‘Recognizing Safety and Liveness’ by Alpern and Schneider

Calvin Deutschbein

17 Jan 2017

1 Intro

1.1 Safety

What is safety?

“Bad things do not happen”

For example, consider the following “safe” program in C:

```
int main() {
return 0;
}
```

It will not cause anything bad to happen.

Safety is here based on invariance arguments, that is, that certain aspects of a system remain unchanged under any circumstance. The invariant parameter is, of course, the one that ensures nothing bad happens.

1.2 Liveness

What is liveness?

“Good things do happen”

Or at least will happen eventually. There is not a timing constraint in this model.

For example, consider the following “live event”: Nuclear Winter.

Sure bad things happen, but it has been an inspiration for much art so something good about it has happened.

Liveness is usually related to well-foundedness.

In logic, well-foundedness means that for a binary relationship on a set, each subset has a minimal element.

Take that minimal element as being the “something good” and it’s fairly easy to imagine that if events always push a system in a certain direction, and it is known that after a certain amount of pushing “something good” would be arrived at, satisfying liveness.

2 Definitions

2.1 Program Model

Programs are modeled by a start state and a series of actions from that start state.

Programs are denoted as "π"

Executions are denoted as "σ"

An execution σ can be expressed as a sequence of program states $s_0 s_1 ...,$.

In general, this sequence is infinite because programs may be infinite. If it is finite, simply repeat the last state infinitely many times.

2.2 Properties

By definition, in this paper properties are sets of infinite sequences of program states.

What does that mean?

Recall that σ denotes an infinite sequence of program states. This is also called an execution of a program.

A specific execution, say σ_1 (of program π_1 for example), might be a member of set P which means that it has property P. This is expressed with double turnstile "$\vDash$" as in: "$\sigma_1 \vDash P$"

Program π_1 "*satisfies*" P if all of its possible executions are in P, that is

$\forall i, \sigma_i \vDash P$

In this paper (as its contribution), properties are expressed as Buchi automata.

2.3 Buchi Automata

History fact: It is named after the Swiss mathematician Julius Richard Bchi who invented this kind of automaton in 1962.

They can, like many automata, be deterministic or non-deterministic. Unlike many automata, non-deterministic Buchi automata can recognize some infinite sequences that deterministic Buchi automata cannot.

Keep in mind that these are designed to accept infinite strings. So in the case of finite length programs we have to massage them a bit.

Now:

A Buchi Automaton m for a program property P is a five-tuple $m = (S, Q, Q_0, Q_\infty, \delta)$:

S is the set of program states of π

Recall program states are denoted s_i and that an infinite sequence of such states forms an execution σ

Q is the set of automaton states of m.

These are graphically represented with circles.

$Q_0 \subset Q$ is the set of start states of m.

In a deterministic automaton, there would be only once such state. These states are graphically represented with incoming arcs from outside the automaton.

THERE ARE TWO KINDS OF STATES.

$Q_\infty \subset Q$ is the set of accepting states of m.

These are graphically represented by double (concentric) circles. In the case of a finite program sequence being accepted, for example, there would be an accepting state with arc containing only the final state s_n being leaving and arriving at the accepting state.

$\delta \in (Q \times S) \to 2^Q$ is the "*transition function*" of m.

The transition function gives what automaton state in Q to transition to given your current automaton state and the new program state. In non-deterministic automata, this may be a set of states or, in deterministic, a single state.

2.4 Transition Predicates

From δ, a transition predicate T_{ij} can be derived that encodes whether a transition is legal given a state of both program and automaton.

T_{ij} is False if $\forall s : \delta(q_i, s) = \text{False}$

2.5 Finite Transition Functions

First some notation.

Recall that an execution σ gives an infinite sequence of program states s_i.

$\sigma[i] = s_i$

$\sigma[...i] = s_0, ..., s_i$

$\sigma[i...] = s_i, s_(i + 1), ...$

$|\sigma| =$ the length of σ or ω is σ is infinite.

Then consider finite transition function δ^*. δ^* is identical to δ except that once the sequence is of length $|\sigma|$ there are no more changes in program state (that is, the states denoted s_i) after a certain point.

2.6 Acceptance

A *run* of m is the sequence of states m passes through, that is, the states denoted q_i while reading σ (or, in the non-deterministic case, any such sequence).

The set of runs is given as $\Gamma(\sigma)$ and is singular in deterministic case.

$INF_m(\sigma)$ is the set of automaton states (that is, the states denoted q_i) that appear infinitely often in any $m \in \Gamma(\sigma)$

σ is accepted by m if and only if:

$\exists q_i : q_i \in INF_m(\sigma) \wedge q_i \in Q_\infty$

Recall that Q_∞ is the set of accepting states of m.

At a high level, this means an execution is accepted by m if an accepting automaton state occurs infinitely many times in its run.

2.7 Examples of Properties

Partial Correctness, Mutual Exclusion, and Starvation Freedom are all trivially represented by Buchi Automata.

The mutex example shows how to encode a system only defined by not violating an invariant.

There's little else of interest in the examples.

2.8 Final Words

The idea of Buchi automata in safety and liveness is to make sure that any failure of safety results in an undefined transition that causes the program not to be accepted and that liveness must occur before reaching an accepting state.

3 Recognizing Safety & Liveness

I would encourage you not to over-think safety. It means bad things don't happen. The authors go so far as to say that bad things are okay if they're fixable. Sure. Just don't over-think it.

3.1 Defining Safety

In plain English, this says that an execution of program (recall that this is denoted $\sigma = s_1, s_2, ...$) is safe if execution at every point, that is, for every i, can be appended with an infinite sequence (called β) that creates an infinite sequence that satisfies safety property P.

Before the formal definition, note that S denotes the set of program states, S^ω denotes infinite sequences of such states, and S^* finite sequences.

Here's the formal definition:

$\forall \sigma \in S^\omega, \sigma \vDash P \Leftrightarrow \forall i \geq 0 \exists \beta \in S^\omega : \sigma[...i]\beta \vDash P$

3.2 Safety in Buchi Automata

Since safety is about bad things not happening, any transition on a Buchi automaton doesn't violate safety (otherwise, such a transition would not be present). Consequently, in the case of safety, all automata states are accepting. This is called the "*closure*" $cl(m)$ of a Buchi automaton.

Therefore, a Buchi automata recognizes a safety property if $m = cl(m)$, that is, that all states of m are accepting states.

Of note, it's important to reduce the automaton first.

This stream of thought is codified in Theorem 1, however, as a simple application of a definition, I didn't feel a need to delve into that too deeply.

3.3 Defining Liveness

In plain English, the liveness definition states that no matter what sequence of states a program has passed through, a sequence exists that will carry it to satisfy a liveness property by reaching an accepting state.

Here's the formal definition:

$\forall \alpha \in S^* \exists \beta \in S^\omega : \alpha\beta \vDash P$

This is a lot more straightforward than safety because it is not concerned with what happens at all times, only at one.

3.4 Liveness in Buchi Automata

Liveness also has a relationship to the closure operation on automata. Recall that safety properties are those that are defined by automata that are their own closures.

As liveness is unconcerned with any occurrences other than the final state, it's closure, that is, $cl(m)$, must accept every input.

I found the theorem here, Theorem 2, similarly redundant as an application of definition.

4 Partitioning into Safety & Liveness

This is the fun part.

Consider Buchi automaton m. The property specified by m, and do recall that in this paper properties are sets of infinite sequences of program states, can be specified as being the intersection of the properties of a safety-only oriented automaton $Safe(m)$ and a liveness-only oriented automaton $Live(m)$.

It makes sense to talk about intersections of properties, because properties are just sets.

This is interesting for a variety of reasons, namely that everything expressed by a Buchi automata can be considered a combination of safety or liveness given these definitions of safety and liveness.

We start by looking at $Safe(m)$.

4.1 *Safe(m)*

Take $Safe(m)$ to be $cl(m)$. Obviously that expresses a safety property. This is established rigorously by Theorems 1 and 3.

4.2 *Live(m)*

The trick for $Live(m)$ is to recognize what is considered a safety property.

For the record, at this point I've begun to grow a bit skeptical of having a liveness classification that is defined simply as being in opposition to a safety property, but I'll leave that one to the philosophers.

Anyway here's the definition given:

$(Live(m)) = L(m) \cup (S^{\omega} - L(cl(m)))$

This is conspicuously similar to just saying "and liveness is everything else!" but, of course, there's nothing wrong with that.

Anyway, in plain English, it says liveness is everything else. But how to build $Live(m)$?

4.3 Building *Live(m)*

For the deterministic case, it is as simple as taking m but sending all previously unaccepted transitions to a new "trap" state that then loops into itself on any input and is accepting. Easy.

For the non-deterministic case, it gets a bit more exciting. The paper gets a bit bogged down here, but the main takeaway is that it is m augmented with a trap state. Don't overthink this either.

A lemma formalizes all of this.

4.4 Verifying *Live(m)*

There's two parts of this. In the paper, these are Theorems 4 and 5.

1. Convince ourselves that $Live(m)$ is a liveness property.

This is easy. $Live(m)$ has no undefined transitions, so its closure clearly accepts everything. This requires a bit more book-keeping in non-deterministic case, but the central thrust is the same.

2. Convince ourselves that $Live(m)$, with $Safe(m)$, is m.

Let's rehash the formal specification here because this theorem is so important.

$L(m) = L(Safe(m)) \cap L(Live(m))$

Recall that $L(Live(m))$ was basically defined to be everything in $L(m)$ except things in $L(Safe(m))$:

$(Live(m)) = L(m) \cup (S^{\infty} - L(cl(m)))$

Then just follow through the logic. It's especially clear when drawing a picture.

The paper then has an extended example that shows partial correctness as $Safe(m)$ and termination as $Live(m)$ form total correctness.

5 Proving Properties

The goal is to get to program verification. So how do we do that? Reverse engineer proof obligations from Buchi automata (deterministic only now).

Consider automaton m for property P. Consider a program π, and recall that the execution of such a program would be represented as an execution σ of infinite program states s_i.

To show π satisfies m, *correspondence invariants* denoted as C_i are used. Recall Q is the set of automaton states in m.

$\forall q_i \in Q, \exists C_i$

$C_i \Leftrightarrow s_i \in \sigma(\pi) \wedge q_i \in \delta(q_{prev}, s_{i-1})$

This means that, if m enters q_i upon reading s, s satisfies C_i.

C_i's are defined inductively.

5.1 Building C_i's

Base case:

Consider the first transitions of the program and the automaton.

The program π begins in state s_o which satisfies $Init_{\pi}$.

The automaton m begins in state q_0 and transitions to q_j.

Now, since m transitions to q_j upon reading s_0, s_0 must satisfy the transition predicate (remember those?) T_{0j}. This predicate captures the presence of this transition in the transition function of m, denoted δ.

This builds up to the base case:

Correspondence Basis: $\forall q_j \in Q : (Init_{\pi} \wedge T_{0j} \Rightarrow C_j)$

Inductive case:

Let m enter q_i upon reading s_k with $k < K$ so s_k satisfies C_i.

Now consider m is in q_i and upon reading s_K enters q_j.

By inductive hypothesis, we have s_{K-1} satisfies C_i and s_K satisfies T_{ij}.

Now, for C_j to hold, we need only that there exists an atomic action, which are the actions that transition s_i to s_{i+1} in σ denoted α, such that $\{C_i\}\alpha\{T_{ij} \Rightarrow C_j\}$.

What does this means?

Well, α goes from s_{K-1} to s_K. C_i is necessarily true for s_{K-1} by hypothesis. T_{ij} is true if this transition would be accepted by m. So then C_j must be true...

Correspondence Induction: $\forall \alpha_i, \forall q_i \in Q, \{C_i\}\alpha\{\wedge_{q_j \in Q} T_{ij} \Rightarrow C_j\}$.

This says the same thing, only for all transitions instead of a specific one.

5.2 Does m satisfy P?

To satisfy P, every σ of π must be accepted by m. This can fail if...

1) m attempts an undefined transition

2) m never reaches an accepting state.

These can be shown to be impossible.

5.3 Transitions

Is there a valid first transition of m?

Transition Basis: $Init_\pi \Rightarrow \vee_{q_j \in Q} T_{0j}$

The inductive step is similar to correspondence.

Transition Induction: $\forall \alpha_i, \forall q_i \in Q, \{C_i\}\alpha\{\vee_{q_j \in Q} T_{ij} \Rightarrow C_j\}$.

5.4 Acceptance

A "Reject Knot" denoted κ is a strongly connected subset of Q that contains no accepting states.

All σ's are accepted if none of them are restricted to non-accepting states, that is, trapped in a reject knot.

To show this, a "variant function" is created, denoted v_κ. It maps automaton and program states to a well-founded set.

Remember well-foundedness? It means all subsets have minimal elements, like the natural numbers.

To prevent being trapped in a reject knot, require that:

$v_\kappa(q, s) = 0 \Rightarrow q \notin \kappa$

This is captured by correspondence invariants as follows:

Knot Exit: $\forall q_i \in \kappa : v_\kappa(q_i) = 0 \Rightarrow \neg C_i$

Then, ensuring an exit is as simple as ensuring each action of the program while in κ decreases v_κ:

Knot Variance: $\forall \alpha, \forall q_i \in \kappa, \{C_i \wedge 0 < v_\kappa(q_i) = x\}\alpha\{\wedge_{q_j \in \kappa}(T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < x\}$

6 All Together

Basis and Induction across Correspondence and Transition, and Knot Exit and Variance take on three forms:

CB, TB, KE are predicate logic.

CI and TI prove invariance of assertions.

KV requires well-foundedness.

6.1 Back to Safety and Liveness

$Safe(m)$ cannot have reject knots because it only has accept states. Easy!

Then there's only invariance arguments, and if $m = Safe(m)$ then safety only requires invariance arguments.

$Live(m)$ cannot have illegal transitions by definition, so it satisfies transitions trivially.

This amounts to a well-foundedness argument on knots!