Temporal Safety Proofs for Systems Code

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar

George C. Necula, Grégoire Sutre, Westley Weimer

EECS Departement, University of California, Berkeley

LaBRI, Université de Bordeaux

Based on the following papers:

- Lazy Abstraction, POPL'02, Jan 2002
- Temporal Safety Proofs for Systems Code, CAV'02, July 2002

Downloadable from:

`http://www.labri.fr/~sutre/Publications`

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

Motivations

- Methodology for the verification and certification of large software
- Crucial for systems code (device drivers)
  - high rate of bugs (especially in device drivers)
  - no runtime protection
  - many third-party providers

Motivations

- Methodology for the verification and certification of large software
- Crucial for systems code (device drivers)
  - high rate of bugs (especially in device drivers)
  - no runtime protection
  - many third-party providers
- We focus on:
  - safety properties
    - locking displines, interface specifications
  - software written in C
    - Linux & Windows NT device drivers

Ingredients

- Model-checking, but...
  - doesn't scale to low-level implementations
  - can only handle (finite) abstractions

Ingredients

- Model-checking, but...
  - doesn't scale to low-level implementations
  - can only handle (finite) abstractions
- Verification using abstraction – refinement based techniques
  - automatic construction and verification of abstract models
  - counter-example guided refinement
  - lazy abstraction (non uniform abstraction)

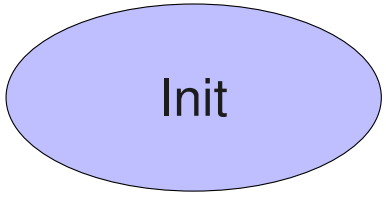
Ingredients

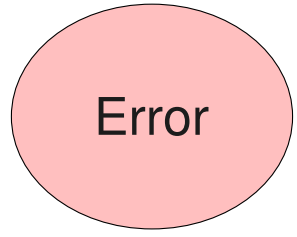
- Model-checking, but...
  - doesn't scale to low-level implementations
  - can only handle (finite) abstractions
- Verification using abstraction – refinement based techniques
  - automatic construction and verification of abstract models
  - counter-example guided refinement
  - lazy abstraction (non uniform abstraction)
- Certification: proof-carrying code
  - verification condition computed from the abstract reachability set
  - proof generation using internal data from verification
  - small correctness certificates

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

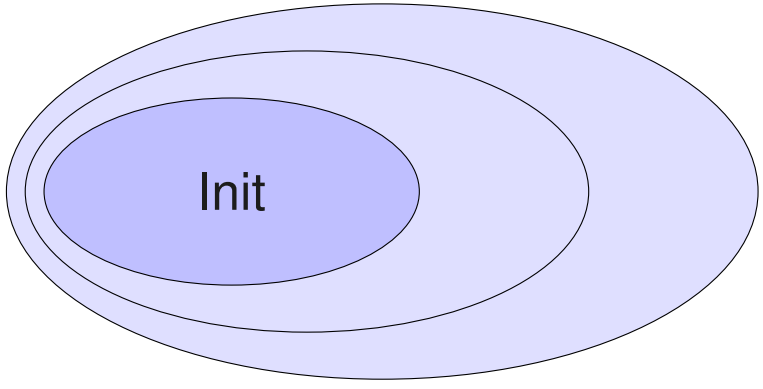
Model-checking: basics

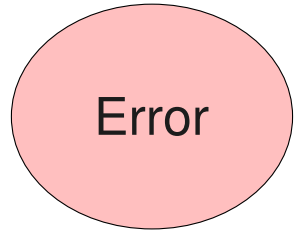




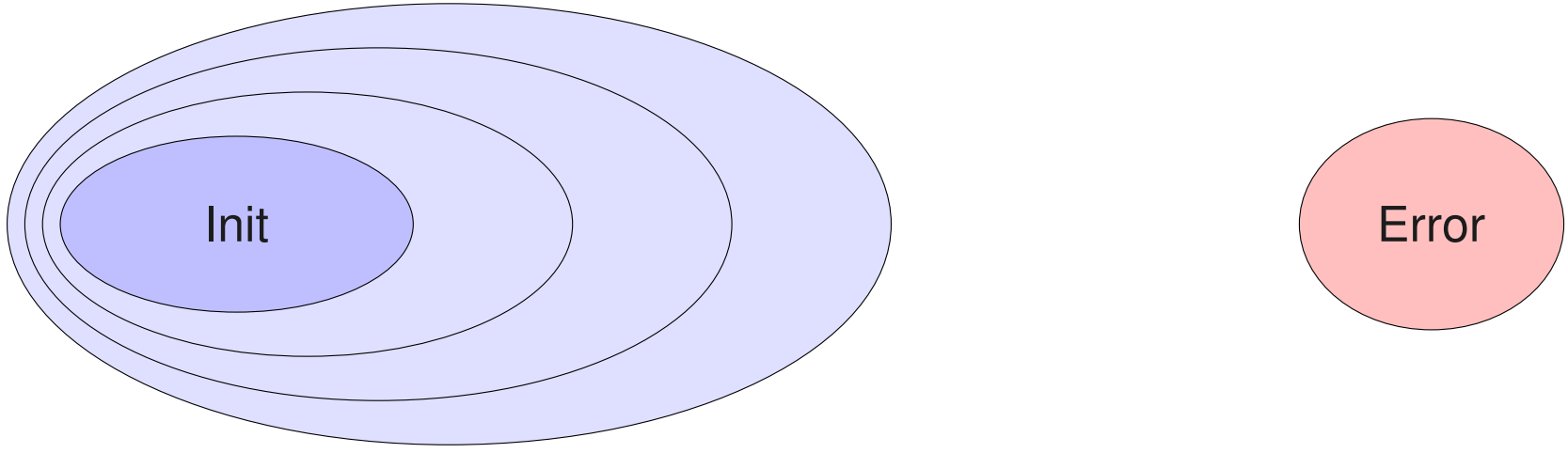
Model-checking: basics

Model-checking: basics

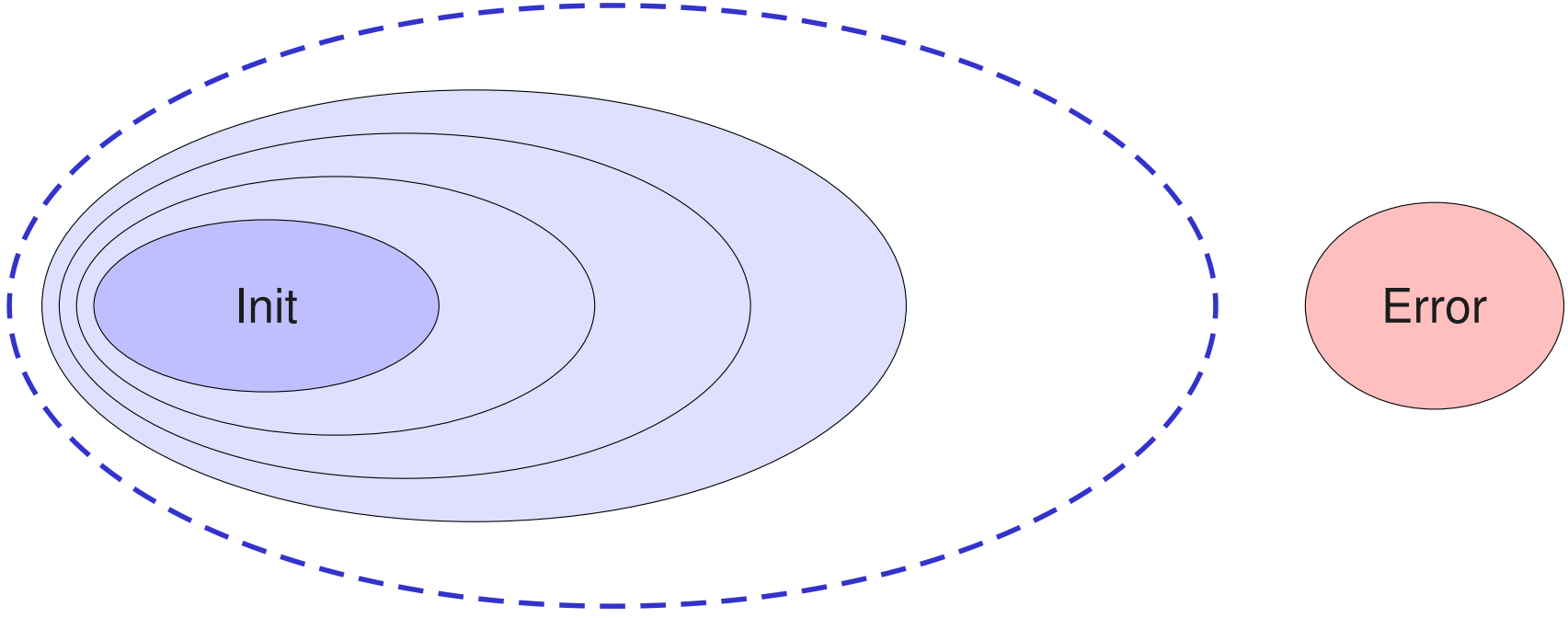




Model-checking: basics

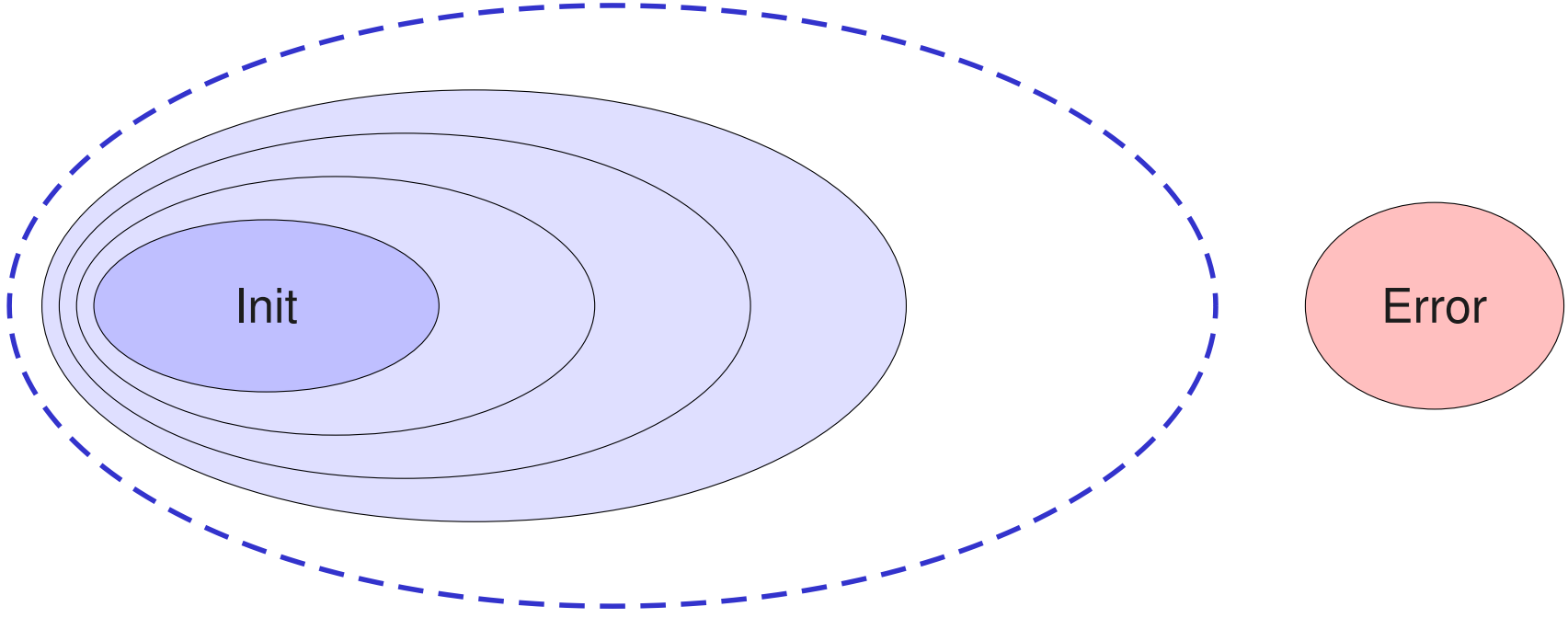


Model-checking: basics



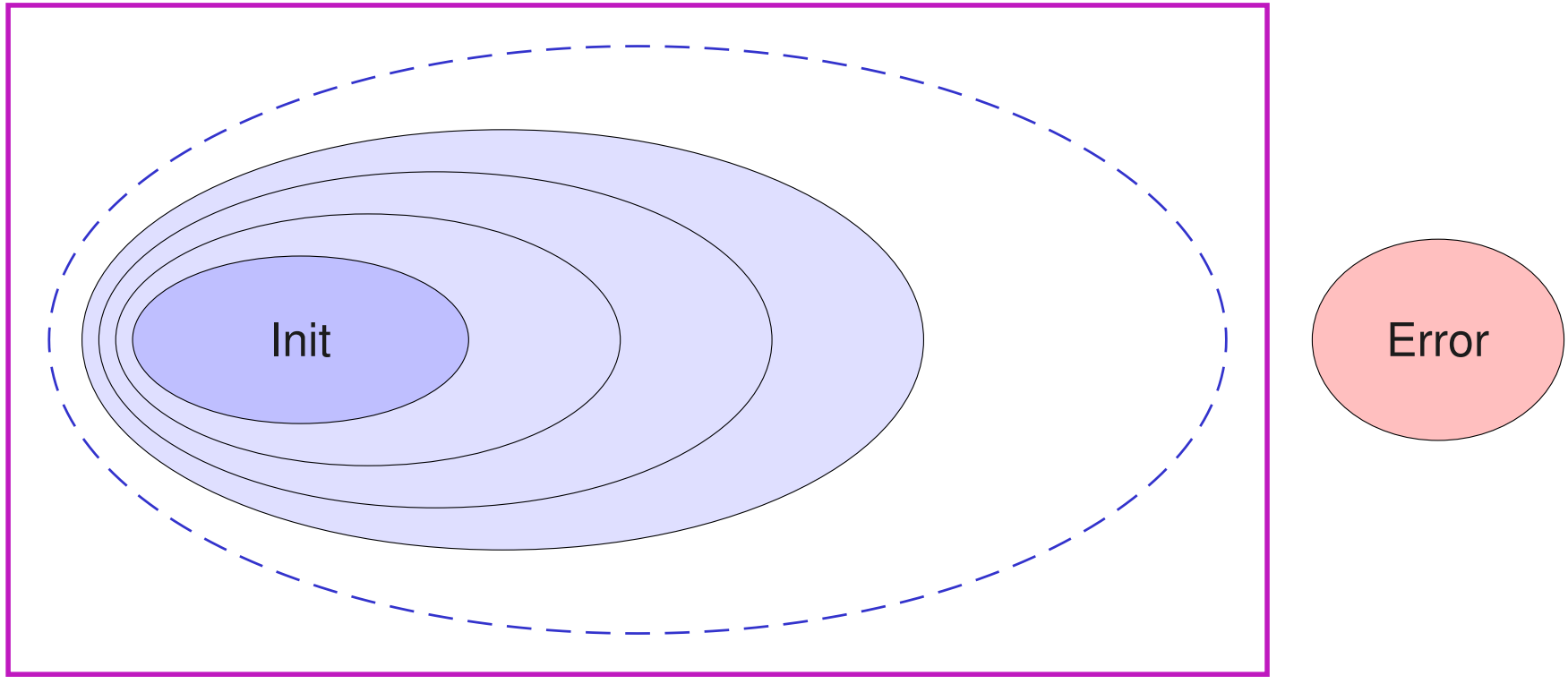
- Iteration doesn't terminate (too many states)

Model-checking: basics



- Iteration doesn't terminate (too many states)
- Solutions:
  - accelerate: exact computation, but too much useless detail

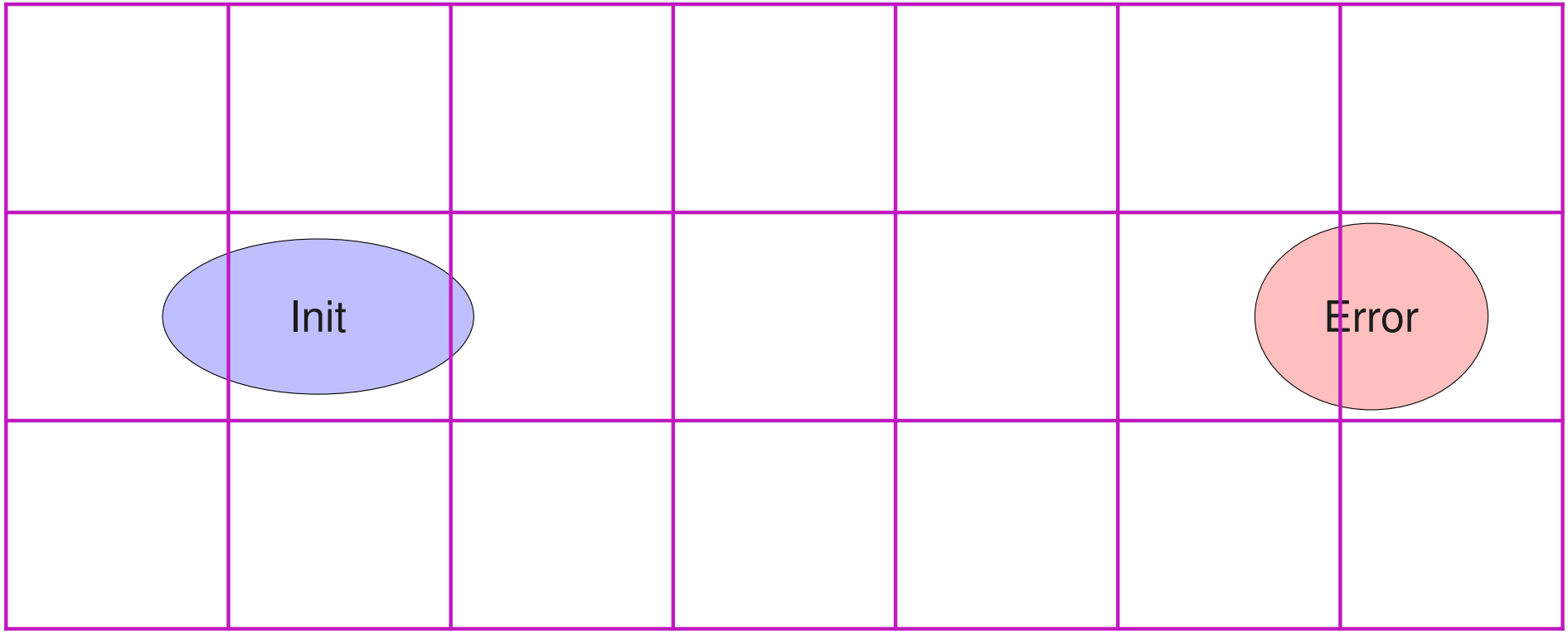
Model-checking: basics



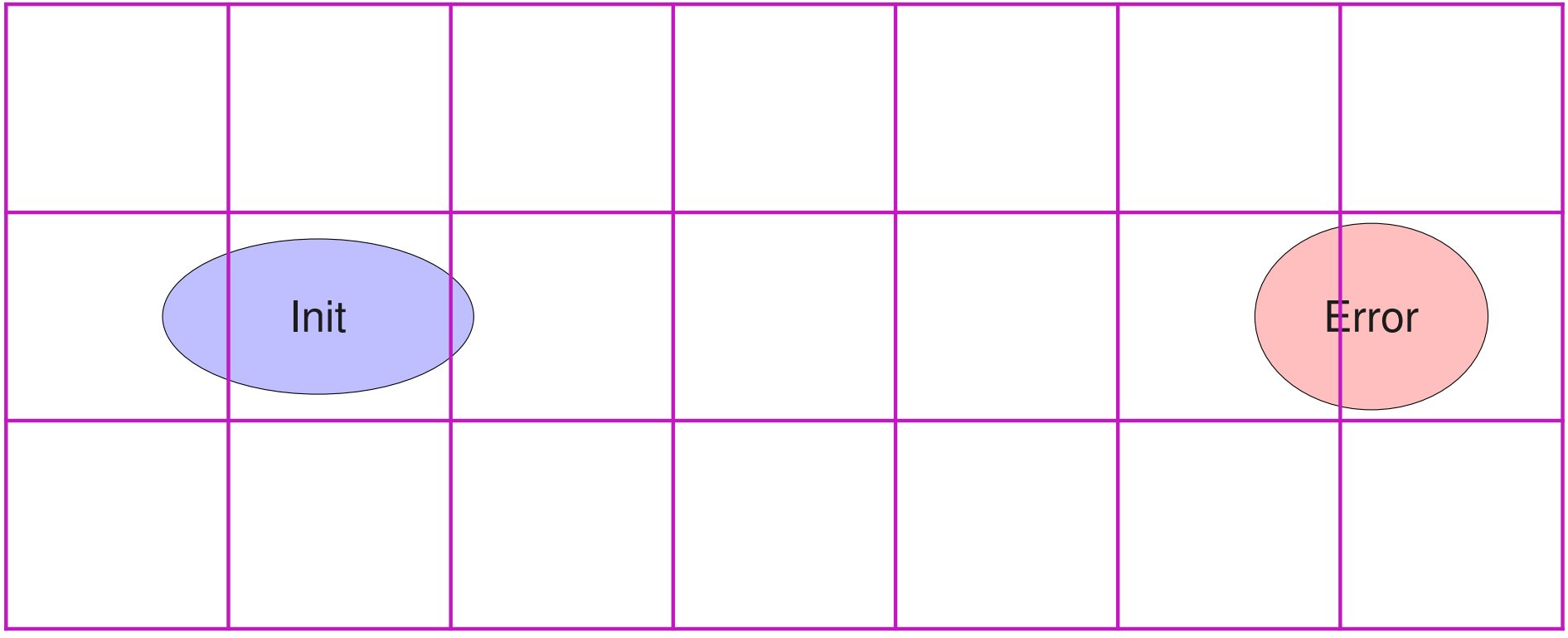
- Iteration doesn't terminate (too many states)
- Solutions:
  - accelerate: exact computation, but too much useless detail
  - abstract: too get rid of this unnecessary detail

Predicate Abstraction [GS97]

Predicate Abstraction [GS97]

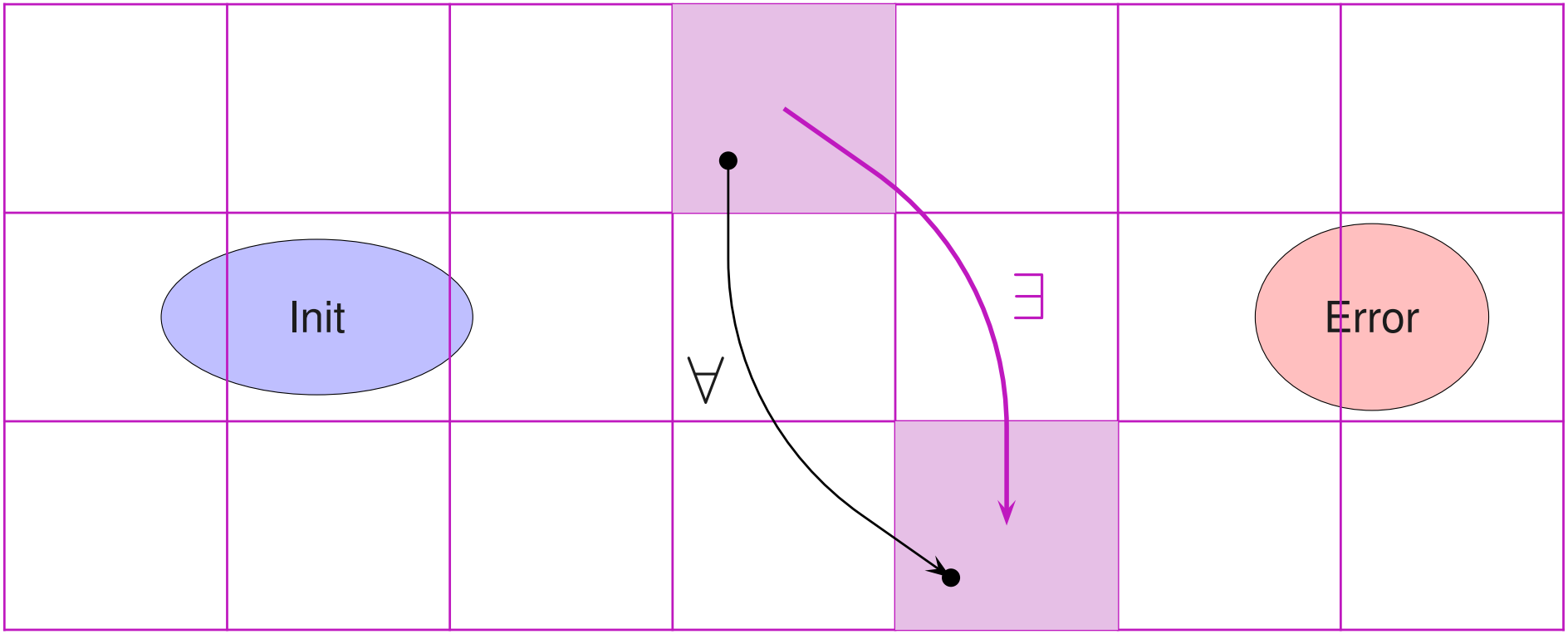


Predicate Abstraction [GS97]



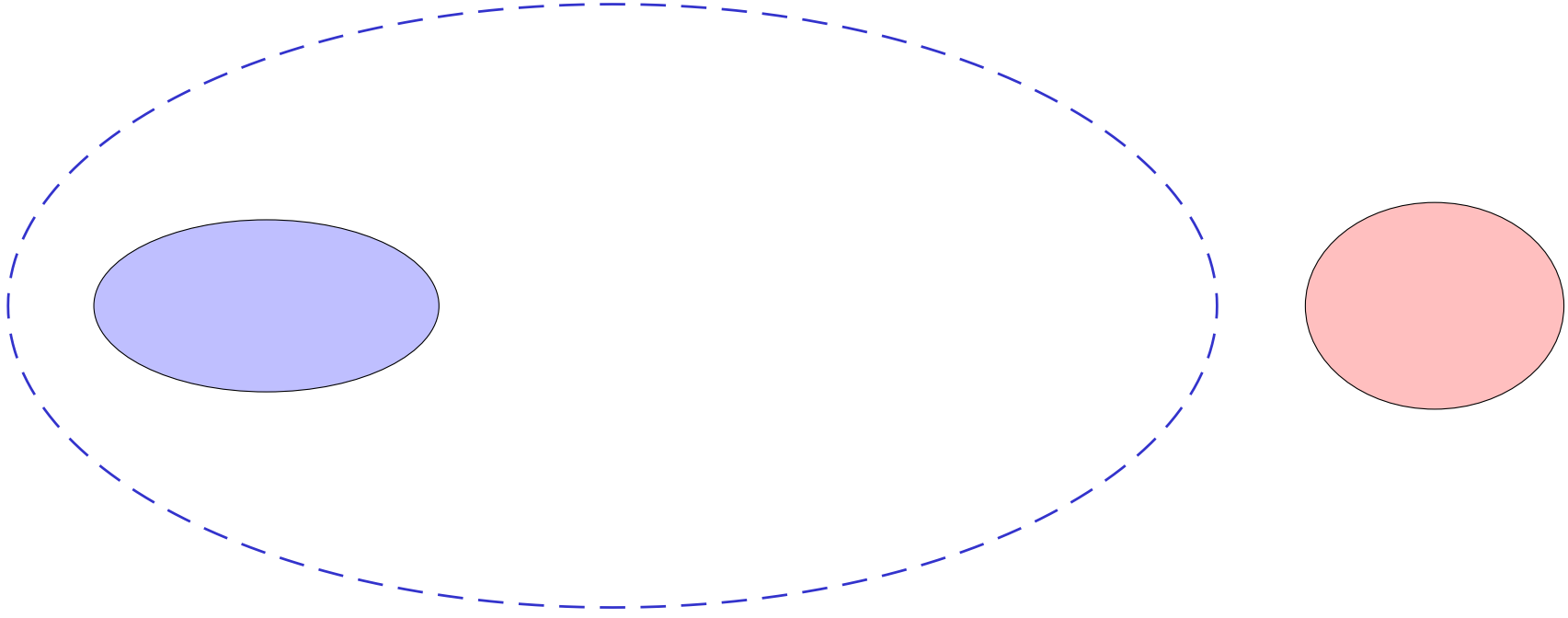
- $P_1, P_2, \ldots, P_n$: predicates ($[\![P_i]\!]$: subset of the state space)
- Abstract states (boxes) are valuations : $\{P_1, P_2, \ldots, P_n\} \rightarrow \{\texttt{true}, \texttt{false}\}$

Predicate Abstraction [GS97]

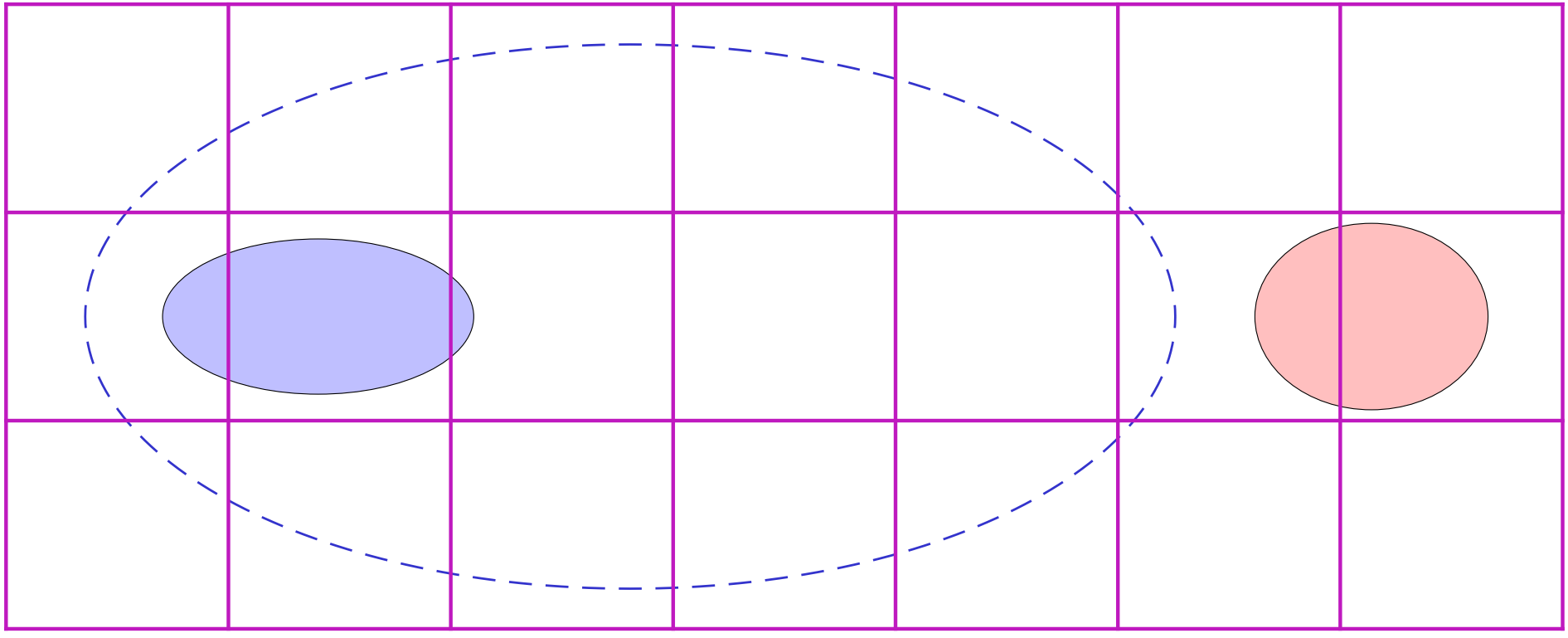


- $P_1, P_2, \ldots, P_n$: predicates ($[\![P_i]\!]$: subset of the state space)
- Abstract states (boxes) are valuations : $\{P_1, P_2, \ldots, P_n\} \rightarrow \{\texttt{true}, \texttt{false}\}$
- Conservative abstraction (computed using decision procedures)

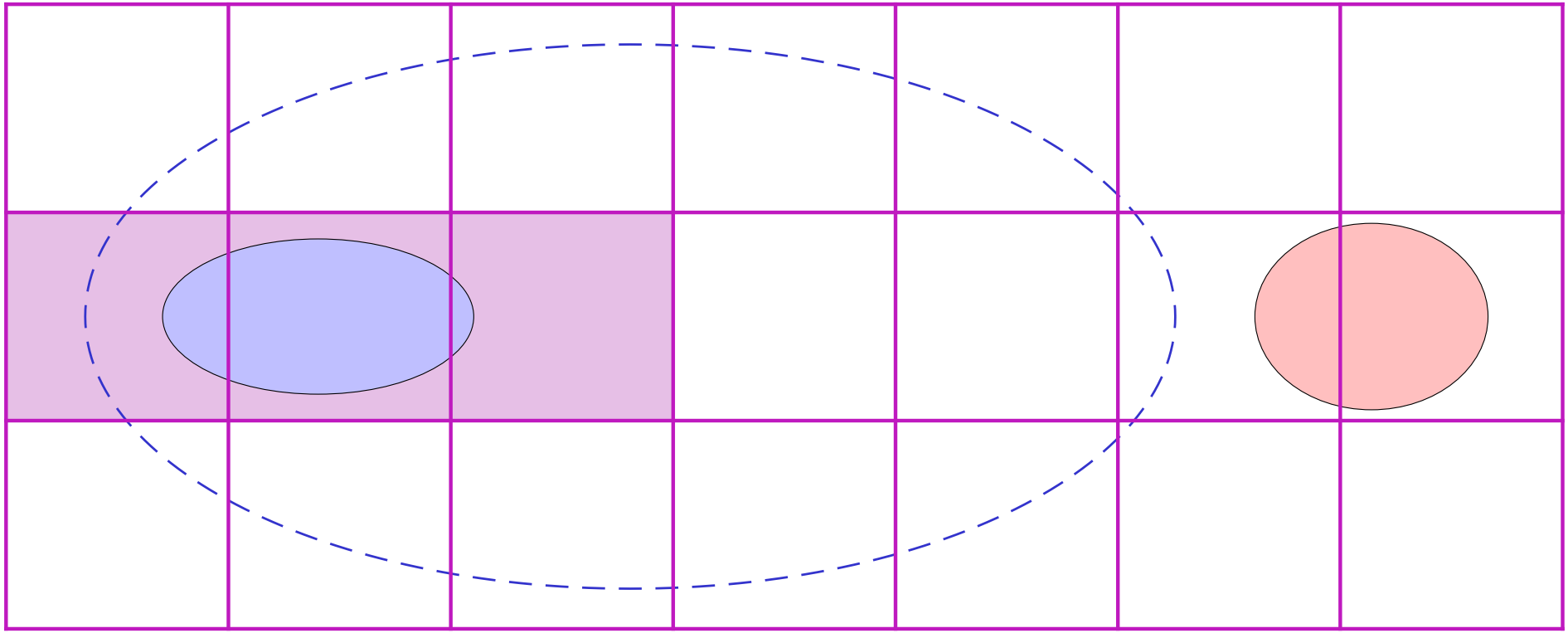
Model-Checking with Predicate Abstraction



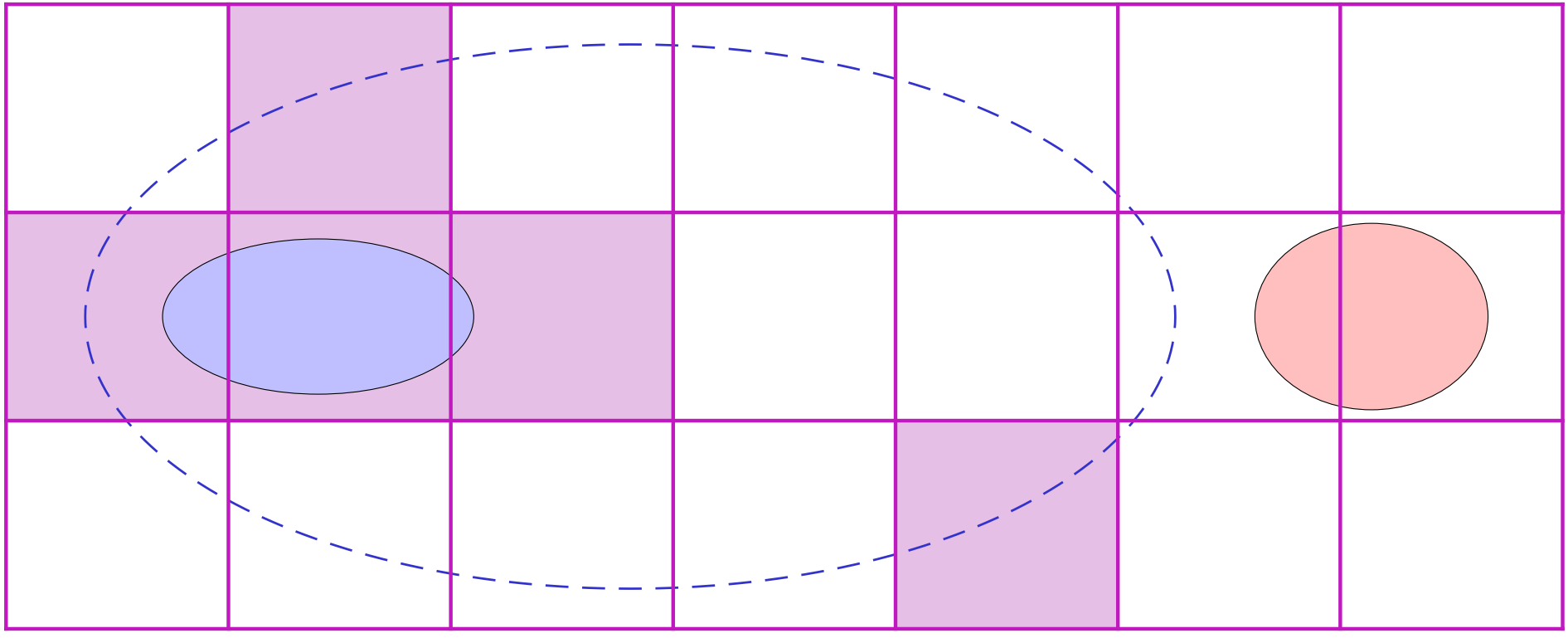
Model-Checking with Predicate Abstraction



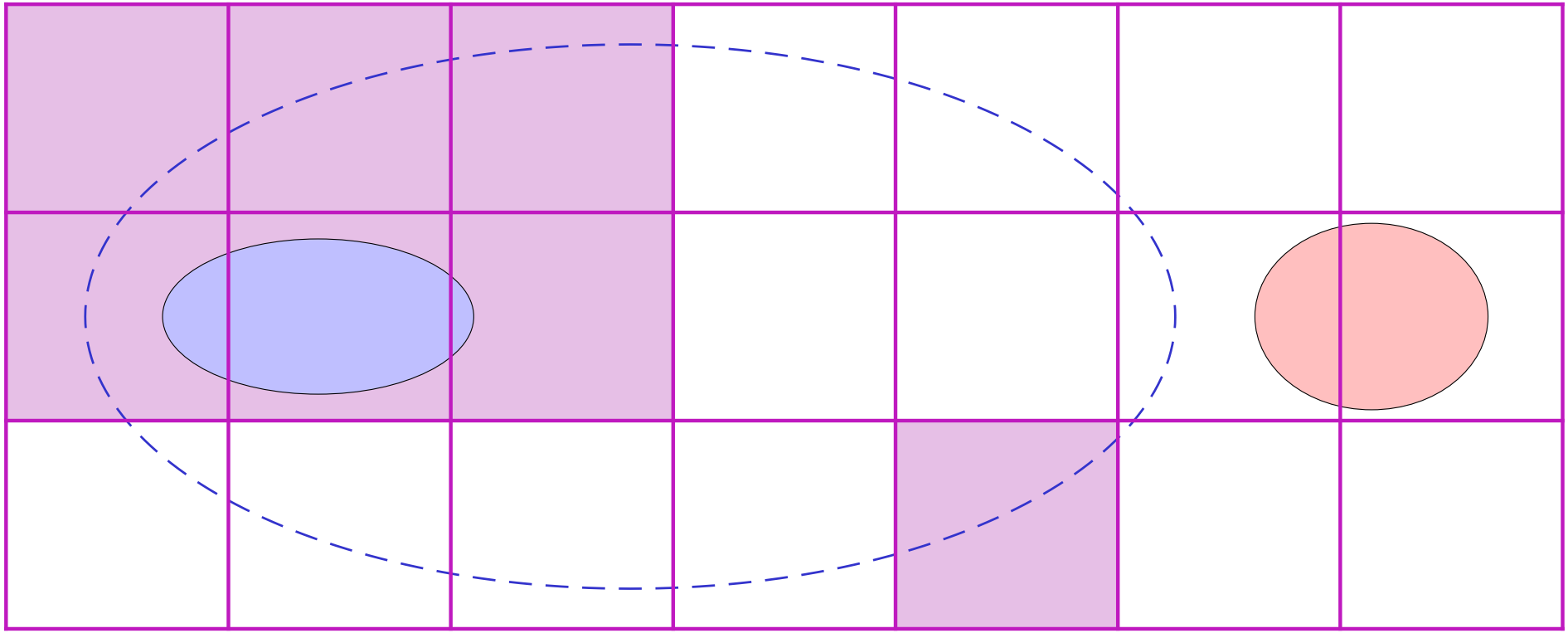
Model-Checking with Predicate Abstraction



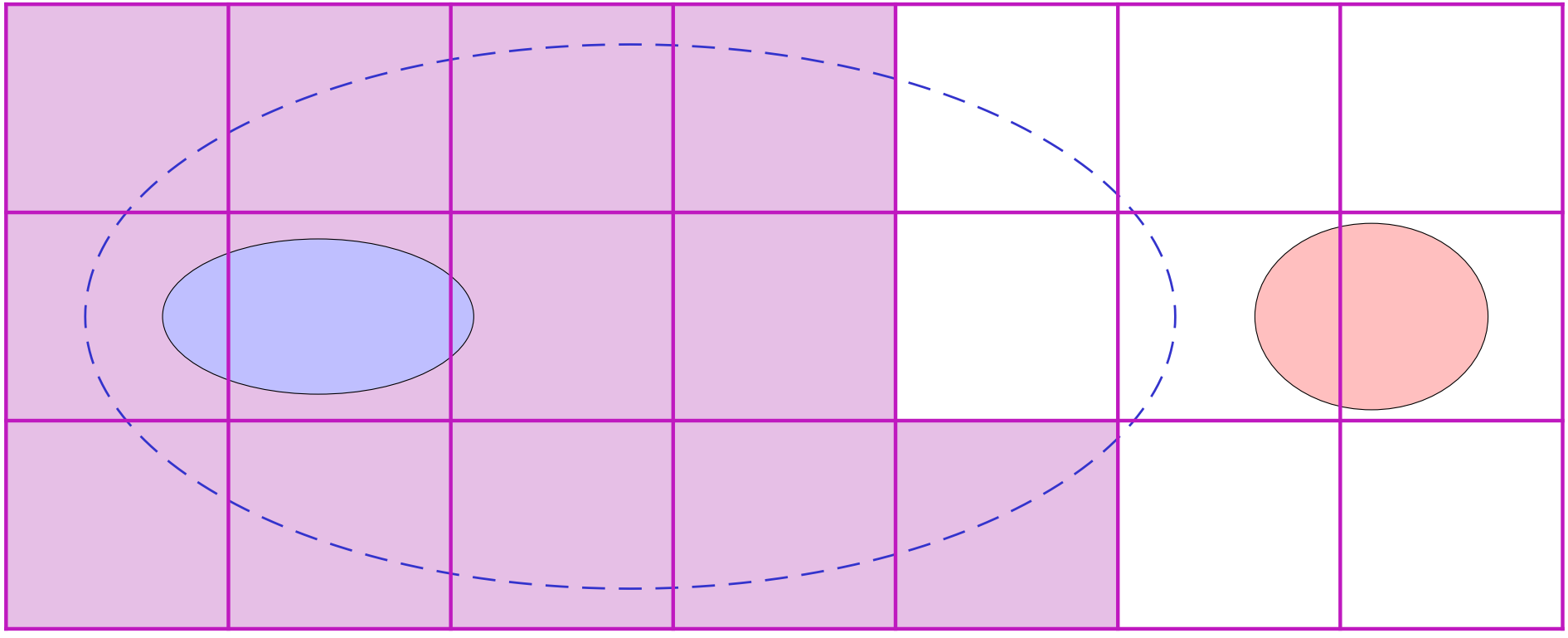
Model-Checking with Predicate Abstraction



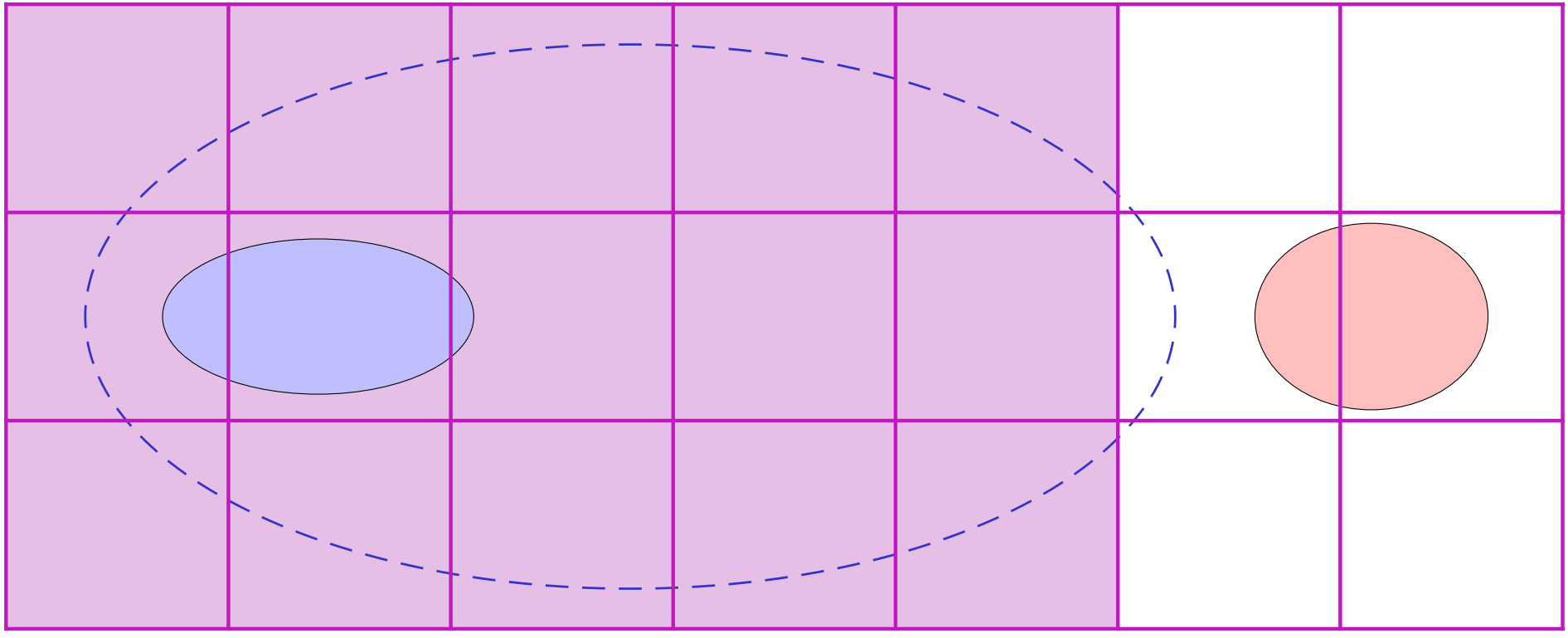
Model-Checking with Predicate Abstraction



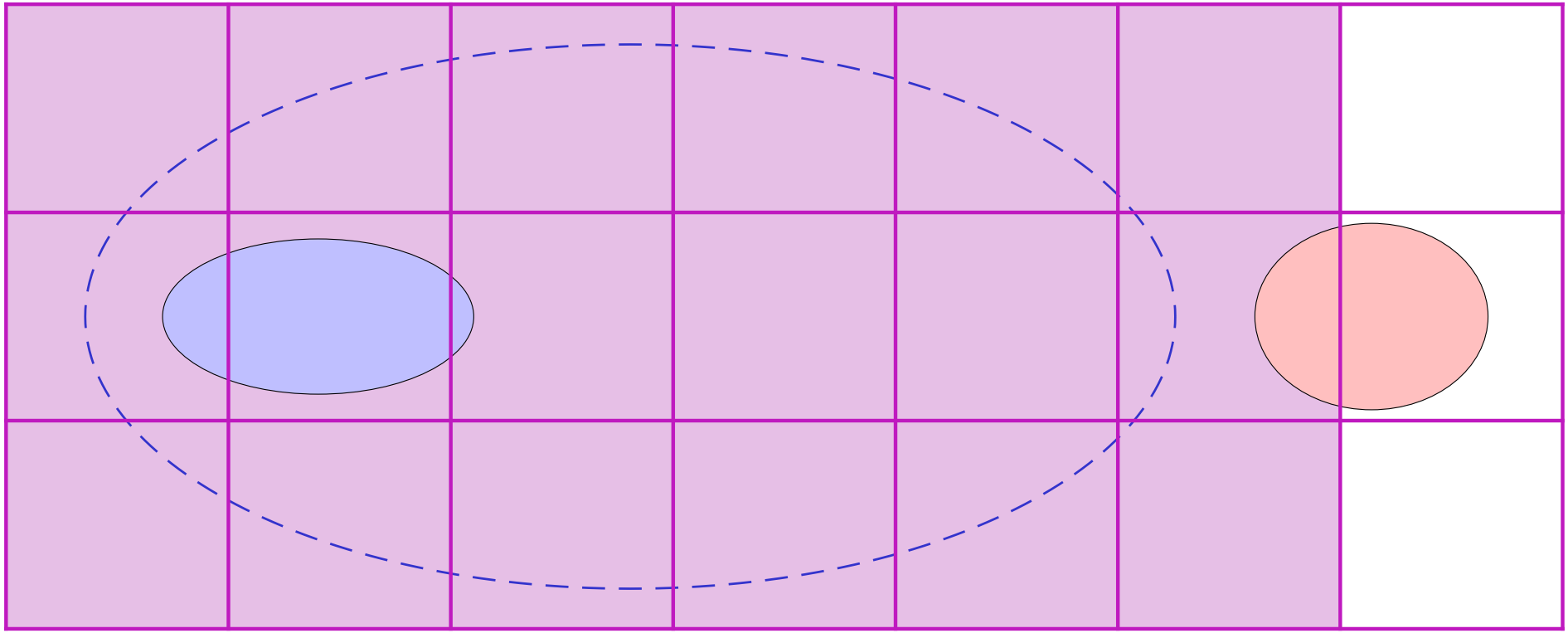
Model-Checking with Predicate Abstraction



Model-Checking with Predicate Abstraction

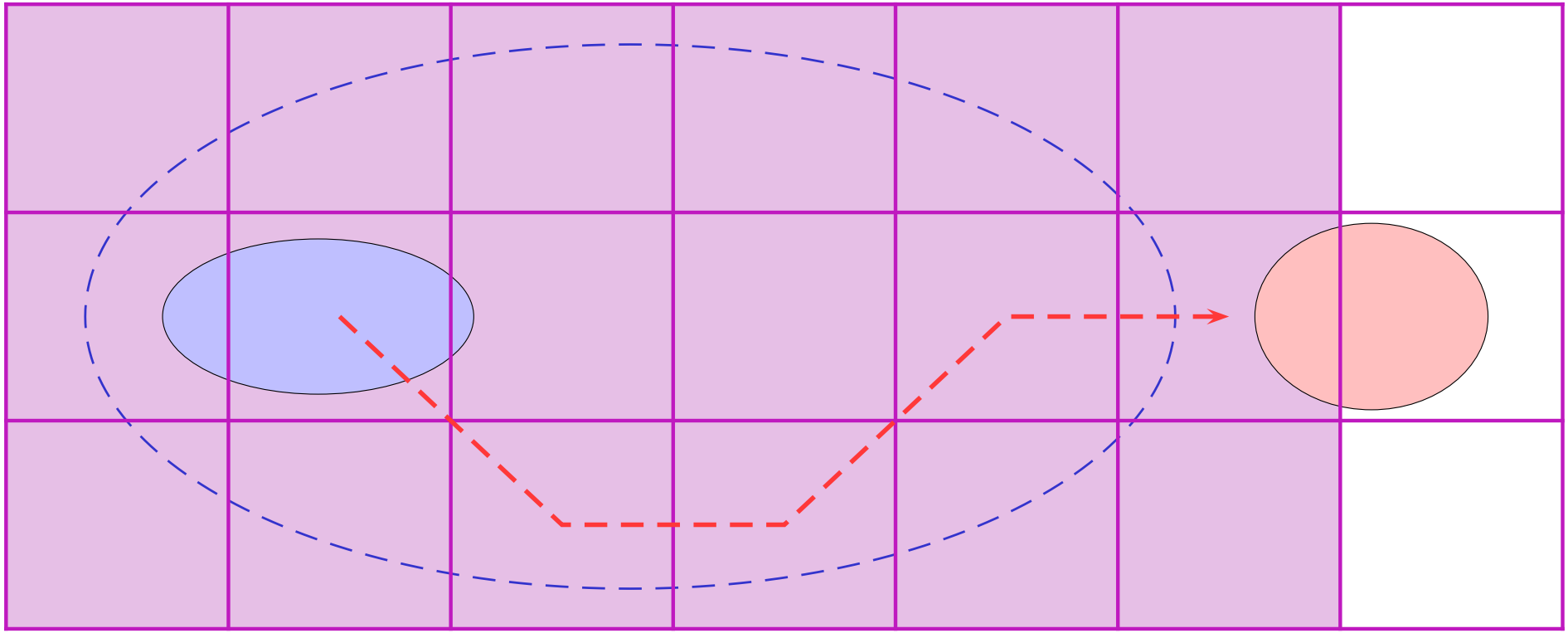


Model-Checking with Predicate Abstraction



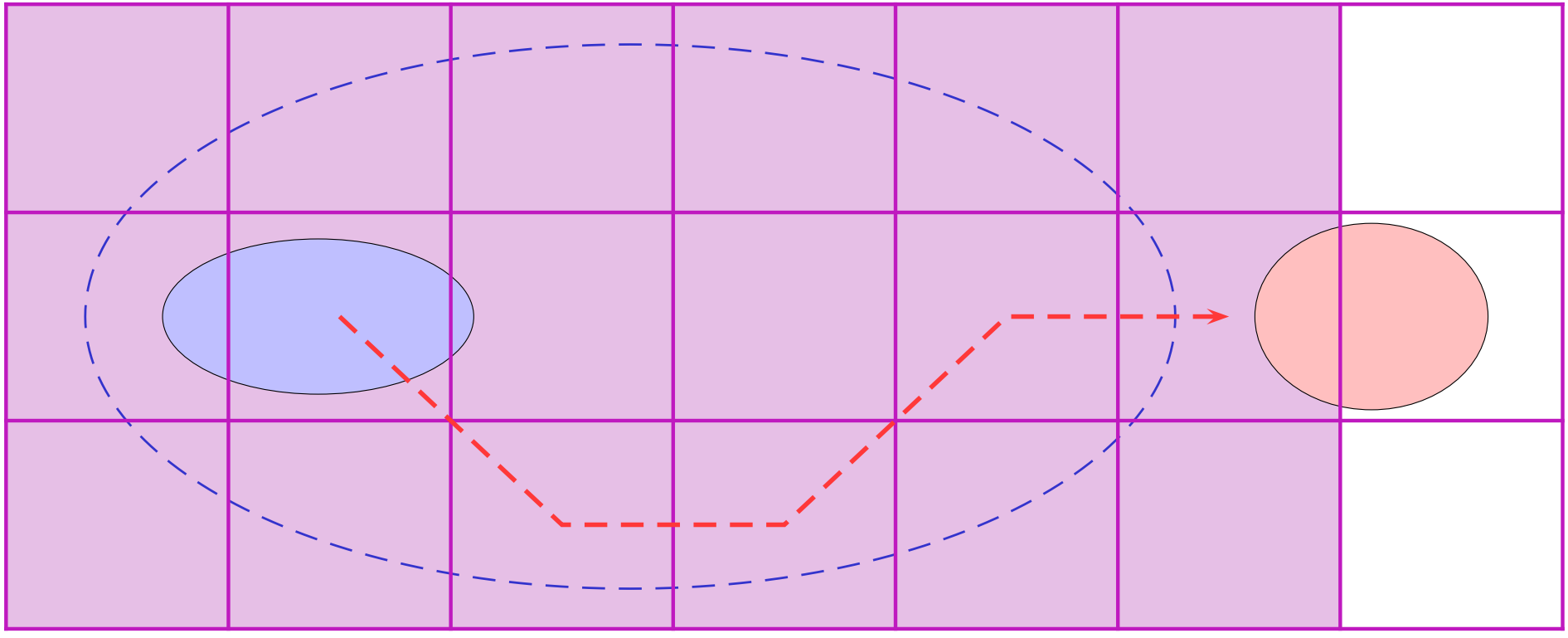
- Abstraction too coarse

Model-Checking with Predicate Abstraction



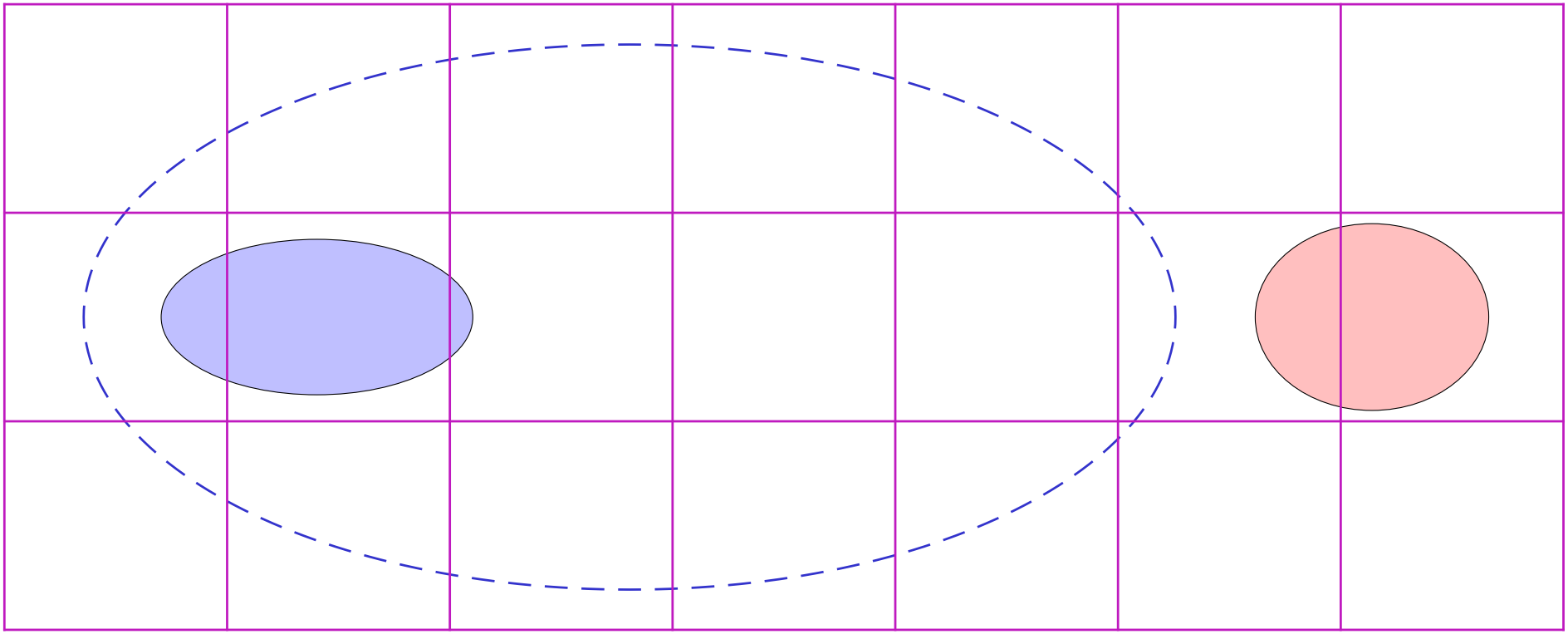
- Abstraction too coarse

Model-Checking with Predicate Abstraction

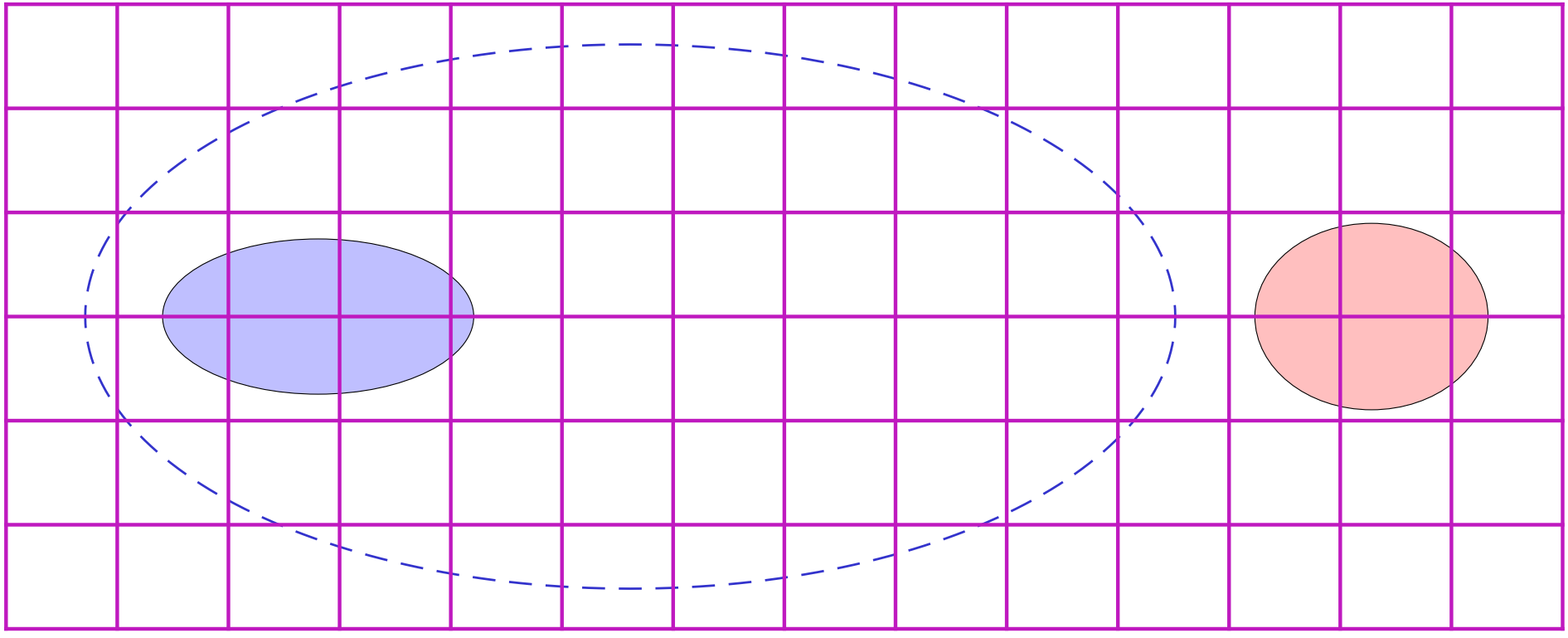


- Abstraction too coarse
- Refinement is needed to avoid this spurious error trace
  - split boxes into smaller ones: add new predicates

Refinement with Predicate Abstraction

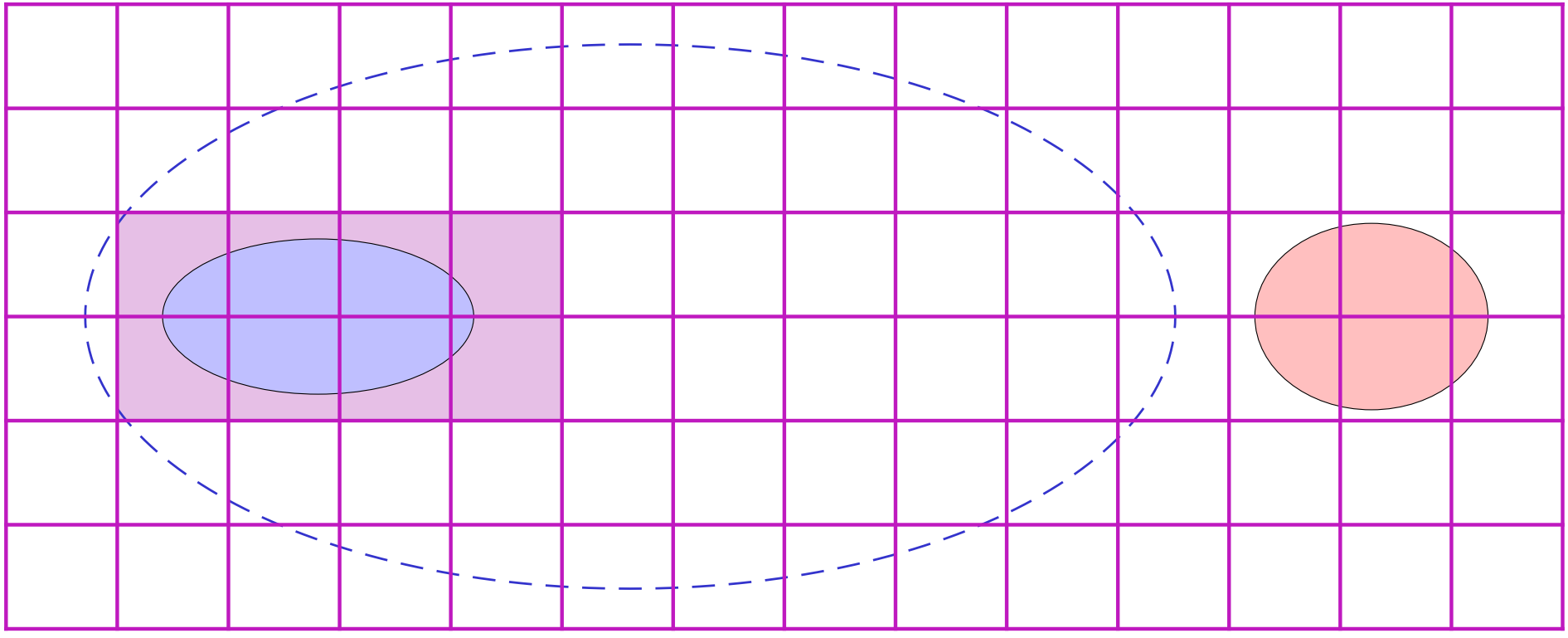


Refinement with Predicate Abstraction



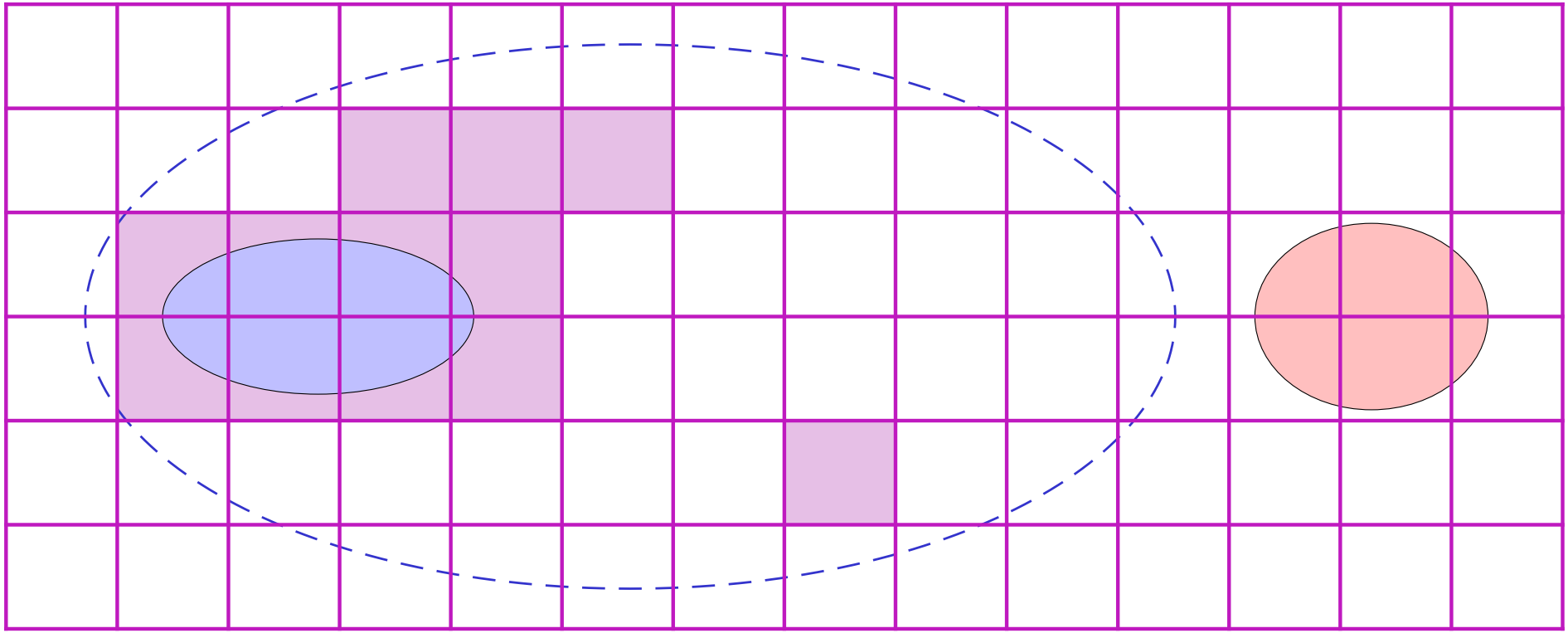
- Refinement: adding new predicates
- Finer grid

Refinement with Predicate Abstraction



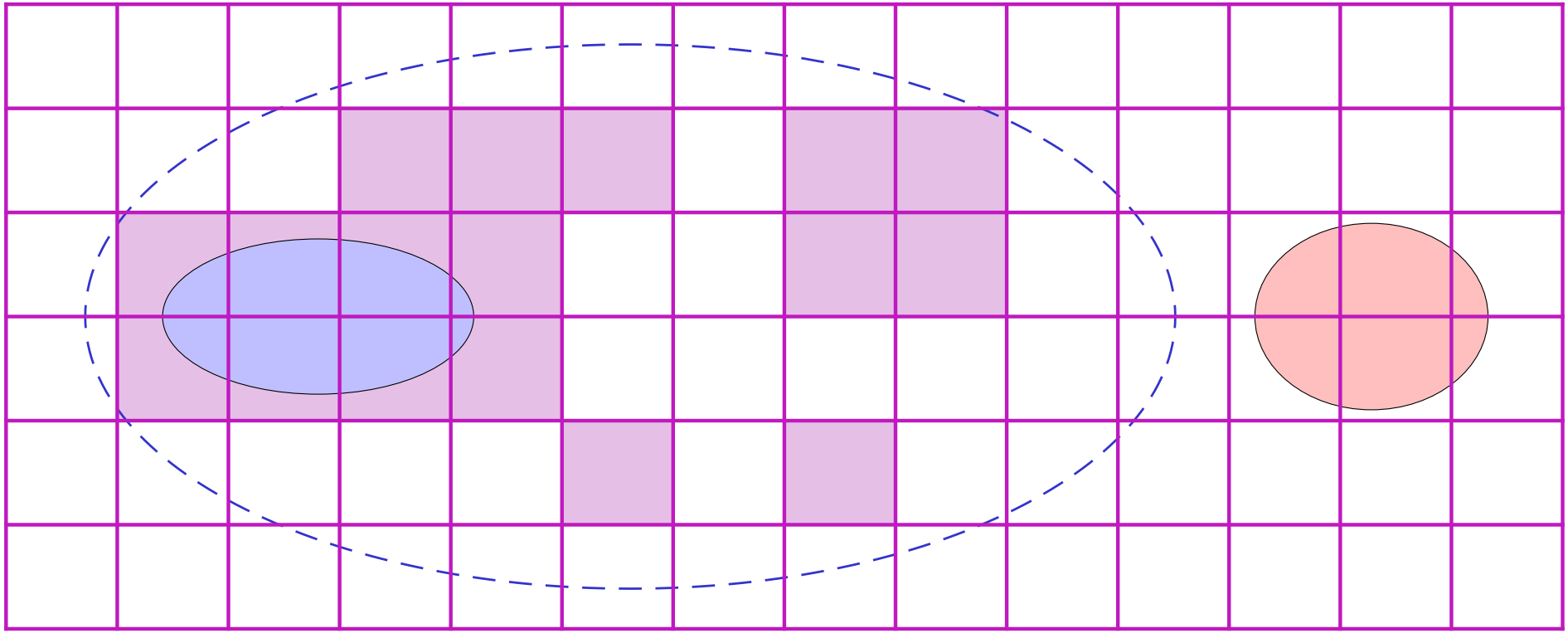
- Refinement: adding new predicates
- Finer grid

Refinement with Predicate Abstraction



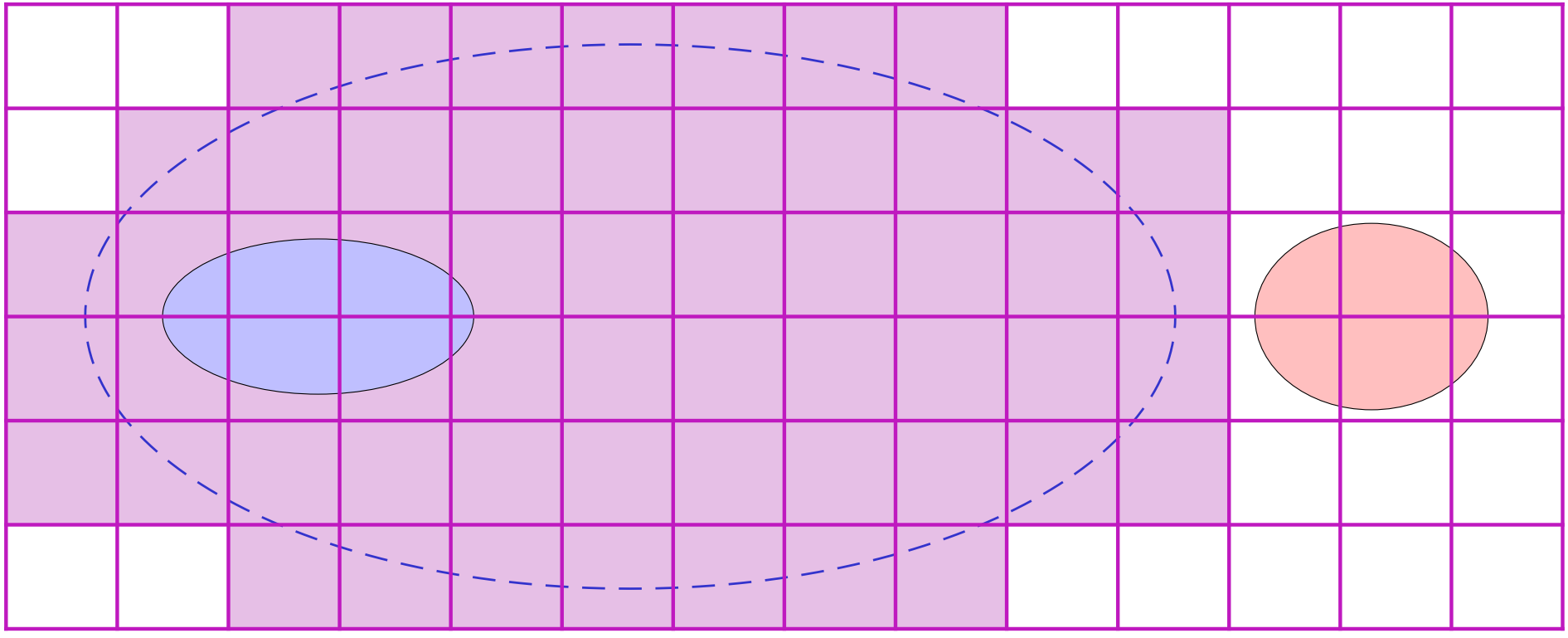
- Refinement: adding new predicates
- Finer grid

Refinement with Predicate Abstraction



- Refinement: adding new predicates
- Finer grid

Refinement with Predicate Abstraction



- Refinement: adding new predicates
- Finer grid
- The refined abstraction is safe!

Abstract-Check-Refine Loop

- Three phases integrated in a completely automatic loop
  - No false negative

Abstract-Check-Refine Loop

- Three phases integrated in a completely automatic loop
  - No false negative
- Automatic abstract counter-example checking
  - for safety properties, reduces to symbolic simulation

Abstract-Check-Refine Loop

- Three phases integrated in a completely automatic loop
  - No false negative
- Automatic abstract counter-example checking
  - for safety properties, reduces to symbolic simulation
- Automatic refinement
  - New predicates automatically inferred from spurious error traces

Abstract-Check-Refine Loop

- Three phases integrated in a completely automatic loop
  - No false negative
- Automatic abstract counter-example checking
  - for safety properties, reduces to symbolic simulation
- Automatic refinement
  - New predicates automatically inferred from spurious error traces
- Previous work: batch-oriented abstract-check-refine loop
  - Clarke et al.: ACTL^* but for finite-state systems [CGJ+00]
  - MSR SLAM Project safety properties for systems code [BR01]

Abstract-Check-Refine Loop (cont'd)

Input: System (program or model) S and seed predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$

Abstract-Check-Refine Loop (cont'd)

Input: System (program or model) S and seed predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$

1. compute the finite predicate abstraction A of S w.r.t. predicates $\mathcal{P}$

Abstract-Check-Refine Loop (cont'd)

Input: System (program or model) S and seed predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$

1. compute the finite predicate abstraction A of S w.r.t. predicates $\mathcal{P}$
2. model-check A
   - **if** A is safe **then return** S is safe
   - **otherwise** extract an (abstract) error path π

Abstract-Check-Refine Loop (cont'd)

Input: System (program or model) S and seed predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$

1. compute the finite predicate abstraction A of S w.r.t. predicates $\mathcal{P}$
2. model-check A
   - **if** A is safe **then return** S is safe
   - **otherwise** extract an (abstract) error path π
3. check whether π is feasible path in S
   - **if** don't know **then return** don't know
   - **if** π is feasible in S **then return** S is unsafe

Abstract-Check-Refine Loop (cont'd)

Input: System (program or model) S and seed predicates $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$

1. compute the finite predicate abstraction A of S w.r.t. predicates $\mathcal{P}$
2. model-check A
   - **if** A is safe **then return** S is safe
   - **otherwise** extract an (abstract) error path π
3. check whether π is feasible path in S
   - **if** don't know **then return** don't know
   - **if** π is feasible in S **then return** S is unsafe
4. refinement
   - extract an "explanation" of unfeasibility of π as new predicates $P'_1, P'_2, \ldots, P'_k$
   - $\mathcal{P} := \mathcal{P} \cup \{P'_1, P'_2, \ldots, P'_k\}$
5. **goto** 1.

Drawbacks

- Batch-oriented integration
  - no sharing of data structures

Drawbacks

- Batch-oriented integration
  - no sharing of data structures
- Abstraction computed completely before the model-checking phase
  - does unnecessary expensive work

Drawbacks

- Batch-oriented integration
  - no sharing of data structures
- Abstraction computed completely before the model-checking phase
  - does unnecessary expensive work
- Computations from earlier loop iterations are wasted
  - redoes work at each iteration
- What if the new inferred predicates are used locally only?
  - error-free parts of the state may have already been explored

Drawbacks

- Batch-oriented integration
  - no sharing of data structures
- Abstraction computed completely before the model-checking phase
  - does unnecessary expensive work
- Computations from earlier loop iterations are wasted
  - redoes work at each iteration
- What if the new inferred predicates are used locally only?
  - error-free parts of the state may have already been explored

→ Need a lazy approach

Outline

1. Introduction
2. Predicate Abstraction
3. **Lazy Abstraction**
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

Our proposal

- Abstract only where required
  - reachable state space is very sparse

  → construct the abstraction on the fly

Our proposal

- Abstract only where required
  - reachable state space is very sparse

  → construct the abstraction on the fly

- Use greater precision only where required
  - non uniform abstraction

  → refine locally, based on false error paths

Our proposal

- Abstract only where required
  - reachable state space is very sparse

  → construct the abstraction on the fly

- Use greater precision only where required
  - non uniform abstraction

  → refine locally, based on false error paths

- Re-use work from earlier phases / iterations

  → tight integration of the 3 phases

Our proposal

- Abstract only where required
  - reachable state space is very sparse

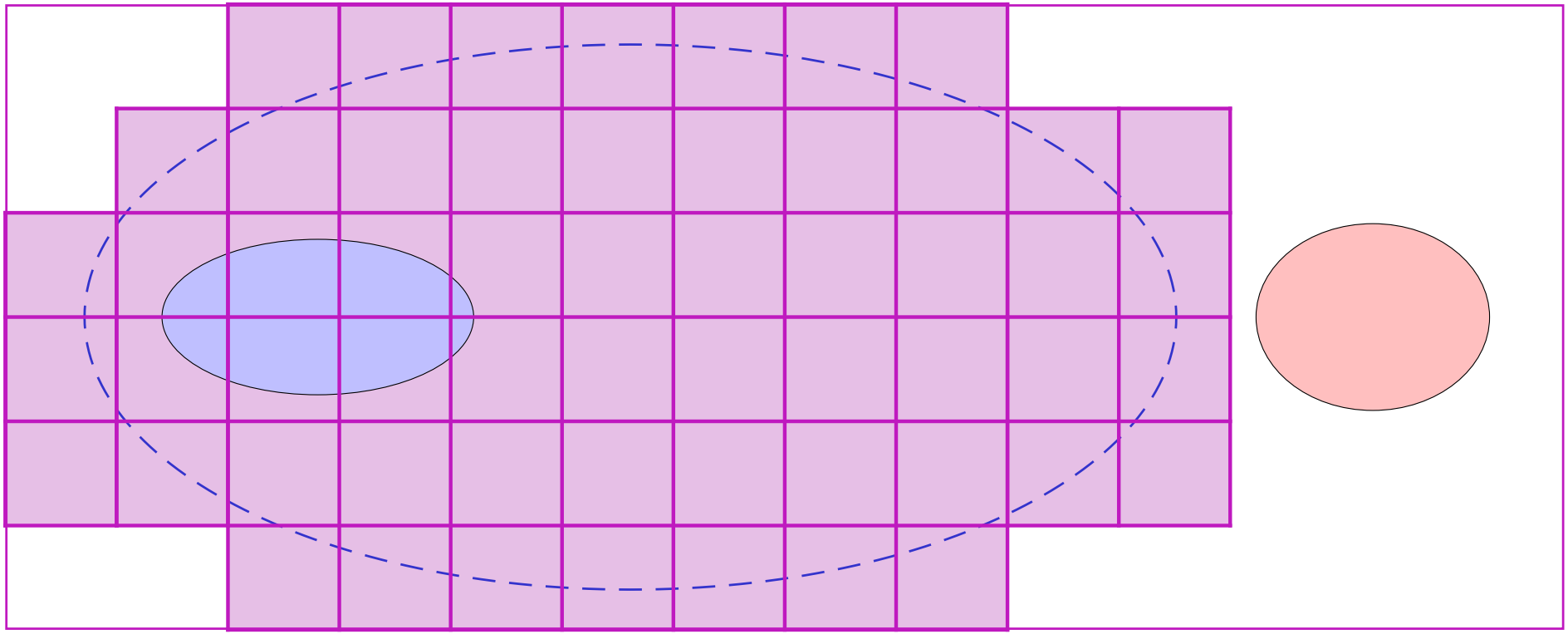
  → construct the abstraction on the fly
- Use greater precision only where required
  - non uniform abstraction

  → refine locally, based on false error paths
- Re-use work from earlier phases / iterations

  → tight integration of the 3 phases

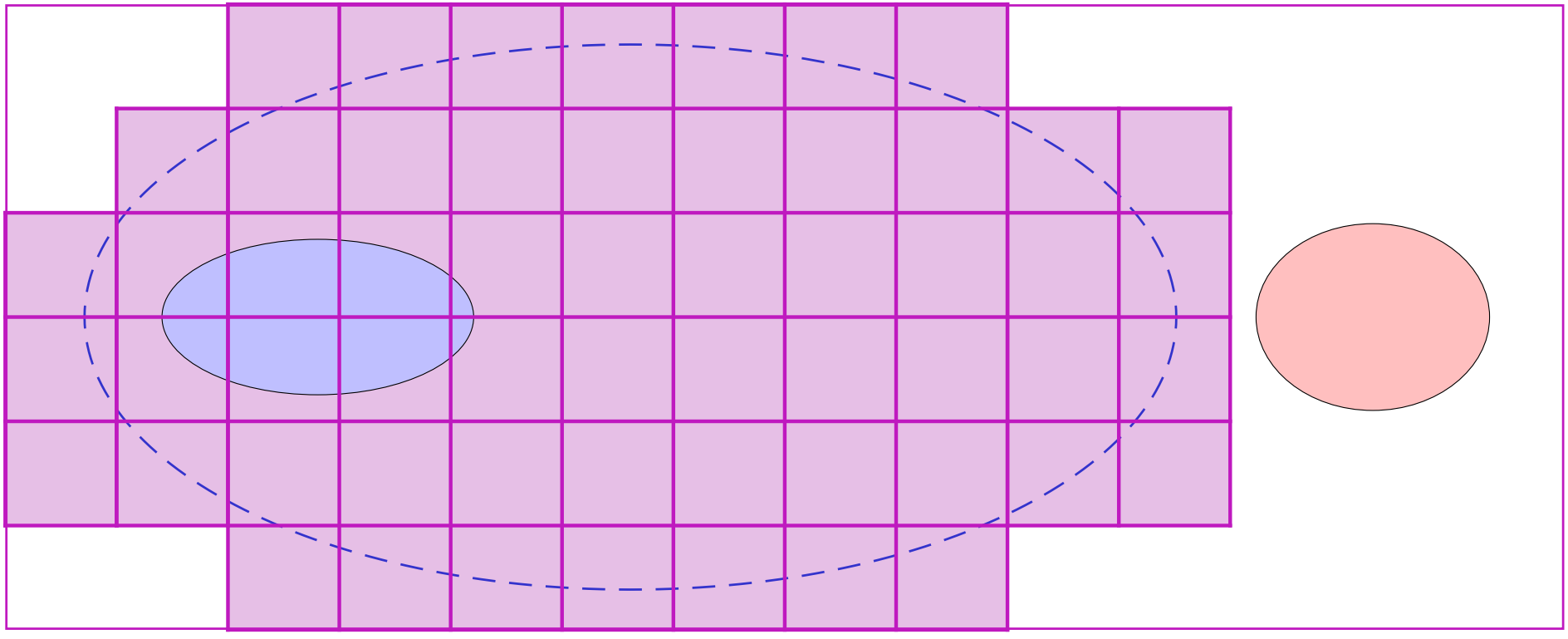
→ Give control to the model-checker

Abstract only where required



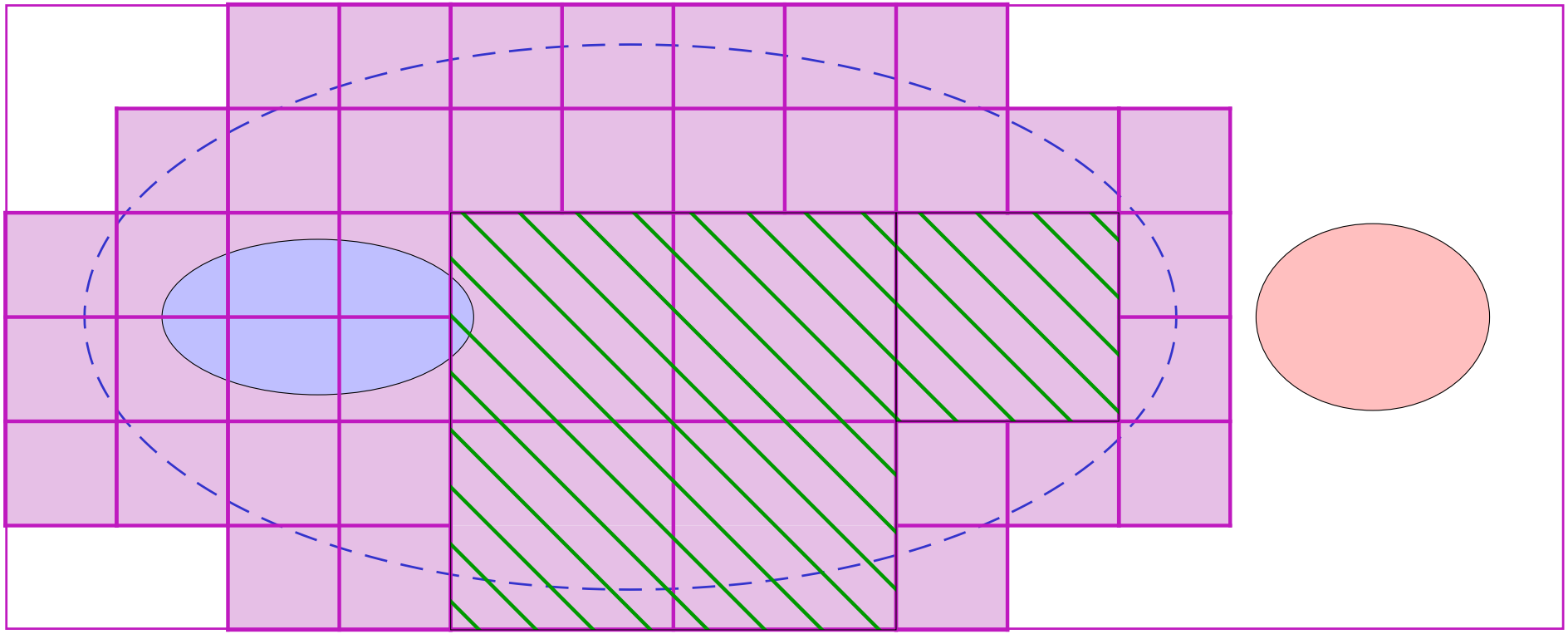
- Computation of the abstract transition relation is very expensive
- Why compute abstract transitions that are never taken?

Abstract only where required



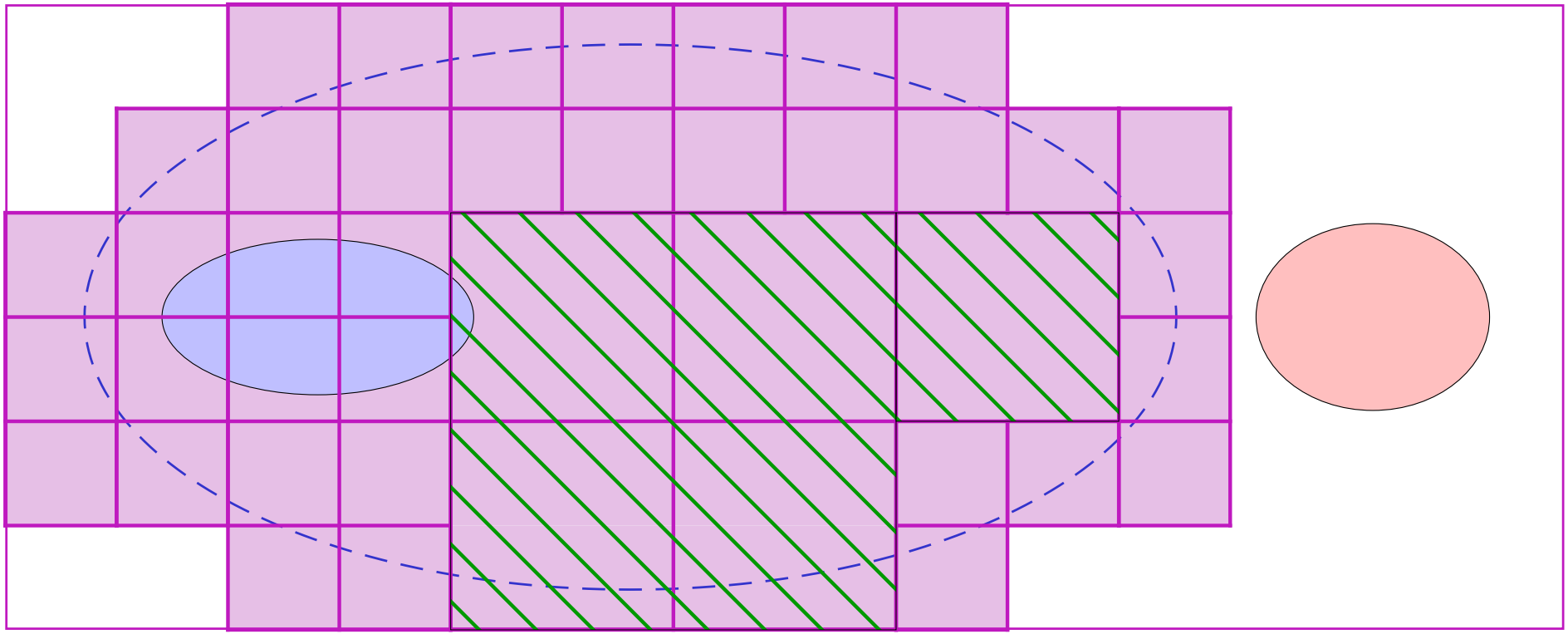
- Computation of the abstract transition relation is very expensive
- Why compute abstract transitions that are never taken?
- On-the-fly abstraction: driven by the search

Refine only where required



- Why be precise everywhere?
- Don’t refine error-free parts of the state space

Refine only where required



- Why be precise everywhere?
- Don’t refine error-free parts of the state space
- Local refinement: driven by the search

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$
- empty region $\bot$ ($[\![\bot]\!] = \emptyset$)

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$
- empty region $\bot$ ($[\![\bot]\!] = \emptyset$)
- union and intersection of regions: $[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!]$, $[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!]$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$
- empty region $\bot$ ($[\![\bot]\!] = \emptyset$)
- union and intersection of regions: $[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!]$, $[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!]$
- concrete pre : $[\![pre(r, l)]\!] = \{s' \in S \mid \exists\, s \in [\![r]\!].\, s' \xrightarrow{l} s\}$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$
- empty region $\bot$ ($[\![\bot]\!] = \emptyset$)
- union and intersection of regions: $[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!]$, $[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!]$
- concrete pre : $[\![pre(r, l)]\!] = \{s' \in S \mid \exists\, s \in [\![r]\!].\, s' \xrightarrow{l} s\}$
- abstract $\widehat{post}$: $[\![\widehat{post}(r, l)]\!] \supseteq \{s' \in S \mid \exists\, s \in [\![r]\!].\, s \xrightarrow{l} s'\}$

Abstraction Structure

- Labelled Transition System: $(S, \Sigma, \rightarrow)$
- Abstraction structure: $(R, \bot, \sqcup, \sqcap, pre, \widehat{post}, [\![\cdot]\!], \trianglelefteq)$
- set R of regions, meaning given by $[\![\cdot]\!] : R \rightarrow S$
- induced inclusion pre-order $\sqsubseteq$ over regions: $r \sqsubseteq r'$ iff $[\![r]\!] \subseteq [\![r']\!]$
- empty region $\bot$ ($[\![\bot]\!] = \emptyset$)
- union and intersection of regions: $[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!]$, $[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!]$
- concrete pre : $[\![pre(r, l)]\!] = \{s' \in S \mid \exists\, s \in [\![r]\!].\, s' \xrightarrow{l} s\}$
- abstract $\widehat{post}$: $[\![\widehat{post}(r, l)]\!] \supseteq \{s' \in S \mid \exists\, s \in [\![r]\!].\, s \xrightarrow{l} s'\}$
- precision pre-order $\trianglelefteq$ over regions: $\widehat{post}$ monotonic w.r.t. $\trianglelefteq \cap \sqsubseteq$
  - usually, $\widehat{post}$ is neither monotonic w.r.t. $\trianglelefteq$, nor monotonic w.r.t. $\sqsubseteq$

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre
- regions: pairs (φ, Γ), where:
  - $\Gamma \subseteq \mathcal{L}$ is a set of support predicates
  - φ boolean formula over predicates in Γ
  - straightfoward meaning $[\![\cdot]\!]$, and empty region: $(\texttt{false}, \emptyset)$

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre
- regions: pairs (φ, Γ), where:
  - $\Gamma \subseteq \mathcal{L}$ is a set of support predicates
  - φ boolean formula over predicates in Γ
  - straightfoward meaning $[\![\cdot]\!]$, and empty region: $(\texttt{false}, \emptyset)$
- $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma' \cup \Gamma)$ and $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma' \cup \Gamma)$

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre
- regions: pairs (φ, Γ), where:
  - $\Gamma \subseteq \mathcal{L}$ is a set of support predicates
  - φ boolean formula over predicates in Γ
  - straightfoward meaning $[\![\cdot]\!]$, and empty region: $(\texttt{false}, \emptyset)$
- $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma' \cup \Gamma)$ and $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma' \cup \Gamma)$
- pre : follows from closure assumption
  - support predicates grow as needed

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre
- regions: pairs (φ, Γ), where:
  - $\Gamma \subseteq \mathcal{L}$ is a set of support predicates
  - φ boolean formula over predicates in Γ
  - straightfoward meaning $[\![\cdot]\!]$, and empty region: $(\texttt{false}, \emptyset)$
- $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma' \cup \Gamma)$ and $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma' \cup \Gamma)$
- pre : follows from closure assumption
  - support predicates grow as needed
- $\widehat{post}(\varphi, \Gamma)$: smallest (φ', Γ) containing $\{s' \in S \mid \exists\, s \in [\![\varphi, \Gamma]\!].\, s \xrightarrow{l} s'\}$

Example: Predicate Abstraction

- Predicate language $\mathcal{L}$
  - predicates interpreted over S
  - boolean closure of $\mathcal{L}$ is decidable and effectively closed under pre
- regions: pairs (φ, Γ), where:
  - $\Gamma \subseteq \mathcal{L}$ is a set of support predicates
  - φ boolean formula over predicates in Γ
  - straightfoward meaning $[\![\cdot]\!]$, and empty region: $(\texttt{false}, \emptyset)$
- $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma' \cup \Gamma)$ and $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma' \cup \Gamma)$
- pre : follows from closure assumption
  - support predicates grow as needed
- $\widehat{post}(\varphi, \Gamma)$: smallest (φ', Γ) containing $\{s' \in S \mid \exists\, s \in [\![\varphi, \Gamma]\!].\, s \xrightarrow{l} s'\}$
- precision pre-order is containment on support predicates

Lazy Symbolic Reachability Tree

create root $\mathtt{r}$ labeled with r_0

while there are unmarked nodes

pick an unmarked node $\mathtt{n}:r$

if $r \sqcap err \not\equiv \bot$

let $\mathtt{n}':r' \xrightarrow{\sigma} \mathtt{n}:r$ be the oldest ancestor of $\mathtt{n}$ with $pre(err,\sigma) \sqcap r' \not\equiv \bot$

if $\mathtt{n}'$ is the root **then return** the "error trace" σ

else `do_refinement()`

else if $r \sqsubseteq \bigsqcup \{u \mid \mathtt{m}:u \text{ is an uncovered marked node}\}$

mark $\mathtt{n}$ as *covered*

else for each label $l \in \Sigma$ **do**

$r' \leftarrow \widehat{post}(r,l)$

construct a son $\mathtt{n}':r'$ of $\mathtt{n}$ and label the arc $\mathtt{n} \xrightarrow{l} \mathtt{n}'$

mark $\mathtt{n}$ as *uncovered*

return the region $\bigsqcup \{u \mid \mathtt{m}:u \text{ is an uncovered marked node}\}$

Lazy Refinement

When entering `do_refinement()`, we have:

- $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ is the oldest ancestor of `n` with $pre(err, \sigma) \sqcap r' \not\equiv \bot$
- $\mathtt{n}' : r'$ is not the root

Lazy Refinement

When entering `do_refinement()`, we have:

- $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ is the oldest ancestor of $\mathtt{n}$ with $pre(err, \sigma) \sqcap r' \not\equiv \bot$
- $\mathtt{n}' : r'$ is not the root

So we have: $\mathtt{n}'' : r'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ and $pre(err, l\sigma) \sqcap r'' \equiv \bot$

- $r'' \xrightarrow{l\sigma} r$ is a spurious error trace

Lazy Refinement

When entering `do_refinement()`, we have:

- $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ is the oldest ancestor of $\mathtt{n}$ with $pre(err, \sigma) \sqcap r' \not\equiv \bot$
- $\mathtt{n}' : r'$ is not the root

So we have: $\mathtt{n}'' : r'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ and $pre(err, l\sigma) \sqcap r'' \equiv \bot$

- $r'' \xrightarrow{l\sigma} r$ is a spurious error trace

we need to refine $\mathtt{n}'' : r''$

Lazy Refinement

When entering `do_refinement()`, we have:

- $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ is the oldest ancestor of $\mathtt{n}$ with $pre(err, \sigma) \sqcap r' \not\equiv \bot$
- $\mathtt{n}' : r'$ is not the root

So we have: $\mathtt{n}'' : r'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ and $pre(err, l\sigma) \sqcap r'' \equiv \bot$

- $r'' \xrightarrow{l\sigma} r$ is a spurious error trace

we need to refine $\mathtt{n}'' : r''$

- we re-label $\mathtt{n}''$ by a new region w'' satisfying $w'' \equiv r''$ and $w'' \trianglelefteq r''$
- w'' should also satisfy $\widehat{post}(w'', l\sigma) \equiv \bot$ to rule out this false negative

Lazy Refinement

When entering `do_refinement()`, we have:

- $\mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ is the oldest ancestor of $\mathtt{n}$ with $pre(err, \sigma) \sqcap r' \not\equiv \bot$
- $\mathtt{n}' : r'$ is not the root

So we have: $\mathtt{n}'' : r'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ and $pre(err, l\sigma) \sqcap r'' \equiv \bot$

- $r'' \xrightarrow{l\sigma} r$ is a spurious error trace

we need to refine $\mathtt{n}'' : r''$

- we re-label $\mathtt{n}''$ by a new region w'' satisfying $w'' \equiv r''$ and $w'' \trianglelefteq r''$
- w'' should also satisfy $\widehat{post}(w'', l\sigma) \equiv \bot$ to rule out this false negative

→ The rest of the tree also needs to be updated...

Lazy Refinement (cont'd)

Updating:

- either the subtree starting at the sons of the refined node $\mathtt{n}''$ are removed
- or only regions along the path $\mathtt{n}'' : w'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ are re-computed

Unmarking recent leaves to garantee correctness:

- all covered leaves that were (last) marked after $\mathtt{n}''$ was last marked are unmarked

Lazy Refinement (cont'd)

Updating:

- either the subtree starting at the sons of the refined node $\mathtt{n}''$ are removed
- or only regions along the path $\mathtt{n}'' : w'' \xrightarrow{l} \mathtt{n}' : r' \xrightarrow{\sigma} \mathtt{n} : r$ are re-computed

Unmarking recent leaves to garantee correctness:

- all covered leaves that were (last) marked after $\mathtt{n}''$ was last marked are unmarked

Theorem When the lazy abstraction algorithm terminates, either:

- it returns an error-free over-approximation of the (concrete) reachability set, or
- it returns a concrete error path.

Application to predicate abstraction

We re-label $\mathtt{n}'' : r''$ by w'' satisfying $w'' \equiv r''$, $w'' \trianglelefteq r''$ and $\widehat{post}(w'', l\sigma) \equiv \bot$

- Refining a region consists in adding more support predicates
- New support predicates should rule out the spurious error trace

  $\rightarrow$ Predicate discovery

Outcomes of Lazy Abstraction

Outcomes of Lazy Abstraction

- Abstract $\widehat{post}$ computed only where required
  - crucial, since reachability set is very sparse

Outcomes of Lazy Abstraction

- Abstract $\widehat{post}$ computed only where required
  - crucial, since reachability set is very sparse
- Local refinement
  - reuse previous work: recompute only the spurious error path (or subtree)
  - for predicate abstraction: local predicates
    - computation of $\widehat{post}$ is very sensitive to the number of predicates

Outcomes of Lazy Abstraction

- Abstract $\widehat{post}$ computed only where required
  - crucial, since reachability set is very sparse
- Local refinement
  - reuse previous work: recompute only the spurious error path (or subtree)
  - for predicate abstraction: local predicates
    - computation of $\widehat{post}$ is very sensitive to the number of predicates
- General setting
  - Application to other classes of systems (e.g. hybrid systems)

Lazy predicate abstraction of C

- Control Flow Automata
  - $\Sigma = \mathtt{Op}$: blocks of assignments, if conditions, function calls
  - safety-monitor automaton
- Predicate language $\mathcal{L}$: theory of equality with uninterpreted functions + integer arithmetic

Lazy predicate abstraction of C

- Control Flow Automata
  - $\Sigma = \mathtt{Op}$: blocks of assignments, if conditions, function calls
  - safety-monitor automaton
- Predicate language $\mathcal{L}$: theory of equality with uninterpreted functions + integer arithmetic
- Control locations kept explicit: nodes labeled with (q, r)

Lazy predicate abstraction of C

- Control Flow Automata
  - $\Sigma = \mathtt{Op}$: blocks of assignments, if conditions, function calls
  - safety-monitor automaton
- Predicate language $\mathcal{L}$: theory of equality with uninterpreted functions + integer arithmetic
- Control locations kept explicit: nodes labeled with (q, r)
- Predicate discovery (from a false error trace):
  - Keep substitutions explicit
    - introduce new (existentially quantified) variables
    - all operations appear explicitly
  - Ask a proof of unsatisfiability
  - Pick predicates appearing in the proof

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. **Proof Generation**
5. Experiments
6. Conclusion and Future Work

Proof Carrying Code [Nec97]

- Verification Condition (VC)
  - first-order formula whose validity ensures correctness of the program
  - automatically computed from the annotated program code
    - annotations are invariants
    - user supplied, or automatically inferred

Proof Carrying Code [Nec97]

- Verification Condition (VC)
  - first-order formula whose validity ensures correctness of the program
  - automatically computed from the annotated program code
    - annotations are invariants
    - user supplied, or automatically inferred
- Code producer sends to the consumer:
  - annotated program code
  - proof of VC

Proof Carrying Code [Nec97]

- Verification Condition (VC)
  - first-order formula whose validity ensures correctness of the program
  - automatically computed from the annotated program code
    - annotations are invariants
    - user supplied, or automatically inferred
- Code producer sends to the consumer:
  - annotated program code
  - proof of VC
- Code consumer:
  - builds the VC from the annotated program code
  - checks the supplied proof

Proof Carrying Code [Nec97]

- Verification Condition (VC)
  - first-order formula whose validity ensures correctness of the program
  - automatically computed from the annotated program code
    - annotations are invariants
    - user supplied, or automatically inferred
- Code producer sends to the consumer:
  - annotated program code
  - proof of VC
- Code consumer:
  - builds the VC from the annotated program code
  - checks the supplied proof
- Proof checking is much easier than verification!

Temporal Safety Certification

- Assumption: the program code is a CFA
  - Specification: control location err is not reachable from q_0
    - but can use safety monitors to express safety properties

Temporal Safety Certification

- Assumption: the program code is a CFA
  - Specification: control location err is not reachable from q_0
    - but can use safety monitors to express safety properties
- Applying PCC
  - compute the invariant annotations
  - compute the VC
  - Generate the proof

Verification Condition for Safety

- Invariant annotations : formulas $Inv(q)$
  - one invariant annotation for each control location

Verification Condition for Safety

- Invariant annotations : formulas $Inv(q)$
  - one invariant annotation for each control location
- VC states that:
  - The global invariant annotation is an inductive invariant of the system
  - $Inv(q_0)$ is equivalent to `true`
  - $Inv(err)$ is equivalent to `false`

Verification Condition for Safety

- Invariant annotations : formulas $Inv(q)$
  - one invariant annotation for each control location
- VC states that:
  - The global invariant annotation is an inductive invariant of the system
  - $Inv(q_0)$ is equivalent to `true`
  - $Inv(err)$ is equivalent to `false`

$$VC \quad = \quad Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\texttt{op}} q'} \left(sp(Inv(q), \texttt{op}) \implies Inv(q') \right)$$

Invariant Generation via Lazy Abstraction

Assume that the lazy abstraction reachability tree, from $(q_0, \texttt{true})$, is computed with no error found. Then:

- for each node $\texttt{n} : (\mathit{err}, r)$ in the tree, r is empty
- if $\texttt{n} : (q, r) \xrightarrow{\texttt{op}} \texttt{n}' : (q', r')$ then $sp(r, \texttt{op}) \implies r'$
- if $\texttt{n} : (q, r)$ is a covered leaf, then there are internal nodes $\texttt{n}_1 : (q, r_1)$, $\texttt{n}_2 : (q, r_2), \ldots, \texttt{n}_k : (q, r_k)$ such that $r \implies r_1 \vee r_2 \vee \cdots \vee r_k$

Invariant Generation via Lazy Abstraction

Assume that the lazy abstraction reachability tree, from $(q_0, \texttt{true})$, is computed with no error found. Then:

- for each node $\texttt{n} : (\mathit{err}, r)$ in the tree, r is empty
- if $\texttt{n} : (q, r) \xrightarrow{\texttt{op}} \texttt{n}' : (q', r')$ then $sp(r, \texttt{op}) \implies r'$
- if $\texttt{n} : (q, r)$ is a covered leaf, then there are internal nodes $\texttt{n}_1 : (q, r_1)$, $\texttt{n}_2 : (q, r_2), \ldots, \texttt{n}_k : (q, r_k)$ such that $r \implies r_1 \vee r_2 \vee \cdots \vee r_k$

Take:

$$Inv(q) \quad = \quad \bigvee_{\texttt{n}:(q,r)} \quad r$$

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Goal : $Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(Inv(q), \mathtt{op}) \implies Inv(q')\right)$

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Goal : $Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(Inv(q), \mathtt{op}) \implies Inv(q')\right)$

Goal : $sp(r, \mathtt{op}) \implies r_1 \vee r_2 \vee \cdots \vee r_k$

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Goal : $Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(Inv(q), \mathtt{op}) \implies Inv(q')\right)$

Goal : $sp(r, \mathtt{op}) \implies r_1 \vee r_2 \vee \cdots \vee r_k$

- If r is the region of an internal node, with $\mathtt{n}:(q, r) \xrightarrow{\mathtt{op}} \mathtt{m}:(q', r_m)$, reduces to:

Goal : $sp(r, \mathtt{op}) \implies r_m$

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Goal : $Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(Inv(q), \mathtt{op}) \implies Inv(q')\right)$

Goal : $sp(r, \mathtt{op}) \implies r_1 \vee r_2 \vee \cdots \vee r_k$

- If r is the region of an internal node, with $\mathtt{n}:(q, r) \xrightarrow{\mathtt{op}} \mathtt{m}:(q', r_m)$, reduces to:

Goal : $sp(r, \mathtt{op}) \implies r_m$

→ Put together proofs of abstract $\widehat{post}$ computation

Proof Generation

- rules: standard deduction rules + special rules (equality, arithmetic, etc.)

Proof decomposition:

Goal : $Inv(q_0) \quad \wedge \quad \neg Inv(err) \quad \wedge \quad \bigwedge_{q \xrightarrow{\mathtt{op}} q'} \left(sp(Inv(q), \mathtt{op}) \implies Inv(q')\right)$

Goal : $sp(r, \mathtt{op}) \implies r_1 \vee r_2 \vee \cdots \vee r_k$

- If r is the region of an internal node, with $\mathtt{n}:(q, r) \xrightarrow{\mathtt{op}} \mathtt{m}:(q', r_m)$, reduces to:

Goal : $sp(r, \mathtt{op}) \implies r_m$

→ Put together proofs of abstract $\widehat{post}$ computation

- The other case is similar

Proof Generation (cont'd)

Optimized in two ways:

- simple proofs of $sp(Inv(q), \texttt{op}) \implies Inv(q')$, where parts of the disjuncts are matched using the tree
- non-uniform predicate abstraction results in fewer proof obligations

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

The BLAST tool

- Berkeley Lazy Abstraction Software Verification Tool

```
http://www-cad.eecs.berkeley.edu/~tah/blast/
```

- Applies lazy abstraction and proof generation to C programs
  - handles all syntactic constructs of C (MSVC and GCC extensions)
  - applied to Linux and Windows NT device drivers
- 10K lines of Objective Caml

The BLAST tool (cont'd)

BLAST uses:

- CIL compiler infrastructure [NMRW02]
  - to parse C programs and produce Control FLow Automata
- CUDD package [Som98]
  - to represent regions as BDDs over support predicates
- Simplify theorem prover [DNS]
  - for abstract post computation and inclusion check
- Vampyre proof generating theorem [Vam]
  - for predicate discovery and proof generation

The BLAST tool (cont'd)

BLAST uses:

- CIL compiler infrastructure [NMRW02]
  - to parse C programs and produce Control FLow Automata
- CUDD package [Som98]
  - to represent regions as BDDs over support predicates
- Simplify theorem prover [DNS]
  - for abstract post computation and inclusion check
- Vampyre proof generating theorem [Vam]
  - for predicate discovery and proof generation

BLAST is modular:

- lazy abstraction algorithm implemented by an OCaml functor
- can use different abstraction structures

Checking drivers with BLAST

- Linux device drivers
  - checked locking discipline
  - safety-monitor automaton with 3 states
  - found interprocedural bugs in the locking behavior
- Windows NT device drivers (from the Windows NT DDK)
  - I/O Request Packet Completion Specification
    - sequence of functions to be called
    - specific return codes
  - safety-monitor automaton with 22 states
  - found bugs involving incorrect status codes

Checking drivers with BLAST (cont'd)

Program	Post-processed LOC	Predicates		BLAST Time (sec)	Proof Size (bytes)
		Total	Active		
`qpmouse.c`	23539	2	2	0.50	175
`ide.c`	18131	5	5	4.59	253
`aha152x.c`	17736	2	2	20.93	
`tlan.c`	16506	5	4	428.63	405
`cdaudio.c`	17798	85	45	1398.62	156787
`floppy.c`	17386	62	37	2086.35	
[fixed]		93	44	395.97	60129
`kbfiltr.c`	12131	54	40	64.16	
[fixed]		37	34	10.00	7619
`mouclass.c`	17372	57	46	54.46	
`parport.c`	61781	193	50	1980.09	102967

- On a 700 MHz Pentium III with 256 MB RAM

Outline

1. Introduction
2. Predicate Abstraction
3. Lazy Abstraction
4. Proof Generation
5. Experiments
6. Conclusion and Future Work

Conclusion

Lazy abstraction:

- optimizes the three phases of the abstract-check-refine loop
  - model-checking driven
  - local refinement
- enables the automatic construction of small proof certificates
  - proof is generated from the error-free symbolic reachability tree
- has been implemented in the tool BLAST
  - ability to analyze real device drivers (60K lines of C)
  - generates small proofs (less than 150Kb)

Problems & Future Work

- Recursion
  - BLAST currently uses an explicit (unbounded finite) stack
- Acceleration of loops
- Application to other classes of systems (hybrid systems, protocols, ...)
- Efficient operations on interpreted BDDs
- Completeness results
- Non predicate based abstraction structures
  - add abstraction and refinement in symbolic model-checkers?

References

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[DNS] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover. `http://research.compaq.com/SRC/esc/Simplify.html`.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[Nec97] G.C. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, LNCS 2304, pages 213–228. Springer-Verlag, 2002.

[Sai00] H. Saidi. Model checking guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*, pages 377–396. LNCS 1824, Springer-Verlag, 2000.

[Som98] F. Somenzi. Colorado University decision diagram package. 1998.

[Vam] Vampyre. `http://www.eecs.berkeley.edu/$\sim$rupak/Vampyre`.