
- Satisfies usual desiderata:
  1. *Obligatory $\rightarrow$ should be regarded as permitted.*
  2. *Forbidden $\rightarrow$ should be regarded as not permitted.*

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond\varphi$
- Satisfies usual desiderata:
  1. *Obligatory $\rightarrow$ should be regarded as permitted.*
  2. *Forbidden $\rightarrow$ should be regarded as not permitted.*
  3. *Explicitly permitted $\rightarrow$ should be regarded as not forbidden.*

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond\varphi$
- Satisfies usual desiderata:
  1. *Obligatory $\rightarrow$ should be regarded as permitted.*
  2. *Forbidden $\rightarrow$ should be regarded as not permitted.*
  3. *Explicitly permitted $\rightarrow$ should be regarded as not forbidden.*
  4. *Explicitly permitted $\rightarrow$ should not be regarded as obligatory.*

Permissions

- Permissions do not affect the set of traces.
- Instead, they are interpreted as "checks" that the rest of the rules must fulfill.
- If not, then there is an incoherence.
- That is, $P(\varphi)$ means:
  $\exists \tau \in \text{system traces}, \tau \models \Diamond\varphi$
- Satisfies usual desiderata:
  1. *Obligatory* $\rightarrow$ *should be regarded as permitted.*
  2. *Forbidden* $\rightarrow$ *should be regarded as not permitted.*
  3. *Explicitly permitted* $\rightarrow$ *should be regarded as not forbidden.*
  4. *Explicitly permitted* $\rightarrow$ *should not be regarded as obligatory.*
- Even with conditional permission, which seems hard according to the literature.

Verification

- Background theory is translated into automata.

Verification

- Background theory is translated into automata.
- Formulae are combined in specific ways so the model checker can detect:

Verification

- Background theory is translated into automata.
- Formulae are combined in specific ways so the model checker can detect:
  - Explicit contradictions

Verification

- Background theory is translated into automata.
- Formulae are combined in specific ways so the model checker can detect:
    - Explicit contradictions
    - Forbidden reparations

Verification

- Background theory is translated into automata.
- Formulae are combined in specific ways so the model checker can detect:
  - Explicit contradictions
  - Forbidden reparations
  - Impossible permissions

Verification

- Background theory is translated into automata.
- Formulae are combined in specific ways so the model checker can detect:
  - Explicit contradictions
  - Forbidden reparations
  - Impossible permissions
  - Etc

Summing up...

- Software specs and regulations seem to be different, but not so much!

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
    - Background theory (translates into automata)

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
  - Background theory (translates into automata)
  - Normative propositions (translate into LTL formulae)

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
  - Background theory (translates into automata)
  - Normative propositions (translate into LTL formulae)
  - Deontic operators

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
  - Background theory (translates into automata)
  - Normative propositions (translate into LTL formulae)
  - Deontic operators
  - Consistency checks

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
  - Background theory (translates into automata)
  - Normative propositions (translate into LTL formulae)
  - Deontic operators
  - Consistency checks
  - Allows to state complex properties

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
    - Background theory (translates into automata)
    - Normative propositions (translate into LTL formulae)
    - Deontic operators
    - Consistency checks
    - Allows to state complex properties
- Permissions interpreted in a novel way.

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
    - Background theory (translates into automata)
    - Normative propositions (translate into LTL formulae)
    - Deontic operators
    - Consistency checks
    - Allows to state complex properties
- Permissions interpreted in a novel way.
- Questions?

Summing up...

- Software specs and regulations seem to be different, but not so much!
- It's possible to use existing model checking tools.
- We devised `FL`: a wrapper for LTL, supporting
  - Background theory (translates into automata)
  - Normative propositions (translate into LTL formulae)
  - Deontic operators
  - Consistency checks
  - Allows to state complex properties
- Permissions interpreted in a novel way.
- Questions?
- Thanks for not being in the Beatles Museum *right now*!

Bonus Track

- Encoding $O(\Diamond_{\text{AcademicYear}}\text{TakeExam})$

Bonus Track

- Encoding $O(\Diamond_{\text{AcademicYear}}\text{TakeExam})$
- Background theory

  actions YearBegins**,** YearEnds
  interval AcademicYear **defined by actions** YearBegins-YearEnds

Bonus Track

- Encoding $O(\Diamond_{\text{AcademicYear}}\text{TakeExam})$
- Background theory

  actions YearBegins**,** YearEnds
  interval AcademicYear **defined by actions** YearBegins-YearEnds

- Automata has inAcademicYear boolean variable. Turns on with YearBegins and off with YearEnds. Those occurr non-deterministically.

Bonus Track

- Encoding $O(\Diamond_{\text{AcademicYear}}\text{TakeExam})$
- Background theory

  actions YearBegins**,** YearEnds
  interval AcademicYear **defined by actions** YearBegins-YearEnds

- Automata has inAcademicYear boolean variable. Turns on with YearBegins and off with YearEnds. Those occurr non-deterministically.
- $\Diamond_{\text{AcademicYear}}\text{TakeExam} =$
  $\text{YearBegins} \rightarrow (\text{inAcademicYear U TakeExam})$