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 24, 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 Formal Overview of $x \times y = k$ Model

Consider a decentralized exchange [2] that trades two tokens X and Y. Let x and y be the number of tokens X and Y, respectively, that the exchange currently reserves. The token exchange price is determined by the ratio of x and y so that the product $x \times y$ is preserved. That is, when you sell Δx tokens, you will get Δy tokens such that $x \times y = (x + \Delta x) \times (y - \Delta y)$. Thus, the price $(\Delta x / \Delta y)$ is the function of x/y. Specifically, when you trade Δx with Δy, the exchange token reserves are updated as follows:

$$
\begin{aligned}
x' &= x + \Delta x = (1+\alpha)x = \frac{1}{1-\beta}x \\
y' &= y - \Delta y = \frac{1}{1+\alpha}y \quad = (1-\beta)y
\end{aligned}
$$

where $\alpha = \dfrac{\Delta x}{x}$ and $\beta = \dfrac{\Delta y}{y}$. Also, we have:

$$
\begin{aligned}
\Delta x &= \frac{\beta}{1-\beta}x \\
\Delta y &= \frac{\alpha}{1+\alpha}y
\end{aligned}
$$

Now consider a fee for each token trade. Let $0 \leq \rho < 1$ be a fee, e.g., $\rho = 0.003$ for 0.3% fee schedule.

$$x'_\rho = x + \Delta x = (1+\alpha)x = \frac{1+\beta(\frac{1}{\gamma}-1)}{1-\beta}x$$

$$y'_\rho = y - \Delta y = \frac{1}{1+\alpha\gamma}y = (1-\beta)y$$

where $\alpha = \dfrac{\Delta x}{x}$, $\beta = \dfrac{\Delta y}{y}$, and $\gamma = 1 - \rho$. Also, we have:

$$\Delta x = \frac{\beta}{1-\beta} \cdot \frac{1}{\gamma} \cdot x$$

$$\Delta y = \frac{\alpha\gamma}{1+\alpha\gamma} \cdot y$$

Note that we have the same formula with the previous one if there is no fee, i.e., $\gamma = 1$. Also, note that the product of x and y slightly increases for each trade due to the fee. That is, $x'_\rho \times y'_\rho > x \times y$ when $\rho > 0$, while $x'_\rho \times y'_\rho = x \times y$ when $\rho = 0$ (no fee).

In the contract implementation [1] of this model, the token X denotes Ether and the token Y denotes the token to trade.

Furthermore, one can invest and divest, sharing the exchange token reserves, which we will formalize later using the concept of liquidity.

Since the implementation uses the integer arithmetic, we will also formally analyze the approximation error caused by the integer rounding, showing that the error is bounded and does not lead to violation of the critical properties[1] denoted by the mathematical model.

State Transition System 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 $\texttt{removeLiquidity}_{\texttt{code}}$,

[1] For example, it is not possible for a malicious user to make "free" money by exploiting the rounding error.

that uses the integer arithmetic, and analyze the approximation errors due to the integer 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 $\alpha = \dfrac{\Delta e}{e}$.[2]

Theorem 2. *Let* $(e, t, l) \xrightarrow{\textit{addLiquidity}_{spec}(\Delta e)} (e', t', l')$. *Let* $(e, t, l) \xrightarrow{\textit{addLiquidity}_{code}(\Delta e)} (e'', t'', l'')$. *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. $\left(\dfrac{l''}{l}\right)^2 < \dfrac{k''}{k}$

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] 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.

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

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

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

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

where

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

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 removeLiquidity$_{\mathtt{code}}$

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

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

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* $(e, t, l) \xrightarrow{\mathtt{removeLiquidity}_{\mathtt{spec}}(\Delta l)} (e', t', l')$*. Let* $(e, t, l) \xrightarrow{\mathtt{removeLiquidity}_{\mathtt{code}}(\Delta l)} (e'', t'', l'')$*. 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. $e' \leq e'' \leq e$
2. $t' \leq t'' \leq t$
3. $l'' = l' < l$
4. $k' \leq k'' \leq k$
5. $\left(\dfrac{l''}{l} \right)^2 \leq \dfrac{k''}{k}$

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 $k'' \leq k$. 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 $\texttt{addLiquidity}_{\texttt{code}}$ is subsequently followed by $\texttt{removeLiquidity}_{\texttt{code}}$ as follows:*

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

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

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

3 Token Price Calculation

We formalize the functions calculating the current token price. Suppose, as Section 1, there are two tokens X and Y, and let x and y is the number of tokens X and Y that the exchange currently reserves, respectively.

We have two price calculation functions: `getInputPrice` and `getOutputPrice`. Given Δx, the `getInputPrice` function computes how much Y tokens (i.e., Δy) can be bought by selling Δx. On the other hand, given Δy, the `getOutputPrice` function computes how much X tokens (i.e., Δx) needs to be sold to buy Δy.

Note that these functions do not update the exchange state.

3.1 getInputPrice

We formalize `getInputPrice` in this section, and `getOutputPrice` in the next section.

3.1.1 $\texttt{getInputPrice}_{\texttt{spec}}$

Definition 5. *Let ρ be the trade fee. $\texttt{getInputPrice}_{\texttt{spec}}$ takes as input $\Delta x > 0$, x, and y, and outputs Δy such that:*

$$\texttt{getInputPrice}_{\texttt{spec}}(\Delta x)(x, y) = \Delta y = \frac{\alpha\gamma}{1 + \alpha\gamma} y$$

where $\alpha = \dfrac{\Delta x}{x}$ and $\gamma = 1 - \rho$. Also, we have:

$$x' = x + \Delta x = (1 + \alpha)x$$
$$y' = y - \Delta y = \frac{1}{1 + \alpha\gamma} y$$

Theorem 7. *Suppose $\texttt{getInputPrice}_{\texttt{spec}}(x' - x)(x, y) = y - y'$. Let $k = x \times y$ and $k' = x' \times y'$. Then, we have:*

1. $x < x'$
2. $y > y'$
3. $k < k'$

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

Definition 6. *Let ρ be the trade fee.* ***getInputPrice$_{code}$*** *takes as input $\Delta x > 0$, x, and $y \in \mathbb{Z}$, and outputs $\Delta y \in \mathbb{Z}$ such that:*

$$\textit{getInputPrice}_{code}(\Delta x)(x, y) = \Delta y = \left\lfloor \frac{\alpha\gamma}{1 + \alpha\gamma} y \right\rfloor$$

where $\alpha = \dfrac{\Delta x}{x}$ and $\gamma = 1 - \rho$. Also, we have:

$$\begin{aligned} x'' &= x + \Delta x = (1 + \alpha)x \\ y'' &= y - \Delta y = \left\lceil \frac{1}{1 + \alpha\gamma} y \right\rceil \end{aligned}$$

In the contract implementation [1], $\rho = 0.003$, and getInputPrice$_{\texttt{code}}(\Delta x)(x, y)$ is implemented as follows:

```
(997 * Δx * y) / (1000 * x + 997 * Δx)
```

where / is the integer division with truncation (i.e., floor) rounding.

Theorem 8. *Suppose* ***getInputPrice$_{spec}$***$(x' - x)(x, y) = y - y'$. *Suppose* ***getInputPrice$_{code}$***$(x'' - x)(x, y) = y - y''$. *Let $k = x \times y$, $k' = x' \times y'$, and $k'' = x'' \times y''$. Then, we have:*

1. $x < x' = x''$
2. $y' \leq y'' \leq y$
3. $k < k' \leq k''$

3.2 getOutputPrice

We formalize getOutputPrice, a companion to getInputPrice.

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

Definition 7. *Let ρ be the trade fee.* ***getOutputPrice$_{spec}$*** *takes as input $0 < \Delta y < y$, x, and y, and outputs Δx such that:*

$$\textit{getOutputPrice}_{spec}(\Delta y)(x, y) = \Delta x = \frac{\beta}{1 - \beta} \cdot \frac{1}{\gamma} \cdot x$$

where $\beta = \dfrac{\Delta y}{y} < 1$ and $\gamma = 1 - \rho$. Also, we have:

$$\begin{aligned} x' &= x + \Delta x = \frac{1 + \beta(\frac{1}{\gamma} - 1)}{1 - \beta} \cdot x \\ y' &= y - \Delta y = (1 - \beta)y \end{aligned}$$

Theorem 9. *Suppose* $\texttt{getOutputPrice}_{\mathit{spec}}(y-y')(x,y) = x'-x$. *Let* $k = x \times y$ *and* $k' = x' \times y'$.

1. $x < x'$
2. $y' < y$
3. $k < k'$

Theorem 10. $\texttt{getInputPrice}$ *is dual to* $\texttt{getOutputPrice}$. *That is,*

$$\begin{aligned}&\texttt{getOutputPrice}_{\mathit{spec}}(\texttt{getInputPrice}_{\mathit{spec}}(\Delta x)(x,y))(x,y) = \Delta x\\ &\texttt{getInputPrice}_{\mathit{spec}}(\texttt{getOutputPrice}_{\mathit{spec}}(\Delta y)(x,y))(x,y) = \Delta y\end{aligned}$$

3.2.2 $\texttt{getOutputPrice}_{\texttt{code}}$

Definition 8. *Let* ρ *be the trade fee.* $\texttt{getOutputPrice}_{\mathit{code}}$ *takes as input* $0 < \Delta y < y$, *x, and* $y \in \mathbb{Z}$, *and outputs* $\Delta x \in Z$ *such that:*

$$\texttt{getOutputPrice}_{\mathit{spec}}(\Delta y)(x,y) = \Delta x = \left\lfloor \frac{\beta}{1-\beta} \cdot \frac{1}{\gamma} \cdot x \right\rfloor + 1$$

where $\beta = \dfrac{\Delta y}{y} < 1$ *and* $\gamma = 1 - \rho$. *Also, we have:*

$$\begin{aligned} x'' &= x + \Delta x = \left\lfloor \frac{1 + \beta(\frac{1}{\gamma} - 1)}{1-\beta} \cdot x \right\rfloor + 1\\ y'' &= y - \Delta y = (1-\beta)y \end{aligned}$$

In the contract implementation [1], $\rho = 0.003$, and $\texttt{getOutputPrice}_{\texttt{code}}(\Delta y)(x,y)$ is implemented as follows:

```
(1000 * x * Δy) / (997 * (y - Δy)) + 1
```

where / is the integer division with truncation (i.e., floor) rounding.

Theorem 11. *Suppose* $\texttt{getOutputPrice}_{\mathit{spec}}(y-y')(x,y) = x'-x$. *Suppose* $\texttt{getOutputPrice}_{\mathit{code}}(y-y'')(x,y) = x''-x$. *Let* $k = x \times y$, $k' = x' \times y'$, *and* $k'' = x'' \times y''$. *Let* $k'' = t''_A * t''_B$, *and we have the following property:*

1. $x < x' < x''$
2. $y' = y'' < y$
3. $k < k' < k''$

Theorem 12. $\texttt{getOutputPrice}_{\mathit{code}}$ *is adjoint to* $\texttt{getInputPrice}_{\mathit{code}}$. *That is,*

1. $\Delta y \leq \texttt{getInputPrice}_{\mathit{code}}(\texttt{getOutputPrice}_{\mathit{code}}(\Delta y)(x,y))(x,y)$
2. $\texttt{getOutputPrice}_{\mathit{code}}(\texttt{getInputPrice}_{\mathit{code}}(\Delta x)(x,y))(x,y) \leq \Delta x$

4 Trading Tokens

Now we formalize the token exchange functions that update the exchange state.

4.1 `ethToToken`

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

4.1.1 $\texttt{ethToToken}_{\texttt{spec}}$

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

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

4.1.2 $\texttt{ethToToken}_{\texttt{code}}$

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

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

4.2 `ethToTokenExact`

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

4.2.1 $\texttt{ethToTokenExact}_{\texttt{spec}}$

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

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

4.2.2 $\texttt{ethToTokenExact}_{\texttt{code}}$

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

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

4.3 tokenToEth

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

4.3.1 $\texttt{tokenToEth}_{\texttt{spec}}$

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

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

4.3.2 $\texttt{tokenToEth}_{\texttt{code}}$

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

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

4.4 tokenToEthExact

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

4.4.1 `tokenToEthExact`$_{\texttt{spec}}$

`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

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

4.4.2 `tokenToEthExact`$_{\texttt{code}}$

`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

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

4.5 `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.

4.5.1 `tokenToToken`$_{\texttt{spec}}$

`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

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

4.5.2 $\texttt{tokenToToken}_{\texttt{code}}$

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

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

4.6 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.

4.6.1 $\texttt{tokenToTokenExact}_{\texttt{spec}}$

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

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

4.6.2 $\texttt{tokenToTokenExact}_{\texttt{code}}$

$\texttt{tokenToTokenExact}_{\texttt{code}}$ 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 contract. `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`.