Formal Specification of Constant Product ($x \times y = k$) Market Maker Model and Implementation

Yi Zhang, Xiaohong Chen, and Daejun Park

Runtime Verification, Inc.

October 20, 2018

Abstract

We formalize the constant product market maker model (aka, $x \times y = k$ model) [2], and formally analyze the integer rounding errors of the implementation in the Uniswap smart contract [1].

1 State Transition System Model

We formalize the market maker model as a state transition system, where the state represents the current asset of the exchange, and the transition represents how each function updates the state.

We define the exchange state as a tuple (e, t, l), where e is the amount of ether (in wei), t is the number of (exchange) tokens, and l is the amount of total liquidity (i.e., the total supply of UNI tokens).

2 Updating Liquidity

We formalize two functions `addLiquidity` and `removeLiquidity` that mints and burns the liquidity, respectively. We first formalize their mathematical definition, $\texttt{addLiquidity}_{\texttt{spec}}$ and $\texttt{removeLiquidity}_{\texttt{spec}}$, that uses the real arithmetic. Then, we formalize their implementation, $\texttt{addLiquidity}_{\texttt{code}}$ and $\texttt{removeLiquidity}_{\texttt{code}}$, that uses the fixed-point arithmetic (i.e., an approximate real arithmetic), and analyze the approximation errors due to the fixed-point rounding.

2.1 Minting Liquidity

An investor can mint liquidity by depositing both ether and token.

2.1.1 `addLiquidity`$_{\texttt{spec}}$

We formulate the mathematical definition of minting liquidity.

Definition 1. *$\mathit{addLiquidity}_{spec}$ takes as input $\Delta e > 0$ and updates the state as follows:*

$$(e, t, l) \xrightarrow{\mathit{addLiquidity}_{spec}(\Delta e)} (e', t', l')$$

where

$$\begin{aligned} e' &= (1 + \alpha)e \\ t' &= (1 + \alpha)t \\ l' &= (1 + \alpha)l \end{aligned}$$

and $\alpha = \dfrac{\Delta e}{e}$.

Here, an investor deposits both Δe ether (wei) and $\Delta t = t' - t$ tokens, and mints $\Delta l = l' - l$ liquidity. The invariant is that the ratio of $e : t : l$ is preserved, and $k = e \times t$ increases, as formulated in the following theorem.

Theorem 1. *Let $(e, t, l) \xrightarrow{\mathit{addLiquidity}_{spec}(\Delta e)} (e', t', l')$. Let $k = e \times t$ and $k' = e' \times t'$. Then, we have the following:*

1. $e : t : l = e' : t' : l'$
2. $k < k'$
3. $\dfrac{k'}{k} = \left(\dfrac{l'}{l}\right)^2$

2.1.2 `addLiquidity`$_{\texttt{code}}$

In the implementation using the integer arithmetic, we have to approximate t' and l' that are not an integer. We formulate the approximation.

Definition 2. *$\mathit{addLiquidity}_{code}$ takes as input an* integer *$\Delta e > 0 \in \mathbb{Z}$ and updates the state as follows:*

$$(e, t, l) \in \mathbb{Z}^3 \xrightarrow{\mathit{addLiquidity}_{code}(\Delta e)} (e'', t'', l'') \in \mathbb{Z}^3$$

where

$$\begin{aligned} e'' &= e + \Delta e &&= (1 + \alpha)e \\ t'' &= t + \left\lfloor \frac{\Delta e \times t}{e} \right\rfloor + 1 &&= \lfloor (1 + \alpha)t \rfloor + 1 \\ l'' &= l + \left\lfloor \frac{\Delta e \times l}{e} \right\rfloor &&= \lfloor (1 + \alpha)l \rfloor \end{aligned}$$

and $\alpha = \dfrac{\Delta e}{e}$.[1]

Theorem 2. *Let* $(e,t,l) \xrightarrow{\mathit{addLiquidity}_{spec}(\Delta e)} (e',t',l')$. *Let* $(e,t,l) \xrightarrow{\mathit{addLiquidity}_{code}(\Delta e)} (e'',t'',l'')$. *Let* $k = e \times t$, $k' = e' \times t'$, *and* $k'' = e'' \times t''$. *Then, we have:*

$$\begin{aligned} e'' &= e' \\ t'' &= \lfloor t' \rfloor + 1 \\ l'' &= \lfloor l' \rfloor \end{aligned}$$

and

1. $e < e' = e''$
2. $t < t' < t'' \leq t' + 1$
3. $l' - 1 < l'' \leq l'$
4. $k < k' < k''$
5. $\left(\frac{l''}{l}\right)^2 < \frac{k''}{k}$

That is, t' is approximated to a larger value t'' but no larger than 1 ($0 < t'' - t' \leq 1$), while l' is approximated to a smaller value l'' but no smaller than 1 ($-1 < l'' - l' \leq 0$). This approximation scheme implies that k' is approximated to a strictly larger value k'', which is desired. This means that an investor may deposit more (up to 1) tokens than needed, but may mint less (up to -1) liquidity than the mathematical value.

2.2 Burning Liquidity

An investor can withdraw their deposit of ether and token by burning their share of liquidity.

2.2.1 `removeLiquidity`$_{\texttt{spec}}$

We formulate the mathematical definition of burning liquidity, being dual to minting liquidity.

Definition 3. $\mathit{removeLiquidity}_{spec}$ *takes as input* $0 < \Delta l < l$ *and updates the state as follows:*

$$(e,t,l) \xrightarrow{\mathit{removeLiquidity}_{spec}(\Delta l)} (e',t',l')$$

where

$$\begin{aligned} e' &= (1-\alpha)e \\ t' &= (1-\alpha)t \\ l' &= (1-\alpha)l \end{aligned}$$

[1]The second column represents the computation model using the integer division with truncation. That is, for example, t'' is computed by `t + ((de * t) / e) + 1` where `de` is Δe and `/` is the integer division with truncation.

and $\alpha = \dfrac{\Delta l}{l}$.

Here, an investor burns Δl liquidity, and withdraws $\Delta e = e - e'$ ether (wei) and $\Delta t = t - t'$ tokens. The invariant is dual to that of minting liquidity.

Theorem 3. *Let* $(e, t, l) \xrightarrow{\texttt{removeLiquidity}_{spec}(\Delta l)} (e', t', l')$. *Let* $k = e \times t$ *and* $k' = e' \times t'$. *Then, we have the following:*

1. $e : t : l = e' : t' : l'$
2. $k' < k$
3. $\dfrac{k'}{k} = \left(\dfrac{l'}{l}\right)^2$

The duality of $\texttt{addLiquidity}_{\texttt{spec}}$ and $\texttt{removeLiquidity}_{\texttt{spec}}$ is formulated in the following theorem.

Theorem 4. *If* $\texttt{addLiquidity}_{spec}$ *is subsequently followed by* $\texttt{removeLiquidity}_{spec}$ *as follows:*

$$(e_0, t_0, l_0) \xrightarrow{\texttt{addLiquidity}_{spec}(\Delta e)} (e_1, t_1, l_1) \xrightarrow{\texttt{removeLiquidity}_{spec}(\Delta l)} (e_2, t_2, l_2)$$

and $\Delta l = l_1 - l_0$, *then we have:*

1. $e_0 = e_2$
2. $t_0 = t_2$
3. $l_0 = l_2$

2.2.2 $\texttt{removeLiquidity}_{\texttt{code}}$

In the implementation using the integer arithmetic, we have to approximate e' and t' that are not an integer. We formulate the approximation.

Definition 4. $\texttt{removeLiquidity}_{\texttt{code}}$ *takes as input an* integer $0 < \Delta l < l$ *and updates the state as follows:*

$$(e, t, l) \in \mathbb{Z}^3 \xrightarrow{\texttt{removeLiquidity}_{code}(\Delta l)} (e'', t'', l'') \in \mathbb{Z}^3$$

where

$$\begin{aligned} e'' &= e - \left\lfloor \frac{\Delta l \times e}{l} \right\rfloor &&= \lceil (l - \alpha) e \rceil \\ t'' &= t - \left\lfloor \frac{\Delta l \times t}{l} \right\rfloor &&= \lceil (1 - \alpha) t \rceil \\ l'' &= l - \Delta l &&= (1 - \alpha) l \end{aligned}$$

and $\alpha = \dfrac{\Delta l}{l}$.

Theorem 5. *Let* $(e, t, l) \xrightarrow{removeLiquidity_{spec}(\Delta l)} (e', t', l')$. *Let* $(e, t, l) \xrightarrow{removeLiquidity_{code}(\Delta l)} (e'', t'', l'')$. *Let* $k = e \times k$, $k' = e' \times k'$, *and* $k'' = e'' \times t''$. *Then, we have:*

$$
\begin{aligned}
e'' &= \lceil e' \rceil \\
t'' &= \lceil t' \rceil \\
l'' &= l'
\end{aligned}
$$

and

1. $e' \le e'' \le e$
2. $t' \le t'' \le t$
3. $l'' = l' < l$
4. $k' \le k'' \le k$
5. $\left(\frac{l''}{l}\right)^2 \le \frac{k''}{k}$

That is, e' and t' are simply approximated to their ceiling $e'' = \lceil e' \rceil$ and $t'' = \lceil t' \rceil$, which satisfies the desired property $k'' \le k$. In other words, an investor may withdraw less amounts of deposit ($e - \lceil e' \rceil$ and $t - \lceil t' \rceil$) than the mathematical values ($e - e'$ and $t - t'$).

One of the desirable properties is that an investor cannot make a "free" money by exploiting the integer rounding errors, which is formulated below.

Theorem 6. *If* $addLiquidity_{code}$ *is subsequently followed by* $removeLiquidity_{code}$ *as follows:*

$$
(e_0, t_0, l_0) \xrightarrow{addLiquidity_{code}(\Delta e)} (e_1, t_1, l_1) \xrightarrow{removeLiquidity_{code}(\Delta l)} (e_2, t_2, l_2)
$$

and $\Delta l = l_1 - l_0$, *then we have:*

1. $e_0 < e_2$
2. $t_0 < t_2$
3. $l_0 = l_2$

2.3 `getInputPrice`

In this section, we present a formal specification of `getInputPrice`. Suppose there are two kinds of tokens in the pool: A and B. $t_A (> 0)$ (or $t_B (> 0)$) represents the total amount of token A (or token B) in the pool.

2.3.1 $\texttt{getInputPrice}_{\texttt{spec}}$

$\texttt{getInputPrice}_{\texttt{spec}}$ takes $\Delta t_A (> 0)$, t_A and t_B as input and returns the amount of token B that Δt_A can exchange for.

$$t'_A = (1+\alpha)t_A$$
$$t'_B = \frac{1}{1+\alpha\beta}t_B$$
$$\texttt{getInputPrice}_{\texttt{spec}}(\Delta t_A, t_A, t_B) = t_B - t'_B = \frac{\alpha\beta}{1+\alpha\beta}t_B$$

and $\alpha = \dfrac{\Delta t_A}{t_A}$ and $\beta = \dfrac{997}{1000}$.

Theorem 7. *Let $k = t_A * t_B$ and $k' = t'_A * t'_B$, and we have the following property:*

1. $t_A < t'_A$
2. $t'_B < t_B$
3. $k < k'$

2.3.2 $\texttt{getInputPrice}_{\texttt{code}}$

$\texttt{getInputPrice}_{\texttt{code}}$ takes **integers** $\Delta t_A (> 0)$, t_A and t_B as input and returns the **maximum integer** amount of token B that Δt_A can exchange for.

$$t''_A = t_A + \Delta t_A$$
$$\texttt{getInputPrice}_{\texttt{spec}}(\Delta t_A, t_A, t_B) = \lfloor t_B - t'_B \rfloor = \left\lfloor \frac{997 * \Delta t_A * t_B}{1000 * t_A + 997 * \Delta t_A} \right\rfloor$$
$$t''_B = t_B - \texttt{getInputPrice}_{\texttt{spec}}(\Delta t_A, t_A, t_B) = \lceil t' \rceil$$

Theorem 8. *Let $k'' = t''_A * t''_B$, and we have the following property:*

1. $t_A < t'_A = t''_A$
2. $t'_B \le t''_B \le t_B$
3. $k < k' \le k''$

2.4 `getOutputPrice`

In this section, we present a formal specification of `getOutputPrice`. Suppose there are two kinds of tokens: A and B in the pool. t_A (or t_B) represents the total amount of token A (or token B) in the pool.

2.4.1 $\texttt{getOutputPrice}_{\texttt{spec}}$

$\texttt{getOutputPrice}_{\texttt{spec}}$ takes $\Delta t_B (0 < \Delta t_B < t_B)$, t_A and t_B as input and returns the number of token A that can exchange for Δt_B.

$$\begin{aligned} t'_B &= t_B - \Delta t_B \\ \texttt{getOutputPrice}_{\texttt{spec}}(\Delta t_B, t_A, t_B) &= \left(\frac{t_A * t_B}{t_B - \Delta t_B} - t_A \right) * \frac{1000}{997} \\ t'_A &= t_A + \texttt{getOutputPrice}_{\texttt{spec}}(\Delta t_B, t_A, t_B) \end{aligned}$$

Theorem 9. *Let $k = t_A * t_B$ and $k' = t'_A * t'_B$, and we have the following property:*

1. $t_A < t'_A$
2. $t'_B < t_B$
3. $k < k'$

2.4.2 $\texttt{getOutputPrice}_{\texttt{code}}$

$\texttt{getOutputPrice}_{\texttt{code}}$ takes integers $\Delta t_B (0 < \Delta t_B < t_B)$, t_A, and t_B as input and returns the **minimal integer** number of token A that can exchange for Δt_B.

$$\begin{aligned} t''_B &= t_B - \Delta t_B \\ \texttt{getOutputPrice}_{\texttt{code}}(\Delta t_B, t_A, t_B) &= \left\lfloor \frac{1000 * t_A * \Delta t_B}{997 * (t_B - \Delta t_B)} \right\rfloor + 1 \\ t''_A &= t_A + \texttt{getOutputPrice}_{\texttt{code}}(\Delta t_B, t_A, t_B) \end{aligned}$$

Theorem 10. *Let $k'' = t''_A * t''_B$, and we have the following property:*

1. $t_A < t'_A < t''_A$
2. $t''_B < t_B$
3. $k < k''$

Theorem 11. *We have the following property between* $\texttt{getInputPrice}$ *and* $\texttt{getOutputPrice}$*:*

1. $\Delta t_B \leq \texttt{getInputPrice}_{\texttt{code}}(\texttt{getOutputPrice}_{\texttt{code}}(\Delta t_B, t_A, t_B), t_A, t_B)$
2. $\texttt{getOutputPrice}_{\texttt{code}}(\texttt{getInputPrice}_{\texttt{code}}(\Delta t_A, t_A, t_B), t_A, t_B) \leq \Delta t_A$

2.5 ethToToken

In this section, we present a formal specification of `ethToToken` (including swap and transfer).

2.5.1 $\texttt{ethToToken}_{\texttt{spec}}$

$\texttt{ethToToken}_{\texttt{spec}}$ takes an input $\Delta e (\Delta e > 0)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{ethToToken}_{\texttt{spec}}(\Delta e)} (e', t', l)$$

where

$$\begin{aligned} e' &= e + \Delta e \\ t' &= t - \texttt{getInputPrice}_{\texttt{spec}}(\Delta e, e, t) \end{aligned}$$

2.5.2 $\texttt{ethToToken}_{\texttt{code}}$

$\texttt{ethToToken}_{\texttt{code}}$ takes an **integer** input $\Delta e (\Delta e > 0)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{ethToToken}_{\texttt{code}}(\Delta e)} (e'', t'', l)$$

where

$$\begin{aligned} e'' &= e + \Delta e \\ t'' &= t - \texttt{getInputPrice}_{\texttt{code}}(\Delta e, e, t) = \lceil t' \rceil \end{aligned}$$

2.6 ethToTokenExact

In this section, we present a formal specification of `ethToTokenExact` (including swap and transfer).

2.6.1 $\texttt{ethToTokenExact}_{\texttt{spec}}$

$\texttt{ethToTokenExact}_{\texttt{spec}}$ takes an input $\Delta t (0 < \Delta t < t)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{ethToTokenExact}_{\texttt{spec}}(\Delta t)} (e', t', l)$$

where

$$\begin{aligned} t' &= t - \Delta t \\ e' &= e + \texttt{getOutputPrice}_{\texttt{spec}}(\Delta t, e, t) \end{aligned}$$

2.6.2 $\texttt{ethToTokenExact}_{\texttt{code}}$

$\texttt{ethToTokenExact}_{\texttt{code}}$ takes an **integer** input $\Delta t (0 < \Delta t < t)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{ethToTokenExact}_{\texttt{code}}(\Delta t)} (e'', t'', l)$$

where

$$\begin{aligned} t'' &= t - \Delta t \\ e'' &= e + \texttt{getOutputPrice}_{\texttt{code}}(\Delta t, e, t) \end{aligned}$$

2.7 `tokenToEth`

In this section, we present a formal specification of `tokenToEth` (including swap and transfer).

2.7.1 $\texttt{tokenToEth}_{\texttt{spec}}$

$\texttt{tokenToEth}_{\texttt{spec}}$ takes an input $\Delta t (\Delta t > 0)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{tokenToEth}_{\texttt{spec}}(\Delta t)} (e', t', l)$$

where

$$\begin{aligned} t' &= t + \Delta t \\ e' &= e - \texttt{getInputPrice}_{\texttt{spec}}(\Delta t, t, e) \end{aligned}$$

2.7.2 $\texttt{tokenToEth}_{\texttt{code}}$

$\texttt{tokenToEth}_{\texttt{code}}$ takes an **integer** input $\Delta t (\Delta t > 0)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{tokenToEth}_{\texttt{code}}(\Delta e)} (e'', t'', l)$$

where

$$\begin{aligned} t'' &= t + \Delta t \\ e'' &= e - \texttt{getInputPrice}_{\texttt{code}}(\Delta t, t, e) = \lceil e' \rceil \end{aligned}$$

2.8 `tokenToEthExact`

In this section, we present a formal specification of `tokenToEthExact` (including swap and transfer).

2.8.1 $\texttt{tokenToEthExact}_{\texttt{spec}}$

$\texttt{tokenToEthExact}_{\texttt{spec}}$ takes an input $\Delta e (0 < \Delta e < e)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{tokenToEthExact}_{\texttt{spec}}(\Delta e)} (e', t', l)$$

where

$$\begin{aligned} e' &= e - \Delta e \\ t' &= t + \texttt{getOutputPrice}_{\texttt{spec}}(\Delta e, t, e) \end{aligned}$$

2.8.2 $\texttt{tokenToEthExact}_{\texttt{code}}$

$\texttt{tokenToEthExact}_{\texttt{code}}$ takes an **integer** input $\Delta e (0 < \Delta e < e)$ and updates the state as follows:

$$(e, t, l) \xrightarrow{\texttt{tokenToEthExact}_{\texttt{code}}(\Delta e)} (e'', t'', l)$$

where

$$\begin{aligned} e'' &= e - \Delta e \\ t'' &= t + \texttt{getOutputPrice}_{\texttt{code}}(\Delta e, t, e) \end{aligned}$$

2.9 tokenToToken

In this section, we present a formal specification of `tokenToToken` (including swap and transfer). Suppose there are two exchange contracts A and B, whose states are (e_A, t_A, l_A) and (e_B, t_B, l_B) respectively.

2.9.1 $\texttt{tokenToToken}_{\texttt{spec}}$

$\texttt{tokenToToken}_{\texttt{spec}}$ takes an input $\Delta t_A (> 0)$ and updates the states as follows:

$$\{(e_A, t_A, l_A), (e_B, t_B, l_B)\} \xrightarrow{\texttt{tokenToToken}_{\texttt{spec}}(\Delta t_A)} \{(e'_A, t'_A, l_A), (e'_B, t'_B, l_B)\}$$

where

$$\begin{aligned} t'_A &= t_A + \Delta t_A \\ \Delta e_{A_{spec}} &= \texttt{getInputPrice}_{\texttt{spec}}(\Delta t_A, t_A, e_A) \\ e'_A &= e - \Delta e_{A_{spec}} \\ e'_B &= e_B + \Delta e_{A_{spec}} \\ \Delta t_{B_{spec}} &= \texttt{getInputPrice}_{\texttt{spec}}(\Delta e_{A_{spec}}, e_B, t_B) \\ t'_B &= t_B - \Delta t_{B_{spec}} \end{aligned}$$

2.9.2 $\texttt{tokenToToken}_{\texttt{code}}$

$\texttt{tokenToToken}_{\texttt{code}}$ takes an **integer** input $\Delta t_A (> 0)$ and updates the states as follows:

$$\{(e_A, t_A, l_A), (e_B, t_B, l_B)\} \xrightarrow{\texttt{tokenToToken}_{\texttt{code}}(\Delta t_A)} \{(e''_A, t''_A, l_A), (e''_B, t''_B, l_B)\}$$

where

$$\begin{aligned} t''_A &= t_A + \Delta t_A \\ \Delta e_{A_{code}} &= \texttt{getInputPrice}_{\texttt{code}}(\Delta t_A, t_A, e_A) \\ e''_A &= e - \Delta e_{A_{code}} \\ e''_B &= e_B + \Delta e_{A_{code}} \\ \Delta t_{B_{code}} &= \texttt{getInputPrice}_{\texttt{code}}(\Delta e_{A_{code}}, e_B, t_B) \\ t''_B &= t_B - \Delta t_{B_{code}} \end{aligned}$$

2.10 `tokenToTokenExact`

In this section, we present a formal specification of `tokenToTokenExact` (including swap and transfer). Suppose there are two exchange contracts A and B, whose states are (e_A, t_A, l_A) and (e_B, t_B, l_B) respectively.

2.10.1 $\texttt{tokenToTokenExact}_{\texttt{spec}}$

$\texttt{tokenToTokenExact}_{\texttt{spec}}$ takes an input $\Delta t_B (0 < \Delta t_B < t_B)$ and updates the states as follows:

$$\{(e_A, t_A, l_A), (e_B, t_B, l_B)\} \xrightarrow{\texttt{tokenToTokenExact}_{\texttt{spec}}(\Delta t_B)} \{(e'_A, t'_A, l_A), (e'_B, t'_B, l_B)\}$$

where

$$\begin{aligned}
t'_B &= t_B - \Delta t_B \\
\Delta e_{B_{spec}} &= \texttt{getOutputPrice}_{\texttt{spec}}(\Delta t_B, e_B, t_B) \\
e'_B &= e_B + \Delta e_{B_{spec}} \\
e'_A &= e_A - \Delta e_{B_{spec}} \\
\Delta t_{A_{spec}} &= \texttt{getOutputPrice}_{\texttt{spec}}(\Delta e_{B_{spec}}, t_A, e_A) \\
t'_A &= t_A + \Delta t_{A_{spec}}
\end{aligned}$$

2.10.2 $\texttt{tokenToTokenExact}_{\texttt{code}}$

$\texttt{tokenToTokenExact}_{\texttt{code}}$ takes an **integer** input $\Delta t_B (0 < \Delta t_B < t_B)$ and updates the states as follows:

$$\{(e_A, t_A, l_A), (e_B, t_B, l_B)\} \xrightarrow{\texttt{tokenToTokenExact}_{\texttt{code}}(\Delta t_B)} \{(e''_A, t''_A, l_A), (e''_B, t''_B, l_B)\}$$

where

$$\begin{aligned}
t''_B &= t_B - \Delta t_B \\
\Delta e_{B_{code}} &= \texttt{getOutputPrice}_{\texttt{code}}(\Delta t_B, e_B, t_B) \\
e''_B &= e_B + \Delta e_{B_{code}} \\
e''_A &= e_A - \Delta e_{B_{code}} \\
\Delta t_{A_{code}} &= \texttt{getOutputPrice}_{\texttt{code}}(\Delta e_{B_{code}}, t_A, e_A) \\
t''_A &= t_A + \Delta t_{A_{code}}
\end{aligned}$$

References

[1] Hayden Adams. Uniswap. `https://github.com/Uniswap/contracts-vyper`.

[2] Vitalik Buterin. The x*y=k market maker model. `https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers`.