Friedrich-Alexander-Universität
Erlangen-Nürnberg

FAU

Extending a Reasoner for Non-Expansive Fuzzy ALC with the `generally` Operator

Felix Lützow

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Informatik

June 9, 2026

1. Theoretical Background
2. The Logic of `generally`
3. Existing System and Architecture
4. Logic Selection
5. Implementation of `generally`
6. Experimental Evaluation

Theoretical Background

Syntax

Classical ALC concepts are generated from concept names and role successors:

$$C, D ::= p \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

The base reasoner uses the following non-expansive fuzzy ALC concepts:

$$C, D ::= p \mid c \mid \neg C \mid C \ominus c \mid C \oplus c \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

where $p \in N_C$, $R \in N_R$, and $c \in [0, 1] \cap \mathbb{Q}$.

Defined operators:

$$C \sqcup D := \neg(\neg C \sqcap \neg D), \qquad \forall R.C := \neg\exists R.\neg C,$$
$$C \oplus c := \neg((\neg C) \ominus c)$$

Theoretical Background
Semantics

An interpretation I consists of a non-empty set Δ^I of individuals and assigns fuzzy valuations to every concept name p and role name R:

$$p^I : \Delta^I \to [0,1], \qquad R^I : \Delta^I \times \Delta^I \to [0,1]$$

For $x \in \Delta^I$, the valuation of compound concepts is defined recursively:

$$\begin{aligned} c^I(x) &= c, & (\neg C)^I(x) &= 1 - C^I(x), \\ (C \ominus c)^I(x) &= \max\{C^I(x) - c, 0\}, & (C \sqcap D)^I(x) &= \min\{C^I(x), D^I(x)\} \end{aligned}$$

$$(\exists R.C)^I(x) = \sup_{y \in \Delta^I} \min\{R^I(x,y), C^I(y)\}$$

For every concept C and individual x, the interpretation assigns a truth degree $C^I(x) \in [0,1]$

Theoretical Background

Concept Assertions and Normalization

Assertion syntax: For $\bowtie \in \{<, \leq, >, \geq\}$, an expression

$$C \bowtie a \qquad (a \in [0,1])$$

is a concept assertion.
Semantics: At an individual x, the assertion means

$$C^I(x) \bowtie a.$$

Interval representation: The constraints on each formula are collected as an interval

$$C^I(x) \in [a, b]$$

Thus every formula has a lower and an upper bound. In the solver, assertions are normalized to non-strict lower bounds $C \geq a$: strict bounds are shifted to the next discrete truth value, and the upper bound $C \leq b$ is stored as the lower bound $\neg C \geq 1 - b$.

- A tableau label stores formula IDs with their strongest lower bounds
- Propositional rules translate lower bounds to sub-concepts:

$$(C \ominus c) \geq a \quad \rightsquigarrow \quad C \geq a + c \qquad (a > 0)$$

$$(C \sqcap D) \geq a \quad \rightsquigarrow \quad C \geq a,\ D \geq a$$

- A branch closes if $C \geq a$ and $\neg C \geq b$ require $a + b > 1$

TBoxes are global constraints and therefore have to be propagated to every state - not supported for generally

Theoretical Background

The One-Step View

Modal or role-successor constructs are handled separately from propositional decomposition.

1. Expand propositional operators until a successor construct is reached
2. Replace concepts below that construct by fresh variables
3. Solve the remaining one-step problem: can the current state have successors satisfying the required bounds?

$$\exists R.C \quad \rightsquigarrow \quad \exists R.v$$

- In fuzzy ALC, $\exists R.C$ is a role-indexed, modality-like operator; the one-step structure is a fuzzy successor relation
- For `generally`, G is a probabilistic modal operator; the one-step structure is a discrete probability distribution over successors
- Successor requirements become new tableau labels and are checked recursively

This is the bridge between the theory and implementation: successor reasoning becomes a finite search over one-step profiles.

1. Theoretical Background
2. The Logic of `generally`
3. Existing System and Architecture
4. Logic Selection
5. Implementation of `generally`
6. Experimental Evaluation

The Logic of generally
Probabilistic Semantics

The modal logic keeps the same non-expansive propositional base, but uses the probabilistic modality G:

$$\varphi, \psi ::= p \mid c \mid \neg\varphi \mid \varphi \ominus c \mid \varphi \oplus c \mid \varphi \sqcap \psi \mid \varphi \sqcup \psi \mid \mathsf{G}\varphi$$

where $p \in At$ is an atom and $c \in [0,1] \cap \mathbb{Q}$

Defined operators:

$$\varphi \sqcup \psi := \neg(\neg\varphi \sqcap \neg\psi), \qquad \varphi \oplus c := \neg((\neg\varphi) \ominus c)$$

A model $M = (X, \tau, \pi)$ consists of states, successor distributions, and fuzzy valuations of atoms:

$$M = (X, \tau, \pi), \qquad \tau : X \to \mathcal{D}(X), \qquad \pi : X \times At \to [0,1]$$

$\tau(x)(A)$ is the probability that a successor of x lies in $A \subseteq X$

$$[\![\mathsf{G}\varphi]\!]_M(x) = \sup_{\alpha \in [0,1]} \min\big(\alpha,\, \tau(x)\big(\{y \in X \mid [\![\varphi]\!]_M(y) \geq \alpha\}\big)\big)$$

- α is a candidate truth threshold for φ
- The probability term measures how likely successors reach that threshold

The Logic of generally
Finite One-Step Search

For a tableau label with n relevant G-formulas, the coalgebraic argument gives a finite representation of the one-step search.

$$\mathsf{G}\varphi_1, \ldots, \mathsf{G}\varphi_n \quad \rightsquigarrow \quad \text{profiles in } \{0,1\}^{2n}$$

- Each profile records which threshold tests are met for the positive and dual constraints
- It is enough to combine at most $2n+1$ vectors of singleton successor profiles
- The solver searches over profile configurations instead of arbitrary probabilistic successor structures
- A candidate configuration is accepted if suitable probability weights exist; implementation-wise this is a linear feasibility check

Example.

$$\begin{array}{rcl} \mathsf{G}(\mathit{safe} \sqcap \mathit{tested}) & \geq & 0.7 \\ \mathsf{G}(\mathit{stable} \ominus 0.2) & \geq & 0.6 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} v_1 := \mathit{safe} \sqcap \mathit{tested} \\ v_2 := \mathit{stable} \ominus 0.2 \end{array}$$

The GEN rule collects both inner formulas in one shared one-step search. Each profile records both threshold tests, and the solver checks weights with

$$\sum_{i:\, P_i(v_1 \geq 0.7)=1} t_i \geq 0.7, \qquad \sum_{i:\, P_i(v_2 \geq 0.6)=1} t_i \geq 0.6, \qquad \sum_i t_i = 1$$

Candidate labels are checked locally before weight feasibility, surviving labels are checked recursively as successor states

1. Theoretical Background
2. The Logic of `generally`
3. **Existing System and Architecture**
4. Logic Selection
5. Implementation of `generally`
6. Experimental Evaluation

Existing System and Architecture

Data Flow

The extension builds on the existing OCaml reasoner for non-expansive fuzzy ALC.

input $\longrightarrow$ parser $\longrightarrow$ normalization $\longrightarrow$ closure and labels $\longrightarrow$ graph tableau

- The frontend parses assertions and optional TBoxes
- Preprocessing computes normal forms, formula closure, negations, and index ranges for relevant formula families
- Labels are represented compactly by formula IDs and bound information
- The graph tableau stores generated states and reuses them when the same label appears again

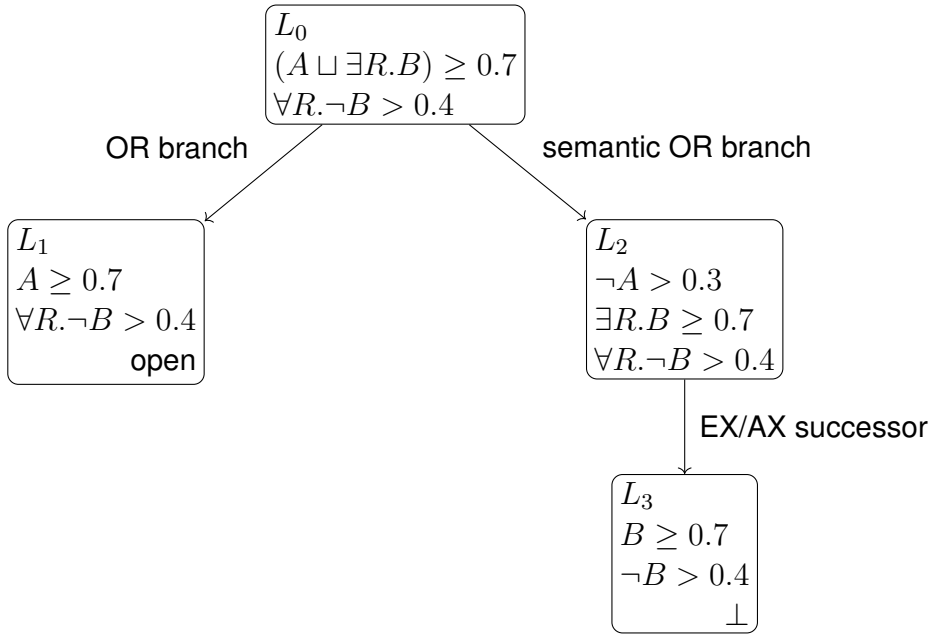
Extension point: `generally` reuses the shared tableau core, concept rules, and tableau optimizations, but adds a new formula class and a new modal expansion step.

For future logics, the current abstraction is strongest when a logic only changes rule availability. New modal operators still affect parser support, closure tables, and the rule case distinction in `FuzzyGraph`.

Existing System and Architecture
Tableau Example

The graph stores labels as nodes. Expanding a principal formula creates new labels; repeated labels are reused instead of copied.

$$L_0 = \{(A \sqcup \exists R.B) \geq 0.7,\ \forall R.\neg B > 0.4\}$$



- OR nodes need one open child; EX/AX expansion creates AND-successors
- Here the successor label closes because $B \geq 0.7$ and $\neg B > 0.4$ demand incompatible intervals

1. Theoretical Background
2. The Logic of `generally`
3. Existing System and Architecture
4. **Logic Selection**
5. Implementation of `generally`
6. Experimental Evaluation

Logic Selection
Why It Was Needed

The original system was a solver for non-expansive fuzzy ALC Adding `generally` changes the modal step, but not the whole tableau infrastructure

- Keep one shared graph-tableau core
- Move logic-specific choices into a small configuration
- Select which formulas may become principal formulas
- Select which implemented modal expansion rule is available
- Reject inputs that a logic variant does not support

input $\longrightarrow$ parser $\longrightarrow$ logic configuration $\longrightarrow$ shared solver

Main idea: `fuzzy` and `generally` are modes of the same solver and share non-modal reasoning. This works best for closely related logics that reuse the existing rule scheme.

Logic Selection
The Configuration

In the OCaml implementation, a logic is represented by a descriptor in `FuzzyGraph`.

`name`	:	CLI and error messages
`select_principal`	:	next expandable formula
`enable_exists_rule`	:	whether $\exists R.C$ is expanded
`enable_gen_rule`	:	whether Gφ is expanded
`allow_tbox`	:	whether global TBox axioms are accepted

- The solver calls `logic.select_principal` whenever it creates or revisits a tableau label
- If the selected formula is an existential formula, the fuzzy-ALC successor rule is used
- The central case distinction in `FuzzyGraph` then dispatches to OR branching or to the modal EX/GEN expansion

This makes rule availability part of the logic, but not a fully generic plugin interface for arbitrary new modal rules

Logic Selection
Supported Modes

Mode	∃ rule	G rule	TBox
`fuzzy`	enabled	disabled	allowed
`generally`	disabled	enabled	rejected

- `fuzzy` is the default mode and matches the original reasoner
- `generally` disables existential expansion and lets `GEN` formulas become principal formulas
- TBoxes are rejected for `generally`, because the implemented theory only covers the TBox-free probabilistic variant

Users can choose the mode with:

`-logic generally` or `-logic fuzzy`

- Future logics can be added with relative ease by turning off unwanted rules
- New operators or one-step rules also require changes in parsing, closure/index tables, and `FuzzyGraph`

1. Theoretical Background
2. The Logic of `generally`
3. Existing System and Architecture
4. Logic Selection
5. **Implementation of `generally`**
6. Experimental Evaluation

Implementation of `generally`

Where It Enters the Reasoner

The implementation keeps the old tableau infrastructure and adds `generally` as a second logic mode.

Part	Change
Formula syntax	new constructor `GEN` and parser token `G`
Formula closure	hash-consing, negation, printing, and indexing know `GEN`
Label machinery	`GEN` formulas get their own index range
Graph tableau	`GEN` can be selected as a principal formula
Executable	`-logic generally` selects the new mode

- The original `fuzzy` mode remains the default
- The `generally` mode disables the existential rule and rejects TBoxes
- All propositional simplification, subsumption, dependency tracking, and graph reuse are shared with the original reasoner

Implementation of `generally`

The GEN Rule

When a tableau label contains `GEN` formulas, the implementation expands all currently active `GEN` bounds together.

$$\mathsf{G}\varphi_i \in [a_i, b_i] \quad \rightsquigarrow \quad (a_i, b_i) \text{ for the one-step solver}$$

- `mkGENList` collects all `GEN` formulas in the label
- The inner formula of each `GEN` node is read from the closure arrays
- Lower and upper bounds are normalized through complement:

$$\mathsf{G}\neg\varphi \geq l \quad \Rightarrow \quad \mathsf{G}\varphi \leq 1 - l$$

- If several constraints mention the same inner formula, the strongest lower and strongest upper bound are kept

Result: the graph rule receives a compact list of interval constraints for the probabilistic one-step problem.

Implementation of `generally`

One-Step Solver

The mathematical one-step part is implemented in `GenerallySolver`.

$$n \text{ GEN constraints} \quad \Rightarrow \quad \text{profiles over } 2n \text{ bound coordinates}$$

- A profile records, for every bound, whether a successor counts towards the lower-threshold test or the upper-threshold test
- Configurations contain at most $2n + 1$ profiles
- For each candidate configuration, the solver introduces probability weights t_i with

$$t_i \geq 0, \qquad \sum_i t_i = 1$$

- The required lower and upper bounds become linear constraints over the sums of those weights
- Feasibility is checked with `ocplib-simplex`

Implementation of `generally`

Enumerating Profiles

The proof uses vectors in $\{0,1\}^{2n}$; the implementation makes this search more structured.

$$(0,1) \longmapsto 0, \qquad (1,0) \longmapsto 1, \qquad (1,1) \longmapsto 2$$

$$\underbrace{(201)_3}_{\text{index } 19} \longleftrightarrow \underbrace{\big((1,1),(0,1),(1,0)\big)}_{\text{local pairs}} \longleftrightarrow \underbrace{\begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}}_{\text{profile vector}}$$

- The locally useless case $(0,0)$ is excluded
- Hard bounds remove impossible digits; the encoding then becomes mixed-radix
- The radix-coded generator decodes stable indices on demand, without materializing all profiles, and can resume after a failed configuration

Important detail: even the benchmark mode called `noopt` still uses these structural reductions

Implementation of `generally`
Building Successor Labels

A candidate configuration is first translated into successor labels.

$$\begin{aligned} P_i \text{ meets } \varphi \geq a &\rightsquigarrow \varphi \geq a \\ P_i \text{ misses } \varphi \geq a &\rightsquigarrow \neg\varphi > 1 - a \\ P_i \text{ meets } \varphi \leq b &\rightsquigarrow \neg\varphi \geq 1 - b \\ P_i \text{ misses } \varphi \leq b &\rightsquigarrow \varphi > b \end{aligned}$$

The table shows semantic bounds. Internally, strict bounds are shifted to the next discrete truth value and stored as $\geq$-assertions.

Example. Continuing the previous example:

$$\begin{aligned} v_1 &:= safe \sqcap tested, & v_2 &:= stable \ominus 0.2 \\ P_i(v_1 \geq 0.7) &= 1, & P_i(v_2 \geq 0.6) &= 0 \end{aligned}$$

$$L_i = \{safe \sqcap tested \geq 0.7,\ \neg(stable \ominus 0.2) > 0.4\}.$$

- Each profile in the candidate configuration yields one successor label
- Locally impossible labels are forbidden before linear feasibility is checked
- After the linear check, remaining labels form a temporary AND node. If a child fails, search resumes with the next configuration

Implementation of `generally`

Optimizations

The direct search is exponential, so the implementation adds several optimizations.

- **Sparse forbidden set:** once a profile is locally impossible or its successor fails, its stable index is skipped in later configurations
- **Resume cursor:** after a failed configuration, `solve_next_with` continues after the last tried configuration
- **State cache:** building the same successor label repeatedly can be avoided, but the implementation keeps this disabled by default because it can increase memory use strongly on deep formulas
- **Fast unsat checks:** conjunction conflicts are detected before profile generation and before linear programs are built

Implementation of `generally`

Testing

The new mode is tested at the solver level and through generated benchmark families.

- Unit tests cover profile enumeration, feasibility checks, and resume behavior after failed configurations
- Generated benchmark series `s9`–`s17` exercise SAT and UNSAT `generally` cases
- The benchmarks separate direct contradictions, deep formulas, memory-heavy cases, and satisfiable instances with large profile search spaces
- Profiling data is collected for time and memory, so optimization changes can be compared across the same generated inputs

1. Theoretical Background
2. The Logic of `generally`
3. Existing System and Architecture
4. Logic Selection
5. Implementation of `generally`
6. **Experimental Evaluation**

Experimental Evaluation

Benchmark Setup

The benchmarks compare the original tableau optimizations with the new `generally`-specific optimizations.

Mode	Meaning
`noopt`	comparable baseline with optional optimizations disabled
`genfast`	early checks for impossible GEN constraints
`gencache`	cache generated GEN successor states
`genforbid`	skip profiles that already produced contradictions
`genall`	all default GEN optimizations
`allop`	GEN optimizations plus general tableau optimizations

- Inputs are generated by `formula_gen.ml`
- Each run uses a timeout of 600 seconds
- The important series are structural UNSAT cases, hard SAT search cases, and memory-heavy deep formulas

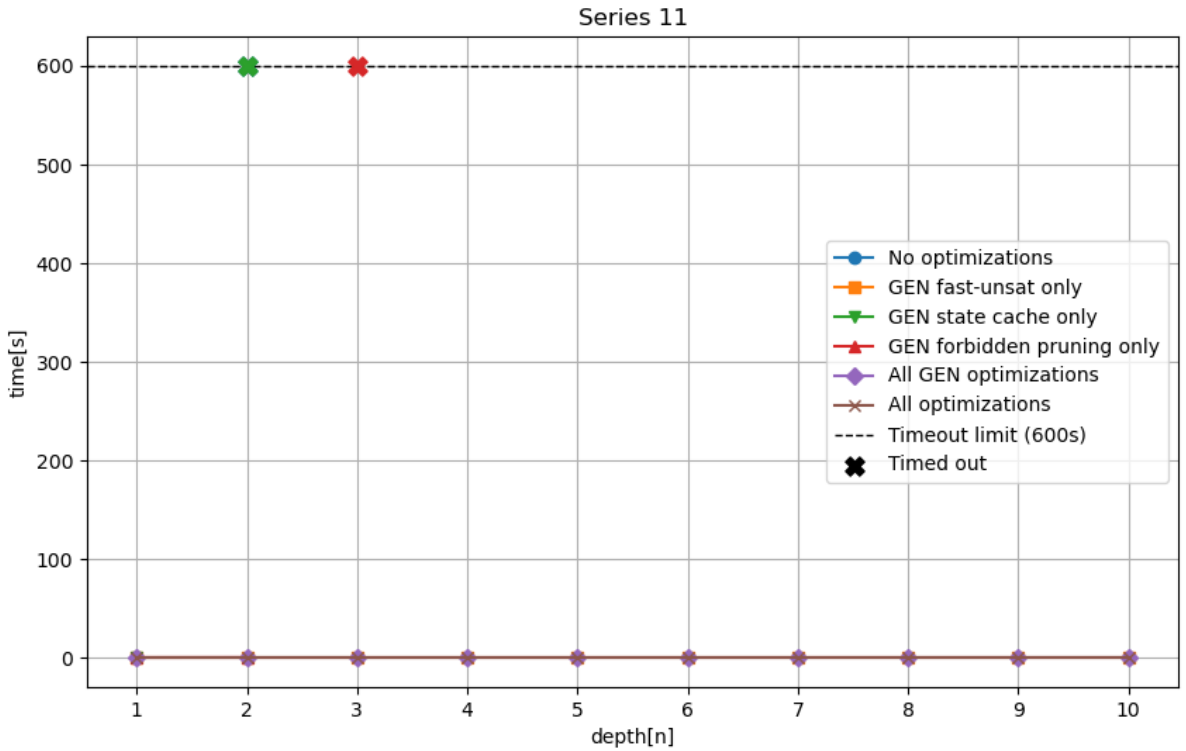
Fast UNSAT Detection

Conjunction Conflict

Input idea:

$$\mathsf{G}p_i \in \left[\frac{1}{4}, \frac{3}{4}\right], \qquad \mathsf{G}\left(\bigwedge_i p_i\right) > \frac{3}{4}.$$

- Without GEN-specific checks, the solver times out very early
- Fast-unsat detects the structural contradiction directly
- General tableau optimizations add almost nothing here



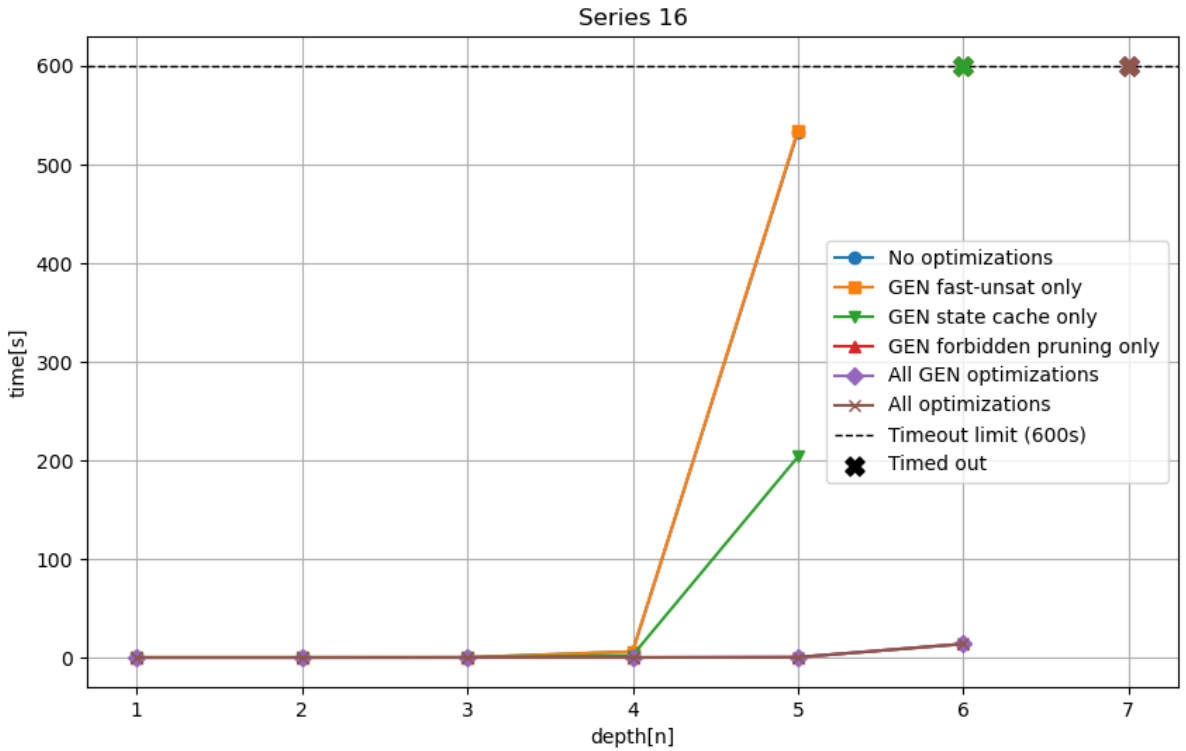
Hard Search Case

Balanced SAT Instances

Input idea:

$$\mathsf{G}p_i \geq \frac{1}{2}, \qquad \mathsf{G}\neg p_i \geq \frac{1}{2}.$$

- The formulas are satisfiable, so the branch cannot close early
- The search has to try many successor profiles
- Forbidden pruning is the main improvement
- It changes the last solved instance from roughly 550 s to roughly 14 s



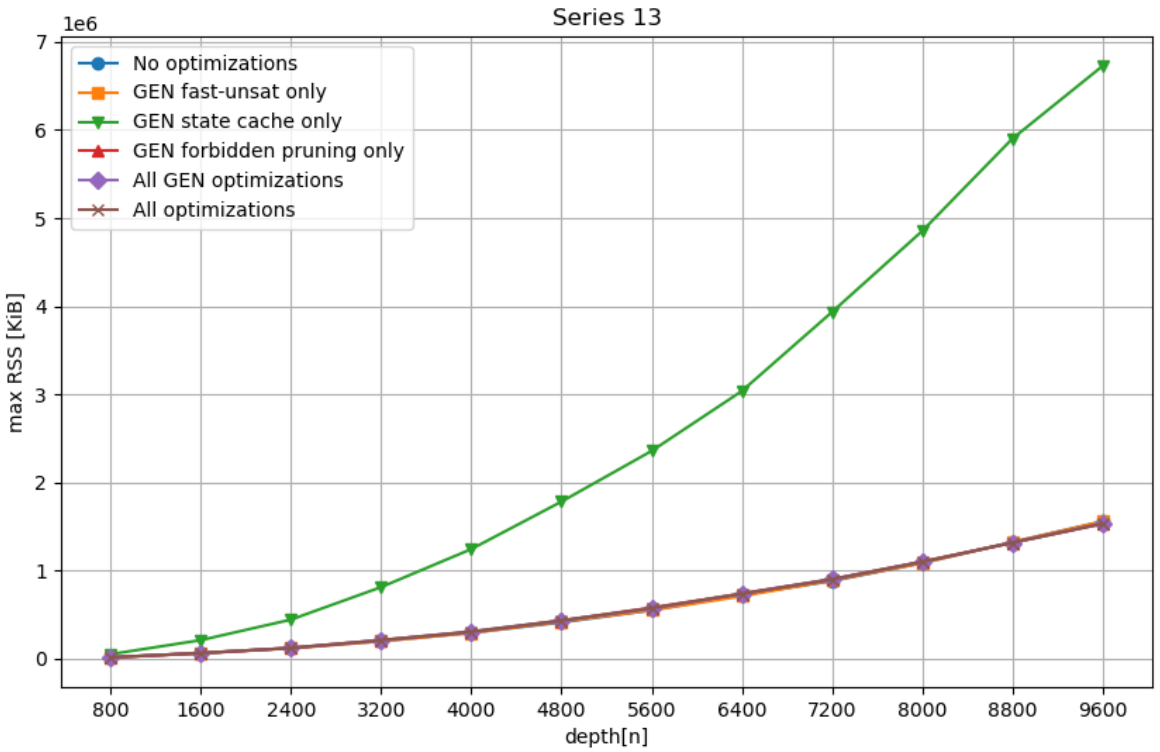
Memory Behavior

State Cache Is Expensive

Series 13:

$$\mathsf{G}^d p_0, \quad \mathsf{G}^d p_1, \quad \mathsf{G}^d(p_0 \sqcap p_1) \in \left[\frac{1}{4}, \frac{3}{4}\right].$$

- Deep formulas stress the internal representation
- The explicit GEN state cache uses much more memory
- The default keeps this cache disabled
- Forbidden pruning gives the useful part without the large memory cost



Experimental Evaluation

Takeaways

- The implementation decides the generated SAT and UNSAT benchmark families as expected
- Fast-unsat detects structural conjunction conflicts before profile search
- Forbidden pruning is best for satisfiable search-space cases
- The optional state cache can help locally, but it is too memory-heavy for the default configuration
- The main bottleneck is the GEN one-step search, not the old tableau infrastructure

Conclusion

- The existing fuzzy-ALC reasoner was extended with explicit logic selection instead of splitting the implementation into separate solvers
- The new `generally` mode adds probabilistic one-step reasoning via finite successor profiles and linear feasibility checks
- GEN-specific optimisations are necessary for practical instances: fast-unsat handles conjunction conflicts, while forbidden-profile pruning helps on satisfiable search cases
- The implemented `generally` mode is intentionally TBox-free
- The logic selection is a useful extension for further similar logics