# Kreiranje složenih transakcija na Eterijum blokčejnu

Nikola Vuković

Master rad

Beograd, 2022.


Matematički fakultet

Univerzitet u Beogradu

Univerzitet u Beogradu - Matematički fakultet

Master rad

**Naslov:** Kreiranje složenih transakcija na *Eterijum* blokčejnu

**Autor:** Nikola Vuković

**Apstrakt:** *Eterijum* blokčejn je distribuirani zapisnik koji omogućava decentralizovano izvršavanje programa. Koncept "pametnih ugovora" predstavlja programe postavljene na blokčejn, čiji kod se ne može menjati i čije izvršavanje je osigurano konsenzusom celog blokčejna. Transakcija predstavlja interakciju jednog naloga (adrese koja predstavlja korisnika) sa jednim nalogom na blokčejnu. Najjednostavnija transakcija predstavlja prenos novčanih sredstava (Etera) između adresa, ali ključna stvar u kojoj se *Eterijum* razlikuje od ranijih rešenja kao što je *Bitkoin*, je mogućnost da transakcija poziva metodu na postavljenom ugovoru. Rad se bavi konceptom složenih transakcija, odnosno transakcija koje u sebi kombinuju pozive ka više pametnih ugovora u cilju obavljanja složene operacije. Obrađen je pristup namenskim ugovorima, kao i generalizovani sistemi koji omogućavaju kreiranje složenih transakcija bez postavljanja novog ugovora na sam blokčejn.

**Ključne reči:** Blokčejn, lanac blokova, *Eterijum*, složene transakcije

**Mentor:**

dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije:**

dr Miodrag Živković, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

Aleksandar Veljković, asistent
Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane:** 30. septembar 2022.

# Zahvalnica

Veliku zahvalnicu dugujem kolegama i prijateljima Nenadu Palinkaševiću i Nikoli Klipi, kao i drugim članovima razvojnog tima projekta *DeFi Saver*, koji su bili raspoloženi za diskusiju oko tehničkih nedoumica na koje sam nailazio tokom pisanja rada.

Neizmerno sam zahvalan i Kristini Krtinić za beskrajno strpljenje i sve vidove podrške od početka do kraja.

Na kraju, ovaj rad ne bi bio napisan da nije roditelja – Nataše i Gorana – koji su od upisa fakulteta do danas bili vetar u leđa.

# Sadržaj

# 1 Uvod

Blokčejn sistemi, distribuirani sistemi za čuvanje podataka sa fokusom na garanciji verodostojnosti, doživljavaju uspon u popularnosti u prethodnim godinama zbog značajnih tehnoloških napredaka. U svom radu o *Bitkoin* blokčejnu, Satoši Nakamoto je spojio prethodno poznate elemente na revolucionaran način i prvi definisao koncept decentralizovanog blokčejna koji se može koristiti kao infrastruktura za prvu primenljivu decentralizovanu digitalnu valutu. Tehnologija koju je Satoši definisao je služila kao osnova za dalji razvoj blokčejn tehnologija za različite primene.

Par godina kasnije, Vitalik Buterin predstavlja decentralizovani blokčejn koji proširuje mogućnosti čuvanja liste transakcija u generalizovanu platformu za decentralizovano izvršavanje programa. Ova ideja omogućava do tada neviđenu paradigmu dizajniranja digitalnih sistema u kojoj nije potrebno autoritativno lice koje osigurava sistem, već je sistem po samoj koncepciji osiguran konsenzusom čvorova u mreži blokčejna. Na ovoj platformi se u narednim godinama razvijaju razni eksperimentalni sistemi kao što su decentralizovano vlasništvo, decentralizovane finansije i drugi. Jedna od glavnih prednosti ove platforme je interoperabilnost svih sistema izgrađenih na njoj. Više o platformi *Eterijum* je razrađeno u poglavlju 1, dok su mogućnosti koje ta interoperabilnost omogućava razrađene u poglavlju 2.

Kako su ovi sistemi sazrevali vremenom, rasla je i njihova složenost i interoperabilnost, pa su i same interakcije s njima postajale kompleksnije. Da bi se podržala ekspanzija ovog ekosistema, bili su potrebni i alati koji bi olakšali i ubrzali razvoj ovih interakcija. Razvoj tradicionalnih specijalizovanih rešenja se pokazao previše sporim, komplikovanim i nepristupačnim za potrebe ovog dinamičnog ekosistema. Vremenom su se pojavila rešenja koja su omogućila veću fleksibilnost i brže iteracije složenih akcija.

Rad za cilj ima upoznavanje čitaoca sa tradicionalnijim, kao i novijim, generalizovanim rešenjima. U 4. poglavlju prikazano je detaljno poređenje tehnologija, uključujući i realizovane primere nekih od češćih složenih interakcija. Priložen je izvorni kod postojećih pametnih ugovora koji ih omogućavaju, kao i njihova varijanta realizovana kroz generalizovane sisteme. Korišćen je alat *Tenderly* za analizu toka i kompleksnosti izvršavanja.

Radi transparentnosti, navodi se da je autor učestvovao u razvoju sistema *DeFi Saver*.

# 2 Sistemi blokčejna

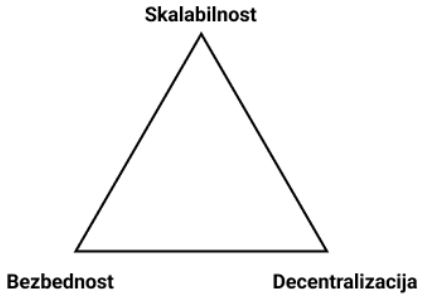
Radi razumevanja potrebe za kompleksnim transakcijama, kao i tehnologijama koje ih omogućavaju, bitno je razumeti osnovnu tehnologiju blokčejna i zašto je ona drugačija od pređašnjih distribuiranih sistema. S obzirom na to da sistemi blokčejna nisu glavna tema rada, koncepti su predstavljeni relativno površno bez ulaska u implementacione detalje kao što su kriptografski algoritmi i komunikacioni protokoli. Nakon konceptualnog uvoda u tehnologiju blokčejna, u ovom poglavlju je razrađena konkretna implementacija blokčejna *Eterijum*, kao i neke od primena tehnologije.

## 2.1 Blokčejn – distribuirani zapisnik

Blokčejn, u bukvalnom prevodu - lanac blokova, je suštinski kriptografska struktura podataka koja za cilj ima garantovanje verodostojnosti (odnosno nepromenjenosti) istorijskih podataka. Osnovni princip se zasniva na tome da je lako dodati nove podatke, dok je izmeniti ili obrisati stare gotovo nemoguće.

Na najjednostavniji način, blok se definiše kao jedinica skladištenja podataka. Svaki blok sadrži nove podatke i kriptografski heš sadržaja prethodnog bloka. Pošto prethodni blok takođe sadrži heš sadržaja njemu prethodnog bloka, blokovi obrazuju lanac gde svaki blok zavisi od prethodnog. Očigledan izuzetak je prvi blok koji nema prethodnika (često nazivan *Genesis* - blok postanka). Ukoliko bi neko želeo da promeni sadržaj određenog bloka u prošlosti, morao bi da promeni heš zapisan u sledećem bloku, a samim tim i u svim blokovima nakon njega, sve do poslednjeg.

Ipak, ako bismo uzeli ovu strukturu i implementirali na postojećim tehnologijama, na primer koristeći bazu podataka da čuvamo sadržaj blokova, ne bismo garantovali tačnost istorijskih podataka. Potencijalnom napadaču bi napad bio u određenoj meri kompleksniji i sporiji za izvođenje, ali bi i dalje bio podjednako moguć kao i da se koristi tradicionalna baza podataka kojoj napadač ima pristup. Kada govorimo o blokčejnu, obično podrazumevamo decentralizovan ili distribuiran sistem gde više entiteta ima pravo upisa. Blokčejn u ovakvom sistemu služi kao mehanizam osiguravanja verodostojnosti podataka bez potrebe za poverenjem između lica.




*Slika 2.1: Ilustracija trileme dizajna blokčejn sistema*

Implementacija blokčejna uključuje definisanje politike konsenzusa, koja zavisi od mesta primene, bezbednosnih i performantnih kompromisa. U zatvorenom sistemu gde postoji centralni entitet koji autentifikuje i autorizuje druge učesnike u mreži, politika konsenzusa rešava probleme istorijski viđene u distribuiranim sistemima [1] (kao što su efikasna komunikacija i koordinacija između učesnika, tolerancija na greške, i sl.). Ipak, obezbeđivanje mreže bez centralnog autoriteta je zapravo glavni cilj otvorenih sistema u kojima svako može da učestvuje kao što su blokčejnovi *Bitkoin* i *Eterijum*. Politika konsenzusa predstavlja način rešavanja problema vizantijskih generala u decentralizovanom sistemu. Vitalik Buterin [2] definiše trilemu skaliranja u kojoj navodi da pri projektovanju sistema nije moguće istovremeno optimizovati više od dve stavke između skalabilnosti, bezbednosti i decentralizovanosti (ilustrovano na slici 2.1). Različiti sistemi sa različitim ciljevima i vrednovanjem ovih stavki u svom dizajnu definišu različite politike konsenzusa. Neki zarad skalabilnosti žrvuju decentralizovanost u manjoj (dokaz o ulogu, eng. *proof of stake*) ili većoj (dokaz o autoritetu, eng. *proof of authority*) meri. Originalni pristup sistema *Bitkoin*, i konceptualno sličan ali implementaciono drugačiji pristup sistema *Eterijum*, optimizuje decentralizovanost mreže, a žrtvuje skalabilnosti. Zasniva se na tzv. principu dokaza o radu (eng. *proof of work*) i biće detaljnije objašnjen u poglavlju o sistemu *Eterijum*.

## 2.2 *Eterijum* – generalizovana platforma za izvršavanje programa

Ideja blokčejna kakvu poznajemo danas je definisana radom o *Bitkoinu* 2008. godine [3] od strane Satoši Nakamota, koji je do danas ostao anoniman. Ipak, suštinske ideje na kojima se rad zasniva postoje dugo pre samog rada [4]. Rad referencira *Heškeš*, sistem Adama Beka iz 2002. koji definiše mehanizam dokazivog uloženog rada (*Proof of work*). Ovaj mehanizam, u radu primenjen na prevenciju neželjene elektronske pošte, uopšteno se primenjuje kao mehanizam zaštite sistema od zloupotrebe tako što efektivno podiže cenu interakcije sa sistemom zahtevajući (dokazivo) ulaganje procesorskog vremena. Ipak, ova ideja prevencije masovnog slanja elektronske pošte se spominje i pre *Heškeša*, čak 1992. godine [4]. Ideja decentralizovane (*peer-to-peer*) mreže za plaćanje takođe postoji godinama – rad o *Bitkoinu* citira i *b-money*, rad Vei Daija [5] iz 1998. godine.

Sistem *Eterijum* blokčejna je u velikoj meri baziran na sistemu *Bitkoin*. U radu u kom predstavlja ideju *Eterijuma*, tzv. *beloj knjizi* [1] [6], Vitalik Buterin uviđa revolucionaran napredak *Bitkoina* kao prve digitalne imovine bez inherentne ili vrednosti koju joj dodeljuje centralni autoritet. Ipak tvrdi da je u stvari sistem distribuiranog konsenzusa najznačajniji

[1] Termin „*bela knjiga*“ (eng. *White Paper*) se najčešće koristi za informativni dokument koji služi da prezentuje projekat, uključujući njegovu motivaciju i okviran tehnički pregled. Često se pojavljuje i termin „žuta knjiga“ (eng. *Yellow Paper*) kojim se prezentuju tehnički detalji projekta.

naučnotehnološki napredak. Vitalik predlaže ideju korišćenja distribuiranog konsenzusa kao osnove za decentralizovanu platformu za kreiranje generalizovanih pametnih ugovora.

Ideja „pametnih ugovora“ koju je par godina ranije definisao Nik Sabo [7] razmatra digitalizaciju koncepta tradicionalnih pravnih ugovora tako da formalizuju i obezbeđuju veze koristeći kriptografske i druge mehanizme. Suština ideje je da se ugovoreni elementi definišu praktično na nivou protokola, tako da se ne ostavlja prostor alternativom tumačenju, već da suštinski nije moguće delovati van ugovorenih pravila. Sabo kao primer ovakvih ugovora ugrađenih u fizički svet daje automat za prodaju grickalica. Pravila su ugrađena u sistem (korisnik mora da ubaci novac da bi dobio limenku soka), a osigurana su fizičkim ograničenjima (zlonamerni korisnik ne može da izvuče limenku na silu).

Da bismo razumeli dalje koncepte, potrebno je da donekle razumemo kako blokčejn *Eterijum* funkcioniše. Nećemo ulaziti u detalje realizacije protokola, kriptografskih elemenata i rudarenja, jer nisu glavna tema rada. Za početak ugrubo definišemo životni vek transakcije – glavnog načina interakcije korisnika sa blokčejnom:

1. Korisnik kreira transakciju na neki način, bilo ručno ili koristeći određeni softver;
2. Korisnik svojim novčanikom (konkretno privatnim ključem) kriptografski potpisuje transakciju;
3. Transakcija se emituje mreži i čuva se u „bazenu“ aktivnih transakcija koje čekaju da budu uključene u blok;
4. Rudar bira njemu najisplativije transakcije i pokušava sa njima da izrudari blok;
5. Blok sa uključenom transakcijom biva izrudaren i time se transakcija smatra izvršenom, a njen efekat se odražava na stanje blokčejna.

Spomenuli smo dosta koncepata mreže koje ćemo dalje konkretnije definisati:

- **Eter:** Osnovna jedinica vrednosti na *Eterijum* blokčejnu. Razmena ove valute je osnova funkcionisanja celog sistema.
- **Adrese:** Jedna korisnička adresa (Često se nazivaju EOA – eng. *Externally owned accounts*) predstavlja jednog eksternog aktera u blokčejnu. Adresa ima svoj balans i sposobnost da kreira transakcije. Ponekad se u praksi naziva novčanikom, što je pojednostavljenje jer je čuvanje vrednosti samo jedna od funkcija. Suštinski, adresa je definisana kombinacijom privatnog i javnog ključa. Sama adresa predstavlja poslednjih 20 bajtova heša javnog ključa algoritmom SHA3. Kontrolu nad tom adresom ima odgovarajući privatni ključ koji se, između ostalog, koristi za kreiranje transakcija, kao i verifikaciju identiteta potpisivanjem poruka. Važno je napomenuti da i pametni ugovori imaju adrese na koje su postavljeni. One su istog 20-bajtnog formata ali predstavljaju drugi tip adresa i one nemaju odgovarajući privatni ključ.
- **Pametni ugovori:** Programi koji su postavljeni na blokčejn sa mogućnošću čuvanja podataka i pružanja usluga. Suštinski, ugovori predstavljaju skup pravila po kojima bilo ko može da interaguje sa njima. Radi ilustracije, ugovori mogu da predstavljaju

registar podataka koji svako može da pročita, mehanizam za kolateralizovan kredit koji svako može da kreira, sistem vlasništva i kupoprodaje jedinstvenih dobara kao što su umetnička dela, itd. Ove funkcionalnosti mogu biti javne i dostupne svima, mogu biti ograničene na određen skup adresa, ili mogu biti dostupne samo adresama koje ispunjavaju neki uslov kao što je, na primer, posedovanje određenih sredstava.

- **Transakcije:** Umesto da posmatramo *Eterijum* prosto kao zapisnik razmena dobara, preciznije je da ga posmatramo kao konačni automat, a transakciju kao prelazak iz jednog stanja u drugo. Transakcija u osnovi predstavlja poruku jedne adrese (pošiljalac) ka drugoj (primalac). Transakcija može da ima vrednost - količinu etera koja se prenosi primaocu od pošiljaoca, i pozivne podatke (*calldata*). Pozivni podaci se ne koriste u transakcijama između korisničkih adresa, ali u transakcijama ka adresama ugovora sadrže komandu koju pošiljalac zahteva da izvrši na ugovoru. Pored pomenutih, transakcija sadrži i parametre za naplatu: cenu po jedinici izvršavanja (cena gasa), kao i gornji limit za trajanje izvršavanja (limit količine gasa). Svrha gasa i mehanizam naplate su objašnjeni u nastavku poglavlja. Kao sistem autentifikacije i autorizacije, svaka transakcija je potpisana privatnim ključem pošiljaoca. Bitno je napomenuti da je transakcija uvek atomična. Transakcija može da bude neizvršiva (na primer kada pošiljalac nema količinu Etera koju pokušava da pošalje, ili je u pametnom ugovoru došlo do greške usled nepoštovanja nekog pravila). U tom slučaju, transakcija i dalje može biti uključena u blok (rudari imaju ekonomski podstrek za to – gas je plaćen), ali će, osim plaćenog gasa od strane pošiljaoca, njeni efekti biti kompletno zanemareni (Eter se neće preneti, operacija na ugovoru neće biti izvršena). Tako izvršena transakcija sa zanemarenim ishodom ima status „opozvane transakcije“ (*reverted transaction*).
- **Blokovi:** Kao što je ranije pomenuto, blok predstavlja jedinicu skladištenja podataka iz koje se sastoji blokčejn. Blok sadrži isključivo transakcije – suštinu bloka, kao i metapodatke o sebi (broj, heš, heš prethodnog bloka, vremensku oznaku).
- ***Eterijum* virtuelna mašina (EVM):** Kôd pametnih ugovora se kompajlira na kod niskog nivoa koji se izvršava u kontekstu virtuelne mašine. EVM je skup pravila po kojima se kod izvršava, odnosno pravila po kojima konačni automat prelazi iz jednog stanja u drugo.
- **Gas:** Mehanizmom gasa se dužina i kompleksnost izvršavanja vezuju za proporcionalnu cenu izvršavanja transakcije. Gas predstavlja jedinicu iteracije izvršavanja programa i proporcionalne naplate. Najjednostavnija transakcija bez podataka „potroši“ 21.000 gasa. Kompleksnija transakcija koja interaguje sa više ugovora i menja zapisane podatke može lako da dostigne više miliona gasa, i samim tim košta višestruko skuplje. EVM je Tjuring-kompletan sistem, što implicira postojanje problema zaustavljanja. Mehanizam gasa takođe definiše način rešavanja problema zaustavljanja, tako što korisnik u okviru transakcije definiše

limit gasa koji transakcija može da potroši svojim izvršavanjem, pa je time ograničena i dužina izvršavanja transakcije, kao i njena cena. Pored ekonomskog podstreka, postoji i fiksan limit za kompleksnost transakcije u vidu ograničenja gasa koji može da potroši ceo blok. Ovo ograničenje se dinamično skalira zavisno od korišćenja mreže, ali je u proseku oko 15 miliona gasa. Kako je transakcija atomična, mora da stane u jedan blok pa samim tim ne može biti veća od njega.

- **Čvorovi mreže:** Čvorovi (eng. *node*) koji aktivno učestvuju u mreži se često nazivaju rudarima. Imaju ažurnu kopiju blokčejna i rade na kreiranju blokova zbog ekonomskog podstreka. Postoje i pasivni čvorovi koji samo sihronizuju svoju kopiju blokčejna novonastalim blokovima. Čvorovi implementiraju standardizovan interfejs kroz koji omogućavaju programima da interaguju sa blokčejnom: da dohvataju postojeće podatke, šalju transakcije, reaguju na nove blokove i sl.
- **Mehanizam uključivanja transakcije u blok:** Transakcija definiše i cenu u Eteru po jedinici gasa. Ilustracije radi, u julu 2021. ova cena je u proseku 20 „gveja“[2]. Da bi se transakcija smatrala kanonski izvršenom, ona treba da bude uključena u blok od strane rudara, i transakcije se međusobno takmiče za uključenje. Rudari mogu da izaberu proizvoljne transakcije za uključivanje u blok, ali glavni podstrek im je finansijski. Zato transakcije uključuju cenu po jedinici gasa koju je pošiljalac spreman da plati rudaru za uključenje u blok. Kada više pošiljalaca želi da pošalje transakciju, dolazi do zagušenja mreže, što dovodi do toga da pošiljaoci podižu cene gasa u svojim transakcijama, ne bi li povećali verovatnoću uključenja. Ovaj fenomen ilustruje jedan od većih problema *Eterijuma* – skalabilnost. Postoji više ideja i aktivnih projekata za unapređivanje skalabilnosti *Eterijuma*, koji prave različite kompromise, ali je materija dovoljno kompleksna da zaslužuje zaseban rad.
- ***Proof of Work*:** Bezbednosni mehanizam kojim se ostvaruje konsenzus mreže i suštinski definiše način na koji rudar „izrudari“ blok i zašto ostatak mreže smatra taj blok validnim. Rudarenje je zapravo rešavanje određenog matematičkog problema čije rešavanje nije moguće optimizovati, već se svodi na nasumično pogađanje rešenja. Provera rešenja je trivijalna. Težina samog problema se lako skalira, pa se na taj način održava konstantno prosečno vreme potrebno za rudarenje bloka. Dakle, mehanizam se zasniva na dokazivanju da je rudar utrošio određeno procesorsko vreme („dokaz utrošenog rada“ u bukvalnom prevodu), a uz taj dokaz se osigurava da se napad na mrežu ne može izvesti ako napadač nema više od ukupne procesorske moći ostatka mreže. Rudar koji prvi dođe do rešenja problema komunicira to sa ostatkom mreže i na taj način kreira novi blok u lancu. Pri kreiranju bloka, rudar dobija određenu nagradu u valuti Eter, pa se na taj način ostvaruje finansijski podstrek zbog kog rudari postoje.

[2] Gvej (eng. *Gwei*) predstavlja $10^{-9}$ deo jednog Etera.

*Bela knjiga* [6] 2014. ilustruje ideju *Eterijuma* navodeći potencijalne aplikacije pametnih ugovora, koje uključuju:

- **Valute:** Moguće je definisati nove valute. One mogu predstavljati neku valutu iz fizičkog sveta (deonice, zlato, dolar, i sl.); mogu da predstavljaju novu, nezavisnu valutu; a čak mogu biti kreirane sa jedinstvenim apoenima gde svaki može da ima unikatno značenje. Valute mogu imati dodatna pravila, kao što su limitacije ili provizija pri transferu.
- **Izvor podataka (eng. *data feed*):** Moguće je definisati sistem gde određeni akteri upisuju podatke iz fizičkog sveta (cene valuta i deonica, količinu padavina na nekoj lokaciji, itd.) u pametan ugovor. Sistem se može obezbediti tako što se od aktera zahteva ulog, na koji dobijaju penal ako je njihov upis značajno različit od ostalih upisa za taj ciklus. Na taj način, za aktere je najpovoljnije da upisuju ono što očekuju da će drugi upisati – odnosno tačan podatak.
- **Sistemi identifikacije i reputacije:** Priroda decentralizovanog sistema omogućava da budemo sigurni da određena adresa predstavlja određenu osobu ili organizaciju. Pametnim ugovorima je moguće kreirati centralni registar i sistem nalik DNS-u, koji bi omogućio mapiranje simboličkih (čitljivih) adresa na kriptografske adrese.

Primeri predviđeni u radu su vremenom zaista realizovani, ali i razrađeni u kompleksnije aplikacije. Neki od primera aktuelnih projekata baziranih na pametnim ugovorima uključuju:

- **Stabilne valute:** Valute koje ciljaju da oponašaju tradicionalne valute fizičkog sveta, kao Dolar i Evro. U nekim slučajevima, određeno autoritetno lice čuva određenu svotu fizičkog novca i za taj novac izdaje digitalnu reprezentaciju za koju ono garantuje. U drugim slučajevima, valuta samo cilja da oponaša tržišnu cenu druge valute različitim mehanizmima i podsticajima, povećavajući i smanjujući likvidnost nezavisno od centralne banke.
- **Decentralizovane organizacije (*DAO* - Decentralized Autonomous Organization):** Organizacije koje upravljaju same sobom koristeći pametne ugovore. Ove organizacije zamenjuju imenovane vođe decentralizovanim glasačkim mehanizmom, kao i blagajnike automatizovanim trezorom.
- **Nezamenljivi tokeni (*NFT – Non-Fungible Token*):** Razne stvari su dobile svoj digitalni oblik na blokčejnu. Određene manifestacije izdaju tokene kao dokaze prisustva. Umetnici koriste blokčejn da digitalizuju vlasništvo svojih umetnina. Neke od daljih ideja uključuju digitalizaciju vlasništva nekretnina, čime bi se eliminisala potreba za fizičkim registrom i olakšala birokratija.

## 2.3 Decentralizovane finansije

Decentralizovane finansije (često skraćeno kao *DeFi*) nemaju formalnu definiciju, već se termin koristi za određene finansijske sisteme kreirane za eru decentralizovanih tehnologija,

baziranih na internetu i blokčejnu. Ti sistemi su primarno transparentni, nisu kontrolisani od strane centralnog autoriteta (vlade, centralne banke, privatnih banki) koji može da zabrani ili ograniči učestvovanje bilo kome - pravila su definisana protokolom i ona su ista za sve. Iz ove formulacije se može naslutiti značaj novih mogućnosti za koje su zaslužni distribuiran konsenzus i pametni ugovori.

Ovakvi moderni sistemi lako mogu da oponašaju tradicionalni finansijski sistem koji se, kada se birokratija izuzme, svodi na transfer novca između lica – najjednostavniju moguću transakciju na blokčejnu *Eterijum*. Štaviše, takav sistem bi radio značajno efikasnije pošto svako može da učestvuje u mreži direktno, pa bi potreba za ljudskim faktorom bankara bila značajno umanjena i sistem ne bi bio ograničen na radno vreme banke. Ipak, uzevši u obzir prethodno spomenute mogućnosti, digitalizacija postojećeg bankovnog sistema bi bila relativno mali cilj za decentralizovane tehnologije.

Jedna jako bitna osobina decentralizovanih finansija i *Eterijuma* uopšte, koja prirodno proizlazi iz otvorenosti svih sistema, je interoperabilnost programa (samim tim i sistema i projekata). Da bismo ilustrovali ovu osobinu i potencijal ovih sistema, prikazaćemo neke od ideja koje ranije suštinski nisu bile moguće:

### 2.3.1 Decentralizovane menjačnice

Koncept razmene jedne valute za drugu bez centralnog autoriteta koji obezbeđuje likvidnost je naravno nemoguć u fizičkom svetu. Ipak, u svetu distribuiranog konsenzusa, centralni autoritet je moguće zameniti protokolom. Tako su neka od prvih dostignuća decentralizovanih aplikacija upravo otvorene menjačnice. Opisaćemo koncept sa malo više detalja jer će biti značajni u razumevanju kasnije materije.

Da bismo razumeli kako decentralizovane menjačnice rade, obratićemo prvo pažnju na tradicionalne menjačnice. Kao najjednostavniji primer imamo menjačnice namenjene fizičkim licima (namenske menjačnice, banke i sl.). Tu centralni autoritet odredi odnos razmene između dve valute, i ta cena važi do kraja dana, a provajder likvidnosti određuje koliku nadoknadu uzima za razmenu (najčešće 1-2%, ali nekada i više). Ovaj jednostavan proces funkcioniše za relativno male razmene koje nisu vremenski osetljive (ni sa strane korisnika ni u pogledu volatilnosti valuta). Kao drugi primer, uzećemo berzu – cena se menja u realnom vremenu jer postoji isključivo kao rezultat ponude i potražnje, na osnovu centralne knjige porudžbina i ponuda. Razmene se dešavaju direktno između krajnjih korisnika, dok naknadu uzima centralni autoritet koji održava knjige i spaja aktere. U realnosti sistem je kompleksniji (na primer postoje stvaraoci tržišta – *market makers*) ali nećemo ga razrađivati pošto su detalji nebitni za ilustraciju. Definisaćemo i neke od uslova po kojima merimo koliko je proces razmene povoljan za korisnika: blizina realnoj tržišnoj ceni, naknada za provajdera likvidnosti, količina i dubina likvidnosti, …

Prve decentralizovane menjačnice kreirane su upravo po uzoru na berzu. Postoji knjiga ponuda i potražnja koja se čuva na ugovoru. Bilo ko može da interaguje sa ugovorom, ispuni drugi deo neke porudžbine na ugovoru, i time izvrši razmenu. Ovaj pristip zamenjuje centralizovane aspekte tradicionalne berze (otvorena platforma koju niko ne kontroliše,

otpornija na hakovanje, itd.) zarad sporije i skuplje interakcije (postavljanje svake ponude je transakcija koja treba da se plati i izrudari).

Nakon preslikanog tradicionalnog sistema, pojavile su se i hibridne varijante bazirane na sistemu knjige ponuda. Suštinski se svode na to da se knjiga ponuda održava van memorije blokčejna (ali i dalje ne centralizovano, već *peer-to-peer*), dok se spajanje ponuđača i potraživača radi na pametnom ugovoru, transakcijom.

Konačno, pojavili su se i kompletno automatizovani stvaraoci tržišta (eng. AMM – *Automated Market Makers*). Celokupna likvidnost se čuva na balansu pametnog ugovora, i svako može da uradi razmenu sredstava koristeći taj ugovor, gde se cena koju će dobiti određuje matematičkom formulom koja zavisi od trenutne likvidnosti na balansu ugovora, kao i od veličine razmene. Svako može da ubaci likvidnost u ugovor, kao i da povuče svoj novac nazad. Za svaku razmenu koja se desi preko ugovora, ugovor uzima nadoknadu (orijentaciono, najčešće između 0.01% i 0.5%) i raspoređuje je na sve davaoce likvidnosti, srazmerno udelu. Kompletna automatizovanost i odsustvo autoritativnog lica, odnosno decentralizovanost ovog sistema ga čine odličnim primerom mogućnosti decentralizovanih finansija.

Jedna od jako bitnih osobina decentralizovanih finansija je to što svi sistemi mogu da komuniciraju međusobno. Tako na primer menjačnica sa centralnom knjigom ponuda može da proveri da li bi korisnik dobio bolju (nižu) cenu na nekoj automatizovanoj menjačnici, i da interno preusmeri njegov zahtev za razmenu na tu eksternu menjačnicu umesto na sopstvenu knjigu ponuda. Na taj način menjačnica pruža bolju uslugu korisniku, pa samim tim i dobija više korisnika.

### 2.3.2 Lutrija bez gubitka – PoolTogether

U prethodnom primeru smo videli da korisnici mogu da deponuju likvidnost u automatizovanu menjačnicu za naknadu, i time ostvarili pasivan prihod. Napominjemo hibridne menjačnice nisu jedina platforma koja pruža naknadu za likvidnost – drugi primeri uključuju tržišta valuta (*Compound*, *Aave*), platforme za trgovanje sa polugom (*dYdX*) i druge.

Lutrija bez gubitka je zanimljiv koncept realizovan projektom *PoolTogether*, koji bi u fizičkom svetu zvučao kao prevara. Ipak, u svetu decentralizovanih finansija je vrlo jednostavan: U opštem slučaju, korisnik može da deponuje likvidnost na pomenute platforme i ostvari godišnju naknadu od, na primer, 5% (naknada se akumulira kontinualno). Dakle, $1000 bi za nedelju dana naraslo na oko $1001. Ako 500 ljudi uloži po $1000 u proseku, za nedelju dana bi se akumulirala naknada od oko $500. Ideja lutrije bez gubitka je da korisnici deponuju novac u pametan ugovor, koji ih automatski deponuje u protokol koji daje naknade za pružanje likvidnosti. Posle nedelju dana, ugovor „izvlači“ jednog dobitnika i njemu sleduje naknada akumulirana za tih nedelju dana, dok ostali mogu samo da podignu deponovan novac. Da bi sistem bio fer, šansa za izvlačenje korisnika je proporcionalna njegovoj deponovanoj svoti.

### 2.3.3 Privremene pozajmice – *Flashloans*

Bitno je ponovo napomenuti da se transakcije dešavaju atomično, što znači da ako dođe do greške tokom izvršavanja, celokupna transakcija se neće uvažiti. Privremene pozajmice koriste ovu osobinu i osobinu mogućnosti međusobne interakcije pametnih ugovora da bi omogućile svim korisnicima privremen pristup velikoj likvidnosti.

Pomenuli smo protokole koji zahtevaju određenu likvidnost za funkcionisanje. Određen procenat ove likvidnosti u bilo kom trenutku jeste slobodan – neiskorišćen (više o ovome kasnije u opisu protokola *Aave*). Ideja privremenih pozajmica je da korisnik na početku svoje transakcije može da pozajmi tu slobodnu likvidnost iz nekog ugovora i iskoristi je kako god želi, dokle god na kraju transakcije vrati pozajmicu. Pošto se transakcija izvršava u kontekstu pametnog ugovora protokola koji nudi pozajmicu, osigurano je da će, ako na kraju ne bude vraćen novac, doći do greške što će poništiti transakciju. Protokoli imaju finansijski podstrek da nude korisnicima ovu mogućnost jer mogu da zahtevaju da korisnici vrate više nego što su pozajmili, time uzimajući nadoknadu. Pošto je likvidnost reda veličine u stotinama miliona dolara, korisnicima ovaj koncept otvara razne mogućnosti – od jednostavnijih arbitraža velikim sumama, do kompleksnijih interakcija između više protokola. Kompleksniji primeri su upravo tema rada, pa će biti prikazani kasnije.

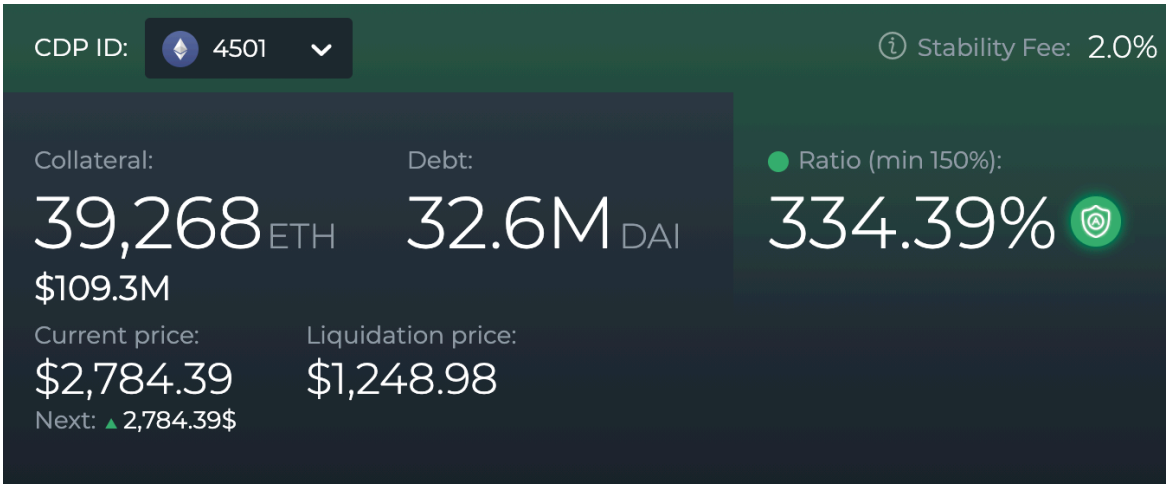
### 2.3.4 Protokol *Maker*

*Maker* je jedan od prvih protokola decentralizovanih finansija (često nazivanih *DeFi*). Njegova valuta, *Dai*, je de-facto valuta decentralizovanih finansija, inkorporirana u osnovu mnogo protokola. *Maker* je nastao kao jedan od prvih pokušaja da se napravi stabilna[3] valuta.

U najjednostavnijoj, inicijalnoj varijanti, protokol omogućava kreiranje kolateralizovanih pozajmica. Funkcioniše je tako što korisnik deponuje *Eter* kao kolateral, i pozajmi *Dai*. Bitno je da je korisnikova pozicija pre-kolateralizovana, odnosno da ima bar 50% više kolaterala u odnosu na dug (ako je deponovano Etera u vrednosti od $150, korisnik može da pozajmi najviše 100 *Dai*). Ako korisnikov dug naraste tako da je kolateral manji od 150% duga, njegova pozicija se likvidira, pri čemu se automatski održava aukcija da se pokrije njegov dug tako što se prodaje njegov kolateral, pri čemu korisnik trpi penal od bar 13% zavisno od ishoda aukcije. Kolateral i dug pozicije se u bilo kom trenutku mogu povećavati i smanjivati deponovanjem, odnosno izvlačenjem sredstava. Bitno je napomenuti da se vrednost kolaterala gleda u dolarskoj vrednosti, odnosno zavisi od trenutne cene Etera, koja se ažurira na sat vremena. Može se desiti da cena značajno padne, i samim tim smanji vrednost kolaterala, a time i kreditnu sposobnost korisnika, i dovede ga do likvidacije. Spomenućemo i kamatu na pozajmicu (tzv. *Stability fee*), koja se kontinualno akumulira tako što se dug korisnika

[3] Postoji određena količina polemike oko definicije stabilnosti. Dai je stabilan u smislu da je vezan za cenu Američkog Dolara. Ipak, uzevši u obzir inflaciju i fluktuaciju između Dolara, Evra i Funte, čak se i Dolar može smatrati nestabilnim. Postoje i druge valute koje se predstavljaju kao stabilne, a koriste druge mehanizme - vezane za vrednosti više tradicionalnih valuta, ili čak isključivo bazirane na kriptovalutama bez ciljne dolarske cene, kao što je npr. Rai.

automatski povećava kroz vreme. Ova kamata je jedan od mehanizama za stabilnost *Dai* valute – kada je tržišna cena *Dai* ispod dolara, kamata se podiže s namerom da smanji cirkulaciju. Obratno se dešava kada je tržišna cena iznad dolara – kamata se spušta kako bi se povećala cirkulacija i zadovoljila potražnja. Primer pozicije u ovom sistemu je prikazan na slici 2.2.




*Slika 2.2: Primer pozicije na protokolu Maker sa 39.268 Etera u kolateralu (oko 109 miliona dolara po ceni u datom trenutku) i 32 miliona duga u valuti Dai – odnos kolaterala i duga je 334% pa se smatra bezbednim jer je značajno veći od minimalnih 150%. Takođe prikazani su kamatna stopa (2%) i cena jedinice kolaterala na kojoj bi se desila likvidacija - $1248.98, odnosno $2784.39/(334.39/150).*

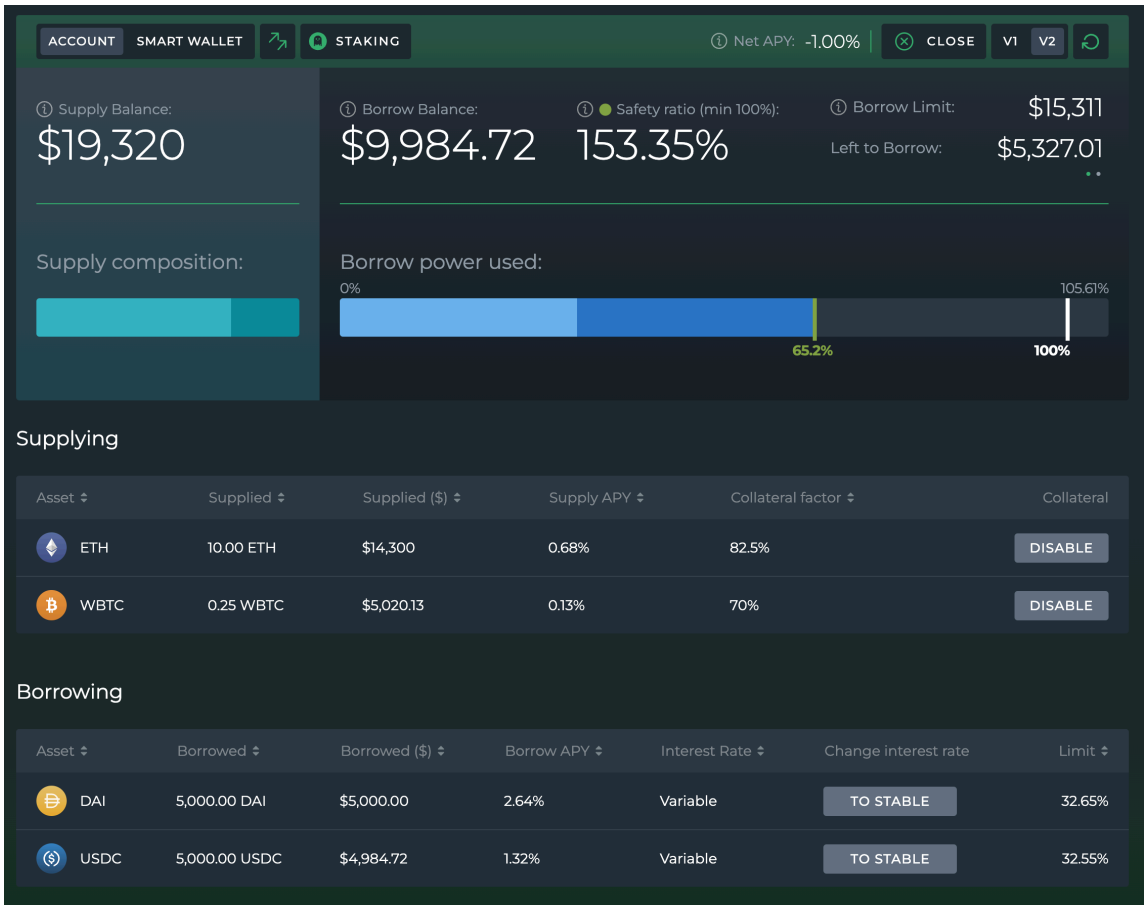
*Maker* ne održava legalni entitet, već ga vodi decentralizovana organizacija (*DAO*). Iako postoji tim koji ga razvija od početka, sve izmene u protokolu su moguće isključivo kroz decentralizovano glasanje koje uključuje celu zajednicu. Kroz ovaj mehanizam se kontrolišu parametri sistema kao što su kamatne stope i granica za likvidaciju. Protokol je vremenom imao i krupnije izmene: razvijena je podrška za kolateral u drugim kriptovalutama koje se dodaju i izbacuju, a izmenjen je i sistem aukcija pri likvidacijama. Trenutno se kao kolateral može deponovati čak i valuta koja predstavlja fizičko zemljište.

### 2.3.5 Protokol *Aave*

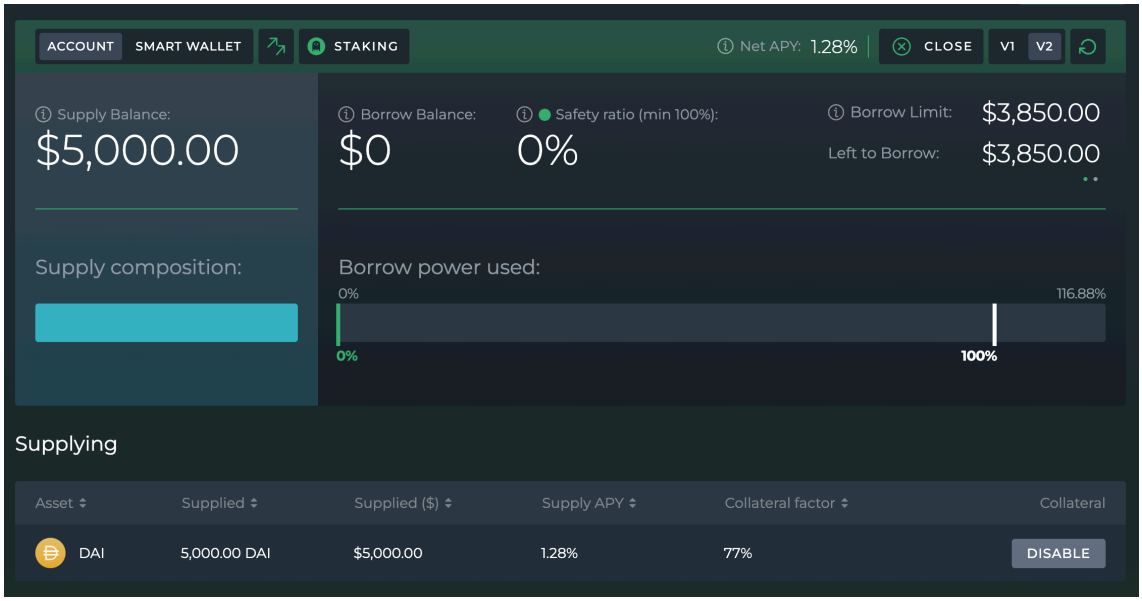
Protokol *Aave* ima slične mogućnosti sa protokolom *Maker* u smislu da dozvoljava korisniku da kreira kolateralizovan dug, ali tu sličnosti prestaju. Kada se dug kreira u poziciji na protokolu *Maker, Dai* tim činom nastaju, i uništavaju se kada se dug vrati – ne postoji fiksan iznos *Dai* u cirkulaciji. *Aave* funkcioniše koristeći korisnička sredstva. S jedne strane ima pružaoce likvidnosti, koji daju svoja sredstva na korišćenje za nadoknadu. Udružena sredstva protokol daje na raspolaganje za pozajmicu. Zajmoprimci plaćaju kamatu na svoju pozajmicu, i upravo ta kamata se isplaćuje pružaocima kao nadoknada. Korisnikov kolateral se takođe računa kao pružanje likvidnosti protokolu, pa on istovremeno ima i pozitivnu kamatu. Štaviše, pružaoci likvidnosti su samo korisnici sa deponovanim kolateralom bez duga (slika 2.5). Još jedna suštinska razlika između pomenutih protokola je da se korisnikov kolateral i dug mogu sastojati od više valuta (različite valute koje sistem podržava su prikazane na slici 2.3, a primer pozicije sa složenim kolateralom i dugom na slici 2.4), dokle god je vrednost kolaterala veća od vrednosti duga.

| Asset | Total supplied | Supply APY | Total borrowed | Borrow APY, variable |
|---|---|---|---|---|
| USD Coin USDC | 1.45B $1.44B | 0.39 % | 432.31M $429.55M | 1.34 % |
| Ethereum ETH | 710.21K $1.01B | 0.69 % | 299.71K $427.15M | 1.83 % |
| Tether USDT | 438.45M $435.29M | 1.19 % | 227.43M $225.79M | 2.33 % |
| DAI DAI | 256.90M $256.04M | 1.29 % | 136.03M $135.58M | 2.68 % |
| Curve DAO Token CRV | 157.72M $164.94M | 2.15 % | 65.27M $68.26M | 6.65 % |
| Wrapped BTC WBTC | 28.33K $562.54M | 0.14 % | 3.33K $66.16M | 1.46 % |
| ChainLink LINK | 17.03M $134.47M | 1.02 % | 4.86M $38.42M | 4.54 % |
| TrueUSD TUSD | 43.18M $43.00M | 0.70 % | 15.77M $15.70M | 1.84 % |

*Slika 2.3: Prikaz nekih valuta u protokolu Aave. Kolone redom prikazuju: Ime valute, ukupnu likvidnost protokola, kamatnu stopu na pružanje likvidnosti, ukupnu pozajmljenu količinu, kamatnu stopu na pozajmicu.*




*Slika 2.4: Primer pozicije na protokolu Aave prikazane u interfejsu aplikacije DeFi Saver. Pozicija ima kolateral sastavljen iz sredstava u valutama ETH i WBTC, u zbirnoj protivvrednosti od $19.320, kao i dug u valutama Dai i USDC, u zbirnoj protivvrednosti od $9.984. Prikazan je maksimalan mogući dug (Borrow limit: $15.311) – veći od njega bi doveo do likvidacije.*




*Slika 2.5: Primer pozicije na protokolu Aave pozicije koja samo pruža likvidnost. Pozicija pruža 5000 Dai i ima godišnju kamatnu stopu od 1,28%.*

# 3 Složene transakcije

Do sada smo zagrebali površinu mogućnosti pametnih ugovora i poznatijih protokola decentralizovanih finansija. Napomenuli smo da su transparentnost, otvorenost i interoperabilnost ključne razlike ovog ekosistema u odnosu na tradicionalni, fizički. Ove razlike omogućavaju daleko složenije interakcije od onih pomenutih u prethodnom poglavlju. Iz tog razloga, iskorišćavanje punog potencijala decentralizovanih sistema zahteva složene mehanizme i procese. Ovo poglavlje će definisati upravo te složene procese, i analizirati alate za njihovo izvršavanje.

## 3.1 Motivacija/potreba

Osnovna motivacija je pogodnost – olakšavanje nekog procesa. Pretpostavimo da je izvršavanje nekog procesa, odnosno skupa akcija, moguće kroz niz sekvencijalnih transakcija. Jasno je da bi korisniku bilo lakše i efikasnije da izvrši samo jednu transakciju koja sadrži sve akcije. Konkretan primer bi bio otvaranje pozicije u protokolu *Maker*, pozajmljivanje određenog iznosa u valuti *Dai*, razmena polovine iznosa za drugu valutu, i deponovanje u fond likvidnosti zarad kamate.

Postoje operacije koje su nebezbedne, ili čak nemoguće, ukoliko se ne izvrše atomično. Pretpostavimo da postoji pozicija u sistemu *Maker* sa kolateralom u valuti Eter sa vrednošću od $20,000, i dugom u vrednosti od $10,000. Ova pozicija ima odnos kolateralizacije od 200%. Ukoliko njen vlasnik smatra da je ovo nebezbedan odnos, može da izvrši parcijalnu likvidaciju kojom bi povećao odnos kolateralizacije bez dodavanja kolaterala ili vraćanja duga dodatnim sredstvima iz novčanika. Naime, korisnik može da:

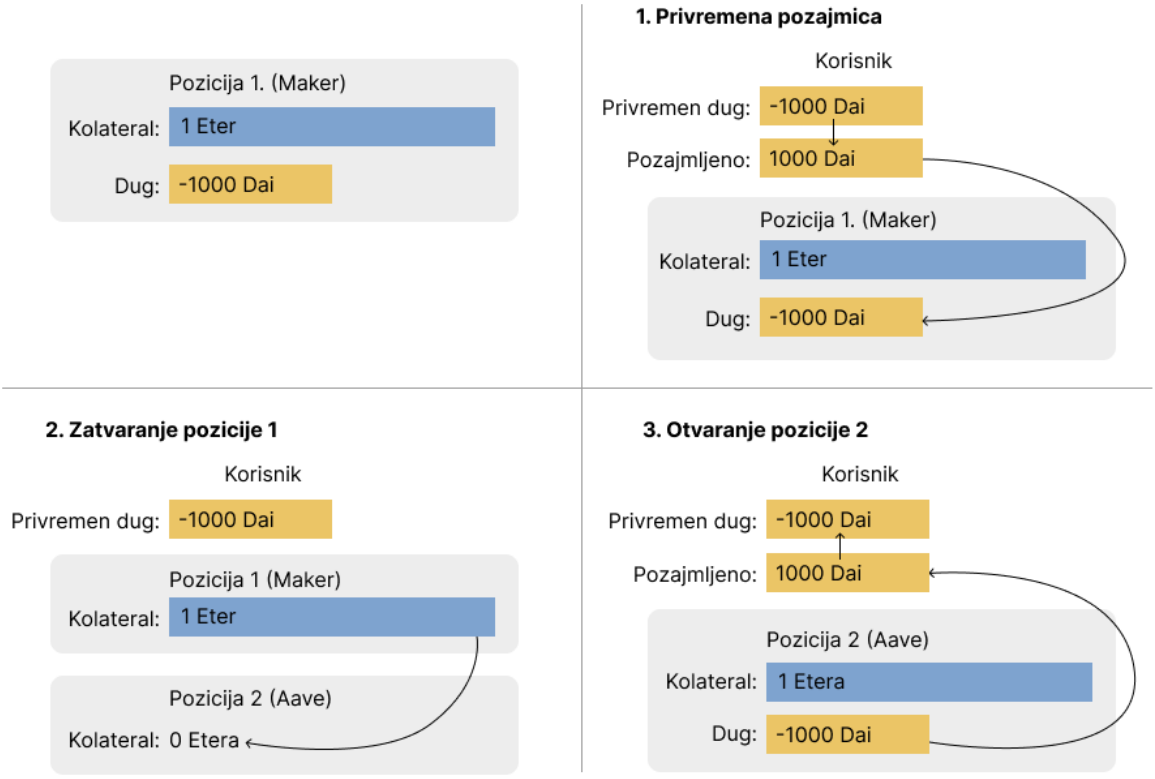
1) iz svoje pozicije povuče kolateral u vrednosti od $5,000;
2) razmeni na menjačnici za okvirno $5,000 u valuti *Dai*;
3) iskoristi kupljen Dai to da vrati dug na svojoj poziciji.

Tako bi se i kolateral i dug smanjili za iznos oko $5,000 ali bi odnos sada bio 300% (kolateral bi bio na $15,000 a dug na $5,000). Ovim procesom, korisnik može da zaštiti svoju poziciju od likvidacije u slučaju pada vrednosti kolaterala. Međutim, ako pretpostavimo da se ovaj proces izvršio u odvojenim transakcijama, primetićemo da se nakon prve transakcije odnos kolateralizacije privremeno spustio na granicu likvidacije, što korisnikovu poziciju čini ranjivom na eksternu likvidaciju sa penalom u slučaju iznenadnog pada vrednosti kolaterala u tom trenutku. Ako bi se ove tri akcije desile atomično u okviru jedne transakcije, proces bi bio sasvim bezbedan.

Neki procesi bi u odvojenim transakcijama bili neefikasni i/ili skupi pa bi bili neizvodljivi. Nadovezaćemo se na prethodni primer: Vidimo da inicijalno korisnik odjednom može da otplati samo oko $5,000 svoje pozicije zato što bi došlo do likvidacije ako bi izvukao više kolaterala, iako tehnički on ima dovoljno kolaterala u poziciji da otplati ceo dug kroz ovakav proces. Ako želi da otplati kompletan dug na ovaj način, morao bi to da radi iz više transakcija. Broj transakcija bi zavisio od početnog odnosa kolateralizacije – ako je ovaj odnos

blizu likvidaciji, u prvoj transakciji bi mogao da otplati mali deo duga jer je mala količina kolaterala „slobodna“, pa bi morao da izvrši veći broj istih transakcija sa rastućom količinom koja se razmenjuje. Dakle, broj transakcija eksponencijalno raste kako se odnos kolateralizacije približava minimalnom, što je posebno nezgodno jer je proces najbitnije izvršiti brzo i efikasno uoči likvidacije. Koncept privremene pozajmice dozvoljava korisniku da pozajmi dovoljno *Dai* da otplati ceo dug, izvuče svoj kolateral, proda ga za *Dai* i time vrati početnu pozajmicu, sve u jednoj transakciji. Koncept privremenih pozajmica (tzv. eng. *flashloans*) se oslanja na ideju da jedna transakcija izvršava više stvari iz prostog razloga što pozajmica traje samo od početka do kraja transakcije.

Spomenućemo još jedan primer ilustrovan slikom 3.1: Pretpostavimo da korisnik ima kolateralizovan dug u protokolu *Maker* kao u prethodnom primeru, i želi da prebaci isti dug na protokol *Aave* iz kog god razloga (manja kamata, niži minimalni odnos kolateralizacije, …). Korisnik može da otplati svoj dug, izvuče kolateral, deponuje ga u drugi protokol i pozajmi originalan dug ponovo. Međutim, ovo nije moguće ukoliko korisnik nema dovoljno sredstava da otplati dug. U teoriji, mogao bi da izvrši proces iz prethodnog primera, a zatim inverzan proces pozajmljivanja i kupovanja kolaterala dok ne dovede poziciju na inicijalno stanje. Međutim, ovo bi zahtevalo veliki broj transakcija, posebno za pozicije sa malo slobodnog kolaterala, a pritom bi uključivalo razmene valuta koje nose sa sobom naknadu. Sve ovo bi činilo proces ekstremno neefikasnim i skupim. Ipak, u svetu decentralizovanih finansija, korisnik može da kreira privremenu pozajmicu, otplati svoj dug, prebaci kolateral na novi protokol, i tamo napravi novi dug kojim bi vratio privremenu pozajmicu.




*Slika 3.1: Ilustracija procesa prebacivanja pozicije na drugi protokol koristeći eksternu privremenu pozajmicu*

Za potrebe ovog rada, složenim transakcijama ćemo smatrati proces sa više koraka, koji potencijalno uključuje interakciju sa više ugovora, u cilju izvršavanja radnje koja bi pri izvršavanju kroz više transakcija bila nemoguća ili značajno manje efikasna ili skuplja.

## 3.2 Pametni ugovori i programski jezik *Solidity*

*Solidity* je najpopularniji, ali ne i jedini programski jezik koji se kompajlira na međukod koji može da se izvršava na virtuelnoj mašini *Eterijum* sistema (EVM). *Solidity* je:

- Objektno-orijentisan
- Statički tipiziran
- Sintaksno sličan jezicima C++ i *JavaScript*.

```
pragma solidity ^0.8.0;
contract MojUgovor {

    uint public brojac = 0;

    event Uvecanje(uint vrednost);

    function pozdrav() public pure returns (string memory) {
        return "Zdravo, svete!";
    }

    function uvecajBrojac() public {
        brojac += 1;
        emit Uvecanje(brojac);
    }
}
```

*Isečak koda 3.1: Primer jednostavnog pametnog ugovora napisanog u jeziku* Solidity

Pametni ugovor je osnovna gradivna jedinica programa na *Eterijum* blokčejnu, analogno klasama u tradicionalnim objektno-orijentisanim jezicima. Ugovor ima svoje stanje (promenljive čije se stanje čuva) i metode. I promenljive i metode pametnog ugovora mogu biti interne, ili javno dostupne. Metode je moguće zaštititi i dodatnim pravilima kroz modifikatore (eng. *modifiers*). Jedan od češćih razloga za ovo je npr. osiguravanje da određene metode može da pozove samo administrator, ili adresa koja je kreirala ugovor. Metode takođe mogu da budu konstantne, odnosno da ne menjaju stanje ugovora, već samo čitaju, obrađuju i vraćaju određene podatke. Ovakve metode se mogu pozvati bez kreiranja transakcije kroz posebnu *RPC* metodu `eth_call` i koriste se za čitanje stanja ugovora. U tom slučaju, klijent nema kontakt sa celom mrežom, već samo sa *RPC* čvorom s kojim komunicira. Ugovori imaju i koncept događaja (eng. *events*) – specifičnih tipiziranih struktura koje mogu da se emituju tokom izvršavanja metode ugovora sa svrhom da komuniciraju stanje ugovora sa eksternim aplikacijama.

## 3.3 Kontekst izvršavanja složene transakcije

Transakcije inherentno imaju sistem autentifikacije tako što je pošiljalac transakcije identifikovan privatnim ključem kojim se potpisuje transakcija. Pošiljalac se, uz druge informacije, može naći u globalnoj strukturi `msg`. Struktura sadrži sledeće informacije:

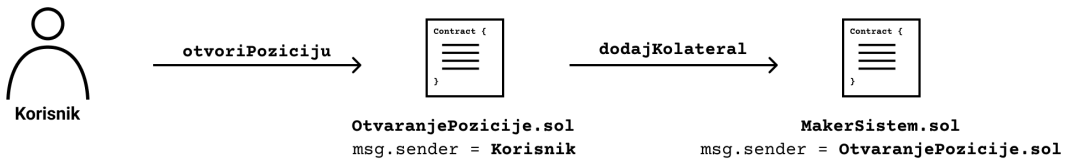
- `sender`: pošiljalac transakcije
- `sig`: prva 4 bajta enkodovanog identifikatora funkcije koja se poziva
- `data`: kodirani argumenti pozvane funkcije
- `gas`: preostao (nepotrošen) gas za izvršavanje transakcije do tog trenutka
- `value`: količina Etera koja je poslata u pozivu/transakciji.

Problem nastaje kada želimo da izvršimo kompleksniju transakciju koja uključuje više ugovora koji autentifikuju pošiljaoca sa `msg.sender`. Ilustrovaćemo problem koristeći pojednostavljen ugovor koji kreira poziciju na protokolu *Maker*, dodaje kolateral i pozajmljuje dug (isečak koda 3.2).

```
contract OtvaranjePozicije {
	function otvoriPoziciju(uint256 kolateral, uint256 dug) public {
		uint256 idPozicije = MakerSistem.novaPozicija();
		MakerSistem.dodajKolateral(kolateral);
		MakerSistem.napraviDug(dug);
		emit OtvaranjePozicije(msg.sender, kolateral, dug);
	}
}
```

*Isečak koda 3.2: Nacrt ugovora koji ovrata poziciju u protokolu Maker*

Unutar funkcije `otvoriPoziciju`, u trećoj liniji, `msg.sender` će zaista biti pošiljalac transakcije. Međutim, kada jedan ugovor poziva metodu drugog, to je svojevrsna, tzv. interna transakcija, i `msg` objekat je poseban za svaki takav poziv. Unutar `MakerSistem` ugovora u funkciji `otvoriPoziciju`, `msg.sender` će u stvari biti adresa ugovora `OtvaranjePozicije` (ilustrovano slikom 3.2). Iz ovog razloga, nije moguće direktno koristiti ovakve ugovore kao „skripte“.




*Slika 3.2: Ilustracija izvršavanja pri pozivu ugovora* `OtvaranjePozicije`

Ovaj problem se rešava konceptom „pametnih novčanika“ i posebnom vrstom internog poziva između ugovora.

Suštinska ideja pametnih novčanika je da emuliraju krajnji korisnički nalog i nadograđuju ga naprednim mogućnostima. Pametni novčanici su uvek ugovori. Isečkom koda 3.3 prikazaćemo pojednostavljen primer jednog ovakvog proksi novčanika i njegovu metodu za izvršavanje.

```
pragma solidity ^0.8.7;

contract Proksi {
   address public vlasnik;

   constructor() {
      vlasnik = msg.sender;
   }

   modifier samoVlasnik() {
      require(msg.sender == vlasnik);
      _;
   }

   Function izvrsi(address ciljniUgovor, bytes memory pozivniPodaci)
      public
      payable
      samoVlasnik
      returns (bytes memory)
   {
      (bool uspeh, bytes memory izlaz) = ciljniUgovor.delegatecall(pozivniPodaci);
      require(uspeh);
      return izlaz;
   }
}
```

*Isečak koda 3.3: Nacrt ugovora koji služi kao jednostavan pametni novčanik*

Prikazana je metoda `execute` koju može da pozove samo korisnik koji je vlasnik ugovora. Glavni detalj ove metode je `delegatecall` koji je identičan standardnom pozivu, osim što se izvršava u kontekstu pozivaoca. U praksi to znači da ugovor može dinamički da učita kod drugog ugovora tokom izvršavanja. Promenljive `msg.sender` i `msg.value` neće biti promenjene, i memorija će i dalje referisati na memoriju pozivaoca. Ako se vratimo na prethodni primer, i zamislimo da je instanca `Proksi` ugovora pozvala `otvoriPoziciju` koristeći `delegatecall`, možemo da zamislimo da se komande iz `otvoriPoziciju` izvršavaju kao da su deo koda funkcije `execute`. Samim tim, `msg.sender` je u svim daljim pozivima adresa instance ugovora `Proksi`. Jasno je da onda korisnik neće direktno biti vlasnik kreirane pozicije, već njegov pametni novčanik, ali je to validno jer isključivo korisnik može njime da upravlja. Možemo da kažemo da je korisnik indirektno vlasnik pozicije. Transakcije koje se izvršavaju koristeći više ugovora gde je potrebna autentifikacija, se u opštem slučaju mogu izvršiti pomoću mehanizma pametnih ugovora koji predstavljaju zastupnike.

Prikazan ugovor je inspirisan `DsProxy` implementacijom koju koristi sistem *DeFi Saver* (više u poglavlju 4). Implementacija pametnog novčanika `DsProxy` u osnovi radi istu stvar kao i prikazan ugovor, ali uključuje značajne optimizacije koje podrazumevaju delove koda pisane na asembler nivou. Razumevanje tih optimizacija bi zahtevalo dubinsko

poznavanje memorijskog rasporeda *Eterijum* virtuelne mašine i mašinskih instrukcija. Kako to nije cilj ovog rada, detalji `DsProxy` i drugi, još kompleksnijih implementacija su izostavljeni. Za potrebe rada zadržaćemo se na mogućnosti ovih ugovora da pruže kontekst za izvršavanje kompleksne transakcije na opisan način. Konceptualno, implementacije svih sistema rade na opisan način koristeći `delegatecall`.

Pametni novčanici, pored spomenute svrhe, mogu biti korisni i na razne druge načine:

- Novčanik može biti obezbeđen za više vlasnika, tako da npr. za izvršavanje transakcije zahteva potpis 2 od 3, ili 3 od 5 vlasnika. Na ovaj način se smanjuje rizik gubitka privatnog ključa, i istovremeno se decentralizuje upravljanje zajedničkim sredstvima. Implementacija se realizuje tako što vlasnici kreiraju kriptografske potpise za datu transakciju, i kada je dovoljno potpisa skupljeno, jedan od njih izvršava transakciju tako što ugovoru šalje parametre za izvršavanje transakcije i prikupljene potpise koji se proveravaju na ugovoru pre izvršavanja.
- Pametni novčanik *Argent* nadograđuje koncept bezbednosnim funkcionalnostima kao što su: dnevni limiti trošenja, povratak pristupa u slučaju gubitka privatnog ključa korišćenjem prijateljskih, unapred autorizovanih adresa.
- Ugovori sistema *Instadapp* su implementirani tako da je ugovor koji korisnik kreira efektivno samo novi pokazivač na implementaciju. Na taj način implementacija lako može da se zameni novom koja ima dodatne funkcionalnosti
- Implementacija `DsProxy` uključuje mehanizam koji omogućava vlasniku da dozvoli drugim adresama da kroz `DsProxy` pozivaju određene metode na određenim ugovorima u njegovo ime.

U primeru na početku poglavlja smo podrazumevali da `MakerSistem` ugovor koristi promenljivu `msg.sender` kao sistem identifikacije, što je najčešće slučaj. Ukoliko bi ugovori uopšteno koristili vrednost originalnog pošiljaoca transakcije za identifikaciju korisnika (što je moguće), implementacija kompleksnih transakcija bi bila trivijalna i ne bi postojala potreba za pametnim ugovorima. Međutim, takav pristup je značajno manje fleksibilan od `msg.sender` pristupa. U određenim slučajevima zapravo ne želimo da koristimo adresu pošiljaoca transakcije kao vlasnika. Primer takvog slučaja su upravo spomenuti *multisig* novčanici za više vlasnika, gde zapravo u `MakerSistem` primeru želimo da *multisig* bude vlasnik pozicije, a ne jedan od njegovih vlasnika koji je poslao transakciju.

## 3.4 Pristup specijalizovanim ugovorima

Najjednostavniji način kreiranja kompleksne transakcije podrazumeva kreiranje ugovora specijalizovanog za određenu akciju koju želimo da izvršimo. Prethodno smo kao primer naveli otvaranje pozicije na protokolu *Maker*.

Ovaj pristup pruža najveću fleksibilnost jer je kreatoru dostupna celokupna mogućnost programskog jezika, uključujući promenljive, petlje, i grananje. To znači da je moguće kreirati kompleksnije akcije koje zavise od međuvrednosti tokom izvršavanja, kao i preuzeti ažurne

podatke u trenutku izvršavanja od drugih ugovora. Pristup takođe dozvoljava razne optimizacije korišćenja gasa. Sa druge strane očigledno zahteva značajno tehničko i domensko znanje pisanja ugovora. Ovaj pristup se može smatrati skupim jer zahteva postavljanje novog ugovora na blokčejn za svaku novu kompleksnu operaciju, kao i svaku izmenu, koliko god mala bila.

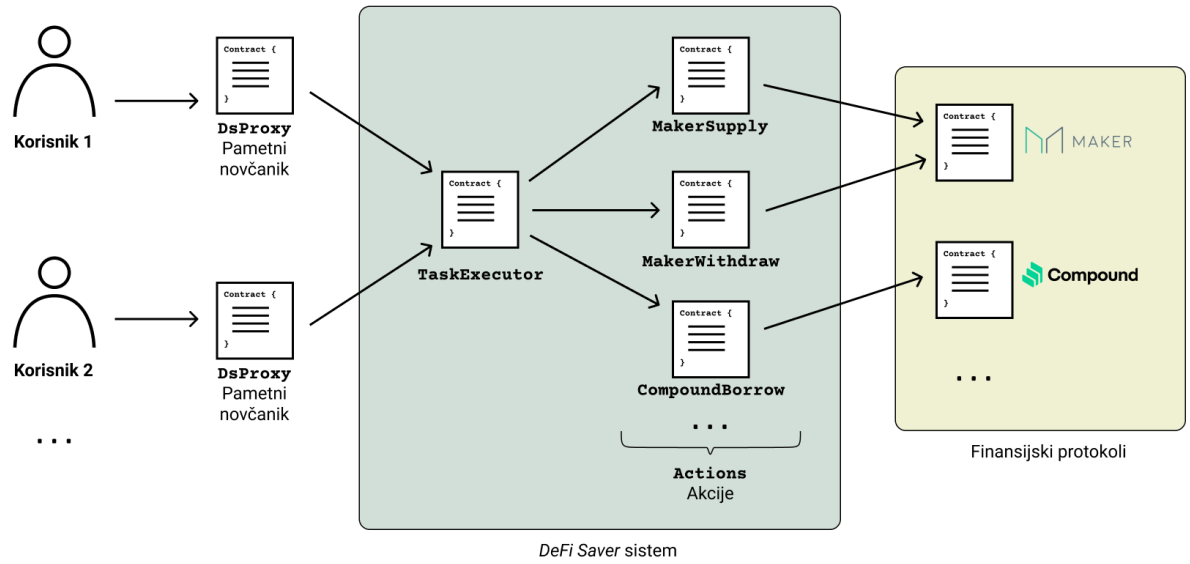
## 3.5 Generalizovani sistemi

U određenim slučajevima presudan je brz i skalabilan razvoj različitih interakcija. Za takve slučajeve realizovani su razni generalizovani sistemi koji u nekoj meri žrtvuju granularnost i moć razvoja na nivou pametnih ugovora zarad lakoće razvoja samih interakcija. U ovom poglavlju biće prikazana 3 sistema kreirana od strane različitih kompanija za sopstvene potrebe.

### 3.5.1 Generalizovani sistem *DeFi Saver*

*DeFi Saver* je softver za menadžment finansijskih investicija sa akcentom na napredne, kompleksne interakcije koje nisu moguće na osnovnim interfejsima protokola. *DeFi Saver* koristi standardizovane pametne novčanike bazirane na ugovoru `DsProxy` koristeći registar koji održava tim protokola *Maker*. Iz ovog razloga, pozicije kreirane kroz ovu platformu su dostupne i u drugim interfejsima.

Za svoje potrebe, tim koji razvija platformu *DeFi Saver* je razvio sistem za generalizovane složene transakcije ilustrovan slikom 3.3. Kreiran je skup akcija, gde svaka akcija predstavlja jednu atomičnu interakciju s jednim protokolom, apstrahovanu tako da može da se uklopi u recept. Metafora recepta se koristi da označi skup akcija koje se izvršavaju atomično i sekvencijalno, sa definisanim parametrima i redosledom.




*Slika: 3.3: Pregled sistema za pametnih ugovora za recepte platforme DeFi Saver*

Formalno, akcija je pametan ugovor koji ispunjava određen interfejs koji dozvoljava prosleđivanje parametara i povratnih vrednosti u kontekstu recepta. Da bi akcija bila deo sistema, mora biti upisana u centralni registar (ugovor *DFSRegistry*). Centralni registar čuva mapiranje imena akcije u adresu njenog ugovora. Postojeće akcije se mogu menjati, i novi protokoli se mogu dodavati. Ovo može da uradi isključivo odobren administrator iz tima. Ovo znači da je sistem zatvoren u smislu da korisnik ne može sam da napravi akciju i koristi je u sistemu, već je ograničen na skup akcija odobrenih od strane administratora.

Izvršavanje transakcije u sistemu *DeFi Saver* počinje u ugovoru *RecipeExecutor* čiji je kod prikazan u isečku 3.4. On je zadužen da pozove svaku akciju, sačuva njenu povratnu vrednost, i prosledi je narednim akcijama. Svaka akcija dobija i vraća vrednosti kodiranu u niz bajtova radi maksimalne interoperabilnosti. Recept se izvršava pomoću metode *executeRecipe* (koji samo poziva internu metodu *_executeActions*). Ova metoda poziva svaku akciju iz recepta sekvencijalno, i čuva njihove povratne vrednosti kako bi mogle da se iskoriste u narednim akcijama. Za svaku akciju se poziva *_executeAction* koji uzima adresu akcije iz registra, i onda poziva *execute* nad sobom (setimo se da se ovo izvršava u kontekstu korisnikovog pametnog novčanika) za tu akciju.

```
struct Recipe {
    string name;
    bytes[] callData;
    bytes32[] subData;
    bytes4[] actionIds;
    uint8[][] paramMapping;
}

contract RecipeExecutor is StrategyModel, ProxyPermission, AdminAuth, CoreHelper {
    /// @notice Called directly through DsProxy to execute a recipe
    /// @dev This is the main entry point for Recipes executed manually
    /// @param _currRecipe Recipe to be executed
    function executeRecipe(Recipe calldata _currRecipe) public payable {
        _executeActions(_currRecipe);
    }

    /// @notice Runs all actions from the recipe
    /// @dev FL action must be first and is parsed separately, execution will go to
_executeActionsFromFL
    /// @param _currRecipe Recipe to be executed
    function _executeActions(Recipe memory _currRecipe) internal {
        address firstActionAddr = registry.getAddr(_currRecipe.actionIds[0]);

        bytes32[] memory returnValues = new
bytes32[](_currRecipe.actionIds.length);

        if (isFL(firstActionAddr)) {
            _parseFLAndExecute(_currRecipe, firstActionAddr, returnValues);
        } else {
            for (uint256 i = 0; i < _currRecipe.actionIds.length; ++i) {
                returnValues[i] = _executeAction(_currRecipe, i, returnValues);
            }
        }

        /// log the recipe name
        DefisaverLogger(DEFISAVER_LOGGER).logRecipeEvent(_currRecipe.name);
    }

    /// @notice Gets the action address and executes it
    /// @param _currRecipe Recipe to be executed
    /// @param _index Index of the action in the recipe array
```

```
    /// @param _returnValues Return values from previous actions
    function _executeAction(
        Recipe memory _currRecipe,
        uint256 _index,
        bytes32[] memory _returnValues
    ) internal returns (bytes32 response) {

        address actionAddr = registry.getAddr(_currRecipe.actionIds[_index]);

        response = IDSProxy(address(this)).execute(
            actionAddr,
            abi.encodeWithSignature(
                "executeAction(bytes,bytes32[],uint8[],bytes32[])",
                _currRecipe.callData[_index],
                _currRecipe.subData,
                _currRecipe.paramMapping[_index],
                _returnValues
            )
        );
    }

    // ...
}
```

*Isečak koda 3.4: Deo koda ugovora RecipeExecutor [8]*

Recept, pored svog imena, za svaku akciju sadrži sledeće podatke:

- *actionId*: identifikator akcije u registru, SHA3 heš njenog imena;
- *callData*: niz bajtova koji se pretvaraju u argumente za izvršavanje;
- *paramMapping*: pokazivači na prethodne akcije, čija povratna vrednost se uzima kao argument (umesto argumenta iz *callData*);
- *subData*: tiče se sistema automatizacije, pa nije relevantan za ovaj rad.

Najlakše je objasniti kako *paramMapping* radi na prostom primeru. Sa decentralizovanim menjačnicama ne možemo unapred biti sigurni koliko ćemo tačno dobiti sredstava, ali često želimo da ta sredstva iskoristimo u narednim akcijama. Zamislimo recept koji sadrži akciju koja prodaje sredstva u jednoj valuti za drugu valutu (*SellAction*), i akciju koja kupljena sredstva deponuje u poziciju (*MakerSupply*). Jasno je da želimo da deponujemo sva sredstva u poziciju, umesto da nam mali iznos ostane u novčaniku. Recimo da *MakerSupply* akcija ima identifikator pozicije (*cdpId* – recimo da je u ovom slučaju 1234) i količinu sredstava za deponovanje kao argumente. U slučaju ovog recepta, za akciju *MakerSupply*, *callData* će sadržati `[1234, -1]`, ali će *paramMapping* sadržati `[0, 1]` tako da drugi element upućuje na povratnu vrednost akcije sa rednim brojem 1. Prvi element koji ima vrednost 0 nema značenje. Vrednost argumenta koji označava količinu (-1) će biti zamenjen povratnom vrednošću akcije sa rednim brojem 1. Na ovaj način, donekle smo rekreirali bazično ponašanje pokazivača.

Svaka akcija u sistemu ima svoj ugovor koji prati određen interfejs (*ActionBase*) radi kompatabilnosti sa sistemom. *RecipeExecutor* poziva *executeAction* prosleđujući sve vrednosti kodirane u niz bajtova. Metoda *executeAction* dekodira vrednosti i menja ih povratnim vrednostima gde je potrebno, a zatim poziva internu metodu koja je u stvari ključni deo akcije.

U isečku koda 3.5 prikazana je implementacija akcije za deponovanje u poziciju na protokolu Maker.

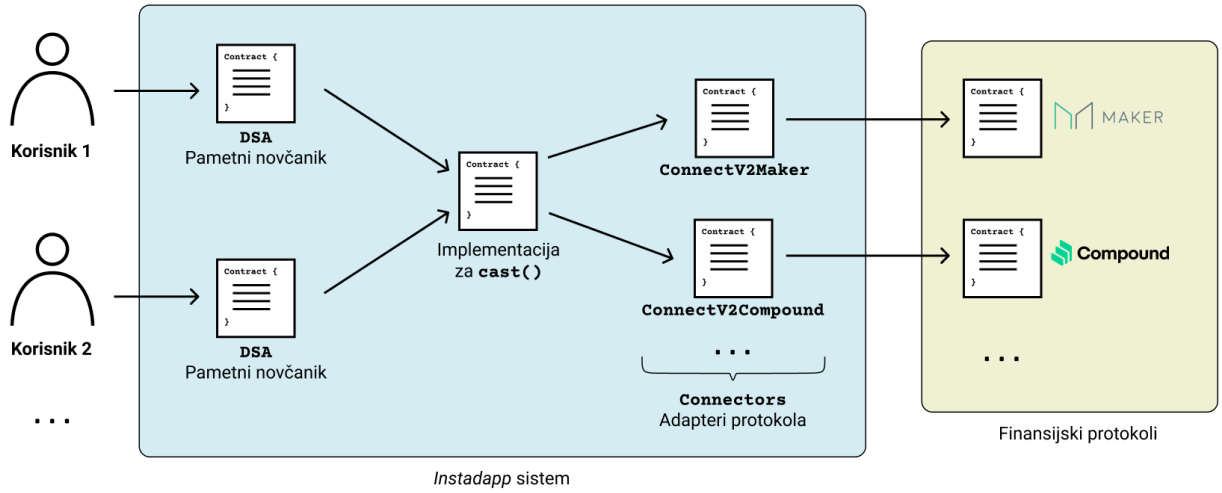
```
contract McdSupply is ActionBase, McdHelper {
    function executeAction(
        bytes[] memory _callData,
        bytes[] memory _subData,
        uint8[] memory _paramMapping,
        bytes32[] memory _returnValues
    ) public payable override returns (bytes32) {
        (uint256 vaultId, uint256 amount, address joinAddr, address from, address
mcdManager) =
            parseInputs(_callData);

        vaultId = _parseParamUint(vaultId, _paramMapping[0], _subData,
_returnValues);
        amount = _parseParamUint(amount, _paramMapping[1], _subData,
_returnValues);
        joinAddr = _parseParamAddr(joinAddr, _paramMapping[2], _subData,
_returnValues);
        from = _parseParamAddr(from, _paramMapping[3], _subData, _returnValues);

        uint256 returnAmount = _mcdSupply(vaultId, amount, joinAddr, from,
mcdManager);

        return bytes32(returnAmount);
    }

    // ...

    //////////////////////////// ACTION LOGIC ////////////////////////////

    /// @notice Supplies collateral to the vault
    /// @param _vaultId Id of the vault
    /// @param _amount Amount of tokens to supply
    /// @param _joinAddr Join address of the maker collateral
    /// @param _from Address where to pull the collateral from
    /// @param _mcdManager The manager address we are using [mcd, b.protocol]
    function _mcdSupply(
        uint256 _vaultId,
        uint256 _amount,
        address _joinAddr,
        address _from,
        address _mcdManager
    ) internal returns (uint256) {
        address tokenAddr = getTokenFromJoin(_joinAddr);
        IManager mcdManager = IManager(_mcdManager);

        // if amount type(uint).max, pull current _from balance
        if (_amount == type(uint256).max) {
            _amount = tokenAddr.getBalance(_from);
        }

        // Pull the underlying token and join the maker join pool
        tokenAddr.pullTokensIfNeeded(_from, _amount);
        tokenAddr.approveToken(_joinAddr, _amount);
        IJoin(_joinAddr).join(address(this), _amount);

        // format the amount we need for frob
        int256 convertAmount = toPositiveInt(convertTo18(_joinAddr, _amount));

        // Supply to the vault balance
        vat.frob(
            mcdManager.ilks(_vaultId),
            mcdManager.urns(_vaultId),
            address(this),
            address(this),
            convertAmount,
```

```
            0
        );

        logger.Log(
            address(this),
            msg.sender,
            "McdSupply",
            abi.encode(_vaultId, _amount, _joinAddr, _from, _mcdManager)
        );

        return _amount;
    }

    function parseInputs(bytes[] memory _callData)
        internal
        pure
        returns (
            uint256 vaultId,
            uint256 amount,
            address joinAddr,
            address from,
            address mcdManager
        )
    {
        vaultId = abi.decode(_callData[0], (uint256));
        amount = abi.decode(_callData[1], (uint256));
        joinAddr = abi.decode(_callData[2], (address));
        from = abi.decode(_callData[3], (address));
        mcdManager = abi.decode(_callData[4], (address));
    }
}
```

*Isečak koda 3.5: Sadržaj ugovora koji implementira akciju za deponovanje u poziciju na protokolu Maker [9]*

### 3.5.2 Generalizovani sistem *Instadapp*

*Instadapp* je u više pogleda konkurent platformi *DeFi Saver*. Iako je tržišni fokus drugačiji, oba projekta omogućavaju korisnicima napredne interakcije sa finansijskim protokolima, a da bi to omogućili, oba projekta imaju sistem za kreiranje generalizovanih transakcija. Sistem *Instadapp, ilustrovan slikom 3.4,* je zatvoreniji od sistema *DeFi Saver* u smislu da je sistem pametnih novčanika kreiran od strane njihovog tima, i administratori imaju određen stepen kontrole nad sistemom u vidu biranja akcija koje se mogu izvršiti unutar sistema, kao i nadograđivanja i menjanja funkcionalnosti novčanika i drugih delova sistema.




*Slika 3.4: Pregled sistema pametnih ugovora za složene transakcije platforme Instadapp*

Kao i u sistemu *DeFi Saver*, transakcija se izvršava u kontekstu korisnikovog pametnog novčanika. Razlike su relativno male.

- *Instadapp* žrtvuje fleksibilnost zarad optimizacija. Logika iskorišćavanja povratnih vrednosti nije implicitna i dostupna kao u receptima sistema *DeFi Saver*. Akcija mora eksplicitno da podrži čuvanje i čitanje vrednosti za pojedinačnu promenljivu. Konkretno, akcije kao parametre primaju `getId` i `setId`, proizvoljne brojeve koji predstavljaju apstrahovane memorijske lokacije iz kojih se čita, odnosno u koje se upisuje vrednost. Spomenuti primer razmene sredstava i deponovanja bi bio realizovan na sledeći način: u akciji koja izvršava prodaju (*Sell*) bi bio unet parametar *setId* sa proizvoljnom vrednošću (npr. 1), i u tu memorijsku lokaciju bi akcija upisala svoju povratnu vrednost. Dalje bi akciji *MakerSupply* bio unet parametar `getId` sa istom vrednošću kao u prethodnoj akciji (1) da pročita vrednost. Ako je parametar `getId` podešen, pročitana vrednost zamenjuje vrednost prosleđenu parametrom *amount*. Ovaj sistem je rudimentarniji ali pokriva većinu jednostavnih slučajeva korišćenja u praksi. Akcije moraju eksplicitno da dozvole prosleđivanje parametara, ali time štede vreme izvršavanja i memorijski prostor u odnosu na *DeFi Saver*, gde se većina povratnih vrednosti nikad ne koristi.
- Sistem podrazumeva da sredstva uvek idu sa pametnog novčanika, naspram akcija u sistemu *DeFi Saver* koje uglavnom traže ulaznu i izlaznu adresu. I ovaj primer ilustruje fokus na pragmatičnost sistema *Instadapp* naspram fokusa na fleksibilnost sistema *DeFi Saver*. Napomena je da postoji odvojena akcija koja eksplicitno deponuje sredstva sa korisničkog naloga na pametni novčanik.

Izvršavanje recepata u sistemu *Instadapp* počinje kroz `cast` funkciju koja poziva funkciju `spell` za svaku akciju. Funkcije su prikazane u isečku koda 3.6. Može se videti da je logika dosta jednostavnija u ovom delu u poređenju sa sistemom DeFi Saver. Odgovornost ovog ugovora je dosta mala – da pozove sve akcije redom i zapiše njihovo izvršavanje. Čuvanje i dohvatanje povratnih vrednosti se dešava kao deo logike svake akcije ponaosob.

```
contract InstaImplementationM1 is Constants {
    function decodeEvent(bytes memory response) internal pure returns (string
memory _eventCode, bytes memory _eventParams) {
        if (response.length > 0) {
            (_eventCode, _eventParams) = abi.decode(response, (string, bytes));
        }
    }

     /**
     * @dev Delegate the calls to Connector.
     * @param _target Connector address
     * @param _data CallData of function.
    */
    function spell(address _target, bytes memory _data) internal returns (bytes
memory response) {
        require(_target != address(0), "target-invalid");
        assembly {
            let succeeded := delegatecall(gas(), _target, add(_data, 0x20),
mload(_data), 0, 0)
            let size := returndatasize()

            response := mload(0x40)
            mstore(0x40, add(response, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
                case 1 {
                    // throw if delegatecall failed
                    returndatacopy(0x00, 0x00, size)
                    revert(0x00, size)
                }
        }
    }

    /**
     * @dev This is the main function, Where all the different functions are called
     * from Smart Account.
     * @param _targetNames Array of Connector address.
     * @param _datas Array of Calldata.
    */
    function cast(
        string[] calldata _targetNames,
        bytes[] calldata _datas,
        address _origin
    )
    external
    payable
    returns (bytes32) // Dummy return to fix instaIndex buildWithCast function
    {
        uint256 _length = _targetNames.length;
        require(_auth[msg.sender] || msg.sender == instaIndex, "1: permission-
denied");
        require(_length != 0, "1: length-invalid");
        require(_length == _datas.length , "1: array-length-invalid");

        string[] memory eventNames = new string[](_length);
        bytes[] memory eventParams = new bytes[](_length);

        (bool isOk, address[] memory _targets) =
ConnectorsInterface(connectorsM1).isConnectors(_targetNames);

        require(isOk, "1: not-connector");

        for (uint i = 0; i < _length; i++) {
            bytes memory response = spell(_targets[i], _datas[i]);
            (eventNames[i], eventParams[i]) = decodeEvent(response);
        }
```

```
        emit LogCast(
            _origin,
            msg.sender,
            msg.value,
            _targetNames,
            _targets,
            eventNames,
            eventParams
        );
    }
}
```

*Isečak koda 3.6: Kôd ugovora zaduženog za izvršavanje interakcija u sistemu Instadapp [10]*

S obzirom da prosleđivanje (odnosno čuvanje i dohvatanje) povratnih vrednosti radi sama akcija, a ne ugovor koji je poziva, ugovor same akcije ne mora da implementira generalizovani interfejs za pozivanje (na kom insistira sistem *DeFi Saver*). Ova fleksibilnost značajno umanjuje količinu šablonskog koda u Instadapp akcijama, i omogućava im da jednostavno budu metode unutar ugovora, a ne zasebni ugovori. U sistemu *Instadapp* najčešće implementira jedan *Resolver* ugovor po protokolu koji podržava. U isečku koda 3.7 je prikazan jedan takav *Resolver* za protokol *Maker,* skraćen tako da samo uključuje metodu za deponovanje.

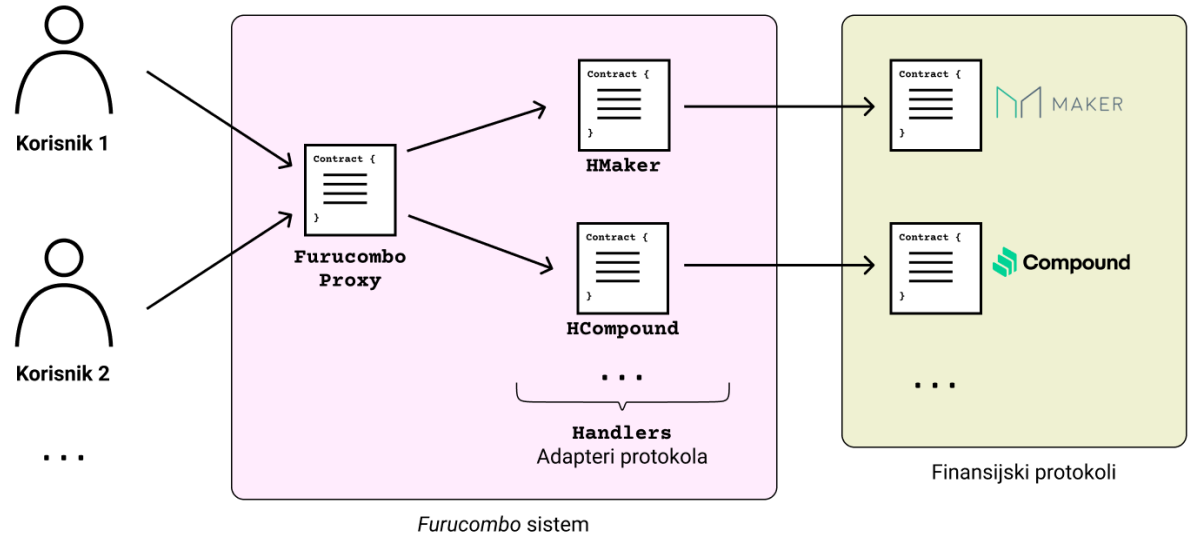
```
abstract contract MakerResolver is Helpers, Events {
    /**
     * @dev Deposit ETH/ERC20_Token Collateral.
     * @notice Deposit collateral to a MakerDAO vault
     * @param vault Vault ID. (Use 0 for last opened vault)
     * @param amt The amount of tokens to deposit. (For max: `uint256(-1)`)
     * @param getId ID to retrieve amt.
     * @param setId ID stores the amount of tokens deposited.
    */
    function deposit(
        uint256 vault,
        uint256 amt,
        uint256 getId,
        uint256 setId
    ) external payable returns (string memory _eventName, bytes memory _eventParam)
{
        uint _amt = getUint(getId, amt);
        uint _vault = getVault(vault);
        (bytes32 ilk, address urn) = getVaultData(_vault);

        address colAddr = instaMapping.gemJoinMapping(ilk);
        TokenJoinInterface tokenJoinContract = TokenJoinInterface(colAddr);
        TokenInterface tokenContract = tokenJoinContract.gem();

        if (isEth(address(tokenContract))) {
            _amt = _amt == uint(-1) ? address(this).balance : _amt;
            tokenContract.deposit{value: _amt}();
        } else {
            _amt = _amt == uint(-1) ?  tokenContract.balanceOf(address(this)) :
_amt;
        }

        approve(tokenContract, address(colAddr), _amt);
        tokenJoinContract.join(address(this), _amt);

        VatLike(managerContract.vat()).frob(
            ilk,
            urn,
```

```
            address(this),
            address(this),
            toInt(convertTo18(tokenJoinContract.dec(), _amt)),
            0
        );

        setUint(setId, _amt);

        _eventName = "LogDeposit(uint256,bytes32,uint256,uint256,uint256)";
        _eventParam = abi.encode(_vault, ilk, _amt, getId, setId);
    }

    // ...
}

contract ConnectV2MakerDAO is MakerResolver {
    string public constant name = "MakerDAO-v1.2";
}
```

*Isečak koda 3.7: Kôd ugovora zaduženog za intereakcije sa protokolom Maker, konkretno funkcije za deponovanje [11]*

### *3.5.3* Generalizovani sistem *Furucombo*

*Furucombo* softver ne cilja da ponudi sopstveno kompletno rešenje za menadžment pozicija, već je namenjen kao koristan, dopunski alat postojećim protokolima. Iz tog razloga je sistem, ilustrovan slikom 3.5, optimizovan za drugačije ciljeve nego druga dva – pogodnost krajnjeg korisnika iznad kompletne funkcionalnosti. Prvenstveno, ne zahteva od korisnika da ima svoj pametni novčanik. Dok na druga dva sistema korisnik radi sve kroz sopstveni proksi, ovde se transakcije za sve korisnike izvršavaju u kontekstu jednog zajedničkog proksija. Sistem *Furucombo* koristi slične zaobilazne načine da ponudi korisniku sličnu, ako ne jednaku, funkcionalnost kao druga dva sistema.




*Slika 3.5: Pregled sistema pametnih ugovora za složene transakcije platforme Furucombo*

Arhitekturalna odluka o zajedničkom proksiju ima sledeće implikacije:

- Korisnici ne moraju da kreiraju pametan novčanik i importuju pozicije da bi koristili *Furucombo* alat, već je alat napravljen da podržava postojeće pozicije. Ipak, to znači

da je opseg akcija koje se mogu izvršiti ograničen. Razlog je jednostavan: u većini slučajeva nije dozvoljeno da adresa interaguje sa pozicijom koja joj ne pripada. Konkretan primer je da, isto kao što adresa A ne može da napravi dug u ime adrese B, *Furucombo* proxy (u čijem kontekstu se izvršava transakcija) ne može da napravi dug u ime korisnika.

- Struktura sa zajedničkim proksijem donosi određene bezbednosne probleme.
    1. Pošto se dozvola za transfer sredstava (*ERC20 approval*) daje proksiju, moglo bi biti opasno ako korisnik pronađe način da izvrši maliciozni kod na proksiju, jer bi time mogao da povuče sva tuđa sredstva za koje proksi ima dozvolu. U sistemima sa odvojenim ugovorima ovakav rizik je smanjen jer postoji izolacija između korisnika.
    2. Sličan problem se dešava i u slučaju operacija za koje je potrebna dozvola nad pozicijom na protokolu *Maker*. *Furucombo* mora da zahteva da korisnik dozvoli zajedničkom proksiju da kontroliše njegovu poziciju. Podešavanje ove kontrole (na nivou ugovora *CdpManager* u protokolu *Maker*) je prilično grubo, pa se zajedničkom proksiju daje kompletna kontrola nad pozicijom, a *Furucombo* onda implementira dodatne sigurnosne mere u njihovom proksiju.
    3. Ako je transakcija takva da je na ugovoru koji izvršava transakciju greškom ostao višak sredstava, u sistemu izolovanih proksija (pametnih novčanika) samo adresa koja je vlasnik može povući taj višak nazad, dok u sistemu sa deljenim proksijem svako ima jednake dozvole da ga povuče. Ovaj problem jeste rešiv, na primer internim čuvanjem balansa tokena koje je korisnik deponovao i iskoristio, ali nije način na koji je Furucombo impelementiran.

Slično prethodnim sistemima, izvršavanje transakcije počinje pozivom ka ugovoru za izvršavanje, čija je odgovornost da pozove sve akcije uključene u transakciju. Kod tog ugovora je prikazan u isečku koda 3.8. U skladu sa grafičkim interfejsom alata, komentari u kodu često nazivaju akcije „kockicama“ („*cubes*“), a transakcije „kombinacijama“ („combo“).

```
/**
 * @title The entrance of Furucombo
 * @author Ben Huang
 */
contract Proxy is IProxy, Storage, Config {

// ...

    /**
     * @notice Combo execution function. Including three phases: pre-process,
     * exection and post-process.
     * @param tos The handlers of combo.
     * @param configs The configurations of executing cubes.
     * @param datas The combo datas.
     */
    function batchExec(
        address[] calldata tos,
        bytes32[] calldata configs,
```

```
        bytes[] memory datas
    ) external payable override isNotHalted isNotBanned {
        _preProcess();
        _execs(tos, configs, datas);
        _postProcess();
    }

    /**
     * @notice The execution interface for callback function to be executed.
     * @dev This function can only be called through the handler, which makes
     * the caller become proxy itself.
     */
    function execs(
        address[] calldata tos,
        bytes32[] calldata configs,
        bytes[] memory datas
    ) external payable override isNotHalted isNotBanned isInitialized {
        require(msg.sender == address(this), "Does not allow external calls");
        _execs(tos, configs, datas);
    }

    /**
     * @notice The execution phase.
     * @param tos The handlers of combo.
     * @param configs The configurations of executing cubes.
     * @param datas The combo datas.
     */
    function _execs(
        address[] memory tos,
        bytes32[] memory configs,
        bytes[] memory datas
    ) internal {
        bytes32[256] memory localStack;
        uint256 index = 0;

        require(
            tos.length == datas.length,
            "Tos and datas length inconsistent"
        );
        require(
            tos.length == configs.length,
            "Tos and configs length inconsistent"
        );
        for (uint256 i = 0; i < tos.length; i++) {
            address to = tos[i];
            bytes32 config = configs[i];
            bytes memory data = datas[i];
            // Check if the data contains dynamic parameter
            if (!config.isStatic()) {
                // If so, trim the exectution data base on the configuration and
stack content
                _trim(data, config, localStack, index);
            }
            // Emit the execution log before call
            bytes4 selector = _getSelector(data);
            emit LogBegin(to, selector, data);

            // Check if the output will be referenced afterwards
            bytes memory result = _exec(to, data);

            // Emit the execution log after call
            emit LogEnd(to, selector, result);

            if (config.isReferenced()) {
                // If so, parse the output and place it into local stack
                uint256 num = config.getReturnNum();
                uint256 newIndex = _parse(localStack, result, index);
                require(
                    newIndex == index + num,
                    "Return num and parsed return num not matched"
```

```
                );
                index = newIndex;
            }

            // Setup the process to be triggered in the post-process phase
            _setPostProcess(to);
        }
    }

    /**
     * @notice The execution of a single cube.
     * @param _to The handler of cube.
     * @param _data The cube execution data.
     */
    function _exec(address _to, bytes memory _data)
        internal
        returns (bytes memory result)
    {
        require(_isValidHandler(_to), "Invalid handler");
        _addCubeCounter();
        assembly {
            let succeeded := delegatecall(
                sub(gas(), 5000),
                _to,
                add(_data, 0x20),
                mload(_data),
                0,
                0
            )
            let size := returndatasize()

            result := mload(0x40)
            mstore(
                0x40,
                add(result, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(result, size)
            returndatacopy(add(result, 0x20), 0, size)

            switch iszero(succeeded)
                case 1 {
                    revert(add(result, 0x20), size)
                }
        }
    }

// ...

}
```

*Isečak koda 3.8: Kôd ugovora zaduženog za izvršavanje interakcija u sistemu Furucombo [12]*

Treba napomenuti da Furucombo nema sistemsku logiku za prosleđivanje povratnih vrednosti. Interfejs može da unapred proceni očekivanu povratnu vrednosti i prikaže je korisniku. Kao parametar pri izvršavanju šalje se upravo ta procenjena vrednost. Kako bi se zaobišao ovaj problem, kao i smanjili drugi bezbednosni rizici, Furucombo ima logiku koja se izvršava nakon svih akcija u transakciji, i služi za validaciju. Validacija najčešće uključuje proveru i slanje preostalog balansa ERC20 sredstava nazad korisniku. Pored toga, svaka akcija (*Handler*) može implementirati svoju validacionu logiku za postprocesiranje. Primer za sopstvenu validaciju bi bio slučaj za otvaranje nove pozicije na protokolu *Maker*. Ona se otvara od strane zajedničkog proksija, a transferuje se korisniku tek na kraju transakcije kako bi proksi

i dalje imao dozvolu da izvršava operacije nad pozicijom[4]. Pošto se akcije izvršavaju u kontekstu (zajedničkog) proksija, dele memoriju sa ugovorom za izvršavanje, pa imaju pristup istoj stek strukturi. Nakon izvršavanja svake akcije, ona ubacuje informacije potrebne za njeno postprocesiranje. U isečku koda 3.9 prikazan je taj proces, dok je čuvanje podataka za postprocesiranje na kraju akcije prikazano u isečcima 3.10 i 3.11.

```
function _postProcess() internal {
   // Handler type will be parsed at the beginning. Will send the token back to
    // user if the handler type is "Token". Will get the handler address and
    // execute the customized post-process if handler type is "Custom".
    while (stack.length > 0) {
        bytes32 top = stack.get();
        // Get handler type
        HandlerType handlerType = HandlerType(uint96(bytes12(top)));
        if (handlerType == HandlerType.Token) {
            address addr = address(uint160(uint256(top)));
            uint256 amount = IERC20(addr).balanceOf(address(this));
            if (amount > 0) IERC20(addr).safeTransfer(msg.sender, amount);
        } else if (handlerType == HandlerType.Custom) {
            address addr = stack.getAddress();
            _exec(addr, abi.encodeWithSelector(POSTPROCESS_SIG));
        } else {
            revert("Invalid handler type");
        }
    }

    // Balance should also be returned to user
    uint256 amount = address(this).balance;
    if (amount > 0) msg.sender.transfer(amount);

    // Reset the msg.sender and cube counter
    _resetSender();
    _resetCubeCounter();
}
```

*Isečak koda 3.9: Funkcija _postProcess u ugovoru za izvršavanje, koja poziva svaku funkciju za postprocesiranje [12]*

```
function swapExactETHForTokens(
    uint256 value,
    uint256 amountOutMin,
    address[] calldata path
) external payable returns (uint256 amount) {
    if (path.length < 2)
        _revertMsg("swapExactETHForTokens", "invalid path");
    address tokenOut = path[path.length - 1];

    // Get uniswapV2 router
    IUniswapV2Router02 router = IUniswapV2Router02(UNISWAPV2_ROUTER);
    value = _getBalance(address(0), value);
    try
        router.swapExactETHForTokens{value: value}(
            amountOutMin,
            path,
```

[4] Davanje dozvole, u slučaju ERC20 sredstava i u slučaju pozicije na protokolu *Maker*, ne može da se uradi u sred neke druge transakcije. U okviru ugovora koji čuva dozvole, koristi se adresa koja je pozvala funkciju (*msg.sender*), a ne adresa koja je poslala transakciju. Ovo je namerno implementirano kao bezbednosni mehanizam, da pametni ugovor ne može da prevari korisnika tako što dodeli sebi dozvolu bez korisnikove eksplicitne potvrde.

```
            address(this),
            now + 1
        )
    returns (uint256[] memory amounts) {
        amount = amounts[amounts.length - 1];
    } catch Error(string memory reason) {
        _revertMsg("swapExactETHForTokens", reason);
    } catch {
        _revertMsg("swapExactETHForTokens");
    }

    _updateToken(tokenOut);
}

// ...

function _updateToken(address token) internal {
    stack.setAddress(token);
}
```

*Isečak koda 3.10: Funkcija swapExactETHForTokens iz ugovora zaduženog za Uniswap menjačnicu, koja menja Eter za drugu valutu. Na kraju funkcije se poziva updateToken sa adresom valute u koju je zamenjen eter. updateToken stavlja tu adresu na stek, a onda Proxy šalje celokupan balans na adresu korisnika. [13]*

```
function postProcess() external payable override {
    bytes4 sig = stack.getSig();
    // selector of openLockETHAndDraw(uint256,address,address,bytes32,uint256)
    // and openLockGemAndDraw(address,address,bytes32,uint256,uint256)
    if (sig == 0x5481e4a4 || sig == 0x73af24e7) {
        _transferCdp(uint256(stack.get()));
        uint256 amount = IERC20(DAI_TOKEN).balanceOf(address(this));
        if (amount > 0)
            IERC20(DAI_TOKEN).safeTransfer(_getSender(), amount);
    } else revert("Invalid post process");
}
```

*Isečak koda 3.11: Funkcija postProcess iz ugovora zaduženog za Maker akcije, koja stavlja na stek posebnu funkciju za postprocesiranje akcija koje otvaraju nove pozicije. Ta funkcija prebacuje vlasništvo novootvorene pozicije sa zajedničkog Furucombo proksija na korisničku adresu. [14]*

### 3.5.4 Poređenje sistema

Sistemi *DeFi Saver* i *Instadapp* su zamišljeni kao ekosistemi gde se transakcije izvršavanju na respektivnim pametnim novčanicima. `DsProxy` je po dizajnu namenjen kao ekstenzija eksternog novčanika. Kao takav je maksimalno generalizovan, i dozvoljava izvršavanje bilo kakve transakcije – njegova `execute` metoda prima adresu i bajtove poziva (*calldata*) koji se izvršavaju direktno. Ipak, akcije koje se mogu izvršiti u okviru sistema *DeFi Saver* su ipak ograničene na one koje su dodate u sistem od strane administratora ugovora. Instadapp pametni ugovori postoje isključivo unutar zatvorenog sistema za menadžment finansijskih pozicija. Kao takvi, ograničeni su isključivo na akcije u sistemu Instadapp (takođe dodate u sistem od strane administratora). Furucombo je namenjen kao dopunski alat isključivo sa svrhom kreiranja složenih transakcija. Što znači da ne zahteva kreiranje pametnog novčanika ili importovanje pozicija kao u slučaju pomenutih sistema. Pozicije koje se menadžuju, za razliku od prethodna dva sistema, nisu pod vlasništvom proksija u čijem kontekstu se izvršava transakcija. Iz tog razloga, za sve protokole posebno su potrebni mehanizmi koji dozvoljavaju

Furucombo proksiju da upravlja pozicijom koju ne poseduje. Ovo nije problem u slučajevima koji su od koristi poziciji (dodavanje kolaterala, vraćanje duga, …) jer su oni često dozvoljeni svima – na primer, korisnik A može da otplati dug korisnika B. Međutim akcije kao što su kreiranje duga i povlačenje kolaterala ne mogu da budu dozvoljeni bilo kome osim vlasnicima. *Aave* protokol na primer omogućava korisniku da dozvoli nekom drugom korisniku da pravi dug u njegovo ime. U protokolu *Maker* postoji mehanizam koji omogućava dozvoljavanje totalnog upravljanja pozicijom od strane drugog lica. Protokol *Compound* (jako sličan protokolu *Aave*), sa druge strane, nema sličan mehanizam, pa zato *Furucombo* nema opciju pozajmice za pozicije protokola *Compound*. Važno je napomenuti da svi ovi mehanizmi koji daju dozvole zajedničkom proksi ugovoru da upravlja pozicijama mogu predstavljati potencijalan bezbednosni problem. Kada se uzme u obzir da administratori sistema mogu da dodaju u sistem novu akciju kojom bi zlonamerno iskoristili te dozvole, problem postaje još bitniji.

Furucombo implementira grafički interfejs za kreiranje kompleksnih akcija kao glavnu funkcionalnost aplikacije. *DeFi Saver* je primarno softver za upravljanje finansijskim pozicijama, ali takođe daje korisnicima mogućnost da kroz grafički interfejs kreiraju svoje recepte. Instadapp se fokusira na upravljanje pozicijama i pametnim novčanicima. Kôd celog sistema je otvoren, pa je ručno kreiranje kompleksnih transakcija moguće kroz grafičke interfejse trećih lica iako Instadapp ne održava interfejs za tu svrhu u okviru svoje aplikacije.

Kao što je pomenuto, sistem *Instadapp* podržava eksplicitno čuvanje povratne vrednosti akcije i njeno ponovno korišćenje kao parametra. Sistem *DeFi Saver* implicitno čuva povratnu vrednost svake akcije. Najčešće je omogućeno spajanje svih ulaznih parametara sa povratnim vrednostima. Akcija za svaki parametar poziva dekodira niz bajtova u traženi tip a onda proverava da li vrednost parametra treba da se zameni nekom od prethodnih vrednosti. U sistemu *Instadapp* je ponovno korišćenje najčešće moguće samo za jedan ili dva parametra koji označavaju količinu. Identifikator otvorene pozicije čak nije dostupan kao povratna vrednost za ponovno korišćenje, već akcije vezane za protokol *Maker* podržavaju posebnu vrednost – nulu, koja označava poslednje otvorenu poziciju. Fleksibilnost sistema *DeFi Saver* naspram minimalističkog pristupa sistema *Instadapp* podržava donekle veći broj mogućnosti u receptima, ali često nepotrebno korišćenje memorije, kao i dodatno dekodiranje svakog parametra ukazuju na potencijalno veću potrošnju gasa. U kasnijim primerima ćemo videti da se ovo ispostavlja tačnim u praksi.

Navedene sličnosti i razlike su sumirane u tablici 3.1.

| | ***DeFi Saver*** | ***Instadapp*** | ***Furucombo*** |
|---|---|---|---|
| **Pametni novčanik** | *DsProxy* | *DSA* | Ne |
| **Fleksibilnost sistema** | Zatvoren sistem, *multisig* | Zatvoren sistem, *DAO* | Zatvoren sistem, *multisig* |
| **Organizacija akcija** | Ugovor po akciji | Ugovor po protokolu | Ugovor po protokolu |
| **Interfejs** | Menadžment pozicija i ručno kreiranje sl. tr. | Menadžment pozicija | Ručno kreiranje složenih transakcija |
| **Korišćenje povratnih vrednosti** | Da, implicitno čuvanje u memoriju | Da, eksplicitno čuvanje u memoriju | Ne |

*Tablica 3.1: Poređenje sistema*

# 4 Primeri složenih transakcija

U nastavku prikazujemo primere složenih transakcija na kojima ćemo ilustrovati prethodno spomenute sličnosti i razlike sistema. Primeri su odabrani tako da prikažu ključne arhitekturalne razlike i njihove posledice.

## 4.1 Kreiranje kolateralizovane pozajmice

Počećemo sa najjednostavnijim primerom otvaranja proste pozicije na protokolu *Aave* sa kolateralom u jednoj valuti, i dugom u drugoj. *Aave* ne zahteva eksplicitno kreiranje pozicije, i suštinski podržava jednu poziciju za ceo nalog. Korisnik jednostavno deponuje valutu u sistem i omogući je kao kolateral. Sistem mu tada dozvoljava pozajmicu do iznosa koji zavisi od konfiguracije valute kolaterala u sistemu (valute stablinije vrednosti dozvoljavaju veću pozajmicu u odnosu na deponovan iznos – na primer do 80% kolaterala za *Dai*, dok je konfiguracija konzervativnija za valute volatilnije vrednosti ili manje likvidnosti – na primer do 50% za YFI).

Recimo da za konkretan primer želimo da deponujemo 20 Etera i pozajmimo 5000 *Dai*. Standardnim pristupom, ova operacija bi zahtevala 3 odvojene transakcije:

- Deponovanje 20 Etera u *Aave* sistem;
- Pozajmljivanje 5000 *Dai*.

U prva sistemima *DeFi Saver* i *Instadapp*, ovo bi bilo izvršeno kroz jednu transakciju koja sadrži ove dve akcije. Važno je napomenuti da se podrazumeva da korisnik poseduje pametni novčanik. U slučaju da ga ne poseduje, potrebno je prethodno ga kreirati, što bi podrazumevalo dodatnu transakciju. U sistemu *Furucombo*, proces je sličan, uz dve razlike:

- Pre izvršavanja transakcije, korisnik mora da dozvoli proksiju sistema *Furucombo* da napravi pozajmicu u njegovo ime. *Aave* implementira mehanizam za ovu svrhu, koji se sastoji iz davanja dozvole pozivom metode `approveDelegation` na ugovoru zaduženom za pozajmice u konkretnoj valuti. Metoda kao argument prima adresu delegata i maksimalno dozvoljenu pozajmicu. Poziv ove metode mora da dođe sa korisničke adrese, pa i ova akcija zahteva zasebnu transakciju.
- Pošto proksi sistema *Furucombo* i deponuje sredstva (u ime korisnika), u sistemu *Aave* ostaje zabeleženo da je taj proksi deponovao kolateral. Potrebno je ovaj kolateral prebaciti na korisnikovu adresu pre nastavljanja na korak pozajmice.
- *Furucombo* pozajmicu prima na zajedničkom proksiju pa je na kraju transakcije implicitno šalje na korisnikov novčanik. Radi ravnopravnijeg poređenja, dodato je slanje na eksterni novčanik i u druga dva recepta.

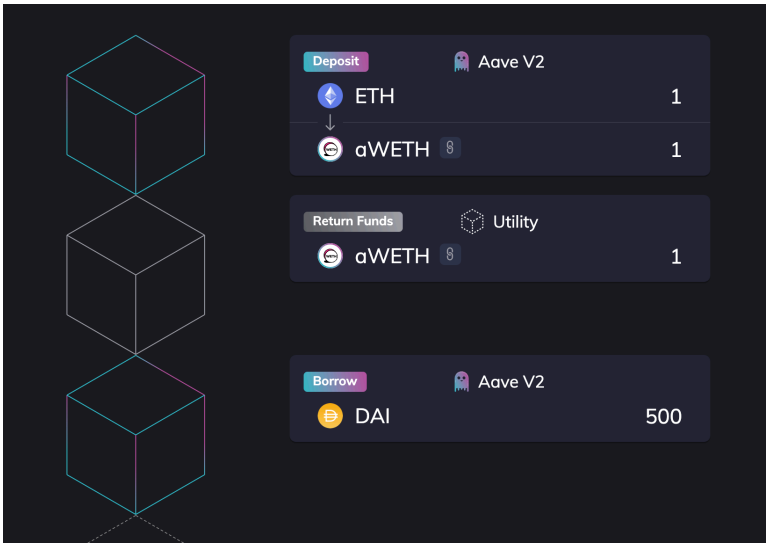
Za kreiranje recepata u sistemima *DeFi Saver* i *Instadapp* korišćene su softverske biblioteke istih sistema (isečci 4.1 i 4.2), dok je za sistem *Furucombo* korišćen grafički interfejs. (slika 4.1).

```
const recipe = new DFS.Recipe('Aave Create', [
  new DFS.actions.basic.WrapEthAction(
    SUPPLY_AMOUNT,
  ),
  new DFS.actions.aave.AaveSupplyAction(
    AAVE_V2_MARKET,
    WETH_ADDR,
    SUPPLY_AMOUNT,
    proxy, // from
    proxy, // onBehalfOf
    true,  // enableAsCollateral
  ),
  new DFS.actions.aave.AaveBorrowAction(
    AAVE_V2_MARKET,
    DAI_ADDR,
    BORROW_AMOUNT,
    2,     // rate mode
    eoa,   // to
  ),
]);
```

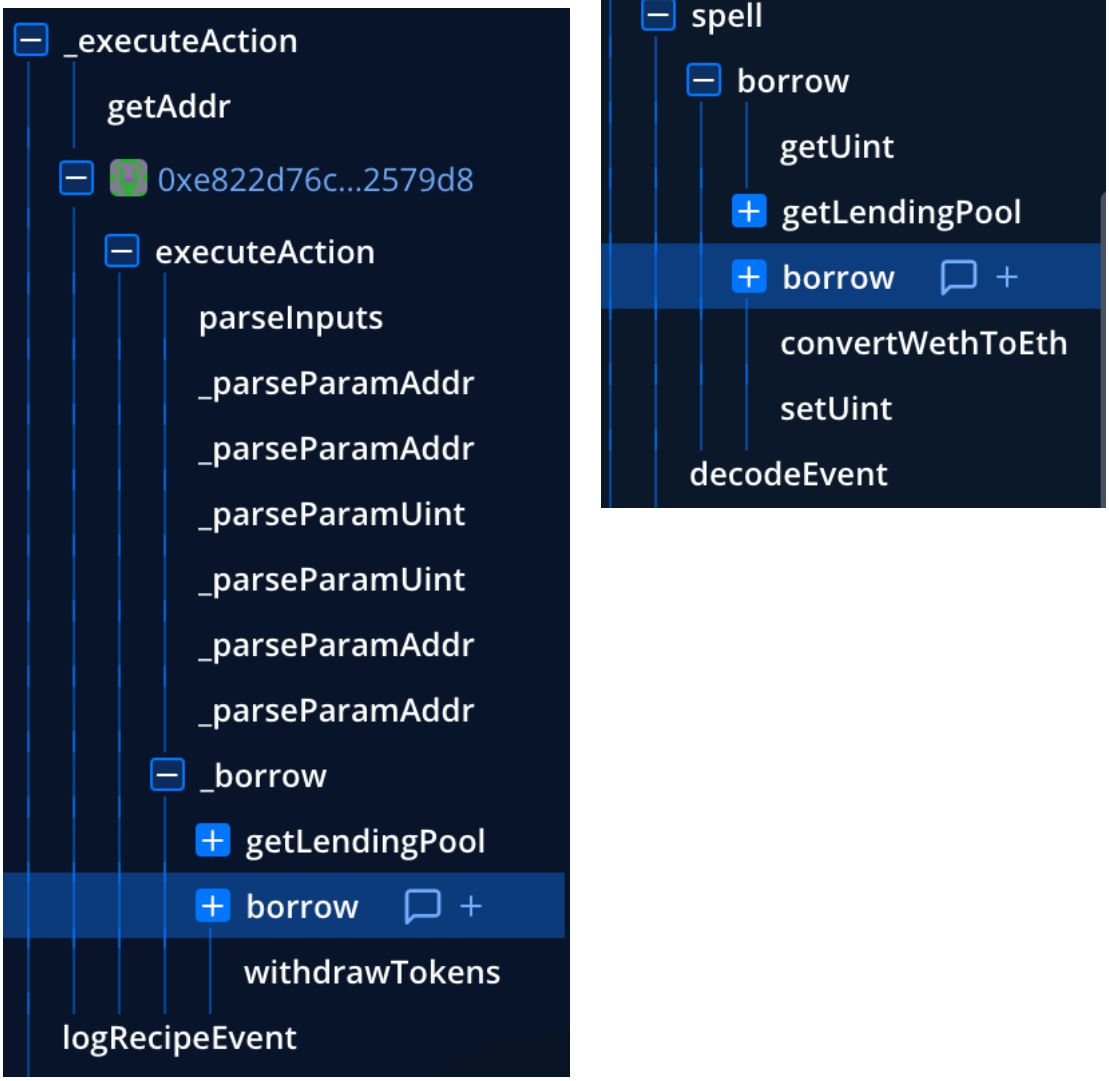
*Isečak koda 4.1: Priprema recepta za izvršavanje – DeFi Saver SDK*

```
const spells = dsa.Spell();
spells.add({
  connector: "BASIC-A",
  method: "deposit",
  args: [ETH_ADDR, SUPPLY_AMOUNT, 0, 0]
});
spells.add({
  connector: "AAVE-V2-A",
  method: "deposit",
  args: [ETH_ADDR, SUPPLY_AMOUNT, 0, 0],
});
spells.add({
  connector: "AAVE-V2-A",
  method: "borrow",
  args: [DAI_ADDR, BORROW_AMOUNT, 2, 0, 0],
});
spells.add({
  connector: "BASIC-A",
  method: "withdraw",
  args: [DAI_ADDR, BORROW_AMOUNT, eoa, 0, 0]
});
```

*Isečak koda 4.2: Priprema recepta za izvršavanje – Instadapp SDK*




*Slika 4.1: Priprema recepta za izvršavanje – Furucombo grafički interfejs*



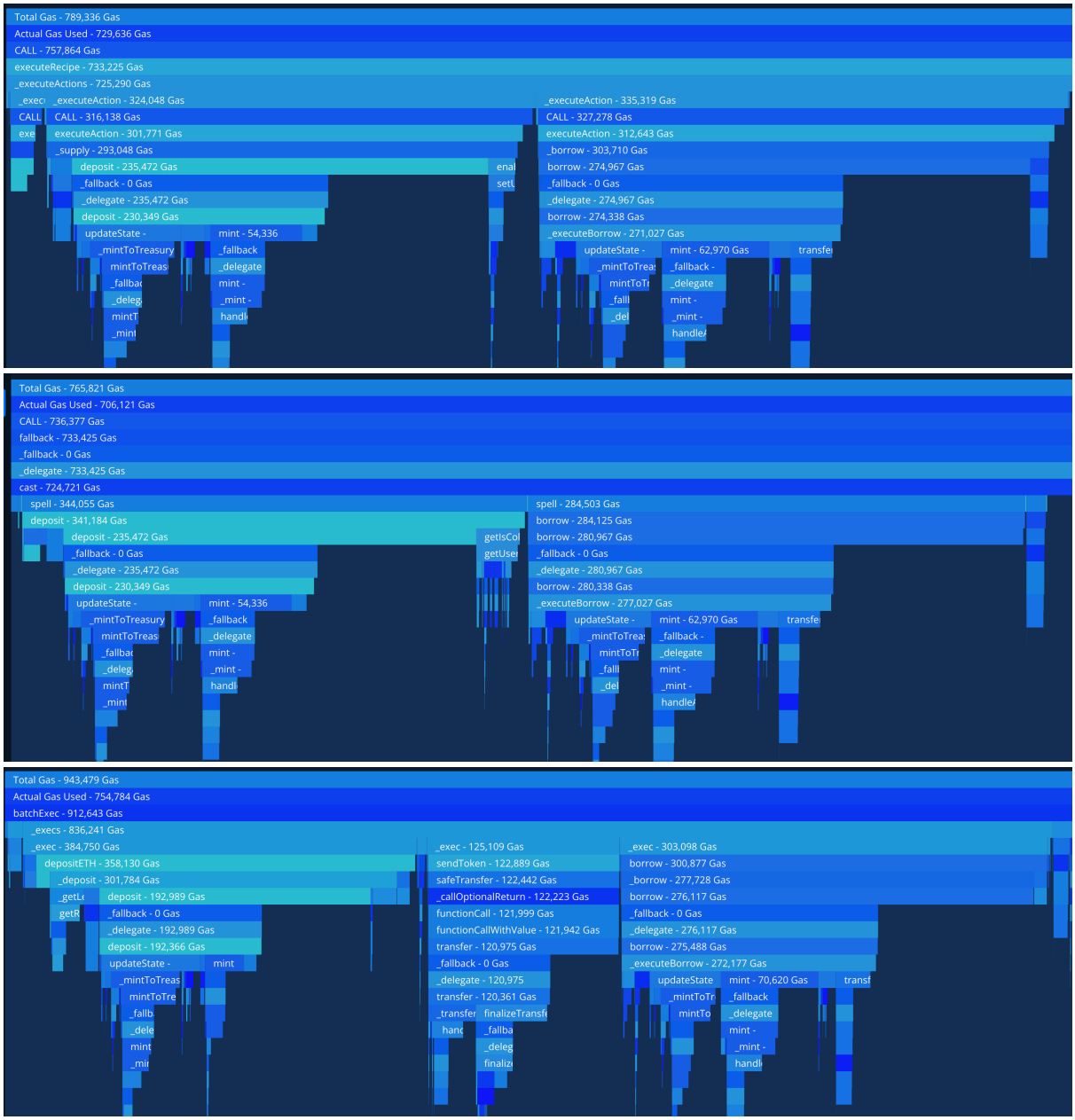

*Slika 4.2: Prikaz steka poziva za akciju pozajmljivanja unutar recepta, od poziva akcije do kontakta sa Aave sistemom (DeFi Saver levo, i Instadapp desno)*

Na priloženoj ilustraciji vidi se velika sličnost između sistema *DeFi Saver* i *Instadapp*. Mogu se primetiti neke sitne razlike. Na primer, sistem *Instadapp* na početku transakcije dohvata adrese svih relevantnih konektora, dok sistem *DeFi Saver* povlači adresu trenutne akcije u svakoj iteraciji petlje. *DeFi Saver* nudi dodatne opcije u okviru akcija za depozit i pozajmljivanje (uključivanje kolaterala, slanje sa, odnosno na drugu adresu), dok su u sistemu *Instadapp* iste stvari omogućene kroz zasebne akcije. Manje razlike na stranu, sistemi suštinski pozivaju istu stvar – `borrow` metodu protokola *Aave* – pa se razlike u potrošnji gasa (ilustrovane slikom 4.3 i tabelom 4.1) mogu pripisati isključivo specifičnostima sistema, a koje su pomenute u nastavku.

Slika 4.2 ilustruje veću kompleksnost koju fleksibilnost sistema *DeFi Saver* donosi u odnosu na *Instadapp*. Pošto se povratne vrednosti mogu proslediti u većinu ulaznih parametara svake akcije, svaka akcija proverava za svaki argument da li treba da ga zameni s nekom od povratnih vrednosti, iako se najčešće zamenjuju jedna ili čak nijedna vrednost. Ovo očekivano rezultuje većom potrošnjom gasa.

Još jedna vidna razlika je što sistem *DeFi Saver* za svaku akciju poziva `execute` nad korisničkim proksijem, što je naizgled nepotrebno jer je izvršavanje već u okviru tog proksija. Ovo takođe rezultira većom potrošnjom gasa zahvaljujući dodatnom pozivu na steku, uz propratnu proveru modifikatora za autorizaciju. Instadapp jednostavno izvršava pomenuti `delegatecall`, što se ispostavlja efikasnijim.

Sistem *Furucombo* na nivou poziva akcija u receptu radi relativno slično kao Instadapp – poziva se metoda `borrow` koristeći `delegatecall`. Jedina razlika je post-validacija za koju sistem odvaja podatke posle izvršavanja akcije. Ta priprema već troši gas a dodatno imamo i samu validaciju nakon svih akcija.




*Slika 4.3: Prikaz potrošnje gasa za naveden primer izvršen u sistemima DeFi Saver, Instadapp i Furucombo*

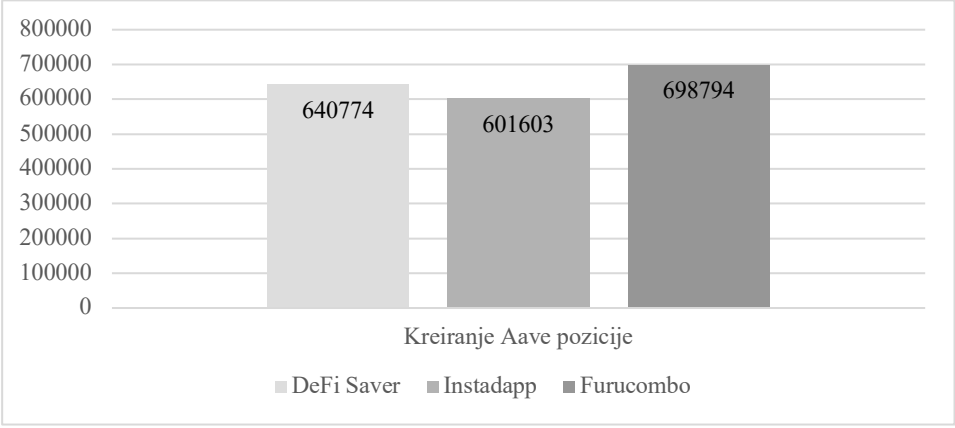
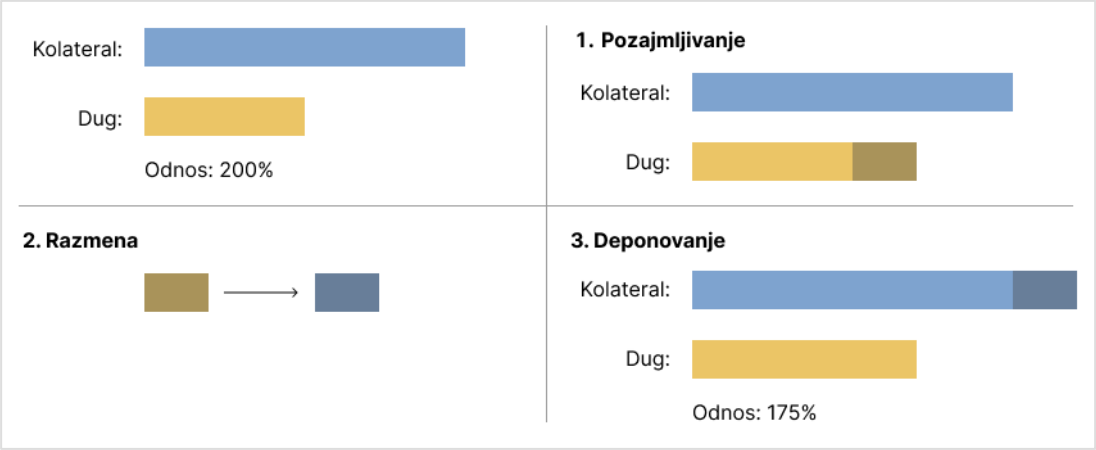



*Tabela 4.1: Uporedni prikaz potrošnje gasa*

## 4.2 Pozajmica, kupovina i deponovanje

Sledeći primer ilustruje akciju koja se često koristi pri investiranju s polugom (eng. *leverage trading*). Cilj akcije je da se korisnikova pozicija poveća, i samim tim poveća stepen izloženosti rasta cene valute kolaterala, tako što se kreira pozajmica, pozajmljen kapital zameni u valutu kolaterala i dobijen kolateral deponuje. Nakon ovakve akcije, korisnik ima veći dug ali i više kolaterala, dok je relativni odnos između kolaterala i duga porastao u korist duga.




*Slika 4.4: Ilustracija recepta*

Sa tehničke strane, primer će koristiti da ilustruje primenu korišćenja povratnih vrednosti. Naime, u drugom koraku ove transakcije dešava se razmena na menjačnici. Kao što smo već pomenuli, s obzirom da se cena algoritamski određuje, ne možemo biti sigurni koja cena će biti aktuelna u trenutku izvršavanja transakcije, a time ni koliko tačno sredstava ćemo dobiti. Taj iznos je bitan jer u trećem koraku želimo da deponujemo celokupan dobijen iznos. Drugim rečima, želimo da koristimo rezultat druge akcije kao ulazni parametar treće. Kao što je prethodno pomenuto, čuvanje izlaznih vrednosti se uvek dešava implicitno u sistemu *DeFi Saver* i vrednost samo treba iskoristiti u predstojećim akcijama. *Instadapp* zahteva da se vrednost eksplicitno sačuva koristeći `setId` i `getId` parametre. *Furucombo* nema implementiran mehanizam za ovu svrhu, već mehanizam postprocesiranja vrati preostala sredstva na korisničku adresu.

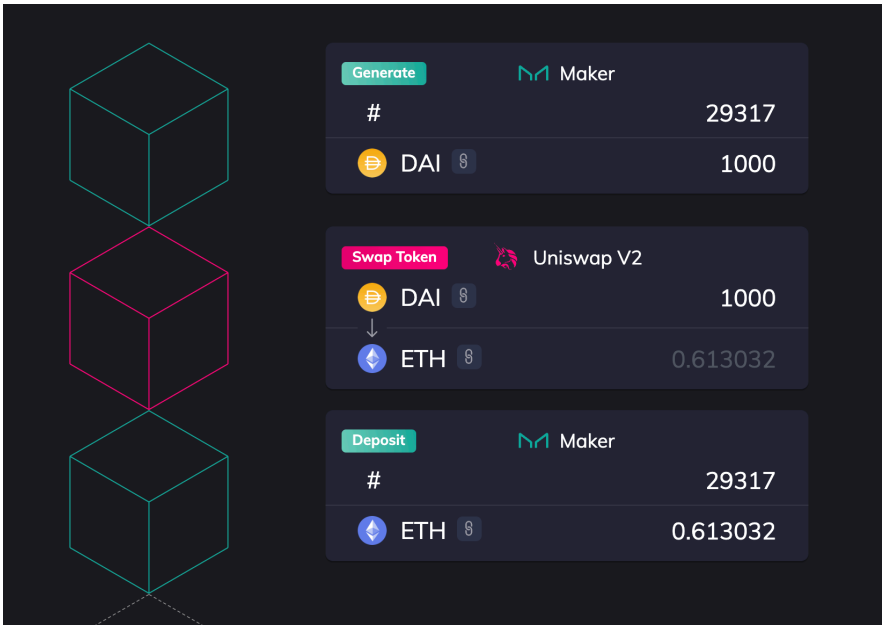
Kao i u prethodnom primeru, za kreiranje recepata u sistemima *DeFi Saver* i *Instadapp* korišćene su softverske biblioteke istih sistema (isečci 4.3 i 4.4), dok je za sistem *Furucombo* korišćen grafički interfejs. (slika 4.5). U poređenje je takođe uključen specijalizovan ugovor sistema *DeFi Saver,* prikazan u isečku 4.5.

```
const recipe = new DFS.Recipe('CDP Boost', [
  new DFS.actions.maker.MakerGenerateAction(
    CDP_ID,
    BORROW_AMOUNT,
    proxy,
  ),
  new DFS.actions.basic.SellAction(
    exchangeOrder,
    proxy,
    proxy,
  ),
  new DFS.actions.maker.MakerSupplyAction(
    CDP_ID,
    '$2',
    ETH_A_JOIN,
    proxy,
  )
]);
```

*Isečak koda 4.3: Priprema recepta za izvršavanje – DeFi Saver SDK*

```
const boostSpell = dsa.Spell();
boostSpell.add({ connector: "MAKERDAO-A", method: "borrow", args: [
    CDP_ID,
    BORROW_AMOUNT,
    0,
    0,
]});
boostSpell.add({ connector: "UNISWAP-V2-A", method: "sell", args: [
    ETH_ADDR,
    DAI_ADDR,
    BORROW_AMOUNT,
    BORROW_AMOUNT.slice(0, -18),
    0,
    123,
]});
boostSpell.add({ connector: "MAKERDAO-A", method: "deposit", args: [
    CDP_ID,
    '0',
    123,
    0,
]});
```

*Isečak koda 4.4: Priprema recepta za izvršavanje – Instadapp SDK*




*Slika 4.5: Priprema recepta za izvršavanje – Furucombo grafički interfejs*

```
function boost(
    ExchangeData memory _exchangeData,
    uint256 _cdpId,
    uint256 _gasCost,
    address _joinAddr,
    ManagerType _managerType
) public payable {
    address managerAddr = getManagerAddr(_managerType);

    address user = getOwner(Manager(managerAddr), _cdpId);
    bytes32 ilk = Manager(managerAddr).ilks(_cdpId);

    uint256 daiDrawn = drawDai(managerAddr, _cdpId, ilk, _exchangeData.srcAmount);

    _exchangeData.user = user;
    _exchangeData.dfsFeeDivider = isAutomation()
        ? AUTOMATIC_SERVICE_FEE
        : MANUAL_SERVICE_FEE;
    _exchangeData.srcAmount = sub(daiDrawn, takeFee(_gasCost, daiDrawn));
    (, uint256 swapedColl) = _sell(_exchangeData);

    addCollateral(managerAddr, _cdpId, _joinAddr, swapedColl);

    // if there is some eth left (0x fee), return it to the caller
    if (address(this).balance > 0) {
        tx.origin.transfer(address(this).balance);
    }

    logger.Log(
        address(this),
        msg.sender,
        "MCDBoost",
        abi.encode(_cdpId, user, _exchangeData.srcAmount, swapedColl)
    );
}
```

*Isečak koda 4.5: Kôd specijalizovanog ugovora ranije korišćenog na platformi DeFi Saver [15]*

Rezultati su slični prethodnom primeru po pitanju potrošnje gasa. Razlike ne prelaze 10% i uglavnom se svode na sitnije implementacione detalje – jedan transfer manje ili više, ili upis na stek poziva. U poređenje je uključen i specijalizovani ugovor koji radi istu stvar. Očekivano, specijalizovani pristup je najefikasniji, ali podržava samo jedan, prost slučaj. Od 3 generalizovana sistema, najefikasnije izvršavanje ima *Instadapp*. Dodatno korišćenje gasa u sistemu *DeFi Saver* vidimo u akciji koja implementira zamenu valuta. S obzirom da je akcija generalizovana da podržava više menjačnica, interno ima transfer na implementaciju razmene na konkretnoj menjačnici. Dodatno, u skladu sa poslovnim modelom platforme, *DeFi Saver* uzima naknadu pri razmeni, pa i taj transfer dodaje na korišćenju gasa. Iako *Instadapp* određene stvari radi efikasnije, odluka programera da u sistemu *DeFi Saver* uvek koriste ERC20 verziju *Eterijuma*, odnosno WETH, se bar u ovom slučaju ispostavlja praktičnom pošto protokoli većinski interno koriste WETH. Naime *Instadapp* ugovori nakon razmene *Dai* za WETH na menjačnici *Uniswap* ta sredstva konvertuje u ETH, da bi ih u trećem koraku ponovo konvertovao nazad u WETH. Slično se dešava i u sistemu *Furucombo*, s razlikom da interfejs ne dozvoljava razmenu u WETH, već direktno u ETH, koji se ipak konvertuje pri deponovanju.

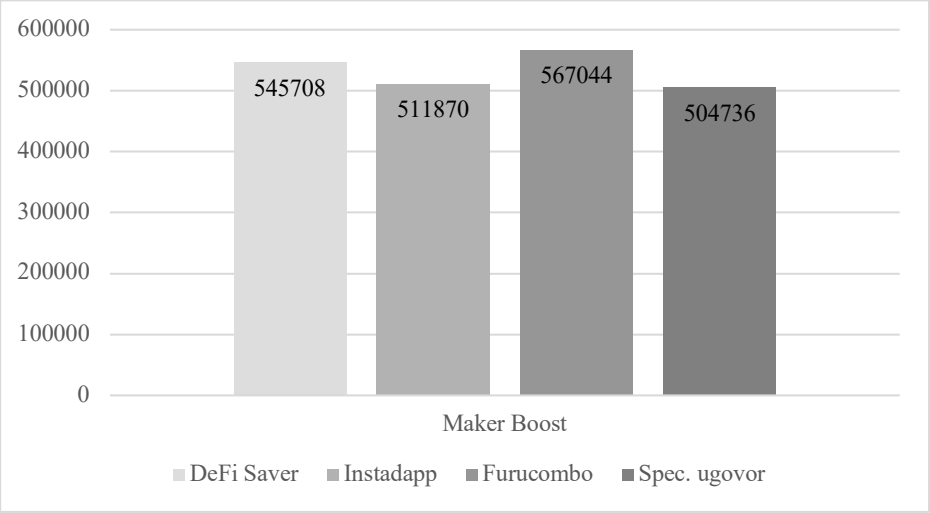
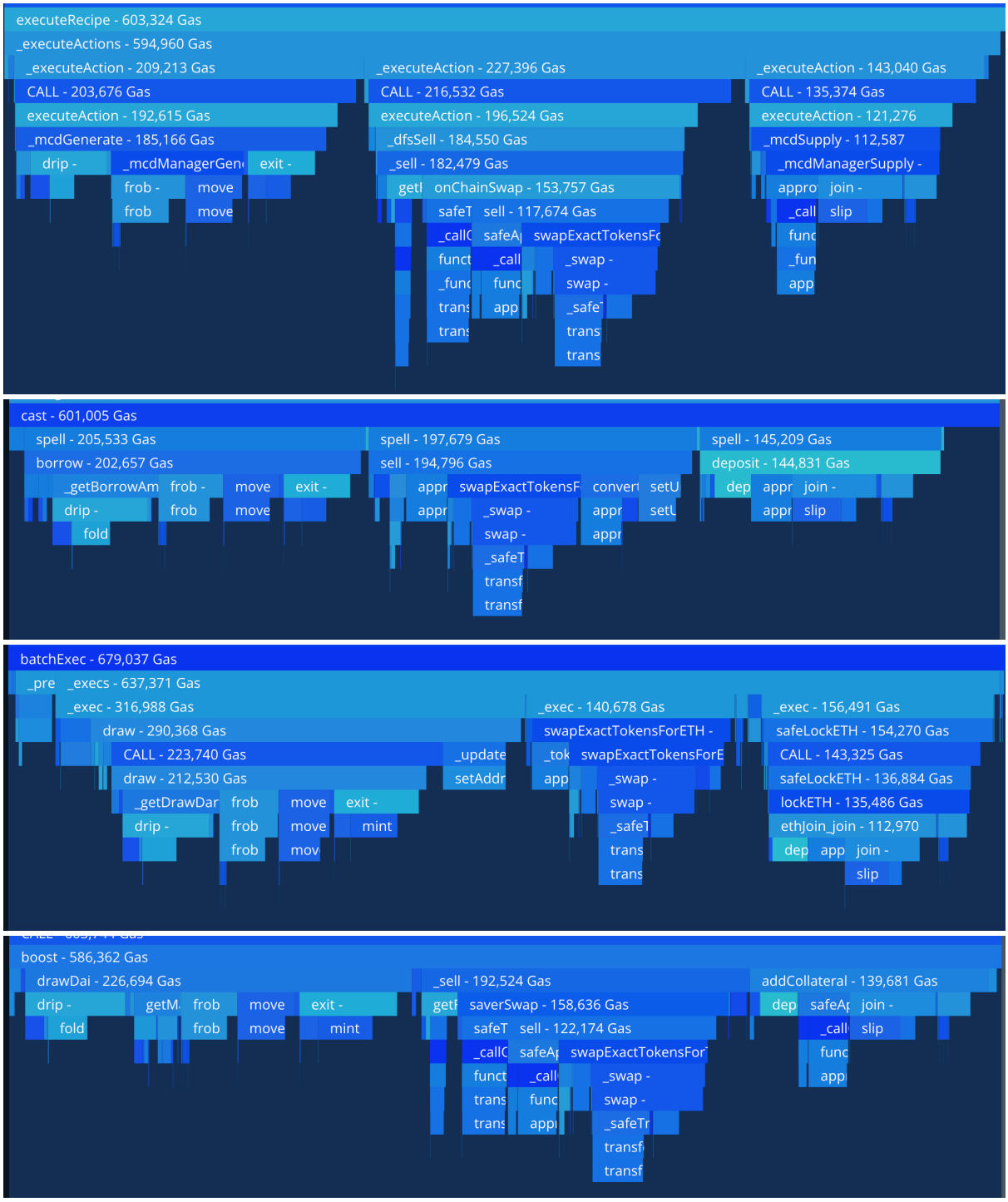



*Tabela 4.2: Uporedni prikaz potrošnje gasa*




*Slika 4.6: Prikaz potrošnje gasa za naveden primer izvršen u sistemima DeFi Saver, Instadapp i Furucombo*

## 4.3 Metodologija testiranja sistema

Sve prikazane transakcije su izvršene u simulaciji. Korišćen je softver lokalnog tima „Tenderly“ koji omogućava kreiranje simuliranog okruženja koje čuva postojeću istoriju blokčejna do tog trenutka (kreira se grananje od određenog bloka – eng. *fork*). Na ovaj način nismo ograničeni raspoloživim finansijskim sredstvima i ne postoji rizik gubitka istih sredstava. Pored toga, softver omogućava detaljnu analizu izvršenih transakcija. Za potrebe rada posebno

su bili korisni alati za praćenje steka kroz izvršavanje (eng. *stack trace* – prikazan na slici 4.2) i analizu potrošnje gasa (prikazan na slikama 4.3 i 4.6).

Decentralizovani sistemi su u pojedinim slučajevima projektovani tako da se određeno održavanje sistema dešava „automatski“ kada se koriste. Na primer, interni množilac koji određuje sakupljenu kamatu za sve u sistemu se periodično uvećava automatski kada bilo ko interaguje sa sistemom. Iz tog razloga se može desiti da ista akcija za dva korisnika ne potroši jednaku količinu gasa, jer prva transakcija u stvari izvrši održavanje sistema a jedan od korisnika plati za to. Bilo je bitno uzeti ovakve slučajeve u obzir, odnosno ne dozvoliti da nam narušavaju tačnost pri upoređivanju sistema. Iz tog razloga, za svaku od 3 prikazane transakcije kreirane su po 3 simulacije koje se granaju iz istog bloka, po jedna za svaki od 3 sistema. Novčanici koji su korišćeni nisu imali prethodne transakcije. Na taj način je osigurano da je u svakom testu prethodno stanje identično, i time su minimizovana odstupanja.

Pored okolnosti izvršavanja, izbegavana su odstupanja i unutar samih transakcija. Pisanje recepata vođeno je najvećim zajedničkim sadržaocem u smislu mogućnosti. Odnosno, ako jedan sistem ne podržava neku funkcionalnost, isti recept na drugim sistemima je bio kreiran bez te funkcionalnosti. Na primer, pošto recept zasnovan na sistemu *Furucombo* u prvom primeru mora na kraju da ima prenos sredstava sa proksija na eksterni novčanik, sličan transfer je dodat i na druga dva recepta, a to ograničenje sistema *Furucombo* je naznačeno u propratnom tekstu.

## 4.4 Poređenje pristupa

Prikazani generalizovani pristupi, kao što je spomenuto, optimizuju različite stvari. Ipak, postoje velika količina konceptualnih i implementacionih preklapanja između ovih sistema. U samim implementacijama postoje značajne optimizacije određenih elemenata koje bi sistemi mogli da uvide jedni od drugih. Neki od takvih primera viđenih prilikom analize sistema uključuju:

- Pri izvršavanju u sistemu *DeFi Saver*, za svaku akciju se unutar metode `_executeAction` poziva `DsProxy.execute` iako je kontekst izvršavanja već sam `DsProxy`. Ova situacija se može optimizovati direktno pozivanjem funkcije `delegatecall` sa istim parametrima. Na taj način ne postoji novi unos u steku poziva, a štedi se i gas potrošen pri proveri `modifikatora` na `execute` metodi.
- U mehanizmu za dozvolu upotrebe sredstava drugim ugovorima, akcije u sistemu *DeFi Saver* koriste metodu koja prvo podesi dozvoljen limit na nulu a zatim na zadat iznos. Ovaj mehanizam je potreban za određene valute koje implementiraju dodatni uslov da se limit ne može podesiti na određen iznos osim ako nije na nuli. Kako je ovo relativno redak slučaj, gas se u većini slučajeva troši bespotrebno. Sličan mehanizam je postignut u sistemu *Instadapp* na elegantniji način. Dozvoljen limit se podešava na zadat iznos unutar bloka `try`, a u slučaju retkih neuspeha se u bloku `catch` izvršava proces podrazumevan u sistemu *DeFi Saver*. Na ovaj način,

optimizovana je potrošnja gasa za većinu slučajeva iako je povećana u retkim slučajevima gde je mehanizam duplog upisivanja potreban.

Postoje i konceptualne razlike koje se pokazuju optimalnijim u određenim slučajevima. Na primer, pristup apsolutne fleksibilnosti i generalizacije u sistemu DeFi Saver deluje nepotrebno u određenim slučajevima. Povratne vrednosti nisu uvek potrebne, a prosleđivanje vrednosti nije relevantno za sve parametre, pa se dosta implicitnog čuvanja i obrade izvršava bespotrebno. Još jedan primer predstavlja korišćenje valute WETH – omotača oko valute Eter koji se uklapa u standard ERC20 koji koriste ostale valute. Iako korisnicima nije prirodno da koriste ovu apstrahovanu verziju Etera, protokoli većinski koriste tu verziju u pozadini. Ispostavlja se da u sistemima Instadapp i Furucombo, koji podrazumevaju da korisnik rukuje sa čistim Eterom, postoji nepotrebna konverzija iz valute ETH u valutu WETH na početku akcije, i obratno na kraju. Ova konverzija se ponavlja u velikom broju akcija pa neoptimalnost raste uz broj akcija u transakciji. Ovakve slučajeve je lakše izbeći u specijalizovanim ugovorima, ali su oni ipak prirodan rezultat raščlanjivanja kompleksnosti i apstrakcije.

# 5 Zaključak

U prethodnim poglavljima su prikazani različiti pristupi kreiranju složenih transakcija. Iako su pristupi značajno različiti, postoje slučajevi u kojima svaki od njih ima smisla. Specijalizovani pametni ugovori su najjednostavniji pristup koji zadovoljava većinu potreba dokle god su one jasno definisane i nepromenljive. Iako ovaj pristup daje najveću fleksibilnost i brzinu pri inicijalnom razvoju, ne pokazuje se optimalnim za skaliranje i održavanje. Uz rast kompleksnosti interakcije eksponencijalno raste i kompleksnost razvoja i testiranja, kao i zahtevnost bezbednosne analize ugovora. Uz to, svaka promena ili dodatna varijacija u interakciji podrazumeva ponovno postavljanje ugovora na blokčejn, što sa sobom nosi određenu cenu. Sa druge strane, u sistemu sačinjenom od atomičnih interakcija na nivou pametnih ugovora značajno je lakše implementirati i testirati nove akcije. Razvoj složenih interakcija je dosta ekonomičniji fleksibilniji jer se sama interakcija ne zasniva na nepromenljivom pametnom ugovoru već nizu koraka koji se može dinamično menjati uz nove potrebe korisnika. Kompromis ovog pristupa je naravno u vidu resursa inicijalno potrebnih za razvoj ovakve arhitekture. Prikazani sistemi su zreli i zasnovani su na godinama prethodnog iskustva. Iako ima mesta za poboljšanja, čitaocima koji razmatraju razvoj sličnih sistema od početka se preporučuje detaljna analiza postojećih u kontekstu njihovih specifičnih potreba.

# Reference


[1] C. Cachin i M. Vukolić, „Blockchain Consensus Protocols in the Wild,“ 2017.

[2] A. S. H. M. S. Abdelatif Hafid, Scaling Blockchains: A Comprehensive Survey, 2020.

[3] S. Nakamoto, „Bitcoin: A Peer-to-Peer Electronic Cash System,“ 2008. [Na mreži]. URI: `https://bitcoin.org/bitcoin.pdf`

[4] A. T. Sherman, F. Javani, H. Zhang i E. Golaszewski, „On the Origins and Variations of Blockchain Technologies,“ 2018.

[5] W. Dai, „b-money,“ 1998. [Na mreži]. URI: `http://www.weidai.com/bmoney.txt`

[6] V. Buterin, „Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform,“ 2014. [Na mreži]. URI: `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_White_Paper_-_Buterin_2014.pdf`

[7] N. Szabo, „Formalizing and Securing Relationships on Public Networks,“ *First Monday,* 1997.

[8] DeFi Saver, „Kod ugovora RecipeExecutor,“ [Na mreži]. URI: `https://github.com/defisaver/defisaver-v3-contracts/blob/569027ffce04db5e42b5fbb84fba827c26322f10/contracts/core/RecipeExecutor.sol`

[9] DeFi Saver, „Kod ugovora McdSupply,“ [Na mreži]. URI: `https://github.com/defisaver/defisaver-v3-contracts/blob/a6f2cb6d28e0857e348791c11d9958d4ce0c7d8d/contracts/actions/mcd/McdSupply.sol`

[10] Instadapp, „Kod ugovora InstaImplementationM1,“ [Na mreži]. URI: `https://github.com/Instadapp/dsa-contracts/blob/c98af0463545d08496fc8f69b93a2cb365832bd8/contracts/v2/accounts/module1/Implementation_m1.sol`

[11] Instadapp, „Kod ugovora MakerResolver,“ [Na mreži]. URI: `https://github.com/Instadapp/dsa-`

connectors/blob/8950a8a53ffc472991725a1802a85fe868ce0573/contracts/mainnet/connectors/makerdao/main.sol

[12] Furucombo, „Kod ugovora Proxy,“ [Na mreži]. URI: https://github.com/dinngodev/furucombo-contract/blob/78aa7db7d4d72306bba915f9b03a626717f239f8/contracts/Proxy.sol

[13] Furucombo, „Kod ugovora HUniswapV2,“ [Na mreži]. URI: https://github.com/dinngodev/furucombo-contract/blob/40a9b1c7651cab0539d8804bc342aeef0faac256/contracts/handlers/uniswapv2/HUniswapV2.sol

[14] Furucombo, „Kod ugovora HMaker,“ [Na mreži]. URI: https://github.com/dinngodev/furucombo-contract/blob/40a9b1c7651cab0539d8804bc342aeef0faac256/contracts/handlers/maker/HMaker.sol

[15] DeFi Saver, „Kod ugovora MCDSaverTaker postavljenog na blokčejn,“ [Na mreži]. URI: https://etherscan.io/address/0x99dde0b130833ce0a6d9efcfe330327acf38006e