Building Dependable Concurrent Systems through Probabilistic Inference, Predictive Monitoring and Self-Adaptation

Gul Agha

University of Illinois at Urbana-Champaign
Predictive Monitoring

Joint work with Koushik Sen and Grigore Rosu
Smart Observer

• Monitoring is limited to what is observed.
• In real world, we learn from near misses
• A single execution trace contains more information than appears at first sight
• Extract other possible runs from a single execution
• Analyze all these runs
Smart Observers

• “Smartness” obtained by capturing causality

• Possible global states generated dynamically \(\rightarrow\) form a lattice

• Analysis is performed on a level-by-level basis in the lattice of global states
Causality

Define the **partial order** $\prec$ on the set of events:

1. $e_i^k \prec e_i^l$ if $k < l$;
2. $e \prec e'$ if
   $\exists x \in S \; e \prec_x e'$
   and at least one of $e, e'$ changes state.
   • $e \prec e''$ if
     $e \prec e'$ and $e' \prec e''$. 
Vector Clocks and Relevant Events

• Consider a subset $R$ of relevant events.

• $R$-relevant causality is a relation $\triangleleft \subseteq \prec$
  • $\triangleleft$ is a projection of $\prec$ on $R \times R$.

• We used a technique based on vector clocks to implements the relevant causality relation.
Causality and Vector Clocks

**Theorem:** If \((e, i, V)\) and \((e', j, V')\) are messages sent then \(e \prec e'\) iff \(V[i] \leq V'[j]\)

If \(i\) and \(j\) are not given, then
\(e \prec e'\) iff \(V \prec V'\)
Actor Creation

• A newly created actor inherits the “vector clock” from its parent at creation time.

• Vector Clock is not really a vector but an association list of actor names and its current estimated local time.
An Example

• Actor a requests certain value from node b

• b computes the value and sends it to a

• Property: no node receives a value from another node to which it had not sent a request
Centralized Monitoring Example

“If a receives a value from b then b calculated the value after receiving request from a”

\[ \text{valRcv} \rightarrow \lozenge(\text{valComputed} \land \lozenge\text{valReq}) \]

\[ \text{valReq} \land \lozenge\text{valReq} \land \text{valCc} \land (\text{valComputed} \land \lozenge\text{valReq} \lor \text{Req}) \]
Decentralized Approach

• Distribute property
  • Properties expressed with respect to a process
  • Local properties at every process

• Decentralize Monitoring
  • Maintain knowledge of global state at each process
  • Update knowledge with incoming messages
  • Attach knowledge with outgoing messages
  • At each process check safety property against local knowledge
Decentralized Monitoring Example

“If a receives a value from b then b calculated the value after receiving request from a”

\[ \text{valRcv} \rightarrow \@b(\Diamond (\text{valComputed} \land \@a(\Diamond \text{valReq}))) \]
Past time Distributed Temporal Logic

• Based on epistemic logic
• Properties with respect to a process, say p
• Interpreted over a sequence of global states that the actor p is aware of
  • Each actor monitors the properties local to it
  • No need for extra messages to create a relevant portion of global state
• KnowledgeVector keeps track of relevant global state that can effect a property.
Remote Expressions in pt-DTL

- Remote expressions – arbitrary expressions related to the state of another actor
- Propositions constructed from remote and local expressions

“If my alarm is set then eventually in past difference between my temperature and temperature at process b exceeded the allowed value”

$$\text{alarm} \rightarrow \Diamond ((\text{myTemp} - \@_b\text{temp}) > \text{allowed})$$
Safety in Airplane Landing

“ If my airplane is landing then the runway that the airport has allocated matches the one that I am planning to use”

landing \rightarrow (\text{runway} = @_{\text{airport}} \text{allocRunway})
Leader Election Example

“If a leader is elected then if the current process is a leader then, at its knowledge, none of the other processes is a leader”

elected $\rightarrow$ (state=leader $\rightarrow$ $\bigwedge_{i \neq j}(\neg i(state = \text{leader}) \land \neg j(state = \text{leader})))$
pt-DTL syntax and semantics

- \( F_i ::= \text{true} \mid \text{false} \mid P(E_i) \mid = F_i \mid F_i \land F_i \) \hspace{1cm} \text{propositional}
  \[
  \mid \cdot F_i \mid \forall F_i \mid \Diamond F_i \mid F_i S F_i \\
  \mid @jF_j
  \]

- \( E_i ::= c \mid v_i \mid V_i \mid f(E_i) \) \hspace{1cm} \text{functional}
  \[
  \mid @jE_j
  \]

- \( \cdot F_i : \text{previously } F_i \)
- \( \forall F_i : \text{always in past } F_i \)
- \( \Diamond F_i : \text{eventually in past } F_i \)
- \( F_i S F_i : F_i \text{ since } F_i' \)
- \( @jF_j : F_j \text{ at process } j \)

- \( c : \text{constant} \)
- \( v_i : \text{variable at process } I \)
- \( P(E_i) : \text{predicate on } E_i \)
- \( f(E_i) : \text{function } f \text{ applied to } E_i \)
- \( @jE_j : \text{expression } E_j \text{ at process } j \)
Interpretation of $@E_j$ at process i

Since, at $s_{23}$ $p_2$ is aware of $s_{12}$ of $p_1$

value of $@E_j$ in $s_{23}$ at $p_2$ = value of $E$ in $s_{12}$ at $p_1$
Example

\[ (Y \geq @_1 X) \text{ at } p_2 \]
Example

\( (Y \geq @_1 X) \) at \( p_2 \)
Predictive Monitoring

• Can predict the violation from the run that did not have the violation.

• Cannot detect a violation if there is no direct communication of intermediate value from p1 to p2
  • *Need time-outs or alarms*
  • *Have to be designed into the system*
Causality Cone Heuristics

- Theory of relativity
  - Speed of light
  - Space-time: causality cone

Not causally related
Past causally connected
Future causality

Not causally related

• Theory of relativity
  • Speed of light
  • Space-time: causality cone
Aggregate Properties

Example:

• A cluster has information distributed across thousands of nodes

• An attacker wants to steal confidential information from the cluster

• Each node sees only a few file transfers per hour, a common usage pattern
Scalable Monitoring

• Local monitoring doesn’t help:
  \[ \text{downloads}(f, C) < \text{limit} \]
  Is not violated at any node.

• Want to monitor for:
  \[ \sum_{n \in \text{Nodes}} \text{@}_n \text{downloads}(f, C) < \text{limit} \]

But monitoring thousands of nodes for millions of events is too expensive!
Statistical Runtime Monitoring

Efficient and effective to monitor probabilistic properties:

$$\Pr\left(\sum_{n \in \text{Nodes}} \forall_n \text{downloads}(f, C) < \text{limit}\right) > 0.999$$

- Monitoring against spatial and temporal variations.
- \textit{cf: Statistical Model Checking}
Adaptive Programs
Computational Reflection

- A meta-level actor describes functionality of actor.
- To change the application’s behavior, modify the relevant meta-actor.
Use Runtime Monitoring to Infer Specifications

• Assume violations are rare events
• Infer concurrency patterns by monitoring traces
• Use Bayesian methods for robustness against outlier observations
  ➢ Incorporate new evidence in a structured way
  ➢ Rare spurious behavior weighed against preponderance of contrary evidence and ignored
• Enforce or monitor for extended specification
• Example: *figure out the intended concurrency patterns and use it to transform to a safer actor program*
Goal: Moving Legacy Concurrent Programs to Clouds

Benefits of moving legacy programs to clouds

- Lower maintenance costs
- Easier and less costly redundancy
- Scalability

Some cloud migration problems

- Most legacy concurrent programs use *shared memory*
- Running on single virtual machine gives few advantages
- Difficult to simulate shared memory in distributed setting
Infrastructure-as-a-Service

Cloud User

End User

Platform

End User

Cloud Provider

Application

Infrastructure

IaaS
Platform-as-a-Service

Cloud User

Cloud Provider

End User

Application

Platform

Infrastructure

PaaS
Scalability: Actor Model of Computation

An actor is an autonomous, concurrent agent which responds to messages.

- Actors operate asynchronously, potentially in parallel with each other.
- Actors do not share state
- Each actor has a unique name (address) which cannot be guessed.
- Actor names may be communicated.
- Actors interact by sending messages which are by default asynchronous (and may be delivered out-of-order).
Actor Behavior

Upon receipt of a message, an actor may:

- create a new actor with a unique name (address).
- use the content of the message or perform some computation and to change state.
- send a message to another actor.
Actor Implementation in Threaded Languages

An actor may be implemented as a concurrent object. Each actor:

- has a system-wide unique name (mailbox address);
- has an independent thread of control; and
- has a message queue and processes **one message at a time**.
Execution of Message-Passing Programs in Networks
Execution of Message-Passing Programs in Networks
Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

Facebook

“[T]he actor model has worked really well for us, and we wouldn’t have been able to pull that off in C++ or Java. Several of us are big fans of Python and I personally like Haskell for a lot of tasks, but the bottom line is that, while those languages are great general purpose languages, none of them were designed with the actor model at heart.” –Facebook Engineering

Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

“When people read about Scala, it’s almost always in the context of concurrency. Concurrency can be solved by a good programmer in many languages, but it’s a tough problem to solve. Scala has an Actor library that is commonly used to solve concurrency problems, and it makes that problem a lot easier to solve.” – Alex Payne, “How and Why Twitter Uses Scala”\(^2\)

\[^2\]http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html
Some Actor Languages and Frameworks

- Erlang: web services, telecom, Cloud Computing
- E-on-Lisp, E-on-Java: P2P systems
- SALSA (UIUC/RPI), Charm++ (UIUC): scientific computing
- Ptolemy (UCB): real-time systems
- ActorNet (UIUC): sensor networks
- Scala (EPFL; Typesafe): multicore, web, banking
- Kilim (Cambridge): multicore and network programming
- Orleans; Asynchronous Agents Library (Microsoft): multicore programming, Cloud Computing
- DART (Google): Cloud Computing
Converting Shared-Memory Programs to Message Passing

- Manual conversion to use message passing requires understanding *concurrency semantics* of programs
  - how locking is used to uphold data structure invariants
  - how thread interference is avoided
- Many different “correct” conversions are possible—but finer granularity gives more concurrency
Converting Shared-Memory Programs to Message Passing

Our approach

1. Use runtime monitoring to infer concurrency semantics in terms of *data-centric synchronization* requirements.
2. Encapsulate objects inside message-passing *actors* based on synchronization analysis.
Converting Shared-Memory Programs to Message Passing

Our approach

1. Use runtime monitoring to infer concurrency semantics in terms of data-centric synchronization requirements.
2. Encapsulate objects inside message-passing actors based on synchronization analysis.
class ArrayList {
    int size;
    Object[] entries;

    Object get(int i) {
        synchronized(lock) {
            if (0 <= i && i < this.size) {
                return this.entries[i];
            } else {
                return null;
            }
        }
    }

    void addAll(ArrayList o) {
        synchronized(lock) {
            this.size += o.size;
        }
        /*... copy elements ...*/
    }
}
class ArrayList {
    atomicset L;
    atomic(L) int size;
    atomic(L) Object[] entries;

    Object get(int i) {
        if (0 <= i && i < this.size) {
            return this.entries[i];
        } else {
            return null;
        }
    }

    void addAll(unitfor(L) ArrayList o) {
        this.size += o.size;
        /*... copy elements ...*/
    }
}
Elements of Data-Centric Synchronization

Atomic Set
Group of fields in a class connected by a consistency invariant

Unit of Work
Method that preserves the invariant when executed sequentially

Alias
Combines atomic sets

Example
In the ArrayList class:
  - Invariant: entries[i] valid if i < size
  - Atomic set $L = \{ \text{size, entries} \}$

Example
In the ArrayList class:
  - Instance methods are units of work for all atomic sets of the object
  - `addAll(ArrayList)` is a unit of work for the other list’s atomic set $L$
Converting a Program to use Atomic Sets Requires Understanding its Concurrency Structure

Must *understand* old synchronization to convert it!

Conversion Experience of Dolby et al. [TOPLAS, 34(1):4, 2012]:

- Takes several hours for rather simple programs
- 2 out of 6 programs lack synchronization of some classes
- 2 out of 6 programs accidentally introduced global locks

Our Algorithm

Avoid conversion errors by automatically determining annotations from program traces using Bayesian probabilistic inference
Synopsis of our Algorithm for Probabilistically Inferring Atomic Sets, Aliases, and Units of Work

Assumptions about Input Programs

- Methods perform meaningful operations (convey intent)
- Data fields that a method accesses are likely connected by invariant

Algorithm Idea

- Observe which pairs of fields a method accesses atomically and their distance in terms of basic operations
  - This is (Bayesian) evidence that fields are connected through a semantic invariant
- Store current beliefs for all field pairs in affinity matrices
Analysis Supports Indirect Field Access and Access Paths

Indirect Access and Distance

- High-level semantic operations use low-level operations
- E.g., `get()` might call `getSize()` instead of accessing field `size`
- Propagate observed access to caller’s scope
- Quantify directness of access as *distance*

Access Paths

- Methods traverse the object graph
- Track *access paths* instead of field names
- Example: `this.urls.size`
Bayes’s Inversion Formula

Bayesian Inference Variables

$H$: “$f$ and $g$ are connected through an invariant” [Hypothesis]
$e_k$: “$f$, $g$ accessed (non-)atomically with distance $d_k$” [evidence]

Consider a sequence of observations $e_1, \ldots, e_n$ w.r.t. $f$ and $g$.
Want to know probability that $H$ holds given $e_1, \ldots, e_n$, i.e.,

$$P(H|e_1, \ldots, e_n) = \frac{P(e_1, \ldots, e_n|H) P(H)}{P(e_1, \ldots, e_n)}$$
Likelihood Ratios and Belief Updating

\[
\frac{P(H|e_1, \ldots, e_n)}{P(\neg H|e_1, \ldots, e_n)} = \frac{P(e_1, \ldots, e_n|H)}{P(e_1, \ldots, e_n|\neg H)} \times \frac{P(H)}{P(\neg H)}
\]

updated info = info from observations \times original info

posterior odds = likelihood ratio \times prior odds

\[
O(H|e_1, \ldots, e_n) = L(e_1, \ldots, e_n|H) \times O(H)
\]
Conditional Independence

If \( e_1, \ldots, e_n \) are conditionally independent given \( H \), we can write

\[
P(e_1, \ldots, e_n | H) = \prod_{k=1}^{n} P(e_k | H)
\]

and similarly for \( \neg H \), whereby

\[
O(H | e_1, \ldots, e_n) = O(H) \prod_{k=1}^{n} L(e_k | H)
\]

Adding one more piece of evidence \( e_{n+1} \), we get

\[
O(H | e_1, \ldots, e_n, e_{n+1}) = L(e_{n+1} | H) O(H | e_1, \ldots, e_n)
\]

Hence, if we have independence, know \( O(H) \), and can compute \( L(e_k | H) \), we can update odds on-the-fly when observing!
Conditional Independence

- Coarse-grained hypothesis space: $H \cup \neg H$
- With conditional independence, $e_1, \ldots, e_n$ should depend only on hypothesis, not on systematic external influence
- However, we have at least the following external factors:
  - workload
  - scheduler

Mitigating Dependencies

- Working assumption: good workload and long executions minimize external influence
- Safe to include $f$, $g$ in atomic set when there is no invariant...
- ...but may result in coarser-grained concurrency
Mapping Observations to Likelihoods

- Given access observation $e_k$ for fields $f$ and $g$ with operation distance $d_k$, need to compute $L(e_k|H)$

- $L(e_k|H)$ should increase as $d_k$ decreases up to some maximum, after which it is flat

- $L(e_k|H)$ should decrease as $d_k$ increases down to some minimum, after which it is flat
Advantages of On-the-Fly Bayesian Inference

- Likelihoods incorporate scope and distance of observations
- Beliefs can be revised by new evidence, and thus improve with longer executions
- Analysis becomes robust and insensitive to outlier observations
- Size of observation data is in the size of the codebase, not size of execution
- Infers aliases similarly to atomic sets, which is hard to do statically
Data-Centric Synchronization Inference Toolchain

start → Program

Instr. Bytecode → Affinity matrices

Workload

Affinity matrices → Atomic sets

Aliases
Actorizing Programs Annotated with Atomic Sets

- **Key property:** messages (method calls) to actors are processed *one-at-a-time*

- Fields in one atomic set *should not* span two actors at runtime

- When object instances are created:
  - instances of class `Thread` end up in separate actor
  - instances with *non-aliased* atomic sets end up in separate actor
  - instances with *aliased* atomic sets end up inside same actor

- Synchronization (two-phase commit) needed to handle some declarations!
Conversion Approach in Practice

Program

→ tool

Annotations

Annotated Program

Message-Passing Program

→ Program Using Message-Passing Library
class DownloadManager {
    // Atomic access ensured by monitors
    ArrayList urls;

    public synchronized URL getNextURL() {
        if (this.urls.size() == 0) return null;
        URL url = (URL) this.urls.get(0);
        this.urls.remove(0);
        announceStartInGUI(url);
        return url;
    }
    /* ... */
}

class DownloadThread extends Thread {
    DownloadManager manager;

    public void run() {
        URL url;
        while ((url = this.manager.getNextURL()) != null) {
            download(url); // Blocks while waiting for data
        }
    }
    /* ... */
}
public class Download {
    public static void main(String[] args) {
        DownloadManager manager = new DownloadManager();
        for (int i = 0; i < 31; i++) {
            manager.addURL(new URL("http://www.example.com/f" + i));
        }
        DownloadThread t1 = new DownloadThread(manager);
        DownloadThread t2 = new DownloadThread(manager);
        t1.start();
        t2.start();
    }
}
class DownloadManager {
    atomicset U;
    atomic(U) ArrayList urls|L=\texttt{this}.U|;
    public URL getNextURL() {
        if (this.urls.size() == 0) return null;
        URL url = (URL) this.urls.get(0);
        this.urls.remove(0);
        announceStartInGUI(url);
        return url;
    }
    /* ... */
}

public class DownloadThread extends Thread {
    DownloadManager manager;
    public void run() {
        URL url;
        while((url = this.manager.getNextURL()) != null) {
            download(url); // Blocks while waiting for data
        }
        /* ... */
    }
}
```java
class DownloadManager implements IDownloadManager {
    ArrayList urls;
    public URL getNextURL() {
        if (this.urls.size() == 0) return null;
        URL url = (URL) this.urls.get(0);
        this.urls.remove(0);
        announceStartInGUI(url);
        return url;
    }
    /* ... */
}

public class DownloadThread extends Thread implements IDownloadThread {
    IDownloadManager manager;
    public void run() {
        URL url;
        while((url = this.manager.getNextURL()) != null) {
            download(url); // Blocks while waiting for data
        }
        /* ... */
    }
    /* ... */
}
```
public class Download {
    public static void main(String[] args) {
        // Actorized initializations
        IDownloadManager manager = new DownloadManager();
        for (int i = 0; i < 31; i++) {
            manager.addURL(new URL("http://www.example.com/f" + i));
        }
        IDownloadThread t1 = new DownloadThread(manager);
        IDownloadThread t2 = new DownloadThread(manager);
        t1.start();
        t2.start();
    }
}
Future Work

- Runtime monitoring to detect specification of process/session types.
- Use runtime verification to detect violations of specifications.
- Enforcement of session types through a meta-actors.
- How to update specifications? Adaptation problem...
References I: Actor Languages, Computational Reflection and Runtime Verification


References II: Probabilistic Programming, Statistical Model Checking, Euclidean Model Checking and Sensor Networks


