Formal Verification of A Market Making Algorithm

By Kanghyuk Lee (kl3768) and Arjun Somekawa (as7423)

Introduction

In the financial industry, market makers are firms that provide liquidity to the market by buying and selling securities. A bid is the price at which a market maker is willing to buy a security, and an ask is the price that it is willing to sell at. The difference between a bid and ask price is known as the bid-ask spread, and it is the source of profit for market makers. For example, a security trading in the open market at $100 may have a bid price of $99.95 and an ask price of $100.05, which could translate into a profit of $0.10 per share if a share can be bought and sold. Currently, the majority of market making is accomplished through high-frequency trading (HFT) firms using automated, low-latency algorithms.

As one could imagine, there is a heavy cost associated with errors in a market making algorithm ranging from not being able to provide liquidity to the market to losing potentially millions or even billions of dollars due to mispricing a security. In 2012, “Knight Capital lost $440 million in 30 minutes” due to “poor software development.” The error was within a “high-frequency trading algorithm designed to buy and sell massive amounts of stock in a short period of time” [1]. Due to the incident, Knight Capital Group, which was one of the largest market makers in the US, had its stock decrease in value by over 70% within the next few days of the incident, and the firm was bought out by another market maker. In order to prevent such issues, our project aims to highlight formal verification as a viable and necessary tool for testing.

Objective

To show that formal verification is a viable method for ensuring the reliability of market making algorithms, we implemented our own simplified market making algorithm and verified several properties using formal verification tools. The market-making algorithm being verified is a simplified, single-stock bid–ask quoting strategy. At each discrete time step, the market midprice is updated using Gaussian noise to model random price movements of the market. Given the updated price and current inventory, the algorithm computes desired bid and ask quotes, places new orders or requests cancellations subject to inventory limits, and then simulates executions of active orders. Active orders may probabilistically fill based on their aggressiveness relative to the midprice, after which any pending cancel requests are resolved as either fill-before-cancel or confirmed cancellation. Inventory, cash, and mark-to-market profit and loss metrics are updated accordingly. The model assumes the following: at most one fill per side can occur per time step, each fill is for a single share, bid and ask orders are treated independently, and any active order remains live until it is either filled or canceled, with cancel requests guaranteed to resolve within the same step.

The goal of our market making algorithm is not to serve as an industry standard, but instead, to serve as a simplified system we can abstract to serve as a model for our formal verification methods. The objective of this project is to see whether formal verification can be used to identify the aforementioned crippling errors in the market, and what strengths and weaknesses come with certain approaches. As such, the properties we will be assessing are a follows:

Safety properties

- AG (bid < ask) : At all times, the bid price is strictly less than the ask price. If bid is greater than the ask, it means that the algorithm is willing to buy a security and sell it for less than the amount it paid for it, which would lead to arbitrage loss as investors would take advantage by continuously selling the security and buying for a lower price.
- AG (net position < max limit) : The inventory should not hold more than a certain number of positions for each security to limit risk. Net position is defined as the total number of shares bought subtracted by the total number of shares sold short.

Liveness properties

- AG (market data received → AF (quote posted)) : When new market data is received, the algorithm should eventually post new bid/ask quotes.
- AG (cancel sent → AF (filled ∨ canceled)) : If a cancel request is sent out, either the order should eventually get filled before cancellation or receive a cancel confirmation.

This may not be an exhaustive list of properties that need to be assessed in a real market environment, but we believe these are key properties of the algorithm that could potentially lead to large losses if violated. We assessed if these properties are satisfied through symbolic model checkers nuXmv and PRISM, and if not, why these properties were violated.

Defining our State Variables and Atomic Propositions

After creating our market making algorithm, we analyzed and abstracted it to identify our key atomic proposition and state variables. For the purposes of brevity, and our aforementioned project objective, we will be defining each of these elements in an abstract manner unbound by any C++ principles. If you wish to explore our market making algorithm in further detail, you can find it in our directory as market-maker.cpp.

1. mid: The current mid-price of the market, representing the fair price around which quotes are placed.
2. bid: The bid quote price currently posted by the market maker.
3. ask: The ask quote price currently posted by the market maker.
4. inv: The market maker's current net inventory.
5. bid_state: Represents the current lifecycle state of the bid order.

6. market_data: Indicates whether new market data has arrived and requires a quote update.
7. quote_posted: Indicates whether a quote has been posted in response to the most recent market data.
8. pending_quote: A liveness flag indicating that market data has arrived and a quote must eventually be posted.
9. pending_cancel: A liveness flag indicating that a cancel has been sent and must eventually resolve as either a fill or cancel confirmation.

Modelling Assumptions and Abstractions for nuXmv

It is key to note that our nuXmv model checker does not analyze or execute the C++ implementation of our market making algorithm directly. Although this may sound intuitive to a seasoned user of SMV, it should be clarified that our model checker operates on a formal, finite-state abstraction of our market making algorithm that we manually construct in the file market-maker.smv. Our abstraction allows us to capture the essential logical structure and control flow of the market-making algorithm while intentionally omitting low-level implementation details.

In our model, market behaviors such as price movements, market data arrival, order fills and cancel confirmations are represented non-deterministically. By doing so, we attempt to mirror reality as many events in the market are controlled by external environmental factors and cannot be predicated nor enforced by a seasoned market maker. By modelling these behaviors non-deterministically, our model checker is able to explore all possible behaviors, including potential worst-case scenarios like the Knight Capital incident.

Additionally, it should be noted that several aspects of real-world market making, such as latency or probabilistic fill rates, are absent from our abstraction. This is on purpose, as the goal for our nuXmv model checker is to verify that the core logic present in market making strategies satisfies our key safety and liveness objectives under all admissible behaviors.

Constructing the nuXmv Model Checker

We constructed a symbolic model checker from nuXmv to formally verify the correctness of our market making algorithm. The nuXmv model is defined using a set of state variables and transition relations that, together, best describe all possible execution of the system.

State variables represent the internal configuration of the market maker and its environment. This includes the market mid-price, bid and ask quotes, inventory level, order lifecycle state, and control flags such as market data arrival and pending quotes. Transitions between states are

specified using next() assignments, which describe how each variable may evolve from one step to the next.

As nuXmv uses synchronous update semantics, which means that all next() expressions are evaluated based on the current state and applied simultaneously to produce the next state. This is of particular importance when modeling time-dependent behaviors such as quote postings and cancelations, as it requires careful handling to ensure that these actions are created and discharged correctly.

The lifecycle of an order is modelled using a finite enumeration that we previously defined as the bid_state. This captures transitions between inactive, active, cancel-sent, filled and cancelled states. To support the verification of our liveness properties, we explicitly track boolean variables, like pending_quotes and pending_cancel, that describe whether the system is required to eventually perform a specific action. This approach allows temporal logic properties to be expressed and verified directly in terms of these boolean values.

Lastly, non-determinism is used extensively in the model to represent choices made by the environment, such as whether market data arrives or whether a cancel resolves as a fill or a cancellation. In contrast to probabilistic models (which we will discuss later), nuXmv takes the worst path in all non-deterministic choices, therefore ensuring that our properties hold under all possible behaviors.

Results of the Bounded Model Checking for nuXmv

In our assessments, we primarily relied on bounded model checking to verify our objective properties. Using nuXmv's SAT-based bounded model checking engine, we verified all specified properties with a bound of 30 steps. No counter-examples were observed for our safety or liveness properties.

The use of BMC proved especially effective when debugging and refining our model. In fact, earlier iterations of our model revealed several counterexamples, particularly for liveness properties, which exposed modelling errors such as those aforementioned time-dependent events being lost if a cancellation occurred mid-process. These counterexamples were incredibly useful in refining our model and identifying situations where market data was lost and required actions were not properly carried out. Once these issues were corrected, our bounded model checker confirmed that the refined model satisfied all specified properties within the given bounds.

Below is a previous log of the aforementioned issue we found.

```
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- specification  G (Pending_quote ->  F Quote_posted)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
```

If you wish to run the bounded model checker yourself, you can do so with the following commands:

1. read_model -i market-maker.smv
2. go_bmc (you may ignore any initialization warnings as it doesn't affect the run)
3. check_ltlspec_bmc -k 30 (you can change 30 to your chosen bound)

You may find additional example logs of our bounded runs in our example directory.

Results of the Unbounded Model Checking for nuXmv

In our earlier attempts of creating our nuXmv model, when it was in an incredibly simple state, we were able to run unbounded property verification to completion. Below is an example of such a run.

```
nuXmv > go
nuXmv > check_ctlspec
-- specification AG bid < ask  is true
-- specification AG (marketDataReceived -> AF quotePosted)  is true
-- specification AG ((cancelSent & orderLive) -> AF (filled | cancelConfirmed))  is true
-- specification AG inventory <= maxLimit  is true
nuXmv > quit
```

However, as we expected, the more complex our model became, the more difficult it became to run our model-checker to completion. Due to our large integer domains for state variables like inventory or mid prices, and the large amount of non-deterministic transitions, our unbounded model proved to be rather computationally expensive.

As a result, our unbounded checks require substantial computation time to reach completion, and runs on more recent iterations of our model do not terminate within a reasonable time frame. You may find examples of our unbounded runs in the examples directory. While we were able to use it to observe and prove the CTL properties, such analysis took more than 24 hours to finish. This

issue, however, was one of the core-reasons we explored bounded model-checking, and we discuss its implications in the next section.

Strengths and Weaknesses for nuXmv for Verifying Market Making Algorithms

NuXmv seems to excel at verifying safety and liveness properties. Its exhaustive exploration of all non-deterministic behaviors allows it to identify edge-case scenarios that would likely be missed by simulation-based testing. The explicit modelling of our algorithm's actions makes it well suited for exploring responsiveness and correctness. And, of course, we should not understate how useful it was in producing counterexample traces that helped us identify issues found in our code base or our model.

However, despite its many strengths, nuXmv is severely lacking in many areas that modern market making schemes would require. As discussed in the previous section, the large numeric domains quickly lead to a state explosion. It is frankly impractical for market makers to use unbounded verification with our nuXmv model. Furthermore, our attempt leaves room for severe debugging issues, as manually abstracting the model from its implementation introduces the risk of a working model proving a bug-filled implementation correct. And, most importantly, our model does not support any probabilistic reasoning. By taking an adversarial approach with our nuXmv model, we fail to support important market making features, such as analyzing expected profits, fill probabilities or risk distribution. That is why we attempted to minimize these weaknesses by introducing a secondary modeling tool, PRISM.

PRISM

PRISM, which stands for Probabilistic Symbolic Model Checker, allows users to formally verify probabilistic abstract models. The four types of mathematical models that PRISM uses are DTMC (Discrete-Time Markov Chain), CTMC (Continuous-Time Markov Chain), MDP (Markov Decision Process), and PTA (Probabilistic Timed Automata). The aforementioned mathematical models allow PRISM to have transitions involving nondeterminism, meaning multiple transitions can be enabled at once without PRISM controlling which one is chosen, and probability. To formally verify probabilistic transitions, PRISM uses PCTL, which stands for Probabilistic Computation Tree Logic. PCTL essentially behaves like CTL with a probability operator added.

For the formal verification of the market making algorithm, the mathematical model chosen to represent the abstract model is MDP, which works with both nondeterminism and probability. As PRISM is a symbolic model checker, it cannot work directly with source code, and the algorithm must be represented as an abstract model using the PRISM language. For PRISM to verify the

aforementioned safety and liveness properties, PCTL equivalents of the properties were constructed as shown below:

Safety

1) P>=1 [G (bid < ask)]
2) P>=1 [G (inv <= MAX_LIMIT & inv >= -MAX_LIMIT)]

Liveness

1) P>=1 [G (pending_bid_quote => (true U !pending_bid_quote))]
2) P>=1 [G (pending_ask_quote => (true U !pending_ask_quote))]
3) P>=1 [G (pending_bid_cancel => (true U !pending_bid_cancel))]
4) P>=1 [G (pending_ask_cancel => (true U !pending_ask_cancel))]

The abstract model for the market making algorithm was further simplified from the source code to keep it bounded. For example, in the source code the midprice is continuous with Gaussian noise updating it randomly, inventory can get up to ±10,000 shares, and fills can occur based on probabilities derived from an exponential intensity function of price distance from the mid price. However, in the abstract model, the mid price has a discrete range of [90, 10], limits inventory to [-100, 100], and fixes bid and ask quotes deterministically to mid price ± 1. Such limits are necessary to prevent the state space explosion problem. Furthermore, the fill uncertainty is simplified to explicit probabilistic branches: 0.5 fill and 0.5 cancel.

Results of PRISM

The initial results of the verification of the safety and liveness properties are shown below in the "Bad Model" table. The reason that three properties were violated was due to the fact that the abstract system was modeled incorrectly. The issues in the model stemmed from two problems: incorrect obligation generation and insufficient enforcement of safety constraints. First, the model was able to create pending quotes when inventory limits had been reached, which is not allowed. After the quote was generated, the obligation could never be fulfilled, leading to the liveness property violation. This error was fixed by making the quote generation depend on inventory bounds, ensuring quote obligations were only created when inventory was within limit instead of whenever new market data had arrived, which was the original case. Secondly, the model allowed states in which the bid was not less than ask, violating the safety property and leading to potential loss through arbitrage. The reason for this violation was because bid and ask updates were not consistently tied to mid price in a way that preserved ordering across all transitions. The fix was to ensure that bids and asks are always derived symmetrically from the mid price. By implementing the fixes to both issues, all of the properties were satisfied as shown in the "Good Model" table. It is important to note that several properties were initially violated

due to poor modeling and not the incorrectness of the source code, portraying a key limitation of symbolic model checking to be discussed in detail later.

“Bad Model”

Property	Result
P>=1 [G (bid < ask)]	False
P>=1 [G (inv <= MAX_LIMIT & inv >= -MAX_LIMIT)]	True
P>=1 [G (pending_bid_quote => (true U !pending_bid_quote))]	False
P>=1 [G (pending_ask_quote => (true U !pending_ask_quote))]	False
P>=1 [G (pending_bid_cancel => (true U !pending_bid_cancel))]	True
P>=1 [G (pending_ask_cancel => (true U !pending_ask_cancel))]	True

“Good Model”

Property	Result
P>=1 [G (bid < ask)]	True
P>=1 [G (inv <= MAX_LIMIT & inv >= -MAX_LIMIT)]	True
P>=1 [G (pending_bid_quote => (true U !pending_bid_quote))]	True
P>=1 [G (pending_ask_quote => (true U !pending_ask_quote))]	True
P>=1 [G (pending_bid_cancel => (true U !pending_bid_cancel))]	True
P>=1 [G (pending_ask_cancel => (true U !pending_ask_cancel))]	True

Limitations of PRISM

A key limitation of PRISM and symbolic model checking in general is that an abstract representation of the model being verified must be developed. If there are differences between the source code and the abstract model, whether it be due to human error or limitations of the model checker, the results of the verification may not be applicable to the source code being tested. An example of such limitation we experienced is the inability to represent continuous variables like Gaussian noise for updating mid-price stochastically. In addition, the abstract model must be bounded using discrete variable domains; otherwise, a state space explosion

would occur and exhaustive checking would be computationally infeasible or impossible. As a result, formally verifying systems with continuous or large finite domains is challenging as it requires aggressive discretization.

Another key limitation of PRISM is that it does not provide counterexamples for probabilistic properties directly. When a property is verified, PRISM will output a result that shows that the property is satisfied in X of N initial states. In the model maker model, there was one initial state so a satisfied property resulted in the following: “Property satisfied in 1 of 1 initial states”, while a dis-satisfied property resulted in the following: “Property satisfied in 0 of 1 initial states.” The results are showing whether the property fails or not starting from the initial state. As stated previously, the limitation is that when a property fails, a counterexample is not directly generated. In order to find the counterexample, one must generate states (.sta) and transitions (.tra) files to manually trace the path of the counterexample.

Although functionally harmless, a few inconvenient syntactic limitations of the PRISM language were recognized. The first limitation is that the temporal operators cannot be nested inside other temporal operators. An example that occurred for the market-making algorithm was that F (eventually) cannot appear directly inside a G (always) as such: G (a => F b). The reason for this limitation is that PRISM expects only a state formula within a path operator. A solution to this limitation is to represent eventually (F Φ) using until (true U Φ) because U consists of two state formulas, which does not have nesting. The second limitation is that refactoring of shared code is not possible as each probabilistic branch must explicitly list all state variable updates, leading to much repetition and potentially technical debt. For example, in the code snippet of the market data update transition below, there is a 50% chance of the mid price increasing and 50% chance of mid price decreasing with the state variable updates being the same between both branches with the exception of mid price change. Since the repetition of updates grows linearly with the number of probabilistic branches per transition, a method of factoring out repetitive code could prove useful and elegant for the PRISM community.

```
[md_update]
   bid_state = 0 & ask_state = 0 ->
     0.5 : (market_data'   = true)
        & (mid'          = min(MID_MAX, mid + 1)) // price went up
        & (bid_quote_posted'  = false)
        & (pending_bid_quote' = true)
        & (ask_quote_posted'  = false)
        & (pending_ask_quote' = true)
```

```
+ 0.5 : (market_data'   = true)
    & (mid'           = max(MID_MIN, mid - 1)) // price went down
    & (bid_quote_posted'  = false)
    & (pending_bid_quote' = true)
    & (ask_quote_posted'  = false)
    & (pending_ask_quote' = true);
```

Use of Artificial Intelligence [2]

Although most view AI as an alternative method to formal verification, our experience has taught us that it can be used hand-in-hand with formal verification tools, especially symbolic model checkers. As discussed, one of the key limitations of symbolic model checking is keeping the structural integrity of the source code or system intact during the construction of the abstract model. Since the implementation of the abstract is a manual process, it is inherently an error-prone process and could introduce discrepancies between the abstract model and source code, potentially limiting the applicability of the formal verification results. An example of this was shown for the bad PRISM model, as it led to violations of several properties when they were not applicable to the source code. To improve the abstract modeling of source code, AI was utilized in two ways: identify inaccuracies in modeling and find suggestions on improving the efficiency of the model. For the first method, ChatGPT 5, a mainstream LLM, was used to compare the source code and the abstract model in order to find inaccurate representations of the source code. By doing so, we were able to find two significant issues discussed previously that caused several properties to inaccurately be violated. Secondly, the LLM was used to identify insignificant aspects of the model with respect to the properties being verified as well as to assess how variable domains could be tightened to reduce the size of the state space.

Static Analysis Tool

For our project, we decided to compare our method of formal verification to a potential alternative, static analysis. Our static analysis tool, clang-tidy, was incredibly useful when it came to detecting dangerous code: uninitialized variables, dead code, violation of coding standards, etc. Most importantly, clang-tidy analyzed the C++ file itself, rather than the abstracted model, which we described as a weakness of our implementation of formal verification.

However, static analysis did have limitations as well. These analyzers were focused on solving issues regarding code structure or data flow, and not temporal behavior in our system. Using it,

we couldn't prove any of our objectives, such as our liveness properties, for example. Overall, it appears that both tools should be viewed complementarily. Static analysis seems well suited for validation code quality and correctness to some degree, while formal verification seems perfect for verifying high level behavior and temporal correctness.

Conclusion

This project demonstrates that formal verification is an effective method for checking safety and liveness properties of market-making algorithms, although there are a few limitations. Using nuXmv and PRISM, we were able to verify key properties of a simplified market-making algorithm and show how symbolic model checking can expose subtle design flaws that could result in severe financial losses. With that being said, the biggest limitation was that both model checkers rely on manually constructed abstract models, making them vulnerable to poor modeling errors. To address this issue, we explored the use of AI as a complementary tool for formal verification. By comparing the abstract model with the source code and identifying inefficiencies and inaccuracies, AI helped improve model integrity and reduce state-space size. Lastly, our comparison with static analysis further shows that these techniques are best utilized together: static analysis for low-level code quality and formal verification for high-level guarantees.

As such, we recommend formal verification of financial algorithms and software but with a grain of salt. As such algorithms consider many variables, continuous domains that can lead to exploding state-space problems, and randomness, we believe it is infeasible to formally verify a real-world market making algorithm. Instead, we believe that formal verification is best suited for verifying modular components or critical subsystems, where abstraction can be tightly controlled and unexpected behavior can be effectively prevented. Taking this a step further, we argue that formal verification need not only be applied to existing systems but can instead be used to define abstract, verified models that real-world implementations are subsequently built upon.

Division of Labor

Arjun worked on the nuXmv implementation and testing and the introduction, nuXmv, and static analysis sections of the final paper. Chris worked on the PRISM implementation, the starter code, and the PRISM, AI analysis, and conclusion sections of the final paper.

References

[1] https://www.cio.com/article/286790/software-testing-lessons-learned-from-knight-capital-fiasco.html?utm_source=chatgpt.com

[2] https://chatgpt.com/share/694b6ea0-d444-8002-948d-dc90446fb33b