Validation of Contracts using Enabledness Preserving Finite State Abstractions

Guido de Caso* Víctor Braberman*

Diego Garbervetsky* Sebastián Uchitel*†

* Departamento de Computación, FCEyN, UBA. Buenos Aires, Argentina

† Department of Computing, Imperial College. London, UK

ICSE 2009, Vancouver, Canada

Why contracts?

Software contracts (**pre/postconditions, invariant,...**) appear in a variety of places:

- As a form of early *specification*: Z, "DbC" technique.
- As *annotations* for analysis tools: Spec#, JML for ESC/Java.
- As the *output* of analysis tools: Daikon, DySy.

But understanding contracts is far from being straightforward...

Contracts are hard to validate

contract *CircularBuffer*
  variable a : array [element]
  variable w, r : integer

  invariant : $0 \leq r < |a| \wedge 0 \leq w < |a| \wedge |a| > 3$

  start : $|a| > 3 \wedge r = |a| - 1 \wedge w = 0$

  action *write*(element e)
    pre : $w < r - 1 \vee (w = |a| - 1 \wedge r > 0)$
    post : $r' = r \wedge w' = (w + 1) \% |a| \wedge a' = \text{store}(a, w, e)$

  action element *read*()
    pre : $r < w - 1 \vee (r = |a| - 1 \wedge w > 0)$
    post : $a' = a \wedge w' = w \wedge r' = (r + 1) \% |a| \wedge \text{rv} = a[r']$

Figure: Pre/post specification for a circular buffer

The abstraction we construct

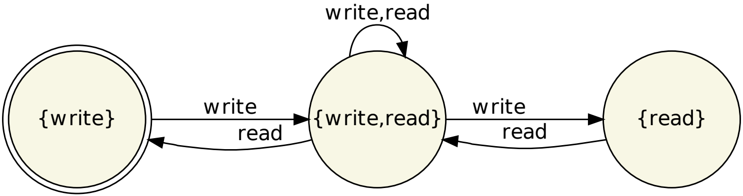


Figure: CircularBuffer contract abstraction

Understanding the CircularBuffer contract

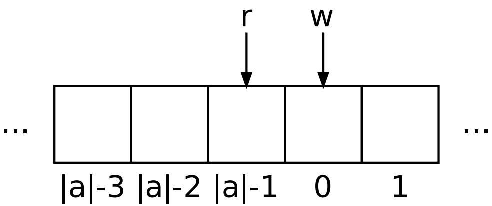


Figure: Empty CircularBuffer

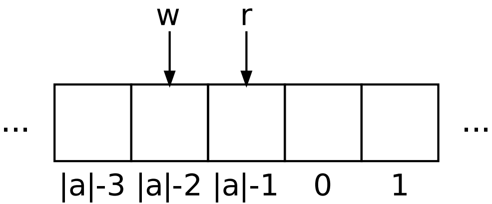


Figure: Full CircularBuffer

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

Problem: When do we stop?

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

  Problem: When do we stop?

- Perform simulations:

Figure: Simulating the circular buffer

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

  Problem: When do we stop?
- Perform simulations:

Figure: Simulating the circular buffer

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

  Problem: When do we stop?
- Perform simulations:

Figure: Simulating the circular buffer

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

  Problem: When do we stop?
- Perform simulations:

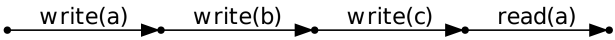


Figure: Simulating the circular buffer

What else can we do?

- Prove properties:
  - Can I read from a newly created buffer?
  - Can I read twice from a buffer where I've just written twice?
  - ...

  Problem: When do we stop?

- Perform simulations:

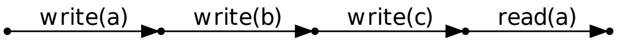
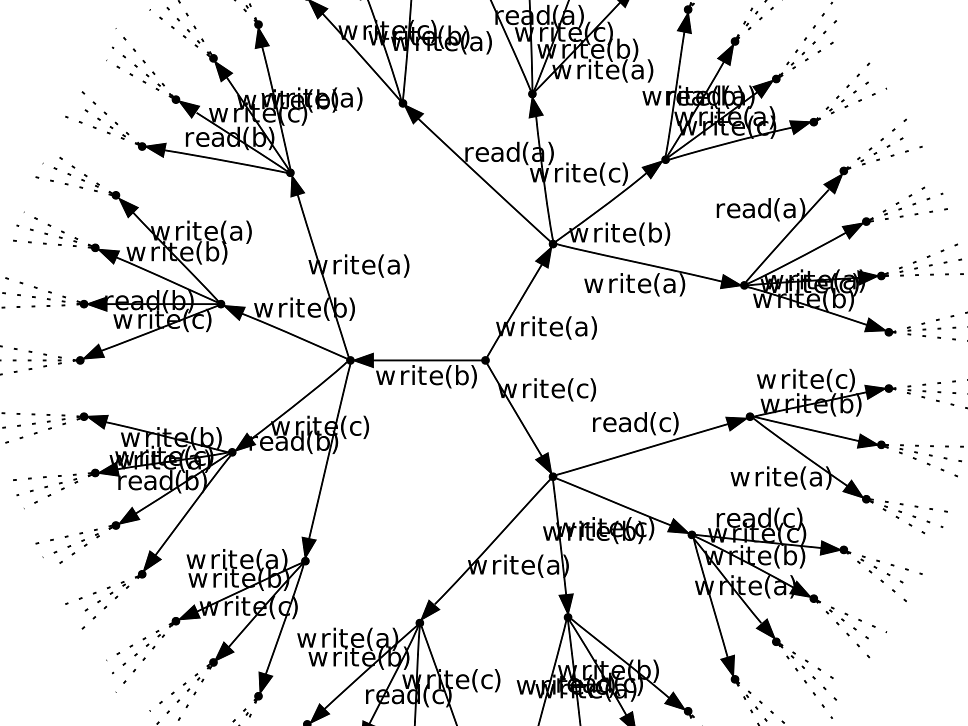


Figure: Simulating the circular buffer

Problem: When do we stop?



write(a)
write(b)
write(c)
read(a)
write(c)
write(b)
write(a)
write(c)
write(b)
write(a)
read(b)
write(c)
write(b)
write(a)
read(b)
write(c)
write(c)
read(b)
write(b)
write(c)
write(a)
read(b)
write(a)
write(b)
write(c)
read(c)
write(c)
write(b)
write(a)
write(a)
read(c)
write(c)
write(b)
write(a)
write(b)
write(a)
write(b)
write(c)
read(c)
write(b)
read(a)
write(c)
write(b)
write(a)
write(a)
write(c)
read(a)
write(b)

Simulation is like using a torch light

Figure: It's dark...

Simulation is like using a torch light

Figure: It's dark, we see some lights...

Simulation is like using a torch light

Figure: It's dark, we see some lights, a fountain...

Abstraction is like a pixelized view

Figure: It looks familiar!

Abstraction is the key, but how?

In order to produce an FSM that abstracts the CircularBuffer contract we must deal with:

- Potentially *infinite* parameter values.
- A *non-regular* underlying language.

Abstraction is the key, but how?

In order to produce an FSM that abstracts the CircularBuffer contract we must deal with:

- Potentially *infinite* parameter values.
- A *non-regular* underlying language.

Precision vs. size (when validating)

We have to be careful when abstracting and...

- avoid creating a lot of states (even infinite) by trying to reduce spurious behaviour.
- avoid creating a trivial abstraction with very few states that produces way too much spurious behaviour.

Enabledness

We need a way to get the "pixelized view" of a contract.

Enabledness

We need a way to get the “pixelized view” of a contract.

Enabledness equivalence

We say two variable valuations are *enabledness-equivalent* if they allow the same set of actions to occur.

Enabledness

We need a way to get the "pixelized view" of a contract.

Enabledness equivalence

We say two variable valuations are *enabledness-equivalent* if they allow the same set of actions to occur.

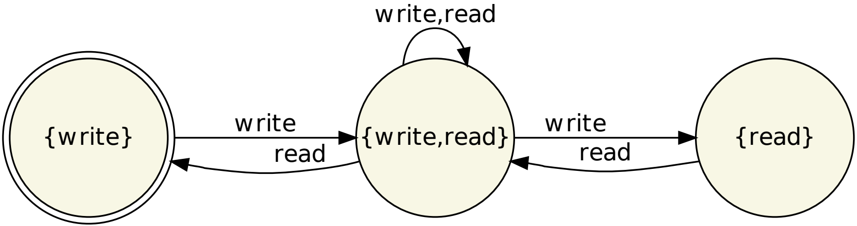


Figure: Enabledness equivalence based abstraction

Formalizing contracts

We say C is a *contract* iff $C = \langle V,\ \text{inv},\ \text{init},\ A,\ P,\ Q \rangle$:

Finite set of variables V

System invariant $\text{inv} \in \mathbb{P}(V)$

Initial predicate $\text{init} \in \mathbb{P}(V)$

Finite set of action labels $A = \{a_1, \ldots, a_n\}$

Preconditions $P : A \rightarrow \mathbb{P}(V \cup \{p\})$

Where p is the (only) action parameter.

Postconditions $Q : A \rightarrow \mathbb{P}(V \cup V' \cup \{p\})$

Where v' denotes the value of v after execution.

Where $\mathbb{P}(X)$ stands for the set of first order logic *predicates* with free variables in X.

Constructing contract abstractions: states

Enabledness-preserving Contract Abstraction (part 1 of 3)

An FSM $M = \langle S,\ S_0,\ \Sigma,\ \delta \rangle$ is an *enabledness-preserving contract abstraction* of $C = \langle V,\ \text{inv},\ \text{init},\ A,\ P,\ Q \rangle$ iff:

1. The set of states is the powerset of actions: $S = 2^A$

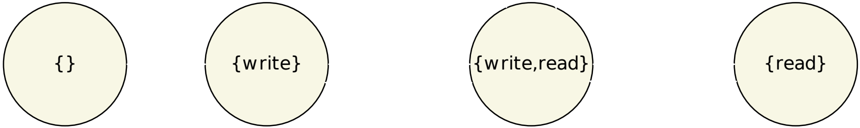


Figure: We use sets of **enabled actions** as states

Constructing contract abstractions: state invariants

State invariant

A state $s \subseteq A$ abstracts system instances on which the **enabled actions** are exactly s, characterized by the *state invariant* inv_s.

$$\mathrm{inv}_s \stackrel{\mathrm{def}}{=} \mathrm{inv} \ \wedge \ \bigwedge_{a \in s} \exists p.\ P_a \ \wedge \ \bigwedge_{a \notin s} \nexists p.\ P_a$$

Figure: We can discard a state s if inv_s is inconsistent.

Constructing contract abstractions: initial states

Enabledness-preserving Contract Abstraction (part 2 of 3)

An FSM $M = \langle S, S_0, \Sigma, \delta \rangle$ is an *enabledness-preserving contract abstraction* of $C = \langle V, \text{inv}, \text{init}, A, P, Q \rangle$ iff:

2. The set of initial states is:

$$S_0 = \{ s \mid \text{init} \Rightarrow \text{inv}_s \}$$

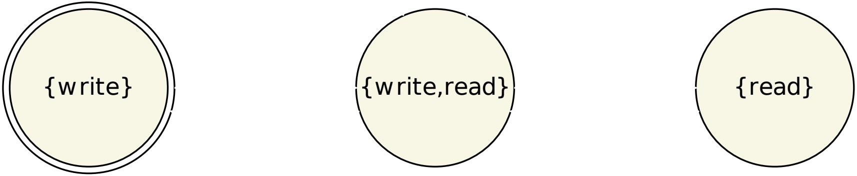


Figure: Initial sets are those implied by the initial predicate

Constructing contract abstractions: transitions

Enabledness-preserving Contract Abstraction (part 3 of 3)

An FSM $M = \langle S,\ S_0,\ \Sigma,\ \delta\rangle$ is an *enabledness-preserving contract abstraction* of $C = \langle V,\ \text{inv},\ \text{init},\ A,\ P,\ Q\rangle$ iff:

3. The alphabet is the set of action labels

$$\Sigma \ = \ A$$

Constructing contract abstractions: transitions

Enabledness-preserving Contract Abstraction (part 3 of 3)

An FSM $M = \langle S,\ S_0,\ \Sigma,\ \delta\rangle$ is an *enabledness-preserving contract abstraction* of $C = \langle V,\ \text{inv},\ \text{init},\ A,\ P,\ Q\rangle$ iff:

3. The alphabet is the set of action labels

$$\Sigma \ = \ A$$

4. The transition function $\delta : 2^A \times A \longrightarrow 2^{2^A}$ satisfies

$$\begin{array}{lll} \delta(s,a) & = \ \emptyset & \text{if } a \notin s \\ \delta(s,a) & \supseteq \ \{s' \mid \text{inv}_s \ \wedge\ Q_a \ \wedge\ \text{inv}'_{s'} \ \text{ is satisfiable}\} & \text{if } a \in s \end{array}$$

(The last item is relaxed due to decidability issues.)

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \to 2^{2^A}$ satisfies

$$\begin{array}{lll} \delta(s,a) & = \emptyset & \text{if } a \notin s \\ \delta(s,a) & \supseteq \{s' \mid \mathrm{inv}_s \wedge Q_a \wedge \mathrm{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s \end{array}$$

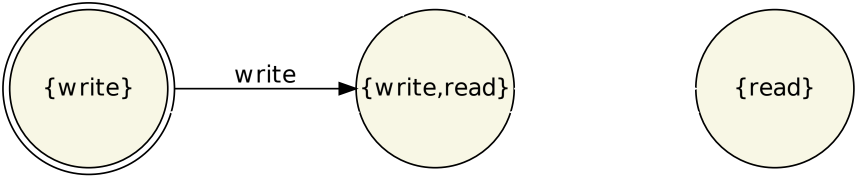


Figure: We add transitions

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \to 2^{2^A}$ satisfies

$$
\begin{array}{lcll}
\delta(s,a) & = & \emptyset & \text{if } a \notin s \\
\delta(s,a) & \supseteq & \{s' \mid \mathsf{inv}_s \wedge Q_a \wedge \mathsf{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s
\end{array}
$$

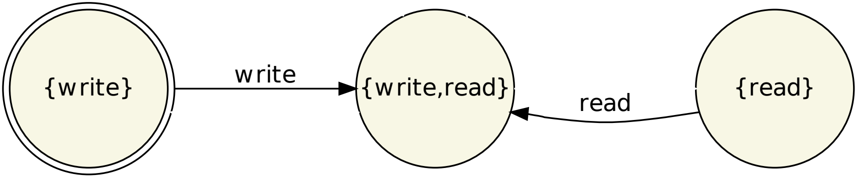


Figure: We add transitions

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \rightarrow 2^{2^A}$ satisfies

$$\begin{array}{lcll} \delta(s,a) & = & \emptyset & \text{if } a \notin s \\ \delta(s,a) & \supseteq & \{s' \mid \mathsf{inv}_s \wedge Q_a \wedge \mathsf{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s \end{array}$$

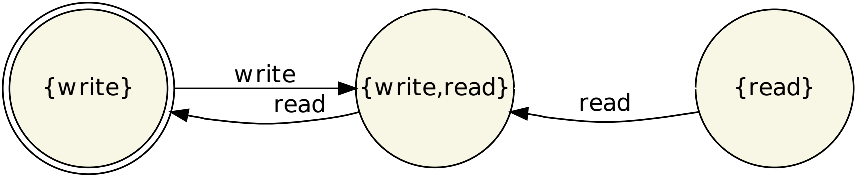


Figure: We add transitions

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \rightarrow 2^{2^A}$ satisfies

$$
\begin{array}{lll}
\delta(s,a) & = \emptyset & \text{if } a \notin s \\
\delta(s,a) & \supseteq \{s' \mid \mathrm{inv}_s \wedge Q_a \wedge \mathrm{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s
\end{array}
$$

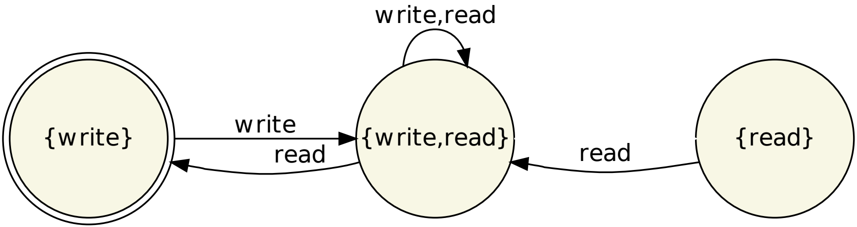


Figure: We add transitions

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \to 2^{2^A}$ satisfies

$$\begin{array}{lll} \delta(s, a) & = \emptyset & \text{if } a \notin s \\ \delta(s, a) & \supseteq \{s' \mid \text{inv}_s \wedge Q_a \wedge \text{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s \end{array}$$

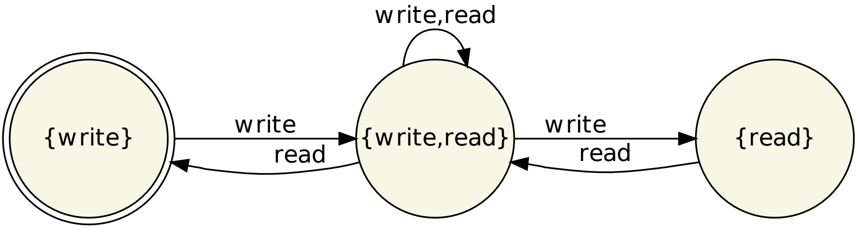


Figure: We add transitions

Constructing contract abstractions: transitions

The transition function $\delta : 2^A \times A \rightarrow 2^{2^A}$ satisfies

$$\begin{array}{lcll} \delta(s,a) & = & \emptyset & \text{if } a \notin s \\ \delta(s,a) & \supseteq & \{s' \mid \text{inv}_s \wedge Q_a \wedge \text{inv}'_{s'} \text{ is satisfiable}\} & \text{if } a \in s \end{array}$$

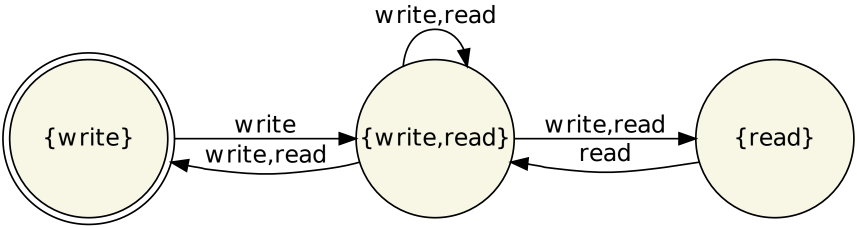


Figure: We add transitions

Validation begins!

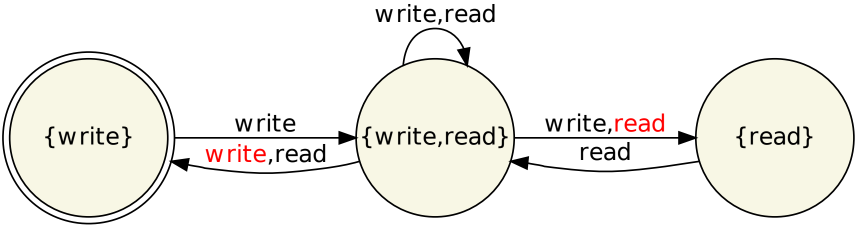


Figure: Finished CircularBuffer contract FSM

What went wrong?

Invariant of state $\{\texttt{write}, \texttt{read}\}$

$$(w < r - 1 \lor (w = |a| - 1 \land r > 0)) \land (r < w - 1 \lor (r = |a| - 1 \land w > 0))$$

What went wrong?

Invariant of state $\{\mathtt{write}, \mathtt{read}\}$

$$(w < r - 1 \vee (w = |a| - 1 \wedge r > 0)) \wedge (r < w - 1 \vee (r = |a| - 1 \wedge w > 0))$$

This is consistent with this position of r and w:

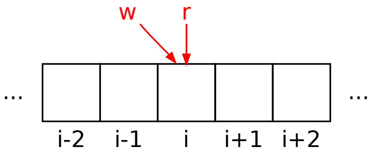


Figure: CircularBuffer with equal pointers

What went wrong?

Invariant of state {`write`, `read`}

$$(w < r - 1 \vee (w = |a| - 1 \wedge r > 0)) \wedge (r < w - 1 \vee (r = |a| - 1 \wedge w > 0))$$

This is consistent with this position of r and w:

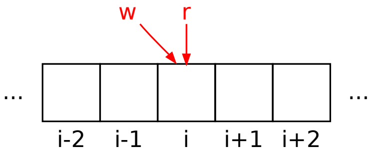


Figure: CircularBuffer with equal pointers

And from this position we can:

- Apply `read()` and go to a full buffer state.
- Apply `write(e)` and go to an empty buffer state.

What did go wrong?

Remember the CircularBuffer contract:

contract *CircularBuffer*
  variable a : array [element]
  variable w, r : integer

  invariant : $0 \leq r < |a| \wedge 0 \leq w < |a| \wedge |a| > 3$

...

What did go wrong?

Remember the CircularBuffer contract:

contract *CircularBuffer*
  variable a : array [element]
  variable w, r : integer

  invariant : $0 \leq r < |a| \wedge 0 \leq w < |a| \wedge |a| > 3$
  ...

The contract omitted saying that $r \neq w$ is part of the invariant!

The enabledness-preserving abstraction helped us to find this bug.

Fixed CircularBuffer abstraction

Adding $r \neq w$ to the specification yields:

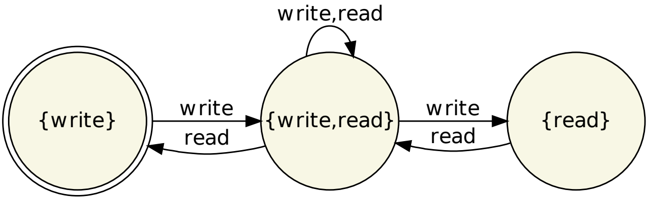


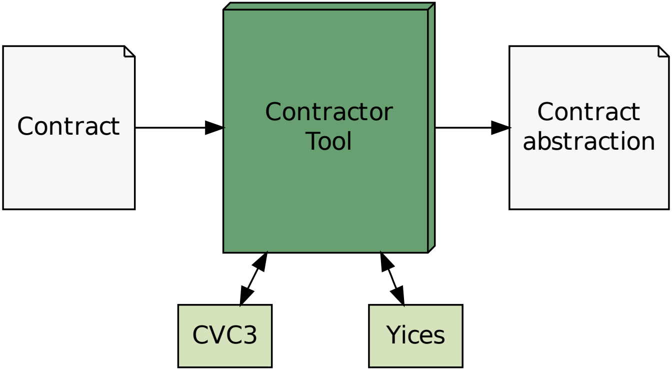
Figure: Finite abstraction of the amended CircularBuffer contract

This abstraction is an intuitive representation of a buffer:

- One state abstracts all the buffers that are empty.
- Other state abstracts all the buffers that are partially full.
- The last state abstracts all the buffers that are full.

Tool support

We implemented a tool called **Contractor**:



Contractor is open source and available at
`http://lafhis.dc.uba.ar/contractor`

Case studies

Using our Contractor tool we were able to carry out a series of case studies, including:

Name	**Source**	**Number of actions**	**Running time**
Web fetcher	DeLine and Fahndrich (ECOOP 2004)	4	0.14 seconds
ATM	Whittle and Schumann (ICSE'00)	8	8 seconds
MS-NSS	Microsoft	13	67 seconds
MS-WINSRA	Microsoft	33	?

Today I will focus on the third one.

The .NET NegotiateStream protocol specification v2.0[1]

.NET NegotiateStream Protocol basics

A protocol for the negotiation of credentials between a client and a server over a TCP stream:

1. The client requests a desired level of security (e.g. encryption).
2. On a first phase a token is passed between the 2 sides.
3. When a special finalization token is received by the client, the second phase starts.
4. During the second phase data is exchanged using the agreed security level.

[1] http://msdn.microsoft.com/en-us/library/cc236723.aspx
Document in version 2.0 during August 2008

Experimental setup

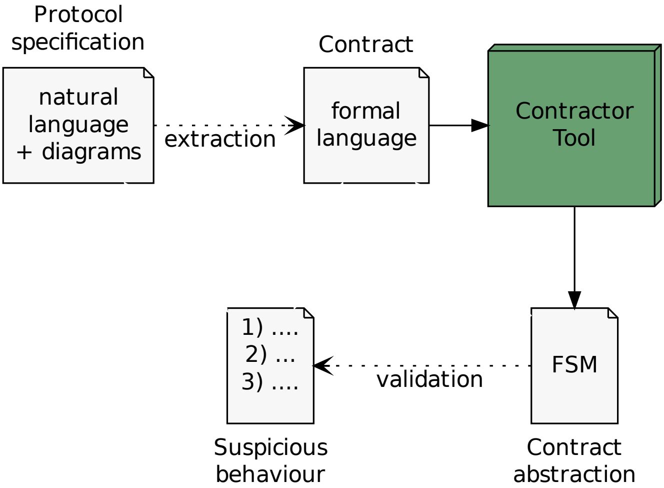


Figure: How the Contractor tool was used

Experimental setup

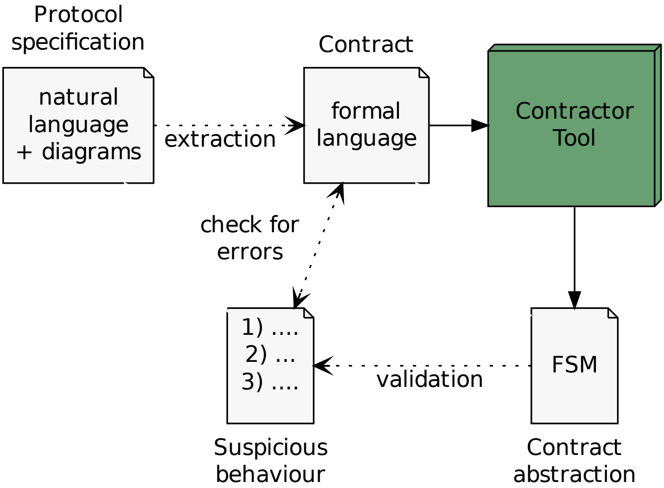


Figure: How the Contractor tool was used

Experimental setup

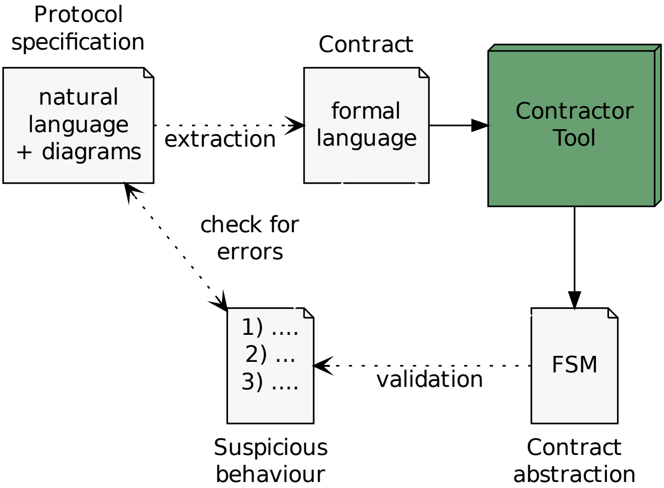


Figure: How the Contractor tool was used

.NET NegotiateStream finite contract abstraction

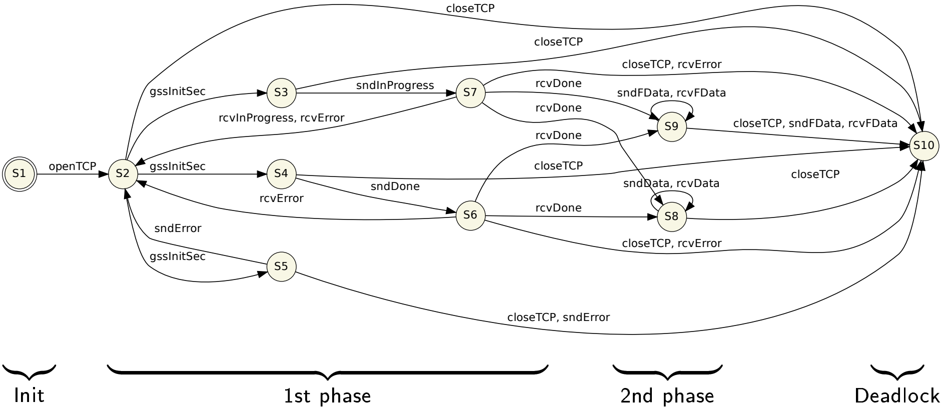


Figure: Finite abstraction of the .NET NegotiateStream protocol contract

.NET NegotiateStream finite: Suspicious behaviour (i)

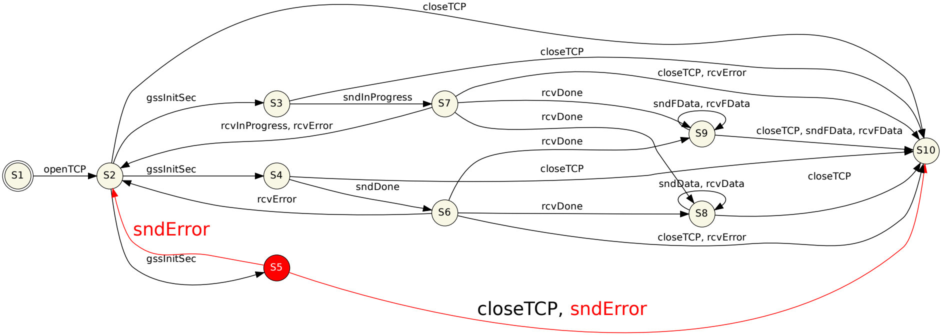
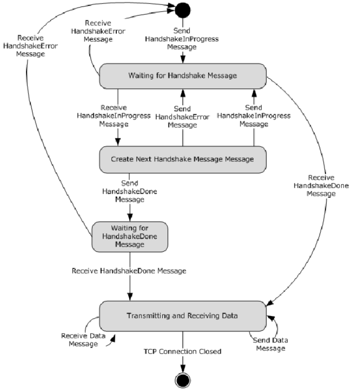
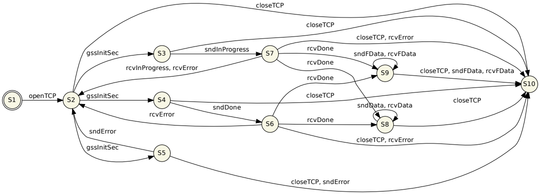


Figure: What happens when error messages occur?

Level of abstraction comparison





These diagrams are not consistent... but *why*?

.NET NegotiateStream finite: Suspicious behaviour (ii)

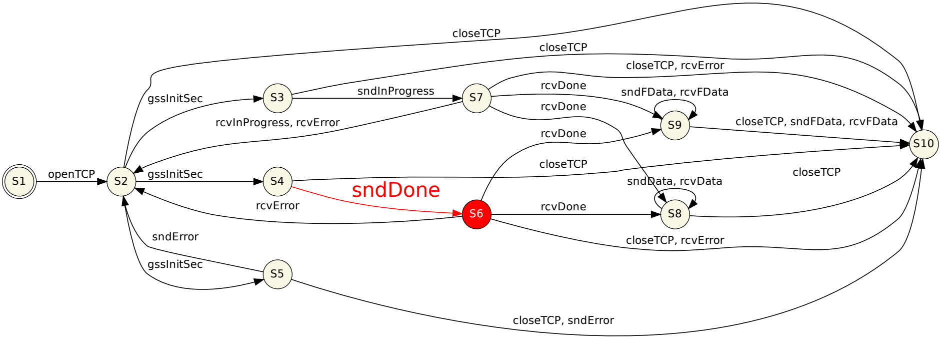


Figure: This FSM deadlocks if composed with the client informal diagram

.NET NegotiateStream finite: Findings

The ambiguities found in the protocol contract abstraction were tracked down in the protocol specification document.

The mentioned errors were corrected in the subsequent official protocol specification.

Contributions

Theoretical

- We formalised the concept of enabledness-based finite behavioural contract abstractions.
- We provided a novel symbolic algorithm to get such abstractions.

Contributions

Theoretical

- We formalised the concept of enabledness-based finite behavioural contract abstractions.
- We provided a novel symbolic algorithm to get such abstractions.

Practical

- We showed their potential validation capacity.
- We implemented our algorithm as a practical tool and used it on a variety of contracts.
- We discovered inconsistencies or omissions in real-life specifications.

Current and future work

Scalability We're working on an on-the-fly multi-threaded algorithm.

Current and future work

Scalability
: We're working on an on-the-fly multi-threaded algorithm.

Precision
: We would like to distinguish transitions depending on whether they are always traversable or not. Modalities.

Current and future work

Scalability We're working on an on-the-fly multi-threaded algorithm.

Precision We would like to distinguish transitions depending on whether they are always traversable or not. Modalities.

Analysability Add simulation support to the tool, together with visual aids such as a hierarchical view of states or decomposition into smaller FSMs.

Related work (i): predicate abstraction

- **Sun and Dong**[2].
  Construction of states using predicates from LSCs and transitions using pre/postconditions.
- **Grieskamp, Kicillof and Tillmann**[3]. **Nebut, Fleurey, Le Traon and Jézéquel**[4]. **Leuschel and Butler**[5].
  Exploration of a contract state space symbollicaly or concretely but no intention to construct a finite abstraction.
- **Van, van Lamsweerde, Massonet and Ponsard**[6].
  Construction of a contract finite abstraction by imposing bounds to the data domains.

---

[2] *Design synthesis from interaction and state-based specifications*, TSE 2006

[3] *Model-based quality assurance of Windows protocol documentation*, ICST 2008

[4] *Automatic test generation: a use case driven approach*, TSE 2006

[5] *ProB: an automated analysis toolset for the B method*, STTT 2008

[6] *Goal-oriented requirements animation*, RE 2004

Related work (ii): other techniques

- **Lee and Yannakakis**[7]. **Tripakis and Yovine**[8].
  Minimisation of LTSs by stabilising state space partitions. Requiring pre-stability may end up in huge or even infinite LTSs in our setting.
- **Alur, Cern, Madhusudan and Nam**[9].
  Conservative construction of finite behaviour models out of Java code. By avoiding exception raising the result is too restrictive when the system language is non-regular.
- **Gabel and Su**[10].
  Mining of finite state automata out of execution traces.
- **Letier, Kramer, Magee, Uchitel**[11].
  Construction of FSMs out of pre/post specifications. Language is propositional and there is no abstraction.

---

[7] *Online minimization of transition systems*, ACM Symposium on Theory of Computing 1992

[8] *Analysis of Timed Systems Using Time-Abstracting Bisimulations*, FMSD 2001

[9] *Synthesis of interface specifications for Java classes*, POPL 2005

[10] *Symbolic mining of temporal specifications*, ICSE 2008

[11] *Deriving event-based transition systems from goal-oriented requirements models*, JASE 2008

Thank you!

- Gracias
- Grazie
- Danke
- Obrigado
- Xie xie
- Merci
- Kamsahamnida
- Toda
- Shukran
- Arigato
- ...