Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems: a case study

F. Alberti[1] S. Ghilardi[3] E. Pagani[3] S. Ranise[2] G.P. Rossi[3]

[1]Università della Svizzera Italiana,
Lugano, Switzerland

[2]FBK-Irst,
Trento, Italy

[3]Università degli Studi di Milano,
Milano, Italy

10th International Workshop on
Automated Verification of Critical Systems
(AVOCS 2010)

Context and Goal

Fault-tolerant systems

Systems that show a correct behavior regardless failures

- Parameterized
- Numerous and heterogeneous applications
  - Network protocols
  - E-commerce
  - Distributed databases
  - Sensor networks
  - Real-time systems

Parameterized verification

Checking that a system satisfies a given property **regardless** the number of processes

Automatic verification of parameterized systems

Desiderata on tools for parameterized verification

- Natural input language
- Counterexample (if the case)
- High degree of automation
  - As much as possible automatic verification
  - Avoid the introduction of bugs from user interactions

Problems

- Number of processes unknown
- Processes manipulate variables defined over unbounded domains, e.g.
  - Integer or real variables
  - Pointers to other processes of the system

⇒ we need a tool to handle infinite state systems

Our case study: the problem of Reliable Broadcast

Ingredients of the problem

- Process p wants to send a message m to **all** processes
- Broadcast primitives not available

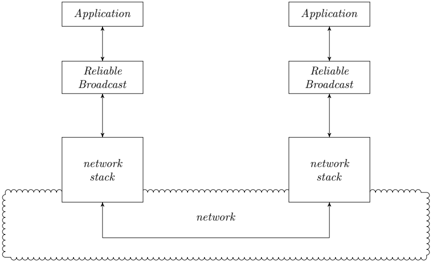
$\Rightarrow$ p must send m to each process **separately**

Failures

Temporary (*omission*) or persistent (*crash*) failure may cause inconsistencies!

Reliable Broadcast: a classical solution

T. D. Chandra and S. Toueg.
Time and message efficient reliable broadcasts.
In *Proceedings of the 4th international workshop on Distributed algorithms*, 289–303, 1991.



Parametric verification

Never been formally verified (to the best of our knowledge)

Our case study: Reliable Broadcast

Safety property: *agreement*

If a correct process delivers a message m, all correct processes deliver m

Correctness is defined w.r.t. different failure models

- Stopping failure
- Send-Omission
- General-Omission
- Arbitrary (Byzantine or malicious)

MCMT - Model Checker Modulo Theories

Why MCMT?

Because it matches the desiderata for infinite state verification tools

- ✔ Natural input language
- ✔ Counterexample (if the case)
- ✔ High degree of automation
  - ✔ As much as possible automatic verification
  - ✔ Avoid the introduction of bugs from user interactions

S. Ghilardi and S. Ranise.
MCMT: a Model Checker Modulo Theories.
In *Proceedings of IJCAR '10*, Springer LNCS, 2010.

MCMT - Model Checker Modulo Theories

Infinite state model checker

Main features

- Symbolic approach: formulae are used to represent set of states
- Declarative specification of topology and data with first order theories
- Predictable symbolic model checking supported by SMT-solvers
- Accept hints from user and use them without compromising correctness
- Easily integrated in the design methodology

MCMT - Model Checker Modulo Theories

Easy declarative language

Example

```
initial (universal p:nat) {
  estimate[p] = unknown  AND  round[p] = 1  AND
  decided[p] = false  AND  faulty[p] = false
}

unsafe (existential p1:nat, existential p2:nat) {
  estimate[p1] = unknown  AND  decided[p1] = true  AND  faulty[p1] = false
  estimate[p2] = message  AND  decided[p2] = true  AND  faulty[p2] = false
}

transition (existential p:nat, universal all:nat) {
  guard: (coord[p] = false) AND (aCoord[p] = false)
  uguard: (coord[all] = false)
  update:
    round := 1;
    coord[p] := true;
    done := lambda (j:nat) { false }
    request := false;
}
```

MCMT - Model Checker Modulo Theories

Predictability

Assumptions on termination

Termination of the algorithm is ensured by some assumption on the topology and data:

- Assumptions on topology are satisfied by many theories of interest like
  - Pure equality
  - Linear orders
  - Trees/forests
  - Graphs

  … but are not met by ring topology.
- We want T_I and T_E to be disjoint

MCMT - Model Checker Modulo Theories

Secure user interaction: the "plan of work"

Using invariants

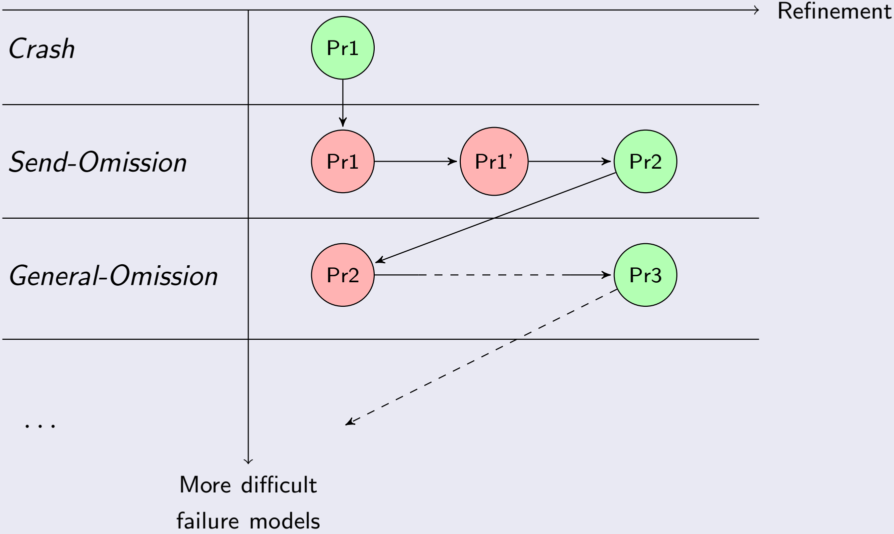
- Large protocols may require big amount of resources (both time and space).
- Using invariants can help to reduce resource requirements, but...
- ...are we sure that we are using *real* invariants?

Solution

- Declarative approach: if S is safe w.r.t. φ, then $\neg\varphi$ is a safe invariant for the system.
- Draw a plan of work!
- We can tell to MCMT:
  1. Try to check these invariants: φ_1, φ_2, φ_3, ...
  2. Use **only** those you have found to be *real* safe invariants in the main verification process.

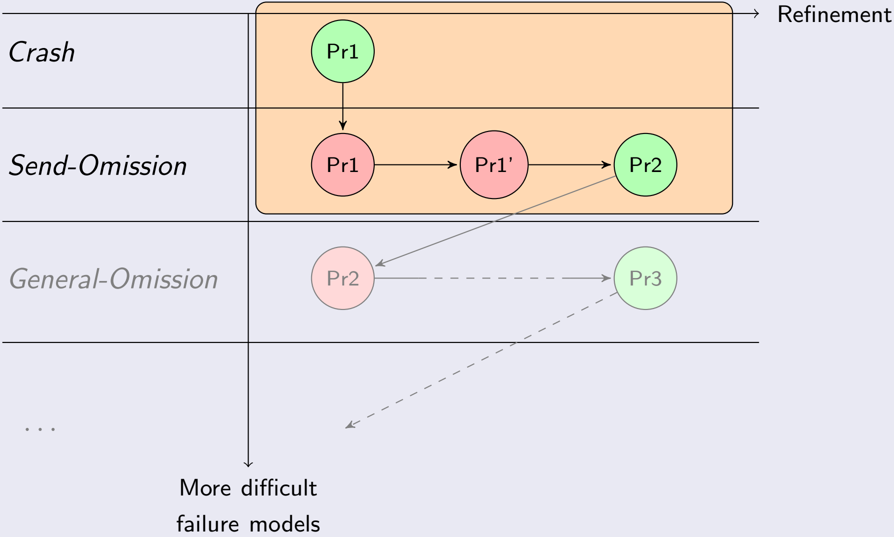
Our case study: Reliable Broadcast

Metodology



Our case study: Reliable Broadcast

Metodology



Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE			
# State variables	8			
# Transitions	13			
Time (s)	1,18			
# Nodes	113 (-21)			
# SMT calls	2.792			
Length unsafe trace	×			
# Invariants	×			
Max # processes	4			

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE		
# State variables	8	9		
# Transitions	13	16		
Time (s)	1,18	17,66		
# Nodes	113 (-21)	464 (-26)		
# SMT calls	2.792	20.009		
Length unsafe trace	×	11 tr.		
# Invariants	×	×		
Max # processes	4	5		

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE		
# State variables	8	9		
# Transitions	13	16		
Time (s)	1,18	17,66		
# Nodes	113 (-21)	464 (-26)		
# SMT calls	2.792	20.009		
Length unsafe trace	×	**11 tr.**		
# Invariants	×	×		
Max # processes	4	5		

Shorter than [1]

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE	UNSAFE	
# State variables	8	9	11	
# Transitions	13	16	22	
Time (s)	1,18	17,66	1.709,93	
# Nodes	113 (-21)	464 (-26)	9.679 (-770)	
# SMT calls	2.792	20.009	1.338.058	
Length unsafe trace	×	11 tr.	33 tr.	
# Invariants	×	×	×	
Max # processes	4	5	6	

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE	UNSAFE	
# State variables	8	9	11	
# Transitions	13	16	22	
Time (s)	1,18	17,66	1.709,93	
# Nodes	113 (-21)	464 (-26)	9.679 (-770)	
# SMT calls	2.792	20.009	1.338.058	
Length unsafe trace	×	11 tr.	**33 tr.**	
# Invariants	×	×	×	
Max # processes	4	5	6	

Same as in [1]

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE	UNSAFE	SAFE
# State variables	8	9	11	15
# Transitions	13	16	22	28
Time (s)	1,18	17,66	1.709,93	4.719,51
# Nodes	113 (-21)	464 (-26)	9.679 (-770)	11.158 (-1.290)
# SMT calls	2.792	20.009	1.338.058	2.558.986
Length unsafe trace	×	11 tr.	33 tr.	×
# Invariants	×	×	×	19 (+7)
Max # processes	4	5	6	6

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Results

	Crash, pr. 1	S-O, pr.1	S-O, pr.1 (e)	S-O, pr.2
Result	SAFE	UNSAFE	UNSAFE	SAFE
# State variables	8	9	11	15
# Transitions	13	16	22	28
Time (s)	1,18	17,66	1.709,93	4.719,51
# Nodes	113 (-21)	464 (-26)	9.679 (-770)	11.158 (-1.290)
# SMT calls	2.792	20.009	1.338.058	2.558.986
Length unsafe trace	×	11 tr.	33 tr.	×
# Invariants	×	×	×	19 (+7)
Max # processes	4	5	6	6

Mandatory!

- Intel Core2 Duo @ 2.66 GHz, 2 GB RAM, Linux Debian
- MCMT (v. 1.0.1) executed in default mode

Our case study: Reliable Broadcast

Remarks

- No toy problems!
- Finding unsafe traces is not trivial!

Our case study: Reliable Broadcast

Adding hints as invariants

The problem

- Verifying large protocols requires a large amount of resources
- Invariants can help reducing time and space requirements
- For the last protocol:
  - Without invariants, MCMT suffers a memory out after 1 day of computation
  - With seven invariants, MCMT ends the computation after 78 mins.

Our case study: Reliable Broadcast

Adding hints as invariants

Features of invariants

The seven invariants concerns how data structures are updated (no deep properties of the protocol!).

An example

- Rotating coordinator paradigm.
- Ciclically one process becomes the coordinator of the network.
- In the network there's only one coordinator.
- Local boolean array `coord` to say which process is the coordiantor;

A natural candidate invariant is this:

$$\forall i, j((\mathtt{coord}[i] = \mathrm{true} \wedge \mathtt{coord}[j] = \mathrm{true}) \to i = j)$$

Our case study: Reliable Broadcast

Adding hints as invariants

Another example

- Send Omission failure model
- Processes become coordinators in order of address
- We use a local boolean array `aCoord` to say which processes have already been coordinator;

$$\forall i, j \left(\begin{pmatrix} \texttt{aCoord}[i] = \text{true} \wedge \texttt{aCoord}[j] = \text{false} \wedge \\ \texttt{crash}[i] = \text{false} \wedge \texttt{crash}[j] = \text{false} \end{pmatrix} \rightarrow i < j \right)$$

Our case study: Reliable Broadcast

Adding hints as invariants

Prooving Invariants

Recall that we don't have to proove invariants by hand! We can use MCMT to proove them...

$$\forall i, j((\mathtt{coord}[i] = \mathsf{true} \wedge \mathtt{coord}[j] = \mathsf{true}) \rightarrow i = j)$$

is a safe invariant if the system is safe w.r.t. this unsafe configuration:

$$\neg\forall i, j((\mathtt{coord}[i] = \mathsf{true} \wedge \mathtt{coord}[j] = \mathsf{true}) \rightarrow i = j)$$

Our case study: Reliable Broadcast

Adding hints as invariants

Prooving Invariants

Recall that we don't have to proove invariants by hand! We can use MCMT to proove them...

$$\forall i, j((\texttt{coord}[i] = \text{true} \wedge \texttt{coord}[j] = \text{true}) \to i = j)$$

is a safe invariant if the system is safe w.r.t. this unsafe configuration:

$$\neg\forall i, j((\texttt{coord}[i] = \text{true} \wedge \texttt{coord}[j] = \text{true}) \to i = j)$$

or, rewriting it,

$$\exists i, j(\texttt{coord}[i] = \text{true} \wedge \texttt{coord}[j] = \text{true} \wedge i \neq j)$$

that is, the unsafe configuration that tells that there are two coordinators in the system!

Conclusions and Future works

Conclusions

- Parameterized verification of fault-tolerant protocols:
  - ✔ Symbolic representation ⇒ heterogeneous applications
  - ✔ *Candidates* invariants ⇒ “secure” interaction with user
  - ✘ Manual search for invariants
  - ✔ Interactive methodology for developing complex protocols
- Analyzed protocols:
  - ✔ First formal parametric verification of this protocols
  - ✔ Completely automatic (except the fourth)
  - ✔ Low resource consumption

Future works

- More difficult failure models (e.g. *general omission*)
- Timing constraints

Why General Omission is difficult?

Formalization

- New assumptions on the protocol:
    - Processes know the size of the network n
    - At least $\frac{n+1}{2}$ are correct

Verification

- Processes handle pointers to other processes identificators
- Processes perform aritmetic operations:
    - Processes count how many message they receives
    - We can formalize it adding a transition that increment a counter
    - Without acceleration, this lead to divergence

Thank you!
Questions?

Francesco Alberti
francesco.alberti@usi.ch