An Agent-Based Modeling Framework and Application for the Generic Nuclear Fuel Cycle

by

Matthew J. Gidden

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Nuclear Engineering & Engineering Physics)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 02/15/2015

Ceci n’est pas un dedication.

ACKNOWLEDGMENTS

Ceci n’est pas un Acknowledgement.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Ceci n'est pas un Abstract. [17]

1.1 The Nuclear Fuel Cycle

1.1.1 Taxonomy

1.1.2 Technologies

1.2 Nuclear Fuel Cycle Simulation

The majority of tools developed to date use a system dynamics approach to modeling the fuel cycle.

System dynamics is good for applications that naturally fit the 'stocks and flows' approach.

1.2.1 Overview

1.2.2 Quasi-Static Models

1.2.3 Dynamic Models

1.3 Tools Used

1.3.1 Modeling & Simulation

Simulation and simulation analysis has a rich history and large body of work that can be leveraged to tackle this problem.

Simulation methodologies can largely be broken down into two groups: discrete time and discrete event simulation.

Cyclus has taken a refined discrete time approach.

Cyclus defines an arbitrary time step, requiring consistency among prototype configurations. Historically, this time step has been one month. Importantly, it is integral, and unchanging within a single simulation.

A time step is comprised of two types of phases: those in which agents observe the simulation environment and update their state, and those in which the kernel queries agents and executes kernel functionality.

The order of phases is:

- building phase (kernel)
- tick phase (agent)
- resource exchange (kernel)
- tock phase (agent)
- decommissioning phase (kernel)

1.3.2 Supply Chain Modeling

1.3.3 Mathematical Programming

These simplifications are important to the computation time required to solve the resulting problem instance. The general solution technique for LPs is the Simplex Method, as previously described. Klee and Minty show that in the worst case, the Simplex Method will execute in exponential time [23], but in practice it is generally considered very computationally efficient. If the problem can be simplified to a TP, then the Transportation Simplex Method can be used [4].

Discussion of P, NP, NP-Hard, NP-Complete, etc.

1.3.4 Multicommodity Transportation Problem

1.4 Motivation

1.4.1 Cyclus Foundations

Deciding how a simulation is structured from an interactions standpoint is a delicate balance of known necessity and perceived future needs. There are basic decisions to make, such as modeling material transfer as discrete or continuous. Discrete transfers more closely match reality and may provide insights in that regard, however they require more of their modeling apparatus due to messaging needs and other structures. More complex decisions include how one wants to determine connections between facilities, and whether such connections are assigned statically and incorporated into the simulation architecture or determined dynamically. Guerin's comment in §**??** stems from this “freedom”. These simulation-engine decisions comprise the art-related portion of fuel cycle simulation, but developers have a goal of making these decisions in as informed a way as possible using domain-level knowledge with respect to our known and perceived requirements. In general, this work tries to minimize the sheer number of choices we make in this regard, instead relying on well known and well documented practices of computer scientists and systems engineers.

The Cyclus team is also interested in fostering a user and developer ecosystem. Accordingly, such concerns also drive the simulation architecture design. The agent-based nature of Cyclus provides an opportunity to reduce barriers to entry into the ecosystem. Given a few basic tenets of agent interaction, other developers should be able to create a new agent to "plug in" to the simulation. Intuitively, a minimal set of behaviors must be defined to sufficiently inform the simulation infrastructure to run the simulation. This freedom allows the simulation program introduce agents at run time, effectively separating the simulation engine's functionality from the agents in the simulation.

Such a framework provides many benefits. First, there is a clear separation of concerns. The Cyclus core is concerned with modeling system dynamics whereas individual agents are concerned with domain-specific issues. Accordingly, developers can focus their attention appropriately, focusing either on the core code base or on agent development. Separating agent development from core development also allows Cyclus to remain a viable open-source candidate to model nuclear fuel cycle dynamics. Because domain-level information is incorporated into agent libraries which are dynamically loaded at runtime, a closed-source developer can focus their efforts entirely on developing agent libraries. Furthermore, developers could participate both privately and publicly, e.g., adding general capability to the Cyclus core that is needed for some functionality without specifying the internals. Such a community paradigm is shown in Figure 1.1.

In order to develop and maintain the core code separate from the agent modules, well-defined interactions must be provided between agents and the Cyclus core. The remaining part of the section provides a proposal for such interactions that allow for a variety facility deployment and supply-demand matching algorithms to be employed. A description of the market resolution interface is provided, and basic agent simulation interaction, such as entering and leaving the simulation is also described.

In the absence of supply constraints, aggregated individual facility behavior and fleet-based models are equivalent. However, any system in which recycling exists will, by definition, have some supply constraints.

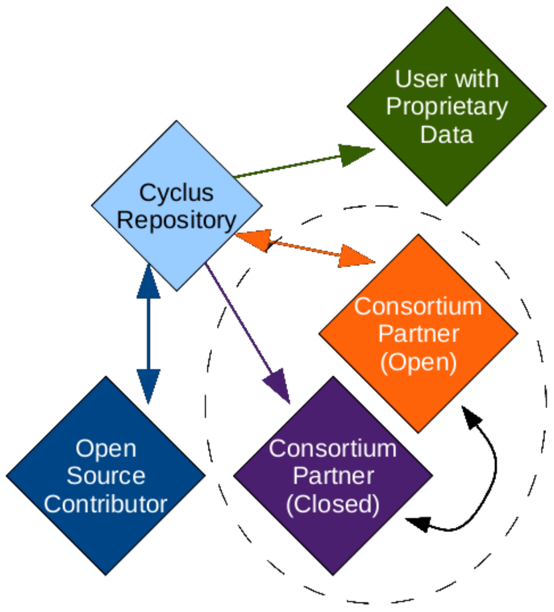
User with Proprietary Data
Cyclus Repository
Consortium Partner (Open)
Open Source Contributor
Consortium Partner (Closed)

Figure 1.1: The Cyclus Participation Paradigm

1.4.2 Statement of Work

2 SIMULATION AND AGENT BASED MODELING IN CYCLUS

Developing a simulator of any complex system is an involved process requiring a solid methodlogical base. A reasonable approach is to define precisely the simulation framework, including a definition of how time moves forward and what events can occur. This chapter lays the foundation for the simulation of nuclear fuel cycles in Cyclus. §2.1 begins the discussion by broadly describing methodological principles on which Cyclus has been developed. The use of agent-based modeling techniques is presented in §2.2, and an agent deployment methodology with a proof-of-principle benchmark is presented. Finally, §2.3 treats the most complex simulation interaction in Cyclus, Dynamic Resource Exchange (DRE), describing its methodology, discussing its implementation, and presenting a set proof-of-principle results.

2.1 Simulation Principles

Cyclus is designed to dynamically model the flow of resources and deployment of facilities in the Nuclear Fuel Cycle (NFC). As such, Cyclus is a *simulator* which models the NFC as a *system*. System simulation is a rich field of study, spanning a variety of disciplines, as described in §1.3.1.

By Law's definition [25], Cyclus is a dynamic, discrete-event simulation that uses a fixed-increment time advance mechanism. In general, fixed-increment time advance simulations assume a time step (Δt). Further they assume that all events that would happen during a time occur simultaneously at the end of the time step. This situation can be thought of as an event-based time advance mechanism, i.e., one that steps from event to event, that executes all events simultaneously that were supposed to have occurred in the time step.

A Cyclus simulation models a collection of *entities* which either trade resources, manage other entities, or perform both actions. The most basic *entity* in a Cyclus simulation is a Facility. Facilities can be used to model processes with arbitrary levels of physical fidelity, and can interact with the simulator and other entities with arbitrary levels of behavioral fidelity. As such, Cyclus can also be described as an *agent-based model* (ABM). Accordingly, the *entities* in a given simulation can be interchangeably referred to as *agents*. Cyclus has an additional notion of an *archetype*. An archetype is the implementation of an entity, whereas an agent is the *in situ* instantiation of a entity. The remainder of this document will use the term archetype when referring to the implementation of an entity and will use the term agent when referring to an entity acting in a simulation.

2.1.1 Events

Two key types of events occur in every Cyclus simulation:

- agent entry into and exit from the simulation
- the exchange of resources between agents

Agent entry and exit events are scheduled by another managing agent, or are scheduled as an initial condition to the simulation. The managing agent and managed agent form a parent-child relationship. Upon entering the simulation, the child entity is constructed and notified of its entry; the parent is then notified. Upon exiting the simulation, the parent is notified; the child entity is then notified and deconstructed. Unlike many of the simulators described in §1.2, the Cyclus simulation kernel naturally treats each agent individually, rather than grouping agents by an attribute and treating like-facilities in an aggregate manner.

While the determination of supply and demand is complex and described further in §2.3, the execution of resource exchange is rather straightforward and a primary event in a Cyclus simulation. When an agent's demand for a resource is matched with another agent's supply of a resource by the Cyclus kernel, a transfer is initiated. Each transfer is treated as discrete, individual trade between two agents.

2.1.2 Timesteps

Simulation entities can have arbitrarily complex state which is dependent on the results of resource exchange and the present discernible status of other agents in the simulation at a given time step. Furthermore, resource exchange necessarily must involve all existing agents in the simulation. Therefore, a well-defined timestep, incorporating agent entry, exit, resource exchange, and agent response to system state must be defined. Cyclus implements a timestep mechanism that deviates slightly from Law's description of fixed-increment time advance by preserving a specific ordering of *event triggers*. Importantly, the following invariant is preserved: *any agent that exists in a given time step should experience the entire time step execution stack.*

This leads to the following *phases* of time step execution:

- agents enter simulation (Building Phase)

- agents respond to current simulation state (Tick Phase)
- resource exchange execution (Exchange Phase)
- agents respond to current simulation state (Tock Phase)
- agents leave simulation (Decommissioning Phase)

The Building, Exchange, and Decommissioning phases each include critical, core-based events, and are called *Kernel* phases. The Tick and Tock phases do not include core-based events, and instead let agents react to previous core-based events and inspect core simulation state. Furthermore, they are periods in which agents can update their own state and are accordingly considered *Agent* phases.

Technically, whether agent entry occurs simultaneously with agent exit or not does not matter from a simulation-mechanics point of view, because the two phases have a direct ordering. It will, however, from the point of view of module development. It is simpler to think of an agent entering the simulation and acting in that time step, rather than entering a simulation at a given time and taking its first action in the subsequent time step.

In the spirit of Law's definition of a fixed-increment time advance mechanism, there is an additional important invariant: *there is no guaranteed agent ordering of within-phase execution*. This invariant allows for:

- a more cognitively simple process
- paralellized implementation

2.2 Agents and Agent Based Modeling in Cyclus

As described in §1.4.1, Cyclus has worked to formally move from a modeling paradigm that does not differentiate between individual facilities, as has been the case historically in FCS, to one that does. Modeling individual facilities in the NFC requires a nuanced approach to determine facility behavior, because such behavior can depend on intricate physical parameters of resources in the simulation as well as complex social-behavioral models of facility interaction.

Agent-based models are defined primarily by two concepts: agents and the simulation environment. Agents in Cyclus are designed to be able to incorporate arbitrary complexity in both physical process models as well as behavioral models. A three-tiered taxonomy has been developed to achieve this aim, specializing agents as either Facilities, Institutions, or Regions. §2.2.1 fleshes out a discussion of this design.

The simulation environment in Cyclus is defined by supply and demand. There is a notion of supply and demand for facility capacity. For example, there can be a demand for power production which drives the deployment of power producing facilities. There is also a notion of supply and demand for resources.

Sufficiently treating resource supply and demand is the primary argument for implementing Cyclus as an ABM simulator. In the NFC, resource supply and demand is a function of both resource quantity and quality, that is, the isotopic composition of material resources. In the extreme in which a high level of detail is required in the notion of resource quality, e.g. tracking an arbitrary number of isotopes, adopting techniques that allow decision-making based on that level of detail is desirable. Modeling the nuclear fuel cycle represents such a level of detail. For example, even in the case of a once-through fuel cycle, many reactors of the same type (e.g., PWRs), may require different resource qualities (i.e., Uranium enrichment). Because resource supply and demand depends on more than the quantity of a resource, moving global management logic becomes difficult. As the complexity of a quality metric increases, an aggregate approach becomes less desirable as it loses such detail through aggregation.

In summary, the arbitrary levels of complexity that can be required for a flexible NFC simulator suggests that ABM is a reasonable tool to use. The remainder of this section describes how agents are provided agency in Cyclus and specifically how agents interact with respect to supply and demand of facility capacity. A proof-of-principle benchmark comparison to a systems dynamics simulator is shown in §2.2.3. Agent interaction with respect to supply and demand of resources is more complicated and therefore treated separately in §2.3.

2.2.1 Agent Taxonomy

The Cyclus kernel implements a basic `Agent` class that provides the minimal interface for agents to be instantiated within a simulation. A `Trader` interface provides a communication layer required for agents to be included in the exchange of resources. Three useful derived classes are provided to be used as basic abstractions of entities in the NFC. `Facility` agents in Cyclus implement both interfaces, while `Institution` and `Region` agents implement only the `Agent` interface. A summary of the conceptual placing of each archetype in a Cyclus simulation is provided below.

2.2.1.1 Facilities

Facilities in Cyclus are either consumers or suppliers of commodities, and some may be both. Supplier agents are provided agency by being able to communicate to the market-resolution mechanism a variety of production capacity constraints in second phase of the information gathering methodology. Consumer agents are provided agency by being able to assign preferences among possible suppliers based on the supplier's quality of product. Because this agency is encapsulated for each agent, it is possible to define strategies that can be attached or detached to the agents at run-time. Such strategies are an example of the Strategy design pattern [34].

2.2.1.2 Institutions

Institutions in Cyclus manage a set of facilities. Facility management is nominally split into two main categories: the commissioning and decommissioning of facilities and supply-demand association. The goal of including a notion of institutions is to allow an increased level of detail when investigating regional-specific scenarios. For example, a consumer facility may prefer to be supplied by a supplier facility in its institution rather than one associated with a different institution. Furthermore, there are international governmental organizations, such as the IAEA, that have proposed managing large fuel cycle facilities that service many countries in a given global region. A fuel bank is an example of such a facility. Accordingly, institutions in Cyclus are able to augment the preferences of supplier-consumer pairs that have been established in order to simulate a mutual preference to trade material within an institution. Of course, situations arise in real life where an institution has the capability to service its own facilities, but choose to use an outside provider because of either cost or time constraints. Such a situation is allowed in this framework as well. It is not clear how such a relationship should be instantiated and to what degree institutions should be allowed to affect their managed facilities' preferences. This issue lies squarely in the realm of simulation design decisions, part of the *art* of simulation. Accordingly, through the course of research, the possible design space will be analyzed in order to determine best practices for this type of design.

2.2.1.3 Regions

Regions in Cyclus provide the forcing function for simulations by requiring that certain parameters be met, e.g., power capacity, fuel cycle service capacity, etc. For example, in the case of nuclear power capacity, a region knows that it needs additional reactors to be built, but leaves the building of those reactors to the institutions that operate in the region. It is important to note here that this abstraction allows for different deployment algorithms to be tested and exchanged in the Cyclus framework without necessitating changes to the simulation engine, as is the case with other simulators described in §1.2.

Regions, like Institutions, are able to affect preferences between supplier-consumer facility pairs in the market information gathering process. The ability to perturb arc preferences between a given supplier and a given consumer allows fuel cycle simulation developers to model relatively complex interactions at a regional level such as tariffs and sanctions.

2.2.2 Methods of Agency

Agency is provided in two primary modes: determining facility deployment and informing resource exchange mechanisms.

Facility deployment involves some combination of an `Institution` agent, a `Facility` agent, and a `Region` agent. `Institution` agents represent a simulation entity abstraction that can deploy `Facility` agents. `Region` agents represent a simulation entity abstraction that have a demand for certain commodities that `Facility` agents provide, for example, reactor-like `Facility` agents provide electrical power.

`Facility` agents are further provided agency by informing market mechanisms of the supply and demand of resource quantity and quality. Cyclus initially used a simple interface and algorithm for determining resource transactions. Individual markets were defined as agents themselves much like `Facility`, `Institution`, and `Region` agents. Many limitations were identified at the time, however, and the market-as-agent approach was eventually abandoned. An enumeration of the observed limitations is described further in §2.2.4.

Dynamic Resource Exchange (DRE), described in detail in §2.3, is the mechanism eventually developed to drive supply-demand transactions. The primary source of agency is provided to `Facility` agents in order to negotiate the quantity and quality of potential resource transactions. `Region`, `Institution`, and

`Facility` agents are then provided agency in the negotiation of preferences of potential transactions, where preference is a proxy for price.

2.2.3 Proof of Principle

Agents were developed to show an initial proof of principle that fuel cycle simulation can be implemented using an agent-based modeling methodology. By definition, dynamic simulators model the deployment of facilities and measure the flow of resources between facilities in the system over time. In the extreme case of unconstrained supply and no competition for resources, resource exchange decisions can be made arbitrarily. In such cases, therefore, only facility deployment agency is required. An initial benchmark case was performed to confirm expected deployment behavior and basic resource routing.

2.2.3.1 Benchmark Cases

The INPRO Business As Usual (BAU) benchmark [3] for the once-through fuel cycle was chosen for three reasons. First, it was the simplest benchmark that demonstrated deployment behavior. Second, no supply or demand constraints were present, so a basic supply-demand framework would suffice. Finally, results from another fuel cycle simulation code, specifically VISION [21], was available for comparison. The INPRO BAU benchmark identified two cases, high electricity demand and moderate electricity demand, as shown in Fig. 2.1. Both cases require that demand met by a composition of 94% Light Water Reactors (LWRs) and 6% Heavy Water Reactors (HWRs). LWRs are fueled with 4% by weight UO_2 while HWRs use natural Uranium fuel.

The goal of this proof-of-principle study was to showcase the capability for a developer to generate the required Facility, Institution, and Region archetypes, and that such archetypes could be deployed in the Cyclus simulation framework and generate satisfactory results. Comparison metrics are based off of similar metrics used in the origin INPRO benchmarking exercise, including deployment patterns, natural uranium consumed, and used fuel produced by all reactors.

2.2.3.2 Agent Archetypes Developed

Each implemented agent is available in the Cycamore repository [35].

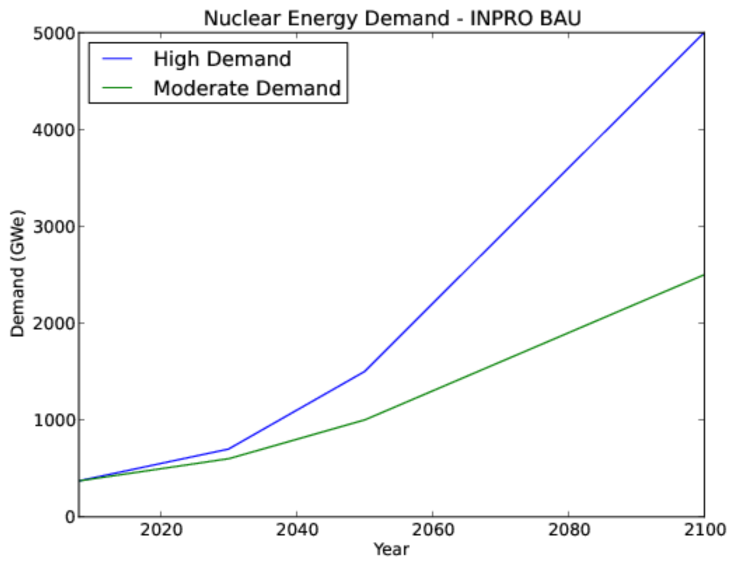

Figure 2.1: The energy demand specification for the INPRO BAU scenarios.

GrowthRegion

The `GrowthRegion` is a Region archetype developed to assist in facility deployment logic. The `GrowthRegion`↩ takes as input a listing of commodities for which it has a demand. For example, the `GrowthRegion` agents in this benchmark demand electrical power. The demand curves for commodities is defined by symbolic functions. Currently, linear functions, exponential functions, and piece-wise combinations of both are supported.

At any time step in which there exists a demand gap, i.e., there exists more demand than supply, a build decision is made. This decision is modeled as the following minimum cost facility deployment integer program:

$$\min_{n} \quad \sum_{i \in I} c_i * n_i \tag{2.1a}$$

$$\text{s.t.} \quad \sum_{i \in I} \phi_i * n_i \geq \Phi \tag{2.1b}$$

$$n_i \in [0, \infty) \quad \forall \ i \in I \tag{2.1c}$$

$$n_i \quad integer \quad \forall \ i \in I \tag{2.1d}$$

where Φ is the unmet demand, I is the set of facilities capable of meeting the demand, and, for each facility in I, c_i is the cost of building, and ϕ_i is the nameplate capacity. Finally, n_i is the optimized number of facilities to build of type i.

ManagerInst

The `ManagerInst` is an Institution archetype also developed to assist in facility deployment. While the `GrowthRegion` places a build order, the `ManagerInst` fulfills the order. Further, the `ManagerInst` determines the set of facilities, I, shown in in Eqn. 2.1, which can be built. Note that the set I can change over time. Once a deployment decision is made, the `GrowthRegion` makes a facility deployment request of the `ManagerInst` which then deploys the chosen facility.

BatchReactor

While a reactor model existed prior to this work, it did not provide the functionality to interchange *batches* of fuel, as required by the INPRO benchmark. A batch of fuel is a fraction of a full reactor core that is extracted and replaced when a reactor is refueled. In general, LWRs replace between a third and a quarter of their assemblies during refueling based on the fuel management scheme used.

The `BatchReactor` used in this work had configurable properties as displayed in Table 2.1. The values used based on the defined INPRO benchmark are described in Table 2.2.

Parameter	Description
Process Time	Active fuel time in the reactor
Refuel Time	Time to refuel the reactor
N Batches	Number of batches in the reactor
Batch Size	Quantity of a batch
Power Capacity	Nameplate Capacity for Power
Power Cost	Cost to build a new reactor

Table 2.1: Configurable input for the `BatchReactor` archetype.

EnrichmentFacility

The `EnrichmentFacility` archetype was developed to provide enrichment-related output for the simulation, namely the amount of separative work units (SWU) and natural uranium used during a given time step. For the INPRO cases, it can be defined quite simply using the values shown in Table 2.3. The

Parameter	LWR Value	HWR Value
Process Time	10	10
Refuel Time	2	2
N Batches	4	4
Batch Size	7.87E4	1.39e5
Power Capacity	1000	600
Power Cost	1000*	600*

(*) Note that the Cost used is arbitrary and set equal to the capacity so that a minimum capacity is built per Eqn. 2.1.

Table 2.2: Configurable input values for reactors used in the INPRO once-through benchmark.

feed assay and product assay, both required for determining output metrics, are defined by the isotopic compositions of resource input, i.e., natural uranium, and resource output, i.e., the isotopic composition of requested fuel.

Parameter	Description	Values
Input Recipe	A description of input isotopics	Natural Uranium
Tails Assay	The U-235 assay of tails.	0.003

Table 2.3: Configurable input values for the `EnrichmentFacility` used in the INPRO once-through benchmark.

2.2.3.3 Results

In aggregate, Cyclus performed well relative to the other benchmark codes. Fig. 2.2 shows the reactor deployment curves for each simulator for the moderate growth scenario while Fig. 2.3 shows reactor deployment for the high scenario. The slight differences are attributed to VISION's look-ahead functionality which builds the required facilities one time step after they are needed, whereas Cyclus builds facilities on the timestep in which they are needed. One can observe that a simple one timestep translation will result in identical output.

Cumulative natural uranium utilization curves for the moderate and high cases are shown in Figures 2.4 and 2.5, and cumulative used fuel inventory curves are shown in Figures 2.6 and 2.7. Slight discrepancies are noted between Cyclus and VISION. These discrepancies are attributed to the implementation of core batch recycling in each of the respective codes. The differences between Cyclus and VISION because the curves show cumulative metrics. In other words, results at time t_1 are added the results at time t_2 and so on. Therefore, a series of small discrepancies appears to be compounded by using this metric. It is not the

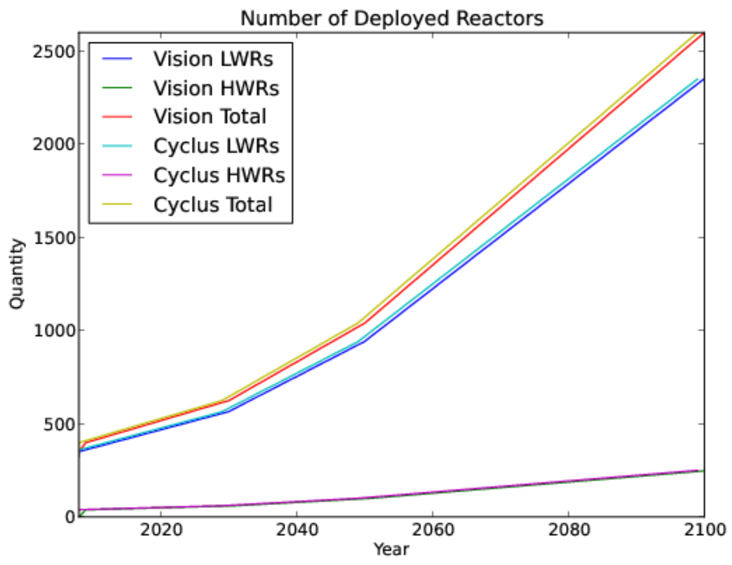

Figure 2.2: The reactor deployment schedule by reactor type for the moderate demand scenario.

metric of choice for general comparisons, but has been used because it was the metric of choice of the benchmark exercise. It is not immediately obvious why there is a greater discrepancy regarding output fuel quantities than natural uranium utilization. Further benchmarking exercises with support from a VISION developer would be required to fully investigate the issue.

2.2.4 Multiple Market Limitations

The proof of principle benchmark described in §2.2.3 utilized the agency provided for facility deployment rather than the agency provided for both deployment and resource exchange. In general informing resource exchange regarding quantity and quality of resources as well as socioeconomic effects is a hard problem.

Cyclus was originally designed to use an addition agent archetype called a `Market`. `Markets` were envisioned to represent markets for specific commodities. For example, the simulation described in §2.2.3 used three commodity markets: natural uranium, enriched uranium fuel, and used fuel. This approach is valid in the absence of supply or demand constraints, competition, and fungibility. However, the inclusion of any one of these features requires a much more involved process.

If supply or demand constraints are to be modeled, each associated `Market` agent must have both

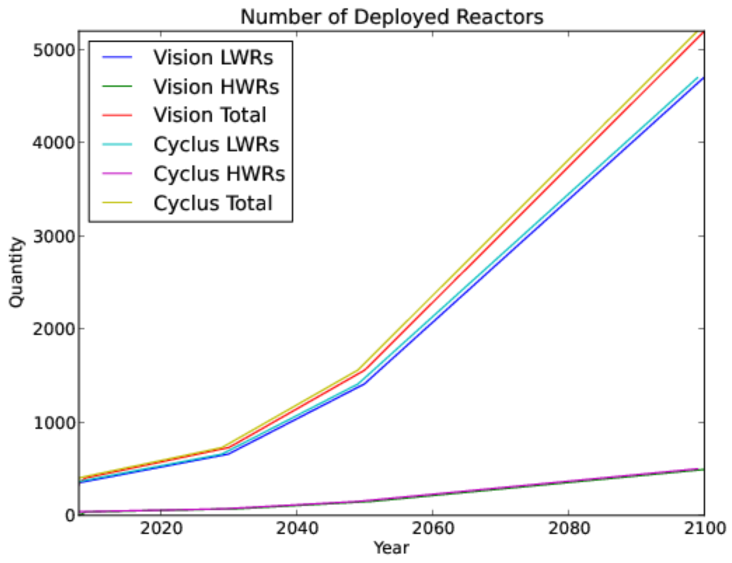

Figure 2.3: The reactor deployment schedule by reactor type for the high demand scenario.

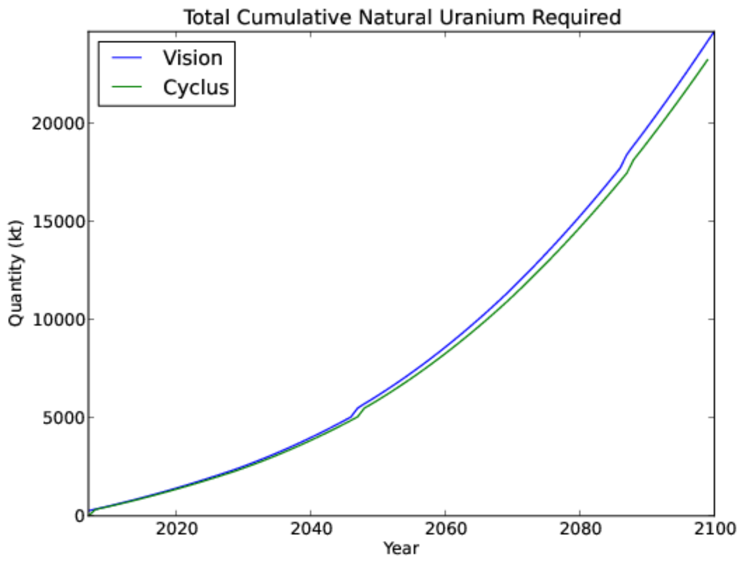

Figure 2.4: The total natural uranium used for the moderate demand scenario.

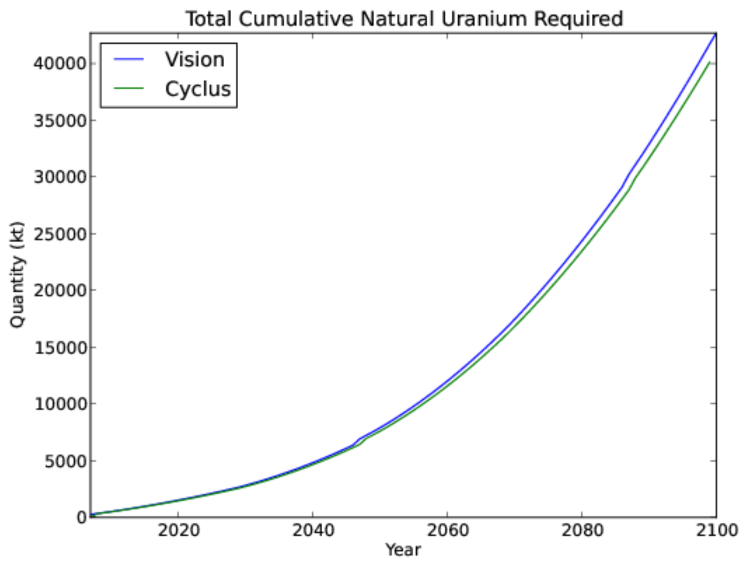

Figure 2.5: The total natural uranium used for the high demand scenario.

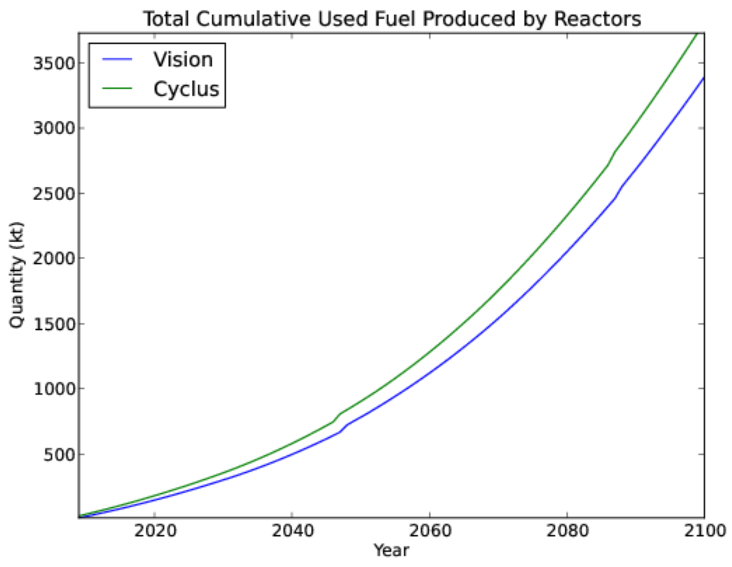

Figure 2.6: The amount of used fuel produced for the moderate demand scenario.

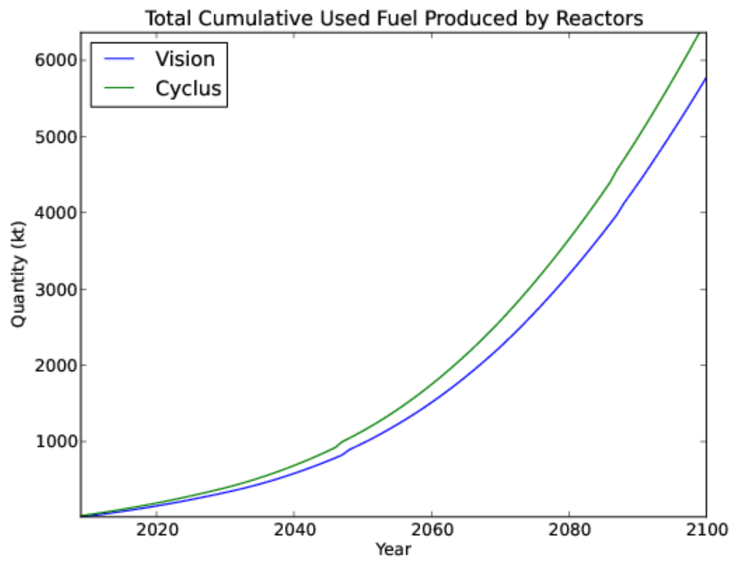

Figure 2.7: The amount of used fuel produced for the high demand scenario.

a corresponding communication interface and an implementation that accounts for such constraints. While quantity constraints are not unreasonable to implement and support, quality constraints are much more difficult. Furthermore, communicating such constraints is difficult. Whereas the `Market` agent can implement a solver algorithm, constraints are more naturally defined by the trader interacting with the `Market` agent. For example, consider the enriched uranium market used in §2.2.3. While the simulation used an agent abstraction for an enrichment facility and fuel fabrication plant, another simulation may wish to model facilities that downblend HEU, rather than enrich LEU. Such a process will have different constraints. Importantly, those constraints are a function of the `Facility` archetype, not of a `Market` archetype.

Assuming that supply or demand is constrained either resource quantity or quality, competition for the resource in question can arise. In fact, resources constraints are only interesting in the presence of competition. When competition for resources exist, there must be some mechanism that determines which transactions are to be executed, i.e., which agents should trade which resources. Determining supply and demand under competition is a well studied problem with many possible formulations and solution frameworks.

Fungibility is the property of a good or commodity to be *capable of being substituted in place of one* *another* [26]. For example, a light water reactor generates power by fissioning nuclei in the thermal energy spectrum. Whether those nuclei are ^{239}Pu , ^{235}U , or ^{233}U makes little difference from a power generation perspective. In other words, those nuclei are *fungible* for light water reactors, given some safety and cycle length considerations. A similar issue arises from a suppliers perspective. Consider a MOX fuel supplier and two requesters: a fast reactor and a thermal reactor. Given the isotopic makeup of Plutonium in the MOX fuel, the supplier's fuel could be potentially be used in either reactor type. Again, Plutonium in this example is a fungible resource. Importantly, the notion of fungibility in a NFC context can refer to both individual isotopes, collections of isotopes, or complete fuel forms. Accordingly, a facility may demand multiple fungible commodities, which must be accounted for by a given market clearing mechanism.

The one-market-per-commodity approach does not treat competition, constrained supply and demand, and fungibility particularly well. Constraints are handled poorly because constraints are best determined by the supplying and demanding agents rather than the market. Because the markets are separate, they must either be solved sequentially or in parallel. If solved sequentially, competition and fungibility are treated poorly, because information involving multiple commodities is not taken into account during the solution of a single market. Accordingly, a solution framework and methodology that incorporates agent querying of supply, demand, and constraints and resolves markets in parallel is required to properly treat resource exchange in the nuclear fuel cycle.

2.3 Dynamic Resource Exchange

Dynamic Resource Exchange (DRE) is the functional bedrock on which Cyclus simulations are built. It defines the interaction mechanisms and methodologies for agents, specifically agents whose archetypes have implemented the `Trader` interface. This section begins by providing a motivating problem statement in §2.3.1. It then details the methodology for querying supply and demand during the information gathering phase of the DRE in §2.3.2. The solution phase, in which the defined DRE is translated into a form of the Multicommodity Transportation Problem (MCTP) and solved, is then described in §2.3.3. Finally, two proof of principle simulations with novel fuel cycle DREs are presented in §2.3.5.

This section represents the culmination of significant previous effort [13, 15, 16]. What follows constitutes the refinement of previous descriptions of the DRE methodology with lessons learned from initial implementation and usage.

2.3.1 Problem Statement

As a next-generation nuclear fuel cycle simulation framework, Cyclus maintains a primary goal of modeling flexibility. As facility, institutional, and regional archetypes are proposed, they should be relatively easily implemented and utilized in the Cyclus simulation framework. Furthermore, the level of modeling abstraction for different facilities in a fuel cycle will be different based on the needs of archetype developer. Any supply-demand resolution framework, therefore, must be able to support arbitrary facilities. One way to approach such a problem is to treat facilities as black boxes, clearly defining a supply-demand communication framework.

As stated previously in §2.2.4, a number of considerations must be taken into account in such a framework. Supply and demand must be able to be solved globally at any given time step. Resources must be able to be treated in a fungible manner. The framework must be able to incorporate arbitrary, agent-defined constraints.

In order to address each of these concerns, the concept of a Dynamic Resource Exchange (DRE) was developed and implemented. That process was motivated by the following problem statement:

> If facilities are treated as individual black boxes and connections between facilities are determined dynamically, how does one match suppliers with consumers considering quantity and quality-based supply constraints, quantity and quality-based demand constraints, supply response to quality-based demands, and issues of fungibility?

2.3.2 Information Gathering

Supply-demand determination at any given time step begins with three *phases*, the terminology of which is influenced from previous supply chain agent-based modeling work [22]. Importantly, this information-gathering step is agnostic as to the supply-demand matching algorithm used, it is concerned only with querying the current status of supply and demand in the simulation. The collective information gathering procedure is shown in Figure 2.8.

The first phase allows consumers of commodities to denote both the quantity of a commodity they need to consume as well as the target isotopics, or quality, by *posting* their demand to the market exchange. This posting informs producers of commodities what is needed by consumers, and is termed the *Request for Bids* (RFB) phase. Consumers are allowed to over-post, i.e., request more quantity than they can

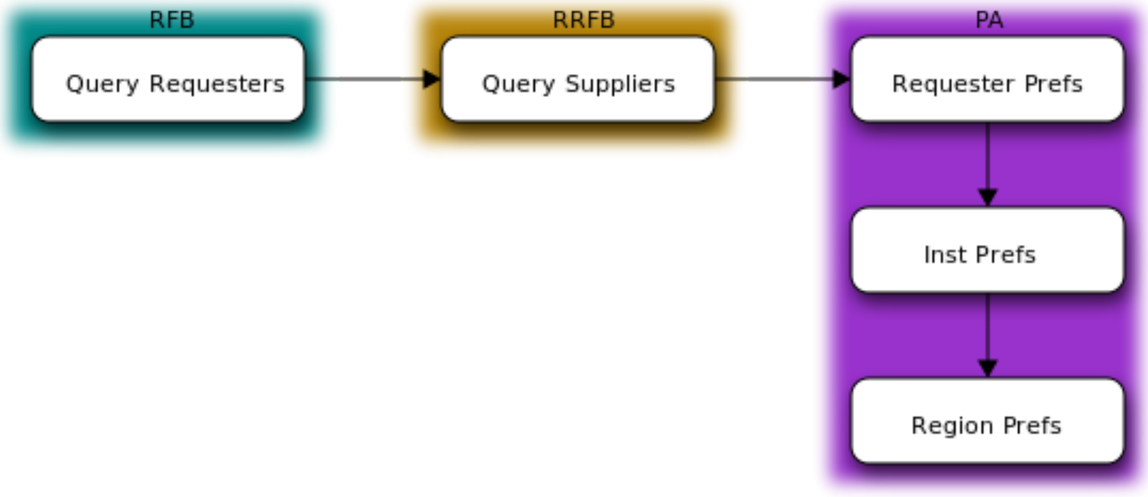

Figure 2.8: Schematic illustrating the DRE's information gathering procedure.

actually consume, as long as a corresponding capacity constraint accompanies this posting. Requests can be denoted as *exclusive*. An exclusive request is one that must either be met in full or not at all. Exclusive requests allow the modeling of quantized, packaged transfers, e.g., fuel assemblies. Further, consumers are allowed to post demand for multiple commodities that may serve to meet the same combine capacity. For example, consider an LWR that can be filled with MOX or UOX. It can post a demand for both, but must define a preference over the set of possible commodities that can be consumed. Such requests are termed *mutual requests*. Another example is that of an advanced fuel fabrication facility, i.e., one that fabricates fuel partially from separated material that has already passed through a reactor. Such a facility can choose to fill the remaining space in a certain assembly with various types of fertile material, including depleted uranium from enrichment or reprocessed uranium from separations. Accordingly, it could demand both commodities as long as it provides a corresponding constraint with respect to total consumption. At the completion of the RFB phase, the market exchange will have a set of request portfolios. Each each portfolio consists of a set requests. Arbitrary constraints over the set of requests can be provided that are functions of quantity or quality. For requests that mutually satisfy a given demand, a preference over those requests is required. Finally, each request portfolio has a specific quantity associated with it.

The second phase allows suppliers to *respond* to the set of request portfolios, and is termed the *Response to Request for Bids* (RRFB) phase (analogous to Julka's Reply to Request for Quote phase [22]). Each request portfolio is comprised of requests for some set of commodities. Accordingly, for each request, suppliers of that commodity denote production capacities and an isotopic profile of the commodity they can provide. Suppliers are allowed to offer the null set of isotopics as their profile, effectively providing no information. Suppliers are also allowed to denote responses as *exclusive*, as is done in the RFB phase. An exclusive offer must be accepted in its entirety or not at all. Again, exclusive offers provides a way to model quantized, packaged resources, such as fuel assemblies. As with requests, supply responses can be grouped into *mutual* responses. Every response in a mutual set is, by definition, exclusive. Of the set of responses, only one may be met. This functionality again supports the notion of quantized orders, e.g., in the case of fuel assemblies. A supplier may have its production constrained by more than one parameter. For example, a processing facility may have both a throughput constraint (i.e., it can only process material at a certain rate) and an inventory constraint (i.e., it can only hold some total material). Further, the facility could have a constraint on the quality of material to be processed, e.g., it may be able to handle a maximum radiotoxicity for any given time step which is a function of both the quantity of material in processes and the isotopic content of that material. Multiple of such constraints are allowed. At the completion of the RRFB phase the possible connections between supplier and producer facilities, i.e., the arcs in the graph of the transportation problem, have been established with specific capacity constraints defined both by the quantity and quality of commodities that will traverse the arcs.

The final phase of the information gathering procedure allows consumer facilities to adjust their set of preferences and for managers of consumer facilities to affect the consumer's set of preferences, as described in the remaining sections. Accordingly, the last phase is termed the *Preference Adjustment* (PA) phase. Preference adjustments can occur in response to the set of responses provided by producer facilities. Consider the example of a reactor facility that requests two fuel types, MOX and UOX. It may get two responses to its request for MOX, each with different isotopic profiles of the MOX that can be provided. It can then assign preference values over this set of potential MOX providers. Another prime example is in the case of repositories. A repository may have a defined preference of material to accept based upon its heat load or radiotoxicity, both of which are functions of the quality, or isotopics, of a material. In certain simulators, limits on fuel entering a repository are imposed based upon the amount of time that has elapsed since the fuel has exited a reactor, which can be assessed during this phase. The time constraint is, in actuality, a constraint on heat load or radiotoxicity (one must let enough of the fission products decay). A repository could analyze possible input fuel isotopics and set the arc preference of any that violate a given rule to 0, effectively eliminating that arc.

2.3.3 The Nuclear Fuel Cycle Transportation Problem

Supply and demand in a nuclear fuel cycle context is inherently a multicommodity problem. A light water reactor can be fueled by both UOX and MOX fuel, for instance. How it is fueled is a result both of fuel availability and associated preferences. Allowing for complex physical and chemical constraints on both processes and inventories, as well as including economics-based approaches for determining exchange preferences is a complicated affair. Determining the optimum solution to such a system is even more complicated. Accordingly, sophisticated tools in both the operations research and agent based modeling realms have been leveraged to accomplish the task.

An instance of supply and demand defined by the DRE information gathering step can be solved in a variety of ways. It can be cast to a constrained, bipartite network, and any heuristic that provides a feasible solution to such networks are valid. To solve the system optimally, however, a formal investigation and solution structure is needed. This section describes the construction of such a formulation, entitled the *Nuclear Fuel Cycle Transportation Problem* (NFCTP).

The basis for the formulation is the Multicommodity Transportation Problem described in §1.3.4 with some departures described in detail below. Two separate formulations are provided. The first is a strictly linear program (LP) while the second is a mixed-integer linear program (MILP). A heuristic is also provided that provides a reasonable solution to most simple problems.

The LP formulation can be solved quickly, but allows split orders. In other words, the LP formulation solves a relaxation of the defined instance that does not take into account *exclusive* requests or bids. The nuclear fuel cycle deals with bundled orders, such as nuclear fuel assemblies, thus this modeling paradigm is only an approximation. The MILP provides a more realistic exchange, but can take much longer to solve.

2.3.3.1 Terminology

Objects and data structures generated in the information gathering procedure are used in the formal definition of the NFCTP and must be defined.

Each portfolio can be considered separately. The set of supply portfolios is denoted as S and the set of request portfolios is denoted as R. Each supply portfolio is comprised of s_M supply nodes, and each request portfolio is comprised of r_N nodes. The set of supply nodes is denoted I, and the set of request nodes is denoted J. The total number of supply and request nodes is then

$$|I| = \sum_{s \in S} s_M \tag{2.2}$$

and

$$|J| = \sum_{r \in R} r_N . \tag{2.3}$$

Each portfolio has a set of commodities, H, associated with it. These are denoted H_s for supply portfolios and H_r for request portfolios. Furthermore, each portfolio has a set of constraints, K, associated with it. Each constraint has a constraining value, b_s^k and b_r^k, respectively. Additionally, each unique combination of portfolio and constraint has an associated *constraint coefficient conversion function*, denoted β_s^k for supply portfolios and β_r^k for request portfolios. Each constraint coefficient conversion function takes as an argument a proposed resource $q_{i,j}$. A clarifying example of the relation between portfolios, commodities, constraints, and coefficient conversion functions is provided in §2.3.3.3. Request portfolios are provided a quantity constraint by default for which coefficients are unity. For a set of *mutual requests*, M, where each request has a request quantity, x_m, the coefficient is defined by the ratio between the the average request quantity over all mutual requests and x_m

$$\beta_{r,m} = \frac{\bar{x_M}}{x_m} . \tag{2.4}$$

The constraint conversion functions are utilized in the NFCTP by applying them to the proposed resource transfers, creating capacity coefficients.

Coefficients for supply constraints are defined as

$$a_{i,j}^k = \beta_s^k (q_{i_j}) . \tag{2.5}$$

Coefficients for request constraints are defined as

$$a_{j,i}^k = \beta_r^k(q_{i_j}). \tag{2.6}$$

Finally, for each supply-request node pair, there is an associated preference, $p_{i,j}$. The set of all preferences is denoted P. Similarly, flow between a node pair is denoted $x_{i,j}$, and the set of all flows is denoted X. The possible flow on an arc is provided an upper bound by the request node quantity, $\tilde{x_j}$.

2.3.3.2 Exchange Graph

Upon completion of the information gathering phase, a *bipartite* network is formed. This network is called the *exchange graph*. The network consists of sending (bid) nodes, I, and receiving (request) nodes, J. For each request node, j, there may be many bid nodes; however, there is a one-to-one mapping between bid nodes and request nodes. In other words, a given bid node, i, is a unique response to a request node, j. An example of a bare exchange graph graph is shown in Figure 2.9.

In the bipartite graph, portfolios act as partitions that group nodes together. Node groups share common constraints, and request node groups share a common notion of satisfiable quantity, i.e., a default mass-based constraint. An example of a partitioned exchange graph is shown in Figure 2.10.

Because of defined constraints, there may not be sufficient supply or demand in the simulated exchange. To ensure a feasible solution, a false supply source and a false demand sink are added to the exchange graph. The false source and sink are unconstrained. Additionally, false nodes are added to each portfolio. Each false node in a request portfolio is connected to the false supply source, and each false node in a supply portfolio is connected to the false request sink. These arcs are denoted as *false arcs*. The preferences given to each false arc, p_f, is defined to be lower than the lowest preference in the system, P.

$$p_f < \min P \tag{2.7}$$

Given the original number of nodes in I and the number of bid portfolios, the total number of bid nodes including false nodes is

$$|I_t| = |I| + |S| + 1 \tag{2.8}$$

Similarly, the total number of request nodes is

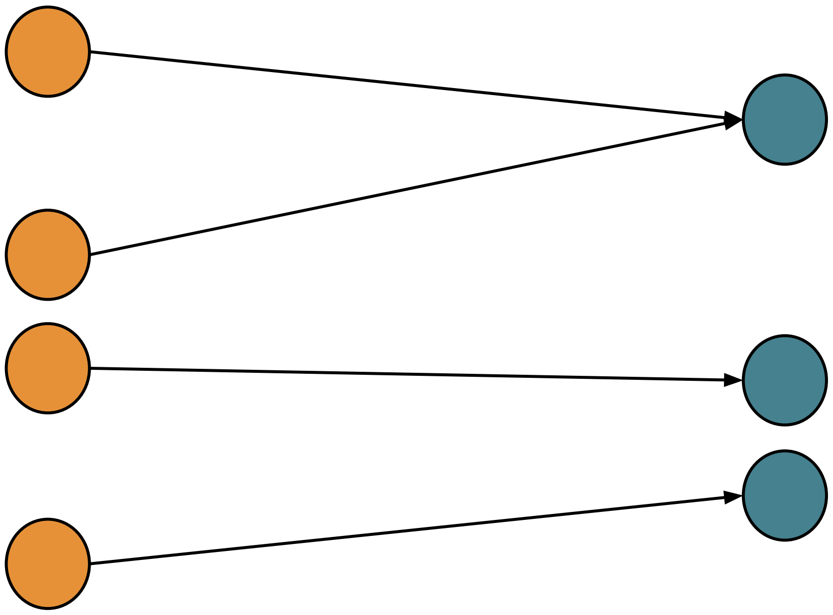

Figure 2.9: A bare example exchange with supply nodes colored orange on left and request nodes colored blue on right. As shown, there can be multiple supply nodes connected to a request node, but each supply node corresponds uniquely to one request node. It is a specific response to that request, as outlined in the RRFB phase.

$$|J_t| = |J| + |R| + 1 \tag{2.9}$$

Finally, the total number of arcs is

$$|A_t| = |A| + |I| + |J| \tag{2.10}$$

Because preferences are defined as in Equation 2.7, any false arc will only be engaged if no other possible arc can be engage, due to capacity constraints. If any flow is assigned to false arcs after the exchange graph is solved, that flow is ignored when initiating transactions. Figure 2.11 shows a fully defined exchange graph.

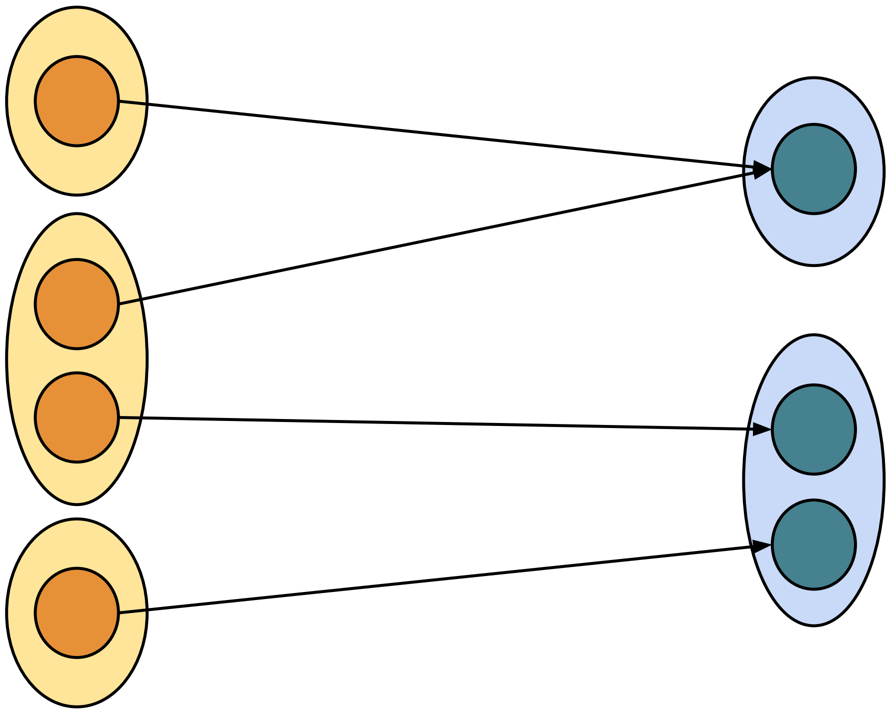

Figure 2.10: The same exchange shown in Figure 2.9 with the inclusion of portfolio partitions. In this example, there are three suppliers and two consumers. The second consumer has two requests which may satisfy its demand. The second supplier can supply the commodities requested by both consumers and has provided two bids accordingly.

2.3.3.3 Arc Properties

The result of the DRE is flow determined along arcs, where arcs connect supply nodes to request nodes. A number of properties are defined on arcs, namely commodities, constraint coefficients, and preferences.

Commodities

During the information gathering step in §2.3.2, consumers and suppliers are queried based on *commodities*. A consumer is allowed to request multiple commodities, and a supplier is allowed to supply multiple commodities. However, each possible resource transfer, i.e., each arc, is based on a single commodity. Accordingly, it is possible to color each arc, given a commodity-to-color mapping.

For example, consider an exchange similar to that shown in Figure 2.10 with two fuel commodities (A, B), two requesters (R_1, R_2), and two suppliers (S_1, S_2, S_3) in the configuration described by Tables 2.4

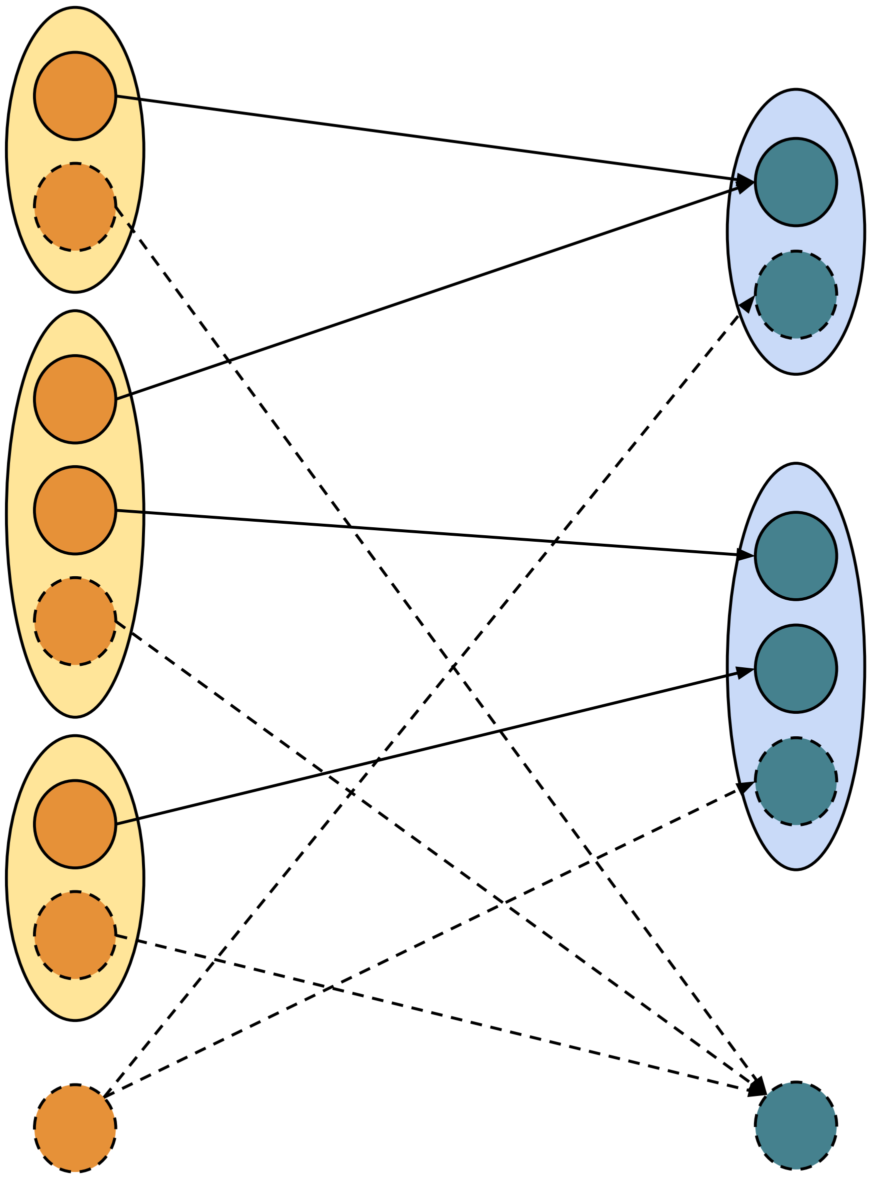

Figure 2.11: The same exchange shown in Figure 2.10 with the inclusion of false arcs. The false supplier and consumer nodes are shown with a dashed outline. Similarly, false arcs are dashed. Note that the false nodes have no associated portfolio structure – there are no constraints associated with false nodes and arcs. The inclusion of a false supplier and consumer guarantees a feasible solution.

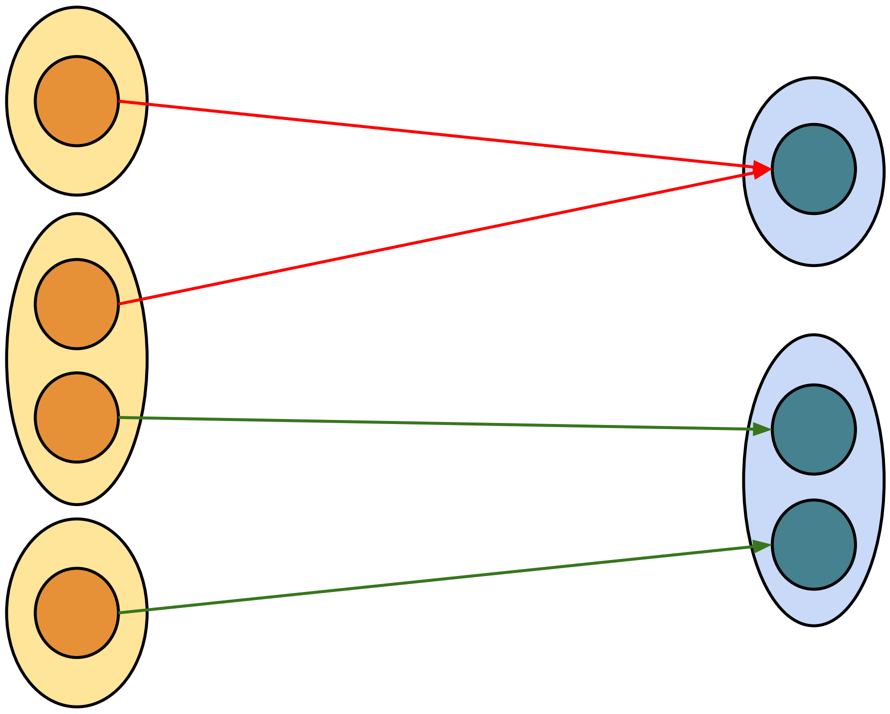

Figure 2.12: The same exchange shown in Figure 2.10 arcs colored by commodity based on Tables 2.4 and 2.5. A red arc corresponds to commodity A; a green arc corresponds to commodity B.

and 2.5.

Supplier	Commodities
S_1	A
S_2	A, B
S_3	B

Table 2.4: A mapping from suppliers to commodities supplied.

Consumer	Commodities
R_1	A
R_2	B

Table 2.5: A mapping from requesters to commodities requested.

Given the color map A: red, B: green, the resulting exchange graph can be colored as shown in Figure 2.12.

The notion of commodities is critical during the information gathering step and can be used to partition arcs in an exchange graph. It also is compelling when generating the formulations shown in §2.3.3.4 and

§2.3.3.5. However, given the constructed network graph, constraint structure, and preference structure, the notion of commodities is not necessary for the exchange graph to be solved.

Constraint Coefficients

Constraint coefficients are determined for an arc based on the proposed resource to be transferred along that arc, the requester's constraint translation functions, and the suppliers constraint translation function. The notion of a capacity translation function is something that has been introduced out of necessity due to the complexity of the DRE. An example of supply-based constraints is provided to help clarify its purpose.

Consider an supplier enrichment facility, s, which produces the commodity enriched uranium (EU). This facility has two constraints on its operation for any given time period: the amount of Separative Work Units (SWU) that it can process, b_s^{SWU}, and the total natural uranium (NU) feed it has on hand., b_s^{NU}. The constraint set for s is then

$$K_s = \{\text{SWU}, \text{NU}\}. \tag{2.11}$$

Note that neither of these capacities are measure directly in the units of the commodity it produces, i.e., kilograms of EU.

Consider a set of requests for enriched uranium that this facility can possibly meet. Such requests have, in general, two parameters: P_j, the total product quantity (in kilograms), and ε_j, the product enrichment (in w/o ^{235}U).[1] For the purposes of this constraint set, the quality of material in question is its enrichment, i.e.,

$$q_j \equiv \varepsilon_j. \tag{2.12}$$

These values are set during a prior phase of the overall matching algorithm, and can therefore be considered constant. Further, let us note that, in general, an enrichment facility's operation, or rather its capacity, is governed by two parameters: $\varepsilon_{f,s}$, the fraction of ^{235}U in its feed material, and $\varepsilon_{t,s}$, the fraction of ^{235}U in its tails material. These parameters determine the amount of SWU required to produce some amount of enriched uranium:

[1]The notation for enrichment, ε_j, is chosen over its normal form, x_p, to limit confusion with the notation of material flow, $x_{i,j}^h$.

$$\begin{aligned} SWU = \;& P(V(\varepsilon_j) + \frac{\varepsilon_j - \varepsilon_{f,s}}{\varepsilon_{f,s} - \varepsilon_{t,s}} V(\varepsilon_{t,s}) \\ & - \frac{\varepsilon_j - \varepsilon_{t,s}}{\varepsilon_{f,s} - \varepsilon_{t,s}} V(\varepsilon_{f,s})) \end{aligned} \tag{2.13}$$

P in Equation 2.13 is the amount of produced enriched uranium, and $V(x)$ is the value function,

$$V(x) = (1 - 2x) \ln \left(\frac{1 - x}{x} \right) \tag{2.14}$$

Utilizing the above equations, one can denote the functional forms of the arguments of this facility's two capacity constraints.

$$\beta_s^{NU}(\varepsilon_j) = \frac{\varepsilon_j - \varepsilon_{t,s}}{\varepsilon_{f,s} - \varepsilon_{t,s}} \tag{2.15}$$

$$\begin{aligned} \beta_s^{SWU}(\varepsilon_j) = \;& V(\varepsilon_j) \\ & + \frac{\varepsilon_j - \varepsilon_{f,s}}{\varepsilon_{f,s} - \varepsilon_{t,s}} V(\varepsilon_{t,s}) \\ & - \frac{\varepsilon_j - \varepsilon_{t,s}}{\varepsilon_{f,s} - \varepsilon_{t,s}} V(\varepsilon_{f,s}) \end{aligned} \tag{2.16}$$

These constraints correspond to the per-unit requirements for enriched uranium of natural uranium feed and SWU. Finally, we can form the set of constraint equations for the enrichment facility by combining Equations 2.12, 2.15, and 2.16.

$$\sum_{j \in J} \beta_s^{NU}(\varepsilon_j)\, x_{s,j} \leq b_s^{NU} \tag{2.17}$$

$$\sum_{j \in J} \beta_s^{SWU}(\varepsilon_j)\, x_{s,j} \leq b_s^{SWU} \tag{2.18}$$

Preferences & Costs

In any network flow problem, of which transportation problems are a subset, the cost of transporting commodities is what drives the solution. Thus, a cost function is necessary to determine a solution.

Because the Cyclus environment is still a nascent simulation platform, accurate pricing metrics, and what such metrics even are in terms of a centuries-long fuel cycle simulation, are generally difficult to ascertain, with the current standard source being the Advanced Fuel Cycle Cost Basis report [31]. Accordingly, the cost function is currently a measure of simulation entity preference, rather than a concrete representation of cost.

The notion of preference extends the work of Oliver's affinity metric [28]. The preference metric is generally consumer centric, i.e., consumers have a preference over the possible commodities that could meet their demand. For example, a reactor may be able to use UOX or MOX fuel, but may prefer to use MOX fuel. Such a preference differential allows the projection of real-world cost into the simulation. Additionally, the managers of a given facility, which in the Cyclus simulation environment include its Institution and Region, also exert an influence over its preference. An obvious example is the concept of affinities given in [28]. In Oliver's work, an affinity or preference existed between facilities in "similar" institutions in order to drive the trading between institutions as a simple model of international relations. This idea is expanded upon to cover a facility's other managers and the commodities themselves. Additionally, a preference can be delineated between the proposed qualities of the same commodity from different vendors, e.g. if two vendors of MOX fuel exist. Finally, the notion of a preference is a positive one, and we require a notion of cost to solve the minimum-cost formulation of the multicommodity transportation problem with side constraints. Therefore one must utilize a translation function.

Formally, a preference function, $p_{i,j}(h)$, is defined which is a cardinal preference ordering over a consumer's satisfying commodity set. A preference is assigned to each arc in the NFCTP.

$$p_{i,j}(h) \;\; \forall i \in I \;\; \forall h \in H_r \tag{2.19}$$

Preference is a function both of the consumer, j, and producer, i, and the proposed resource transfer from consumer to producer. The dependence on producer encapsulates the relationship effects due to managerial preferences, i.e., the effects of the Preference Adjustment phase described in §2.3.2.

A cost translation function, f, is defined that operates on the commodity preference function to produce an appropriate cost for the NFCTP.

$$f : p_{i,j}(h) \rightarrow c_{i,j} \tag{2.20}$$

For the purposes of this work, any operator that preserves the preference monotonicity and cardinal ordering is suitable. The inversion operator has been chosen because it preserves required features and also allows for easy translation from preference to cost as well as translation from cost to preference.

$$f(x) = \frac{1}{x} \tag{2.21}$$

If cost data and a valid cost assignment methodology is developed in the future, costs may be used directly, and the preference-to-cost translation may be ignored.

2.3.3.4 Linear Programming Formulation

Combining the previous discussions, the LP Formulation of the NFCTP, denoted the NFCTP-LP, can be constructed. In general, the NFCTP is a minimum cost transportation problem that includes custom constraints as described in previous sections. Including all of the discussion in the previous sections, the formulation is straightforward and shown in Equation 2.22.

$$\min_x \; z = \sum_{i \in I} \sum_{j \in J} c_{i,j} x_{i,j} \tag{2.22a}$$

$$\text{s.t.} \; \sum_{i \in I_s} \sum_{j \in J} a^k_{i,j} x_{i,j} \leq b^k_s \qquad \forall\, k \in K_s, \forall\, s \in S \tag{2.22b}$$

$$\sum_{j \in J_r} \sum_{i \in I} a^k_{i,j} x_{i,j} \geq b^k_r \qquad \forall\, k \in K_r, \forall\, r \in R \tag{2.22c}$$

$$x_{i,j} \in [0, \tilde{x_j}] \qquad \forall\, i \in I, \forall\, j \in J \tag{2.22d}$$

The variables and sets used to define Equation 2.22 have been described in detail in previous sections. A short synopsis of the sets used is provided in Table 2.6, and a corresponding synopsis of the variables used is provided in Table 2.7.

Notably, a feasible solution to the formulation provided in Equation 2.22 is guaranteed due to the presence of false arcs. Accordingly, the DRE using this formulation will never fail within a simulation.

Set	Description
S	suppliers
R	requesters
I	all supply nodes
I_s	nodes for a supplier s
J	all request nodes
J_r	nodes for a requester r
K_s	constraints for a supplier s
K_r	constraints for a requester r
X	the feasible set of flows between producers and consumers

Table 2.6: Sets Appearing in the NFCTP-LP Formulation

Variable	Description
$c_{i,j}$	the unit cost of flow from producer node i to consumer node j
$x_{i,j}$	a decision variable, the flow from producer node i to consumer node j
$a_{i,j}^k$	the constraint coefficient for constraint k on flow between nodes i and j
b_s^k	the constraining value for constraint k of supplier s
b_r^k	the constraining value for constraint k of requester r
$\tilde{x_j}$	the requested quantity associated with request node j

Table 2.7: Variables Appearing in the NFCTP-LP Formulation

2.3.3.5 Mixed Integer Linear Programming Formulation

The previous linear program (LP) formulation of the Generic Fuel Cycle Transportation Problem fully describes many of the types of transactions that arise at any given time step. However, it does not allow the critical case of reactor fuel orders, which comprise a large amount of material orders within the simulation context. Specifically, it allows reactor fuel orders to be met by more than one supplier with an arbitrary amount of the order met by each supplier. Put another way, the LP formulation does not contain the discrete material information required to model the transaction of fuel assemblies. In order to provide this capability of quantizing orders, binary decision variables must be introduced and integer programming techniques must be utilized to solve the resulting mixed integer-linear program.

The addition of integer variables changes both the complexity of the formulation and the complexity of the solution technique. Such a change requires a Mixed Integer-Linear Program (MILP) formulation and solution via the branch-and-bound method which solves NP-Hard combinatorial optimization problems. The Linear Program (LP) version requires the simplex method which is much more efficient.

Binary Variables

The primary difference between the LP and MILP formulations is the inclusion binary decision variables $y_{i,j}$. A variable $y_{i,j}$ has a value of 1 if flow occurs between producer node i and consumer node j. If flow occurs, its quantity will be equal to the equivalent flow upper bound along that arc, $\tilde{x}_j$, which denote the quantity of a quantized order.

Binary variables, representing quantized flow, are directly related to the notion of *exclusive* bids and requests discussed in §2.3.2. In the MILP formulation, an arc (i, j) is considered exclusive if either node i or node j was defined as exclusive in the information gathering phase of the DRE. Accordingly, it is useful to partition all arcs based on this characteristic. Given the set of arcs A, a partition exists such that A can be separated into exclusive arcs, A_e, and non-exclusive arcs, or arcs that allow partial flow, A_p.

$$A = A_p \cup A_e \tag{2.23}$$

Similarly, each partition can be further subdivided into partitions based on supplier and requester.

$$A = \bigcup_{r \in R} A_{p_r} \cup A_{e_r} \tag{2.24}$$

$$A = \bigcup_{s \in S} A_{p_s} \cup A_{e_s} \tag{2.25}$$

Mutually Exclusive Constraints

Mutual requests and responses were described in §2.3.2. These are defined as a set of requests or responses, of which only one may be satisfied. This is represented in the formulation as a constraint on the associated variables. Again, if a variable $y_{i,j}$ is set to 1, flow is sent along arc (i, j). If it is 0, no flow occurs. A *mutually exclusive* constraint simply says that only one arc in a mutual set may have a value of 1.

The set of mutually satisfying arcs is denoted M_s and M_r for suppliers and requesters, respectively. The associated constraints are then defined by Equations 2.26 and 2.27.

$$\sum_{(i,j) \in M_s} y_{i,j} \leq 1 \; \forall \; s \in S \tag{2.26}$$

$$\sum_{(i,j) \in M_r} y_{i,j} \leq 1 \, \forall \, r \in R \tag{2.27}$$

Formulation

Using the above arc partition notation allows for a much simpler written formulation of the MILP that looks quite close to the related LP formulation shown in Equation 2.22. The full formulation of the NFCTP is shown in Equation 2.28

$$\begin{aligned}
\min_{x,y} \; z \; &= \sum_{(i,j) \in A_p} c_{i,j} x_{i,j} \; + \sum_{(i,j) \in A_e} c_{i,j} \tilde{x_j} y_{i,j} && && \text{(2.28a)} \\
\text{s.t.} \; & \sum_{(i,j) \in A_{p_s}} a^k_{i,j} x_{i,j} \; + \sum_{(i,j) \in A_{e_s}} a^k_{i,j} \tilde{x_j} y_{i,j} \leq b^k_s && \forall \, k \in K_s, \forall \, s \in S && \text{(2.28b)} \\
& \sum_{(i,j) \in M_s} y_{i,j} \leq 1 && \forall \, s \in S && \text{(2.28c)} \\
& \sum_{(i,j) \in A_{p_r}} a^k_{i,j} x_{i,j} \; + \sum_{(i,j) \in A_{e_r}} a^k_{i,j} \tilde{x_j} y_{i,j} \geq b^k_r && \forall \, k \in K_r, \forall \, r \in R && \text{(2.28d)} \\
& \sum_{(i,j) \in M_r} y_{i,j} \leq 1 && \forall \, r \in R && \text{(2.28e)} \\
& x_{i,j} \in [0, \tilde{x_j}] && \forall \, (i,j) \in A_p && \text{(2.28f)} \\
& y_{i,j} \in \{0, 1\} && \forall \, (i,j) \in A_e && \text{(2.28g)}
\end{aligned}$$

The sets and variables involved in Equation 2.28 are described in Tables 2.8 and 2.9.

The examples of the various constraints from the previous section also apply here. The only difference is the notion of the binary variables, $y_{i,j}$, which act as on/off switch as to whether a consumer's entire requested amount of a resource is met by a supplier or not.

It should be noted that this advanced formulation adds significant complexity to the resolution method at every time step. However, simple heuristics exist. A common heuristic for MILPs is to solve a relaxed version of the problem in the form of a linear program, and to round values to form an integer solution. A heuristic used in Cyclus is provided in §2.3.3.6.

Note that each constraint coefficient for binary variables can be rewritten as Equation 2.29 and each

Set	Description
S	suppliers
R	requesters
A_p	arcs that allow *partial* flows
A_e	*exclusive* flow arcs
A_{p_s}	arcs that allow *partial* flows for supplier s
A_{e_s}	*exclusive* flow arcs for supplier s
A_{p_p}	arcs that allow *partial* flows for requester r
A_{e_p}	*exclusive* flow arcs for requester r
M_s	arcs (i, j) associated with *mutually exclusive* supply for supplier s
M_r	arcs (i, j) associated with *mutually exclusive* requests for requester r
X	the feasible set of flows between producers and consumers
Y	the binary variable set of flows between producers and consumers

Table 2.8: Sets Appearing in the NFCTP Formulation

Variable	Description
$c_{i,j}$	the unit cost of flow from producer node i to consumer node j
$x_{i,j}$	a decision variable, the flow from producer node i to consumer node j
$y_{i,j}$	a decision variable, whether flow exists from producer node i to consumer node j
$a_{i,j}^k$	the constraint coefficient for constraint k on flow between nodes i and j
b_s^k	the constraining value for constraint k of supplier s
b_r^k	the constraining value for constraint k of requester r
$\tilde{x_j}$	the requested quantity associated with request node j

Table 2.9: Variables Appearing in the NFCTP Formulation

objective coefficient can be rewritten as Equation 2.30.

$$a_{i,j}^{k\prime} = a_{i,j}^k \tilde{x_j} \tag{2.29}$$

$$c_{i,j}^{\prime} = c_{i,j} \tilde{x_j} \tag{2.30}$$

Using both updated efinitions, a simpler formulation can be written and is shown in Equation 2.31.

$$\min_{x,y} z = \sum_{(i,j)\in A_p} c_{i,j}x_{i,j} + \sum_{(i,j)\in A_e} c'_{i,j}y_{i,j} \tag{2.31a}$$

$$\text{s.t.} \sum_{(i,j)\in A_{p_s}} a^k_{i,j}x_{i,j} + \sum_{(i,j)\in A_{e_s}} a^{k\prime}_{i,j}y_{i,j} \le b^k_s \qquad \forall\, k \in K_s, \forall\, s \in S \tag{2.31b}$$

$$\sum_{(i,j)\in M_s} y_{i,j} \le 1 \qquad \forall\, s \in S \tag{2.31c}$$

$$\sum_{(i,j)\in A_{p_r}} a^k_{i,j}x_{i,j} + \sum_{(i,j)\in A_{e_r}} a^{k\prime}_{i,j}y_{i,j} \ge b^k_r \qquad \forall\, k \in K_r, \forall\, r \in R \tag{2.31d}$$

$$\sum_{(i,j)\in M_r} y_{i,j} \le 1 \qquad \forall\, r \in R \tag{2.31e}$$

$$x_{i,j} \in [0, \tilde{x_j}] \qquad \forall\, (i,j) \in A_p \tag{2.31f}$$

$$y_{i,j} \in \{0,1\} \qquad \forall\, (i,j) \in A_e \tag{2.31g}$$

2.3.3.6 A Heuristic Solution

With full simulation domain knowledge of supply and demand, including false arcs, a feasible solution can be found. By definition a feasible solution is a *solution* to the possible flow of resources, but not necessarily an *optimal* solution. Many heuristics may be applied to bipartite graphs with constrained flows. A simple *greedy* heuristic is presented here and implemented.

The maximum flow along an arc, x_{max}, depends on the constraints associated with each node on the arc. For nodes i and j belonging to portfolios s and r, respectively, the maximum allowable flow is defined as

$$x_{max} = \min\{\min\{\frac{b^k_s}{a^k_{i,j}}\ \forall k \in K_s\},\ \min\{\frac{b^k_r}{a^k_{i,j}}\ \forall k \in K_r\}\}. \tag{2.32}$$

The Greedy Exchange Heuristic matches maximum flow along arcs, up to the requested amount defined by each request portfolio, q_r, after having sorted all arcs. The constraining values of each arc, b_k, are updated upon declaration of a match (via an `AddMatch` function) in Algorithm 1.

```
**Data**: A resource exchange graph with constraints and preferences.
**Result**: A valid set of resource flows.
sort request partitions by average preference;
forall the $r \in R$ do
    sort requests by average preference;
    matched $\leftarrow$ 0;
    while matched $\leq q_r$ and $\exists$ a request do
        get next request;
        sort incoming arcs by preference;
        while matched $\leq q_r$ and $\exists$ an arc do
            get next arc;
            remaining $\leftarrow q_r$ - matched;
            to_match $\leftarrow \min\{$remaining, $x_{max}\}$;
            AddMatch(arc, to_match);
            matched $\leftarrow$ matched + to_match;
        end
    end
end
```

Algorithm 1: Greedy Exchange Heuristic

2.3.3.7 Departure from the MCTP

The classic MCTP includes the coloring of flows based on commodity type. For example, for a commodity, h, the unit cost of flow would be $c_{i,j}^h$ rather than $c_{i,j}$. This is included because multiple commodities can flow along the same arc in the MCTP. In other words, the node-arc incidence matrix includes an extra commodity dimension.

The multicommodity nature of the NFCTP is included in constraints, rather than arcs. Because each node pairing, (i, j), corresponds to a specific, proposed resource transfer, it can only have one commodity associated with it. Instead, the constraint set, K, is applied over multiple arcs, where each arc is assigned its own commodity.

Take the enrichment facility example, expanding on the previous discussion. Note that an enrichment facility takes feed uranium and then enriches its ^{235}U content. This feed uranium can come from different sources which have different feed enrichments. In practice, the most likely sources of feed uranium are natural uranium (NU) or recycled uranium (RU), a product of reprocessing light water reactor fuel. Recycled uranium may be advantageous to use if it has a higher weight percent of ^{235}U than does natural uranium. We can now state the set the values for H_r for this facility:

$$H_r = \{\text{NU}, \text{RU}\} \tag{2.33}$$

One or more constraints would then accompany any requests. For example, one could constraint total ^{235}U content needed, which would include both NU and RU flows.

2.3.4 Implementation

The DRE and its solution framework are implemented in three layers. The first layer includes information for specific `Resource` types. For example, a `Material`-based exchange is used for agents to communicate supply and demand information regarding `Material` objects. The *resource layer* is the point of entry and exit of the DRE framework. It is the agent-facing interface of the DRE: supply and demand is provided to the DRE as input during the information gathering step, and trades to be executed are provided to agents as output.

The second layer, called the *exchange layer*, is a `Resource`-agnostic implementation of a specialized bipartite graph. Supply/demand constructs in the first layer are translated into stateful objects representing nodes, arcs, constructs that carry constraint information, *et cetera*. The collection of objects and structures combine to create an `ExchangeGraph`. Any custom, Cyclus-aware solver can be applied to an `ExchangeGraph` to determine a feasible solution to the DRE.

In order to use sophisticated, 3rd party LP and MILP solving libraries, the `ExchangeGraph` must be translated into an appropriate data structure representing an instance of the NFCTP, resulting in the *formulation layer*. The Open Solver Interface (OSI) [12] is used to create the necessary formulation structures, including a constraint matrix and objective coefficient vector. The NFCTP instance is then solved.

After a feasible, perhaps optimal, solution to the NFCTP is found, whether in the exchange or formulation layer, the solution is back-translated to the resource layer. The agents associated with successful supply-demand connections are informed, and trades of resources between agents are executed. A graphic of the entire workflow is shown in Figure 2.13.

2.3.4.1 Resource Layer

The resource layer utilizes *templated* classes in order to reduce the amount of code required for implementation. Each object is templated on the concrete `Resource` type, e.g., the `Material` and `Product` classes. The fundamental data structures in the resource layer reflect the constructs of the information gathering procedure described in §2.3.2.

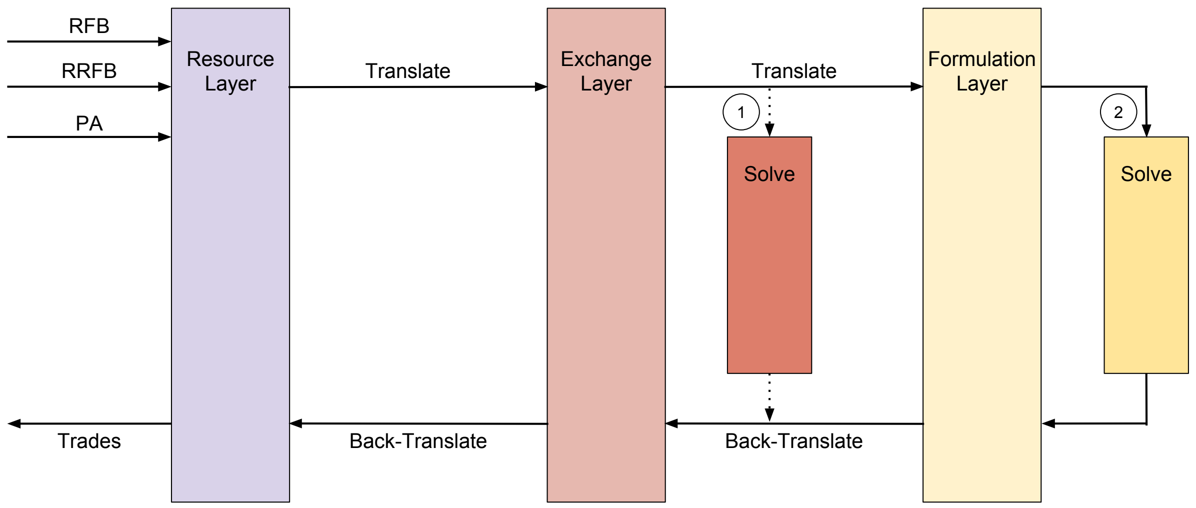

Figure 2.13: The full DRE workflow is shown. The information gathering phase results in the resource layer. The resource layer is translated to the exchange layer; a decision is made whether to continue translation or to directly solve, marked by the number 1. If the exchange is not solved, it is translated into an instance of the NFCTP resulting in the formulation layer. A choice of solver is made, marked by the number 2, and the instance is solved. The solution is back-translated through the exchange and resource layers. The result is a series of resource trades to be executed in the simulation.

In the RFB phase of the DRE, agents populate `RequestPortfolio<T>`s with `Request<T>`s and `CapacityConstraint<T>`s. A `Request<T>` defines a desired `Resource<T>`, communicating quantity, quality, and preference. Any number of `CapacityConstraint<T>`s may be added to a `RequestPortfolio<T>`. A `CapacityConstraint<T>` defines a capacitating value and a conversion function that takes as an argument a `Resource` and returns a value in units of the conversion function. For `RequestPortfolio<T>`s, constraints are assumed to be demand constraints, i.e., take the form of a greater-than constraint. In the RRFB phase of the DRE, agents populate `BidPortfolio<T>`s with `Bid<T>`s and `CapacityConstraint<T>`s. Agents can inspect the population of `Request<T>`s and associated `Resources`. A `Bid<T>` targets a specific `Request<T>`, responding with a proposed `Resource` to transfer to the requester. `CapacityConstraint<T>`s are applied to all `Bid<T>`s in a portfolio. For bidders, constraints are assumed to be less-than constraints. Before continuing, requesting agents and their managers are allowed to alter the preference associated with each `Request<T>`-`Bid<T>` pair in the PA phase of the DRE. When a solution to the DRE is found, bidders associated with successful `Request<T>`-`Bid<T>` pairs are informed, and a trade of the bidder's `Resource` is initiated.

Future work can be focused on providing more features to the DRE implementation. A natural extension of the present work is to support both kinds of constraints, greater and less-than, in `Portfolio` data strutures. Additionally, the PA procedure could use a negotiation model that involves both suppliers and requesters in order to define a final preference for an arc. Such an extension would allow for more seamless and natural usage of arc costs in addition to preferences.

2.3.4.2 Exchange Layer

The exchange layer is constructed by an `ExchangeTranslator` object that translates the resource layer objects into an instance of an `ExchangeGraph`. Request and bid objects are translated to `ExchangeNodes`, and portfolio objects are translated to `ExchangeGroups`. Constraint coefficient and preference information is recorded on `ExchangeArcs`, which store a reference to a supply `ExchangeNode` and a demand `ExchangeNode`. Finally, constraint values are stored on the appropriate `ExchangeGroup` object.

An `ExchangeContext` object is tasked with storing a mapping from `Request<T>` and `Bid<T>` objects to their associated `ExchangeNode`. Importantly, the exchange layer does *not* depend on resource type, i.e., the resource type is abstracted away during translation. Finally, a general solver can be implemented that operates on the `ExchangeGraph`. A solution to the `ExchangeGraph` instance is a mapping from `ExchangeArcs` to flow quantities that does not violate the provided constraints. After a solution is found, it is back-translated to the resource layer.

2.3.4.3 Formulation Layer

While a solver may operate on the exchange layer, an instance of an `ExchangeGraph` can be translated fully into the NFCTP. Once in an LP or MILP form, the DRE instance can be solved by sophisticated 3rd party libraries. In order to interface with a large number of the possible solvers, including COIN-OR and CPLEX, the COIN-OR Open Solver Interface API [12] is utilized.

The translation from the exchange layer to formulation layer is managed by the `ProgTranslator` class. A variable in the NFCTP is associated with each `ExchangeArc`, with variable bounds set by request values on `ExchangeNodes`; a binary variable is used if the arc is exclusive, otherwise a linear variable is used. Capacity coefficients and preference values defined for `ExchangeArcs` are translated into an objective coefficient vector and constraint matrix. The right-hand-side b constraint vector is determined by `ExchangeGroup` constraining values.

A solution to the NFCTP instance is determined by the identified solver, assigning values to linear and integer variables. Linear variable values map directly to assigned resource flow quantity. If a binary variable is set to unity in a solution, the maximum possible flow value is assigned, analogous to $\tilde{x_j}$ in the NFCTP formulation. The variable-flow value assignment is then back-translated into an equivalent `ExchangeArc`-flow value assignment by the `ProgTranslator`.

2.3.5 Proof of Principle

In order to demonstrate the correctness of the methodology and implementation, two test cases were developed and analyzed. These test cases are entire Cyclus simulations in which the full DRE procedure is executed at each time step. Both scenarios validate the ability of the DRE to model preferences, preference adjustment, and unresolved markets. The first scenario is a simulation including quantity-based constraints and dynamic commodity-based preferences. The second scenario illustrates a simulation that involves quality constraints and dynamic quality-based requests. In each scenario, fuel quantities are treated using arbitrary units without loss of generality. For the purposes of the enrichment example, a unit of fuel is equivalent to a kilogram.

Each scenario is comprised of archetypes defined in Cycamore [35]. The minimal Institution and Region archetypes are used because no complicated facility deployment logic is needed. The facility archetypes used include the *SourceFacility*, *BatchReactor*, and *EnrichmentFacility*.

Finally, each instance of the DRE is solved using the greedy heuristic described in §2.3.3.6. In each case, requests and supplies are *not* exclusive, and thus multiple sources of supply may be matched to a request. In general, these exchanges are very small and have a unique objective solution which corresponds to the solution determined by the greedy heuristic.

2.3.5.1 Test Cases

2 Sources, 3 Reactors

As shown in Figure 2.14, this scenario includes three *BatchReactors* and two *SourceFacilities*. The *BatchReactors*, denoted as `Reactor1`, `Reactor2`, and `Reactor3`, each have a unique fuel preference. One *SourceFacility* supplies MOX while the other supplies UOX; these are denoted as `MOX_Source` and `UOX_Source`, respectively. Any reactor may be fueled from MOX or UOX fuel; both fuel types are *fungible* in this scenario.

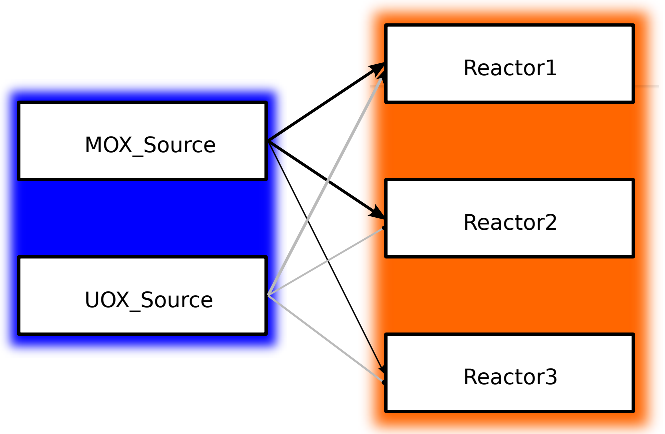

Figure 2.14: Schematic illustrating the first fuel cycle scenario. The thickness of the arrows represents the preference value and the grey color indicates that a material transfer is possible.

In this example case `Reactor1`, `Reactor2`, and `Reactor3` are deployed sequentially over 3 time steps. Each of these has a full core when built and requires 1 unit of fresh fuel at each subsequent time step. Both source facilities have a capacity of 2.5 units each time step.

The simulation begins with the following facilities: `MOX_Source`, `UOX_Source`, and `Reactor1`. At time step 2, `Reactor2` is deployed, followed by `Reactor3` at time step 3. `Reactor1` and `Reactor2` both are given a stronger preference for MOX requests than `Reactor3`. At time step 4, `Reactor1`, `Reactor2`, and `Reactor3` all request to refuel with MOX. At time step 5, `Reactor1` changes its preference to UOX. Table 2.10 summarizes the reactor preferences as a function of time.

Table 2.10: Time sequence of reactor preferences and the total MOX requested. The MOX capacity for each time step is 2.5 units.

Time step	`Reactor1`		`Reactor2`		`Reactor3`		Total MOX Requested [units]
	Fuel	Preference	Fuel	Preference	Fuel	Preference	
1	MOX	1.0	none		none		0.0
2	MOX	1.0	MOX	1.0	none		1.0
3	MOX	1.0	MOX	1.0	MOX	0.5	2.0
4	MOX	1.0	MOX	1.0	MOX	0.5	3.0
5	UOX	2.0	MOX	1.0	MOX	0.5	2.0

Enrichment, 2 Reactors

As pictured in Figure 2.15, this scenario includes one *EnrichmentFacility* and two *BatchReactors*. The *EnrichmentFacility*, denoted as `Enrichment`, has a designated capacity at each time step. The two *BatchReactors*, denoted as `Reactor1` and `Reactor2`, request a given amount and quality of enriched uranium upon refueling.

The `Enrichment` facility is constrained by a constant capacity of 10 SWU per time step. For this entire simulation, trade between `Enrichment` and `Reactor1` is preferred over trade between `Enrichment` and `Reactor2` with preference values of 1.0 and 0.5, respectively. Each reactor requests 1 unit of enriched uranium at each time step.

Initially, both reactors are present in the simulation and have a full core of 3% enriched uranium. On time step 1, `Reactor1` requests uranium enriched to 5% U-235 while `Reactor2` requests uranium at a 3% enrichment level. At time step 2, `Reactor1` reduces its enrichment request to 3%. Table 2.11 summarizes the reactor requests as a function of time.

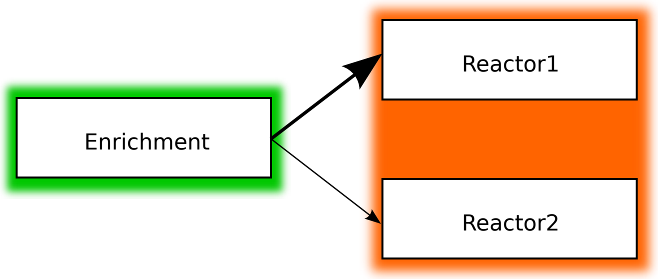

Figure 2.15: Schematic illustrating the second fuel cycle scenario. The thickness of the arrows represents the preference value.

Table 2.11: Time sequence of reactor preferences and the total SWU requested. The SWU capacity for each time step is 10.

Time step	`Reactor1`		`Reactor2`		Total SWU Requested
	Recipe	Preference	Recipe	Preference	
0	3% U-235		3% U-235		0
1	5% U-235	1.0	3% U-235	0.5	10.6
2	3% U-235	1.0	3% U-235	0.5	6.8

2.3.5.2 Results

The cases outlined in §2.3.5.1 have been designed to provide different conditions at each point in time. The results for these cases are discussed below. In each of these cases, the total Cyclus run time was ~0.1 seconds and the output database size was ~68 kB.

Note that the resource flows in Figures 2.16-2.22 have been generated automatically from Cyclus output using Cyan [11]. Due to this, these figures only show facility agents which participated in a resource exchange. For instance, Figure 2.16 does not show the `UOX_Source` facility, even though it is present in the simulation.

2 Sources, 3 Reactors

Initially present are the source facilities and `Reactor1`. `Reactor1` has a preference for accepting MOX fuel over UOX. The `MOX_Source` capacity of 2.5 units is more than enough to handle the 1 unit of MOX requested by `Reactor1`. This matching may be seen in Figure 2.16. The `MOX_Source` only provides the 1 unit of material requested by `Reactor1`. It, correctly, does not oversupply.

At time step 2 in this simulation, `Reactor2` is deployed and also requests fuel with the preference for MOX. Figure 2.17 displays that the `MOX_Source` indeed has the required capacity to meet the requests of both of the reactors. This may seem trivial at first glance but it is important to emphasize that the resource exchange solver was not altered in any way to handle both time steps 1 and 2. Furthermore, the solver on time step 1 had no future knowledge that `Reactor2` would be deployed on time step 2. This is significantly different than the traditional system dynamics approach.

On time step 3, `Reactor3` is deployed. This facility still prefers to accept MOX fuel over UOX fuel. At this point, 3 units of MOX are requested (1 unit from each facility) but the `MOX_Source` may only provide

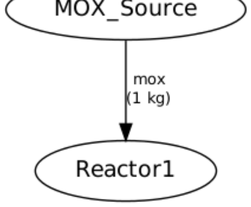

Figure 2.16: Time step 1 for the 2 Sources, 3 Reactors case.

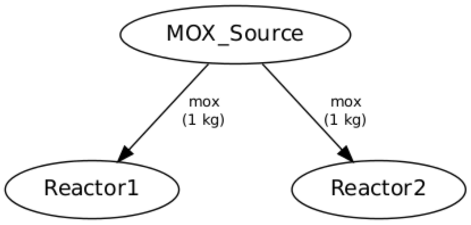

Figure 2.17: Time step 2 for the 2 Sources, 3 Reactors case.

Table 2.12: Resource exchange preferences for agents on time steps 3 and 4 for reactors in the 2 sources, 3 reactors case.

	$t = 3$		$t = 4$	
Agent	UOX	MOX	UOX	MOX
`Reactor1`	0.0	1.0	2.0	1.0
`Reactor2`	0.0	1.0	0.0	1.0
`Reactor3`	0.0	0.5	0.0	0.5

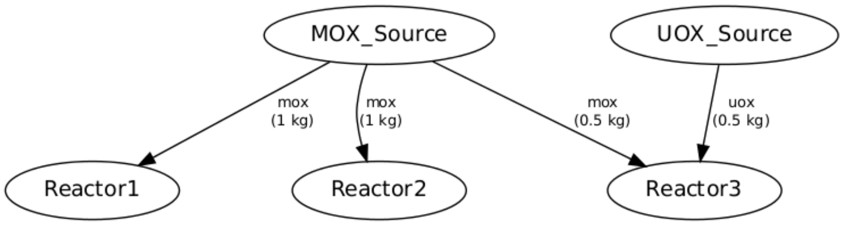

Figure 2.18: Time step 3 for the 2 Sources, 3 Reactors case.

2.5 units. Because of this, the `UOX_Source`, which has been present in the simulation since the beginning, now enters the exchange to make up for the missing 0.5 units of fuel not obtainable from the `MOX_Source`. `Reactor3` is selected to receive the UOX fuel rather than `Reactor1` and `Reactor2`. These preferences are detailed in Table 2.12. In time step, 3 because all agents tie for UOX, `Reactor1` and `Reactor2` tie for MOX, and the `Reactor3` preference for MOX is less than the others, `Reactor3`'s full request for MOX is not met and it must top-up with UOX. This situation is displayed in Figure 2.18.

Finally, on time step 4 the preference of `Reactor1` for UOX changes from 0.0 to 2.0. This alteration causes the tie previously present for UOX to be broken. Furthermore, the value of 2.0 makes this the most preferred arc in the system so it is attempted to be satisfied first. As may be seen in Figure 2.19, the `UOX_Source` capacity of 2.5 units is more than enough to satisfy the request from `Reactor1` for 1 unit of

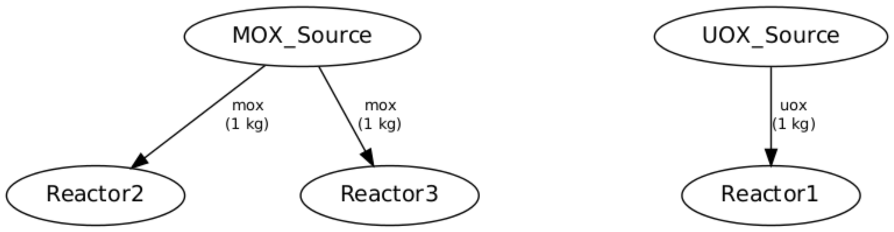

Figure 2.19: Time step 4 for the 2 Sources, 3 Reactors case.

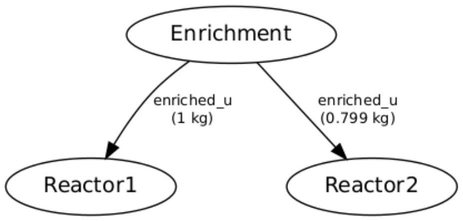

Figure 2.20: Time step 1 for the Enrichment and 2 Reactors case.

UOX. Since `Reactor1` does not diminish the capacity of the `MOX_Source` both `Reactor2` and `Reactor3` are able to obtain their first choice fuel.

This simple 2 source, 3 reactor simulation shows how the resource exchange can dynamically and correctly adapt to the both facility deployments and the preferences that these agents have for requested resources.

Enrichment and 2 Reactors

Initially, `Enrichment`, `Reactor1`, and `Reactor2` are all present. The reactors both begin with cores composed of 3% enriched uranium.

Figure 2.20 shows the result of the resource exchange for time step 1. Here the SWU capacity of `Enrichment` is not sufficient to meet the requests for 5% and 3% enriched fuel simultaneously but is enough to meet either of them individually. Therefore, the bid for at least one of the reactors must be partially unmet. Due to the preferences, the requests of `Reactor1` will be met first. Thus in Figure 2.20 `Reactor1` receives 1 unit of 5% enriched fuel. However, `Reactor2` only receives $\sim$80% of its request for 3% enriched uranium. This is not enough to run on and so this material is saved for the future.

On time step 2, `Reactor1` now switches from requesting 5% enriched fuel to requesting 3% enriched

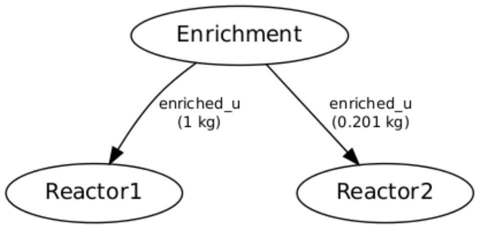

Figure 2.21: Time step 2 for the Enrichment and 2 Reactors case.

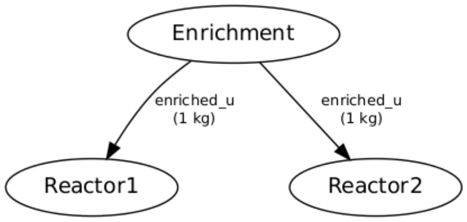

Figure 2.22: Time step 3 for the Enrichment and 2 Reactors case.

fuel. However, the reactor archetypes are implemented such that if they have stored fuel from a previous time step, they will instead only request a mass needed to create 1 unit of fuel. Therefore, `Reactor2` only requests $\sim$0.2 units from `Enrichment`. `Reactor1` continues to request 1 unit of material and this is met because the SWU capacity constraint is not exceeded. These resource flows may be seen in Figure 2.21.

No further adjustments of requests were made on time step 3. Thus the results displayed in Figure 2.22 represent the same dynamic resource exchange procedure in time step 2. The key differences here however are that now the system has returned to a steady state and - unlike in time step 1 - the SWU capacity of `Enrichment` is enough to meet the 2 units of 3% enriched fuel coming from both reactors. Thus `Reactor1` and `Reactor2` each receive the kilogram of fuel that they request. Without further adjustments this system will continue *ad infinitum*.

3 EXPERIMENTATION METHODOLOGY

The NFCTP is the first attempt at solving the supply and demand of the nuclear fuel cycle in a dynamic manner within a NFC simulation. Accordingly, there is no precedent for investigating the performance and efficacy of a given approach. This chapter describes an experimental methodology to assess the NFCTP in which rules for generating instances of exchanges are defined, exchanges are generated, exchanges are executed, and results are analyzed. The goal of this work is twofold: demonstrate the generation of a large number of exchange instances under reasonable assumptions, and analyze the effects and performance of different solvers applied to those exchange instances.

The chapter begins with a discussion of the generation of exchanges, in §3.1. Two species of exchanges are included: one in which reactors are requesting fuel, and one in which reactors are supplying used fuel. In NFC parlance, these are called the *front end* of the fuel cycle and *back end* of the fuel cycle. Notably, both of these exchanges occur in the same time step.

Generating and solving instances of exchanges at a large scale is a difficult problem. The Cyclopts ($\underline{\text{Cycl}}$us $\underline{\text{Opt}}$imization $\underline{\text{S}}$tudies) framework was implemented for this purpose, consisting of both a Python and C++ layer. The Python layer is largely responsible for generating exchanges and interfacing with an associated persistence mechanism. The C++ layer is compiled and linked against the Cyclus kernel shared object library, `libcyclus`, and is responsible for calling directly into the kernel's resource exchange API. §3.2 describes the implementation of Cyclopts and its varied modes of operation.

3.1 Generating Exchanges

Instances of resource exchanges are required to analyze the effects and performance of the NFCTP formulation and its solvers. In the absence of large Cyclus simulations with interesting facility and relationship models, instances must be generated given some set of rules and parameters. Two distinct *species* of exchanges are generated, those related to the *front end* of the nuclear fuel cycle and those related to the *back end* of the nuclear fuel cycle. Broadly, the front end of the fuel cycle is concerned with fueling reactors, and the back end is concerned with either recycling or disposing of used fuel exiting reactors. Common features to both types of exchange generation are described in §3.1.1.

Previously, §2.3, described the methodology for solving a single exchange. If a large exchange is separable, i.e., it can be completely separated into two or more smaller exchanges, sub-exchanges can be solved independently. §3.1.2 provides an argument for why it is valid to split exchange instances into, at minimum, the front and back ends of the NFC.

Exchange generation, absent full-scale simulation, is a naturally parameterized process. Some generation parameters are common to any NFCTP instance, and are described in §3.1.3. Specific species may additionally define their own set of parameters. §3.1.4 describes the parameters generation methodology associated with front-end exchanges. §3.1.5 follows with a similar discussion for back-end exchanges.

3.1.1 Common Features

3.1.1.1 Fuel Cycles and Commodities

Three types of fuel cycles can be generated: a once-through fuel cycle, labeled OT; a plutonium-recycle fuel cycle, labeled MOX; and a plutonium and thorium-recycle fuel cycle, labeled MOX-ThOX. As fuel cycles increase in complexity, the number of commodities that exist increases, as shown in Table 3.1. The commodities are referred to by abbreviation: Uranium Oxide (UOX), Mixed Plutonium Oxide for Thermal Reactors (TMOX), Mixed Plutonium Oxide for Fast Reactors (FMOX), Thorium Oxide for Fast Reactors (FThOX).

Table 3.1: A mapping between fuel cycles to the commodities that exist in each one.

Fuel Cycle	Commodities
OT	UOX
MOX	UOX TMOX FMOX
ThOX	UOX TMOX FMOX FThOX

3.1.1.2 Reactors

All reactors are modeled as either thermal or fast reactors. It is necessary to estimate the amount of fuel exchanged by reactors each time step. Accordingly, thermal reactors are simplified models of AP-1000 reactors [1], and fast reactors are simplified models of BN-600 reactors [20]. Using the dimensions in Table 3.2, one can estimate that the AP-1000 core volume is approximately 12.5 times larger than the BN-600 core.

Table 3.2: Primary Reactor Parameters

Reactor	Core Height (m)	Core Diameter (m)	Number of Assemblies
AP1000	4.27	3.04	157
BN600	0.75	2.05	369

Reactors operate in a batch mode, where each batch is approximately one quarter of the reactor core, an assumption which similar to other analyses [29]. Additionally, a single AP-1000 fuel assembly is assumed to contain 450 kg of material [24]. Therefore, a single batch of thermal reactor fuel is assumed to be

$$450\frac{kg}{assembly} * \frac{t}{1000\ kg} * 157\frac{assemblies}{core} * \frac{1}{4}core = \sim 17.6t. \tag{3.1}$$

The amount of fuel required and number of assemblies by each reactor type is shown in Table 3.3. The number of assemblies is taken as the ratio of total number of assemblies and number of batches per core rounded to the nearest integer. The batch size for the BN600 reactor is estimated by dividing the AP1000 batch size by the relative core volume.

Table 3.3: Reactor Batch Size

Reactor Type	Quantity (t)	Number of Assemblies
AP1000	17.6	39
BN600	1.41	92

The reactors that operate in a given exchange is also a function of the fuel cycle being modeled. In a OT fuel cycle, only thermal reactors exist. In the MOX case, fast reactors that prefer MOX-based fuel are added and denoted as FMOX reactors. Finally, in the MOX-ThOX case, an additional class of fast reactor is added that prefers ThOX-based fuels and is denoted as FThOX. A summary of available types of reactors as a function of the fuel cycle being modeled is shown in Table 3.4, and a summary of the reactor models used for each reactor type is shown in Table 3.5.

Table 3.4: A mapping between fuel cycles to the reactor types that exist in exchange instances.

Fuel Cycle	Reactor Types
OT	Thermal
MOX	Thermal FMOX
ThOX	Thermal FMOX FThOX

Table 3.5: A mapping between surrogate reactor models and reactor types.

Reactor Model	**Reactor Types**
AP1000	Thermal
BN600	FMOX FThOX

Reactors may be fueled by different fuel types, i.e., fuel commodities. The set of commodities that reactors can use is a modeling assumption and a proxy for how reactors may behave in simulations; this particular reactor-to-commodity mapping may not be true for other analysts' fuel cycle models. However, it is appropriate to make certain broad assumptions for such an exploratory study. A mapping of reactors to acceptable commodities is provided in Table 3.6. Note that there is still a preference distribution associated with each reactor-commodity pair as well as constraint coefficient effects. Accordingly, each reactor-commodity pair provides a unique effect on an exchange instance.

Table 3.6: A mapping between reactor types and the commodities allowed to fuel each reactor type.

Reactor Types	**Fuel Commodities**
Thermal	UOX TMOX FMOX
FMOX	UOX TMOX TMOX FThOX
FThOX	UOX TMOX TMOX FThOX

3.1.1.3 Support Facilities

In a front-end exchange, fuel suppliers exchange material with reactors. In a back-end exchange, reprocessing and storage facilities exchange material with reactors. In either case, facilities that are not reactors are referred to as *support* facilities, as they support the reactors which generate power. Support facilities for front-end exchanges are described in §3.1.4.1, and support facilities for back-end exchanges are described in §3.1.5.1.

3.1.1.4 Preferences

Preferences for all transactions have a default value, $p_c(i, j)$, based on the proposed commodity to be transferred between a supplier, i, and consumer, j. However, a large exchange with a small preference distribution results in problem degeneracy. Further, a primary application for Cyclus is the modeling of regional and location effects on fuel cycles. Accordingly, a location proxy is provided for preferences, as shown in Equation 3.2, in order to simulate both location-based preferences and non-degenerate exchange instances.

Preferences can also be a function of facility location. Each facility is assigned a location value, $\text{loc}_i \in [0, 1)$. The domain is then divided evenly into ten regions, where the first region comprises all location values in $[0, 0.1)$, *et cetera*. For example, a facility at location of 4.6 is in the fifth region. $\delta_{\text{reg}}$and $\delta_{\text{loc}}$are binary variables which are activated based on the parameters described in §3.1.3. If $\delta_{\text{reg}}$is zero, no location-based preference is used. If $\delta_{\text{loc}}$is zero, only coarse, region-based preference is used. In both cases, preference is a function of the euclidean distance between regional and location values. The inverse exponential functional form was chosen in order to model a preference gradient that decays as distance increases.

$$p_l(i, j) = \delta_{\text{reg}} \frac{\exp(-|\text{reg}_i - \text{reg}_j|) + \delta_{\text{loc}} \exp(-|\text{loc}_i - \text{loc}_j|)}{1 + \delta_{\text{loc}}} \tag{3.2}$$

The preference for a given arc is then a weighted, linear combination of location and commodity preferences as shown in Equation 3.3. The weighting factor, $r_{l,c}$, is a parameter of exchange generation and described further in §3.1.3.

$$p(i, j) = p_c(i, j) + r_{l,c} p_l(i, j) \tag{3.3}$$

3.1.2 Splitting Exchanges

A well known simplification of the Multicommodity Transportation Problem occurs when supply and demand is separate for separate commodities. The large multicommodity problem can then be decomposed into n single commodity subproblems, where n is the number of commodities. Each subproblem can be solved separately from the others.

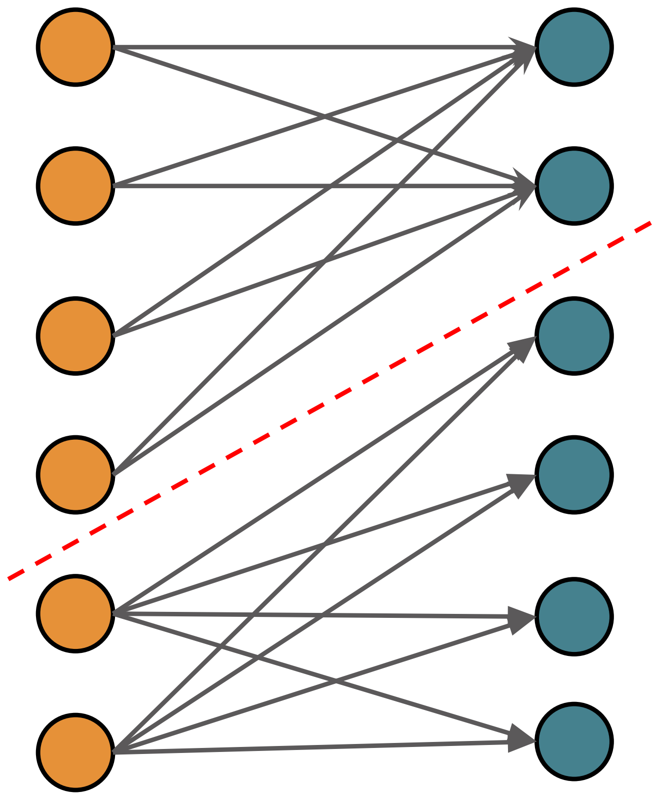

Figure 3.1: A separable bipartite graph with the partition shown as a red dashed line.

An analog exists in the NFCTP when the Exchange Graph is *separable*. A bipartite graph with directed arcs, A, consisting of sending nodes, U, and receiving nodes, V, is separable if there a partition

$$A = A_1 \cup A_2 \tag{3.4}$$

$$U = U_1 \cup U_2 \tag{3.5}$$

$$V = V_1 \cup V_2 \tag{3.6}$$

such that no node in U_1 is connected to a node in V_2 and no node in U_2 is connected to a node in V_1. The graph shown in Figure 3.1 is an example of a separable bipartite graph.

The Exchange Graph of the NFCTP, however, has additional structure in the form of portfolios and

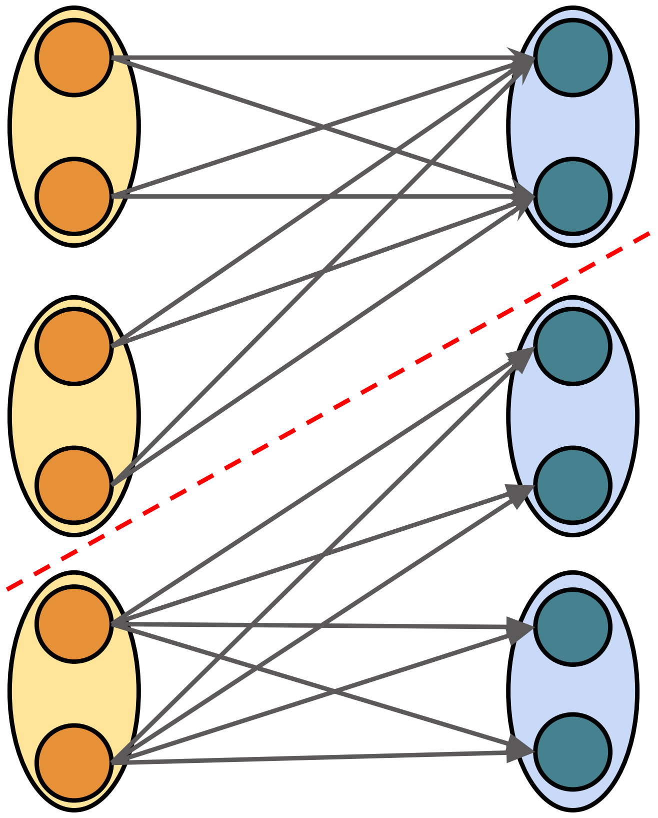

Figure 3.2: A separable Exchange Graph with nodes grouped by portfolio and the separating partition shown as a red dashed line.

thus has a stricter notion of separability. Specifically, the partition must also separate the set of supplier portfolios, S, and requester portfolios, R, as in Equations 3.7 and 3.8, respectively.

$$S = S_1 \cup S_2 \tag{3.7}$$

$$R = R_1 \cup R_2 \tag{3.8}$$

Figure 3.2 depicts a separable Exchange Graph, for example, while Figure 3.3 shows an Exchange Graph where the underlying bipartite graph is separable, but full separability is broken by the overlaid portfolio structure.

The Exchange Graph resulting from the information gathering phase of the DRE will be minimally

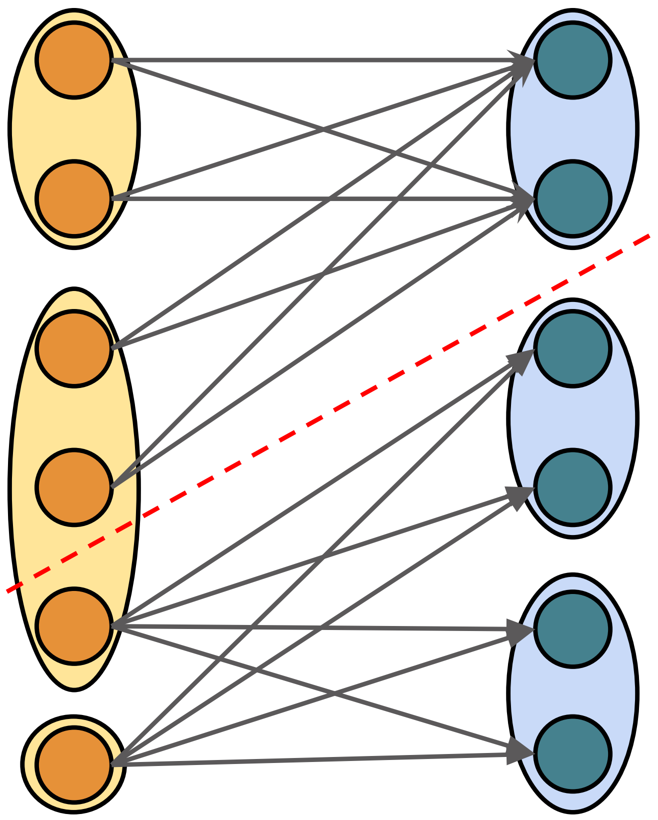

Figure 3.3: An Exchange Graph with nodes grouped by portfolio that is *not* separable because a portfolio crosses the node partition.

separable into front-end and back-end exchanges if two conditions are true:

1. Reactor output commodities can *not* be sent to both other reactors and supporting facilities.

2. Supporting facility output commodities can *not* be sent to both other supporting facilities and reactors.

In the first case, separability is broken by a supplier providing bids across a separating partition. A minimal example is shown in Figure 3.4. This case can arise if reactors can somehow directly refuel other reactors. In the NFC domain, such an arrangement only occurs in an abstraction of a self-recycling system in which there is a dedicated recycling complex associated with a fast reactor. It is reasonable for a self-recycling reactor system to be implemented in such a way that it does not participate in the DRE for self-refueling purposes. Accordingly, this condition is expected to be met in most use cases of the DRE.

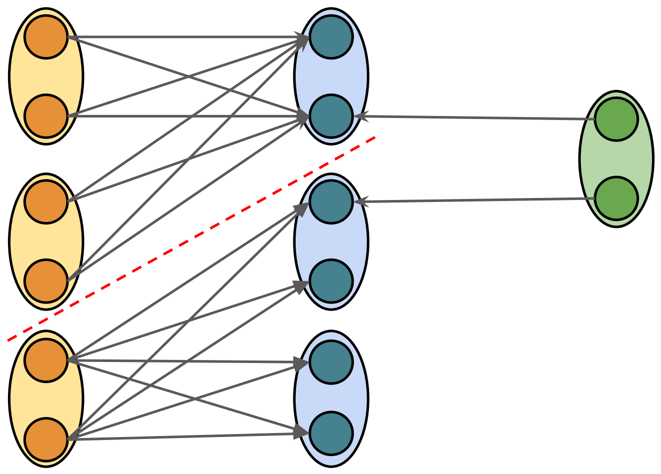

Figure 3.4: An Exchange Graph where separability broken by a supplier. This occurs in NFC modeling if assumption 1 is broken.

In the second case, separability is broken by a requester requesting commodities across a separating partition. Again, a minimal example is shown in Figure 3.5. This case can arise in practice when modeling an NFC system where both a reactor and a repository compete for some commodity. While this is a valid modeling case under certain assumptions and simplifications, it is not very realistic. In general fuel that can be used by a reactor has been processed differently than material to be sent to a repository. If an instance of a DRE does not meet this requirement, it will not be able to be subdivided into smaller instances.

Because the majority of fuel cycles analyzed will meet both conditions, most of DRE instances will be able to be separated into at least two distinct instances which can solved independently of one another. One instance will be associated with the front end of the fuel cycle where reactors are requesting fuel. The other instance will be associated with the back end of the fuel cycle, where reactors are supplying used fuel.

3.1.3 Exchange Parameters

The generation of exchanges is naturally a parameterized process. For instance, a critical parameter is the number of reactors in an exchange. Exchange generation parameters can be divided into two classifications, *fundamental* parameters and *instance* parameters. All exchange species share some fundamental parameters and instance parameters, a discussion of which is the focus of this section. Species also define

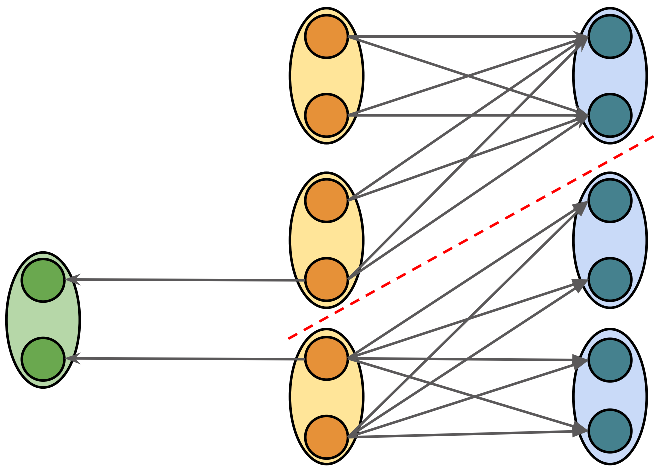

Figure 3.5: An Exchange Graph where separability broken by a requester. This occurs in NFC modeling if assumption 2 is broken.

their own set of instance parameters to complete the full set of parameters needed to define an instance of an exchange.

3.1.3.1 Fundamental Parameters

The fundamental parameters are related to the common features of all instances described in §3.1.1. Each fundamental parameter is a switch that sets the level of *fidelity* of a given exchange. As such, they are each denoted as f_x, where the x subscript describes the parameter.

The most critical parameter is related to the fidelity of the fuel cycle being modeled, f_{fc}. A value of zero indicates modeling the OT fuel cycle, one is used for the MOX fuel cycle, and two the ThOX fuel cycle. The parameter-to-fuel-cycle is summarized in Table 3.7. As fuel cycle fidelity increases, the number of commodities increases, and thus the number of possible connections between suppliers and consumers that exist increases, because some entities trade in multiple commodities.

Table 3.7: A mapping between fuel cycles and f_{fc}values.

Fidelity (Fuel Cycle)	f_{fc}
UOX	0
MOX	1
ThOX	2

The second parameter is reactor fidelity, f_{rx}. Reactors can make requests or provide supply based either on their entire batch or for each assembly in a batch. An $f$rx value of zero indicates reactors trading full batches, and a value of one indicates reactors trading individual assemblies. Trading individual assemblies is of higher fidelity because the number of possible trades, and thus variables in the NFCTP formulation, increases by an order of magnitude. The parameter-to-reactor-fidelity mapping is shown in Table 3.8.

Table 3.8: A mapping between reactor fidelity and f_{rx}values.

Fidelity (Reactor)	$f_{\textbf{rx}}$
Batches	0
Assemblies	1

Finally, the fidelity with with objective value coefficients are generated can be varied. This parameter is denoted f_{loc}because it governs the degree to which location is taken into account in Equation 3.2. The mapping between $f_{\text{loc}}$and parameters in Equation 3.2 is shown in Table 3.9. As f_{loc}increases, the size of the distribution of possible objective coefficient values increases. When $f_{\text{loc}}$is zero, the number of possible objective coefficient values is equal to the product of the number of requester types and the number of commodities. Increasing $f_{\text{loc}}$by one, the total possible values increases by a factor of ten, because there are ten possible regional-preference values. Finally, when $f_{\text{loc}}$is two, the number of possible objective values is uncountably infinite [10].

Table 3.9: f_{loc}Effects on Objective Coefficient Values in Equation 3.2.

Fidelity (Location)	$f_{\textbf{loc}}$	$\delta_{\textbf{reg}}$	$\delta_{\textbf{loc}}$
No region or location data	0	0	0
Region data	1	1	0
Region and location data	2	1	0

3.1.3.2 Instance Parameters

Fundamental parameters represent switches that change the notion of the fidelity of the exchange being generated, for example the difference between a once-through fuel cycle and a fuel cycle with recycling. Instance parameters, on the other hand, change the *shape* and *size* of instances in a given population. In addition to an instance's shape and size, instance parameters can also affect *coefficient generation*. While fundamental parameters are related basic modeling assumptions, instance parameters are related to the specifics of an instance, given those basic modeling assumptions. Both species of exchange instances share some instance parameters, namely those related to the population of reactors in a given exchange and objective coefficient generation.

Reactor Population

Instances are broadly defined by a parameter representing the number of reactors that exist in an exchange instance, n_{rx}. Next, the split between thermal and fast reactors is defined by a parameter defining the ratio of thermal reactors to all reactors in the system, $r_{rx,\text{Th}}$. Assuming $f_{\text{fc}} > 0$, the number of thermal and fast reactors is given by

$$n_{rx,\text{Th}} = r_{rx,\text{Th}} n_{rx}, \tag{3.9}$$

and

$$n_{rx,f} = n_{rx} - n_{rx,\text{Th}}. \tag{3.10}$$

If $f_{\text{fc}}$is zero, a OT fuel cycle is modeled, thus n_{rx} is equal to $n_{rx,th}$ as there are only thermal reactors in the exchange. If a MOX fuel cycle is modeled, the number of FMOX reactors, $n_{rx,\text{FMOX}}$, is trivially equal to the number of fast reactors. However, for a ThOX fuel cycle, i.e., $f_{\text{fc}} > 1$, the number of FMOX and FThOX reactors is determined by a parameter defining the ratio of Thorium-fueled fast reactors to the total population of fast reactors, $r_{rx,\text{FThOX}}$, such that

$$n_{rx,\text{FThOX}} = r_{rx,\text{FThOX}} n_{rx,f} \tag{3.11}$$

and

$$n_{rx,\text{FMOX}} = n_{rx,f} - n_{rx,\text{FThOX}}. \tag{3.12}$$

In the event that the determined number of reactors is non-integral, the value is rounded to the nearest integer, with an imposed minimum value of unity.

Objective Coefficients

As shown in Equation 3.3, the value of an objective coefficient has two components, preference due to a commodity, p_c, and preference due to the relative location between two entities, p_l. It is not obvious to what degree, if any, the relative values of the two components affect formulation performance. Accordingly, a ratio parameter, $r_{l,c}$, is introduced to allow for investigating such effects.

3.1.3.3 Parameter Summary

A summary of species-independent parameters is provided in Table 3.10.

Table 3.10: Parameter Description Summary for Species-Independent Parameters.

Parameter	Type	Description
f_{fc}	Fundamental	The fuel cycle fidelity of an instance (which fuel cycle is being modeled).
f_{rx}	Fundamental	The reactor fidelity of an instance (whether individual assemblies are modeled or whole batches are modeled).
f_{loc}	Fundamental	The location fidelity of an instance (to what degree is facility location included in objective coefficients).
n_{rx}	Instance	The number of reactors in an instance.
$r_{rx,\text{Th}}$	Instance	The ratio of thermal reactors to all reactors in an instance, if appropriate.
$r_{rx,\text{FThOX}}$	Instance	The ratio of ThOX-based fast reactors to all fast reactors, if appropriate.
$r_{l,c}$	Instance	The weight given to location preference with respect to commodity preference.

3.1.4 Front-End Exchanges

A front-end exchange is one in which reactors request fuel and supporting facilities supply fuel resources. Given a specified reactor population, a supporting facility population is determined, as described in §3.1.4.1. Conceptually, the information gathering procedure for this exchange begins with the RFB phase where reactors make requests for commodities with a given quantity and enrichment. Enrichment in this case is a simple resource quality proxy for an isotopic vector. Supporting facilities are then polled to provide a response to these requests during the RRFB phase. Managers of reactors would then adjust preferences based on implemented strategies. The remainder of this section describes how front-end exchange generation models the information gathering procedure, starting with the generation of requests in §3.1.4.2, followed by the generation of supply responses in §3.1.4.3. The PA phase is modeled using the location proxy described in §3.1.1.4. Throughout the discussion on generating front-end exchanges, instance parameters are defined. A summary of all front-end specific instance parameters is described in §3.1.4.4.

3.1.4.1 Support Facility Population

It is assumed that there is a single type of support facility, or supporter, for each type of commodity used in the fuel cycle. Further, each supporter is paired with a reactor type, i.e., there a reactor type which is the *primary consumer* of each supporter type. The primary consumer-supplier relationship is modeled within the formulation by choosing preferences such that there is a maximum preference for the provided relationship (described in the following sections). A summary of these relationships is provided in Table 3.11.

Table 3.11: A mapping between commodities and the supporter type of that commodity.

Commodities	**Supporter**	**Primary Consumer**
UOX	UOX	Thermal
TMOX	TMOX	Thermal
FMOX	FMOX	FMOX
FThOX	FThOX	FThOX

The number of each type of supporter in a front-end exchange instance is a function of of the number of primary consumers as well as configurable parameters. Supporter types are divided into two groups: those who primarily support thermal reactors and those who primarily support fast reactors. The number of thermal fuel supporters is determined to be the product of the number of thermal reactors and a ratio parameter, $r_{s,\text{Th}}$, i.e.,

$$n_{s,\text{Th}} = r_{s,\text{Th}} n_{rx,\text{Th}}. \tag{3.13}$$

The number of TMOX supporters, assuming $f_{\text{fc}} > 0$, is then determined by a parameter defined as the ratio of TMOX to UOX supporters, $r_{s,\text{TMOX,UOX}}$, such that the number of UOX and TMOX supporters is

$$n_{s,\text{UOX}} = \frac{n_{s,\text{Th}}}{1 + r_{s,\text{TMOX,UOX}}} \tag{3.14}$$

and

$$n_{s,\text{TMOX}} = n_{s,\text{Th}} - n_{s,\text{UOX}}. \tag{3.15}$$

The number of fast reactor fuel supporters is determined directly from the number of associated fast reactors in the exchange using ratio parameters, $r_{s,\text{FMOX}}$ and $r_{s,\text{FThOX}}$. Assuming $f_{\text{fc}} > 0$, the number of FMOX supporters is given as

$$n_{s,\text{FMOX}} = r_{s,\text{FMOX}} n_{rx,\text{FMOX}}. \tag{3.16}$$

Similarly, assuming $f_{\text{fc}} > 1$, the number of FThOX supporters is given as

$$n_{s,\text{FThOX}} = r_{s,\text{FThOX}} n_{rx,\text{FThOX}}. \tag{3.17}$$

3.1.4.2 Request Generation

Reactors make *mutual* requests for all commodities that they can consume as described in Table 3.12. Again, a mutual request set is a group of requests of which any single request will meet a given demand. When reactors request a single batch, i.e., when $f_{\text{rx}}$is zero, a single request is made per commodity. When requesting n_a assemblies, a request is made per assembly per commodity, with the number of assemblies denoted previously in Table 3.2. A single request portfolio encompasses all requests, with a portfolio quantity equal to the reactor's batch size.

It is assumed that fuel is requested at some enrichment level dependent on the reactor type. Each reactor in an exchange will choose a batch enrichment level given a uniform distribution. Recycled fuel is modeled as being composed of a target element oxide and topped up with natural uranium oxide; the mixing ratio is again based on reactor type. For recycled fuel, the associated enrichment level describes the enrichment of the fissile isotope in the *target* element. For example, MOX fuel with 45% enrichment implies that of the elemental Plutonium in the mixture, 45% is comprised of isotopic ^{239}Pu . Finally, each reactor has a preference assignment over its set of consumable commodities.

Thermal reactors can consume UOX fuel as well as both MOX variants. It is assumed that thermal reactors would prefer to consume thermal MOX fuel in order to maintain any equilibrium status of the cycle. UOX fuel is next preferred, and finally fast MOX, a fuel type usable by but not meant for thermal reactors. Preference values for each commodity are summarized in Table 3.12. A normal operating enrichment range of $[3.5, 5.5]$ is used for UOX fuel. MOX-based fuels are assumed to be comprised of 7% Plutonium-oxide with 93% Uranium-oxide top up [8] and an enrichment range of $[55, 65]$ [6]. In practice, many reactor concepts restrict the fraction of an LWR's core that can be made up of MOX fuel rather than UOX fuel due to a reduced safety margin. Accordingly, a tuneable parameter is added to the model, f_{mox}, which denotes the fraction of a request that can be made up of MOX-based fuel. This fraction is only relevant if reactors are operating in assembly mode, i.e., if $f_{\text{rx}}$is unity.

MOX and ThOX fast reactors utilize the same governing request parameters but have a different preference distribution over commodities. It is assumed that a Thorium-based fast reactor prefers Thorium-based fast reactor fuel over MOX-based fast reactor fuel and *vice versa*. Additionally, both fast reactor types can utilize thermal MOX fuel or medium-enriched UOX, but prefer fast reactor-based fuels. Preference values for each commodity type and reactor are summarized in Table 3.12. Both fast reactor types select a UOX enrichment in $[15, 20]$[6], with an upper limit set by LEU legal enrichment limits. All recycled fuels commodities are assigned an enrichment range of $[55, 65]$ [6] and have a composition of 20% of the target element (Plutonium or Thorium) and 80% Uranium top up [6].

A summary of chosen chosen request parameters based on reactor and commodity types is shown in Table 3.12.

Table 3.12: A summary of reactor request parameters.

Reactor Type	Commodity	Enrichment Range	Target Element Fraction (%), f_{el}	Commodity Preference, p_c
Thermal	UOX	$[3.5, 5.5]$	100	0.5
	TMOX	$[55, 65]$	7	1
	FMOX	$[55, 65]$	7	0.1
FMOX	UOX	$[15, 20]$	100	0.1
	TMOX	$[55, 65]$	20	0.5
	FMOX	$[55, 65]$	20	1
	FThOX	$[55, 65]$	20	0.25
FThOX	UOX	$[15, 20]$	100	0.1
	TMOX	$[55, 65]$	20	0.25
	FMOX	$[55, 65]$	20	0.5
	FThOX	$[55, 65]$	20	1

3.1.4.3 Supply Generation

With all requests known, each supporting supply facility responds to all requests for their assigned commodity, creating an associated supply node and an arc between the supply node and request node. Constraint coefficients are determined for each arc based on the requested enrichment associated with that arc. Furthermore, a right-hand side (RHS), b_s^k, is provided for each constraint in addition to a coefficient conversion function.

Each supplier has two types of constraints for which coefficients must be calculated: a *process* constraint and an *inventory* constraint. A process constraint models a situation in which the amount of supplied fuel is constrained physically; only so much fuel can be made in one time step. An inventory constraint models a situation in which a supplier is constrained by the available material inventory on hand. Both constraints are a function of requested quantity and fuel enrichment.

UOX Constraints

A UOX supplying facility is assumed to be constrained by a SWU process constraint and a natural Uranium inventory constraint. Assuming general operating parameters, including a tails assay of 0.3 and a feed assay of natural Uranium, 0.711, constraint coefficients can be applied to arcs. The SWU coefficient conversion function is previously described in Equation 2.16 while the natural Uranium conversion function is described in Equation 2.15. Therefore, for UOX supplying facilities,

$$\beta_s^{\text{proc}}(\epsilon) = \beta_s^{\text{SWU}}(\epsilon) \tag{3.18}$$

and

$$\beta_s^{\text{inv}}(\epsilon) = \beta_s^{\text{NU}}(\epsilon). \tag{3.19}$$

In order to determine a constraining RHS, the proposed Eagle Rock Enrichment Plant is chosen as a model. It purports to have a SWU capacity of $3.3E6$ Million SWU per year. Accordingly, the process constraint RHS is chosen to be an approximate monthly value,

$$b_s^{\text{SWU}} =\sim 2.75E5 \frac{\text{SWU}}{\text{month}}. \tag{3.20}$$

Any inventory constraint will be based on the present state of a facility at a given simulation time step. Therefore, a sufficiently reasonable value must be provided without actual simulation data. Because two constraints are added, investigating their relative effects is of interest, which leads to a strategy for generating an inventory constraining value by deriving it from the process constraining value. In order to make such comparisons, the two RHS values must be equivalent in both units and with respect to the expected coefficient values associated with each constraint. Accordingly, a translation constant is defined to achieve both aims. The translation , τ_s, constant is taken to be a ratio of constraint coefficients for the average enrichment of a support facility's primary consumer, i.e.,

$$\tau_s = \frac{\beta_s^{\text{inv}}(\bar{\epsilon_r})}{\beta_s^{\text{proc}}(\bar{\epsilon_r})}. \tag{3.21}$$

Thus, a UOX supporting facility uses a average enrichment, $\bar{\epsilon_r}$, of 4.5 because that is the median of the thermal reactor enrichment range. Further, a ratio coefficient parameter, $r_{inv,proc}$, is added in order to investigate interesting cases from a formulation point of view. If $r_{inv,proc} > 1$, then the process constraint RHS is smaller and thus the process constraint is more likely to be engaged in an feasible solution than the inventory constraint. On the other hand, if $r_{inv,proc} < 1$, the inventory constraint is more likely to be engaged. The determination of the inventory RHS is identical for all supporting facilities and is defined in Equation 3.22.

$$b_s^{\text{inv}} = r_{inv,proc} \tau_s b_s^{\text{proc}}. \tag{3.22}$$

Recycled Commodity Constraints

Due to the lack of commercially viable, well documented fast reactor fuel suppliers, a simple linear surrogate model is assumed for an inventory constraint. The primary inventory of any recycling facility is the amount of fissile material it has on hand. Therefore, using constants defined in Table 3.12, the coefficient function conversion function is chosen to be

$$\beta_s^{\text{inv}}(\epsilon) = f_{el}\epsilon. \tag{3.23}$$

There are many possible process constraints that could be used, such as heat production or radiotoxicity; however, each of these requires a detailed isotopic composition to be relevant. Accordingly, a commodity-informed mass throughput constraint is used. Per the current IAEA practice [19], and extrapolating the same effect for reprocessing ^{233}U , a factor of 100 is added for for Plutonium and Thorium-based commodities. The process constraint coefficient function is defined as

$$\beta_s^{\text{proc}} = 100 f_{el}. \tag{3.24}$$

From previous conversations with industry representatives [27], a reasonable size for a processing plant is 800 tonnes per year, which is similar to the Rokkasho plant in Japan [19]. Given the request parameters defined in Table 3.12, an 800 t Uranium / 8 t Plutonium facility could service on the order of 2-3 fast reactors or ~2 thermal reactors with $\frac{1}{3}$ a request as MOX. The yearly process limit is again translated to a monthly limit, resulting in a constraint RHS value of

$$b_s^{\text{proc}} =\sim 66.7 \frac{\text{t}}{\text{month}}. \tag{3.25}$$

The inventory constraint RHS is determined identically to the UOX case.

3.1.4.4 Parameter Summary

A summary of front-end exchange species-dependent instance parameters is provided in Table 3.13.

Table 3.13: Parameter Description Summary for Front-End Exchange Instance Parameters.

Parameter	Description
f_{mox}	The fraction of thermal reactor requests that can be met with mox fuel.
$r_{s,\text{Th}}$	The ratio of thermal support facilities to thermal reactors.
$r_{s,\text{TMOX,UOX}}$	The ratio of TMOX to UOX support facilities.
$r_{s,\text{FMOX}}$	The ratio of FMOX support facilities to FMOX reactors.
$r_{s,\text{FThOX}}$	The ratio of FThOX support facilities to FThOX reactors.
$r_{inv,proc}$	The ratio of the inventory RHS to the process RHS.

3.1.5 Back-End Exchanges

A back-end exchange models the transfer of used fuel from reactors to supporting facilities, such as reprocessing facilities and repositories. During the information gathering process, supporting facilities make requests for commodities that can either be used directly in the recycling process or need to be stored, temporarily or permanently. Reactors then respond based on output fuel to each request during the RRFB phase. Throughout the discussion on generating back-end exchanges, instance parameters are defined. A summary of all back-end specific instance parameters is shown in Table 3.15.

3.1.5.1 Support Facility Population

Four classes of supporting facilities are modeled in back-end exchanges: a thermal fuel recycling facility, a facility that recycles fast MOX fuel, a facility that recycles fast ThOX fuel, and a repository. As thermal fuel recycling facilities are the only thermal supporting facilities, the number of such facilities in a given back-end exchange is trivially $n_{s,\text{Th}}$. As is the case with front-end exchanges, there is a class of supporting facility for each fast fuel commodity. The methodology for determining the population of each facility type is identical to front-end exchanges:

$$n_{s,\text{FMOX}} = r_{s,\text{FMOX}} n_{rx,\text{FMOX}} \tag{3.26}$$

and

$$n_{s,\text{FThOX}} = r_{s,\text{FThOX}} n_{rx,\text{FThOX}}. \tag{3.27}$$

Back-end exchanges include repositories, a facility type not present in front-end exchanges. A simple ratio parameter, r_{repo} is applied based on the total number of other supporting facilities, i.e.,

$$n_{s,\text{repo}} = r_{s,\text{repo}}(n_{s,\text{Th}} + n_{s,\text{FMOX}} + n_{s,\text{FThOX}}). \tag{3.28}$$

3.1.5.2 Request Generation

It is assumed that any recycling facility will accept UOX fuel. However, MOX recycling facilities can not process ThOX-based fuels, and ThOX facilities can not process MOX-based fuels. Additionally, fast MOX facilities prefer fast MOX fuel, while thermal facilities prefer thermal MOX fuel. Finally, repositories can accept all commodities; however, it is a consumer of last resort. The assigned preference value as a function of commodity type and supporting facility type, p_c is shown in Table 3.14.

Table 3.14: p_c Value Mapping between Back-End Supporting Facilities and Commodities.

Supporting Facility \ Commodity	UOX	TMOX	FMOX	FThOX
TMOX	1	1	0.5	N/A
FMOX	0.5	1	1	N/A
FThOX	0.3	N/A	N/A	1
Repo	0.1	0.1	0.1	0.1

A single request for the facility's processing capacity is made for each commodity. Recycling facilities define their request quantity using the same 800 ton per year limit discussed in §3.1.4. Repositories, however, use a limit based on the Yucca Mountain statutory limit of 17,000 tons and assuming a 30-year operating lifetime, i.e., period of time in which fuel can enter the facility. Thus, a repository's monthly request quantity is determined to be ~ 48t.

A fissile quantity constraint is added for each recycling facility (*not* added for reprocessing facilities). The fissile constraint models a situation in which recycling facilities have a demand for fissile material. The amount of fissile material required by recycling facilities is based on their primary consumer. It is determined to be the product of the facility's mass constraint and the mean amount of fissile material in a primary consumer's request per unit mass, as shown in Equation 3.29. This constraint can be considered as "recycling facilities request fissile material quantities as if its capacity was completely met by average primary reactors".

$$b_r^{\text{fiss}} = \bar{\epsilon} f_{el} b_r^{\text{mass}}. \tag{3.29}$$

The fissile constraint coefficient is simply the amount of fissile material for a given supply, as described in Equation 3.30.

$$\beta_r^{\text{fiss}}(\epsilon) = \epsilon f_{el}. \tag{3.30}$$

3.1.5.3 Supply Generation

A key difference between the front-end and back-end exchanges is that in front-end exchanges, reactors request fuel, and thus can make a single request per commodity per assembly. In back-end exchanges, commodities must be assigned to each assembly. Accordingly, a key parameter in back-end exchanges is the commodity distribution for assemblies. A normalized uniform distribution parameter is provided for each reactor type with a value for each commodity type that reactor can consume as defined in Equation 3.31.

$$
\begin{aligned}
d_{\text{Th}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}],\ x_i \in [0,1) \\
d_{\text{FMOX}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}, x_{\text{FThOX}}],\ x_i \in [0,1) \\
d_{\text{FThOX}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}, x_{\text{FThOX}}],\ x_i \in [0,1)
\end{aligned}
\tag{3.31}
$$

If an exchange is in batch mode, i.e., $f_{\text{rx}}$is zero, then this distribution acts as a selection distribution, where each x_i represents a probability that the batch will be of that commodity. If in assembly mode, then commodities are assigned to each assembly given the relative x_i values. The assignment of commodities to number of assemblies for a given reactor type is done by rounding the product of x_i and the total number of assemblies, starting with the lowest value of x_i. The final assignment is then taken as the difference between the total number of assemblies and the previously assigned values.

For example, consider a thermal reactor with a distribution $d_{\text{Th}} = [\frac{2}{3}, \frac{1}{3}, 0]$ and number of assemblies $n_a = 39$. The assembly-commodity breakdown would be calculated as

$$
\begin{aligned}
n_{\text{FMOX}} &= \text{round}(x_{\text{FMOX}} n_a) = 0 \\
n_{\text{TMOX}} &= \text{round}(x_{\text{TMOX}} n_a) = 13 \\
n_{\text{UOX}} &= n_a - n_{\text{TMOX}} - n_{\text{FMOX}} = 26.
\end{aligned}
$$

Once a commodity is assigned either to a single batch or a selection of assemblies, the remaining supply generation methodology is identical. If $f_{\text{rx}}$is zero, the following discussion uses the term assembly to mean either an individual assembly or a batch. That is, a reactor in a back-end exchange has a single assembly to supply. If $f_{\text{rx}}$is one, then it has n_a assemblies to supply, where n_a is defined in Table 3.3 for each reactor type.

In order to assign enrichment values to each assembly, a single random value is chosen, $x \in [0,1)$. Each assembly is then assigned an enrichment based on the assembly's commodity and enrichment range, as defined in Table 3.12. This modeling assumption supports a situation in which, for a given batch, equivalent fissile enrichments were used across commodities. For example, consider a Thermal reactor with x chosen to be 0.55. All UOX assemblies would be assigned an enrichment value of 4.6, and each

MOX-based assembly would be assigned an enrichment value of 60.5.

A bid portfolio is assigned to each assembly. Given the commodity of each assembly, a supply response is provided to each supporting facility that requests that commodity. For example, given a UOX assembly, a reply is sent to each supporting facility, as all supporting facilities accept UOX, shown in Table 3.14. The set of supply responses associated with a single assembly is denoted a *mutual* set. That is, each supply node corresponds to a single assembly that should not be split between supporting facilities.

3.1.5.4 Parameter Summary

A summary of back-end exchange species-dependent instance parameters is provided in Table 3.15.

Table 3.15: Parameter Description Summary for Back-End Exchange Instance Parameters.

Parameter	Description
d_{Th}	thermal reactor assembly distribution
d_{FMOX}	fast mox reactor assembly distribution
d_{FThOX}	fast thox reactor assembly distribution
r_{repo}	repository to supporting facility ratio

3.2 Experimental Tools

In order to explore the large number of possible exchange instances described in §3.1, a sophisticated instance generation and solving framework is needed. This section describes the design principles and implementation details of a new software package called Cyclopts ($\underline{\text{Cycl}}$us $\underline{\text{Opt}}$imization $\underline{\text{S}}$tudies). Cyclopts, written primarily in Python with a C++ layer used to interface with Cyclus, provides a general framework for sampling a parameter space, defining problem instances for a given point in parameter space, and solving a problem instance under a variety of conditions. While this section focuses on the Cyclopts workflow, implementation, and high-throughput computing (HTC) capabilities, details specific to the database layout and command line interface (CLI) are treated lightly. A full treatment of the the database layout is provided in Appendix A, and the CLI is detailed in Appendix B.

3.2.1 Terminology

Cyclopts supports a two-tier definition of problem instances, borrowing terms from biological classification. Problem *families* describe a general form of problem instance. For example, the Traveling Salesman

Problem (TSP) could be implemented as a problem family. In this analysis, the NFCTP is considered the problem family, since any given instance of the NFCTP will have the same general structure. Whether or not the LP or MILP formulation is used is dependent on whether or not arcs in the Exchange Graph are labeled as exclusive or not. If there are no exclusive arcs, the LP formulation is used; otherwise, the MILP formulation is used.

Each problem family can have any number of *species*. One can conceptualize the relationship as a tree structure, in which families are parent nodes and species are child nodes. A problem species defines the methodology for generating *instances* of a problem family. Using the TSP example above, a problem species may be "the greater Atlanta metropolitan area", for which the effect of regional gas prices may be studied. For the NFCTP study, front-end and back-end exchanges form two separate species. Each species can have unique parameters in addition to family-related parameters, as is the case for the two species studied.

3.2.2 Design

The full Cyclopts stack is comprised of three phases: generation of parameter space, generation of instances, and execution of instances. The workflow begins with user input detailing a range of values for a set of parameters. Cyclopts then translates the input into a parameter space by enumerating all possible combinations of parameters. For example, if parameters x and y have defined values of $[1, 2]$ and $[3, 4, 5]$, respectively, Cyclopts will generate a parameter space comprised of six points in (x, y) notation: $(1, 3)$, $(1, 4)$, $(1, 5)$, $(2, 3)$, $(2, 4)$, and $(2, 5)$. Each point is then then provided to a problem species in order to generate one or more problem instances. Species are expected to define defaults for all parameters as user input may define values for only a subset of available parameters.

Given a point in parameter space, an instance can be generated. If there are any stochastic effects during instance generation, many instances may be generated. Again, because parameters are species dependent, the logic of instance generation from a set of parameters is the task of a problem species. Following instance generation, instances can be executed. Cyclopts supports multiple solution options by design. The same instance may be solved with both a heuristic and a full optimization solver, for example. Once an instance of a problem is defined, it is independent of any species-level effects. Accordingly, instance execution and related logic is the domain of problem families.

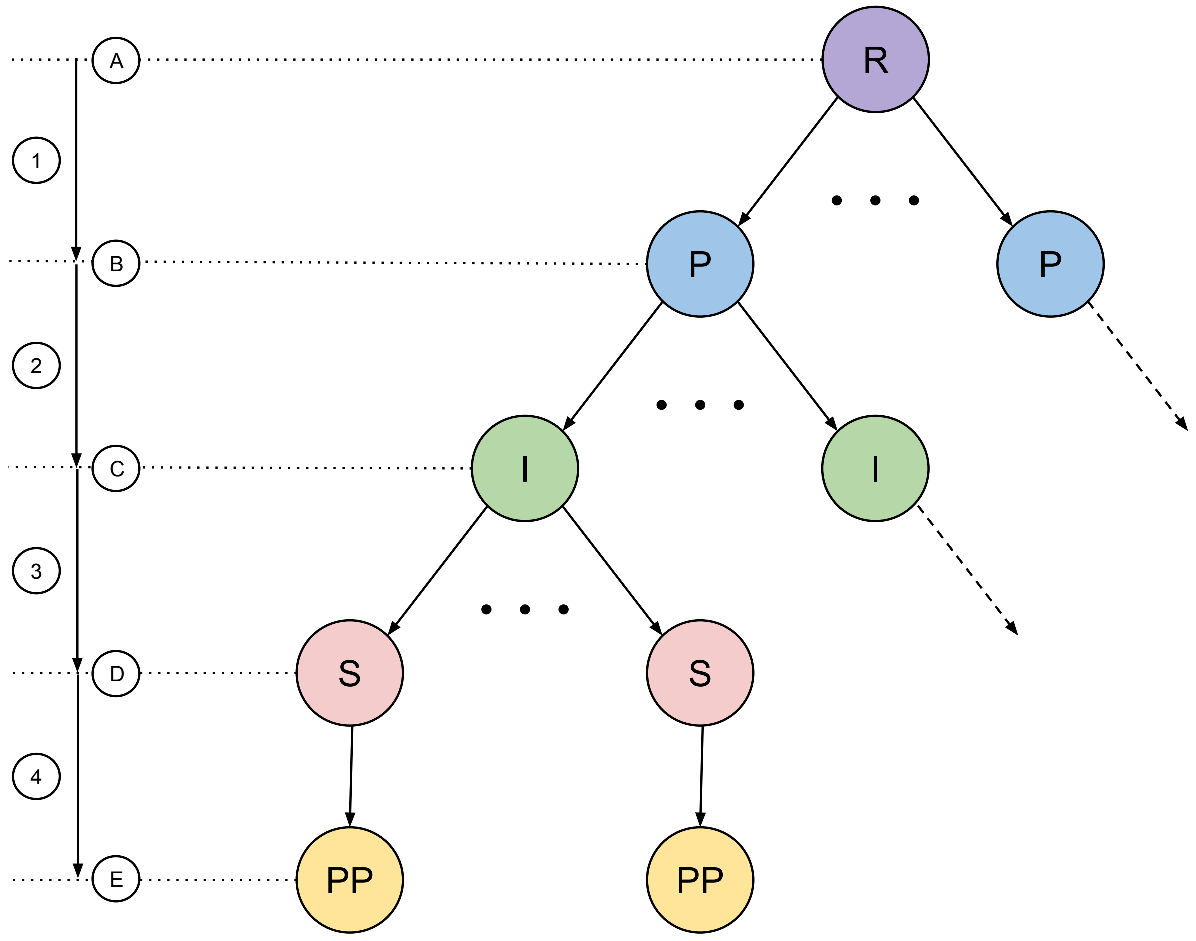

Figure 3.6: A graphical representation of the Cyclopts object tree-structure. For any parent node, there is a one-to-many mapping of children nodes. The node types in the tree structure are defined in Table 3.16. The actions associated with moving between each level are explained in Table 3.17.

A summary of the high-level Cyclopts workflow and entities is presented in Figure 3.6. Note that objects generated as the workflow moves from parameter space to instance solution form a tree structure.

Table 3.16: Cyclopts object tree structure node types as shown in Figure 3.6.

Label	**Node Type**	**Description**
A	Root	A definition of the full parameter space as provided by a user.
B	Parameter	A fully defined point in the parameter space.
C	Instance	A fully defined instance of a given problem.
D	Solution	A solution to an instance of a problem determined by an appropriate solver.
E	Post-process	Post-processed information, given all parent nodes.

Table 3.17: Cyclopts actions generating child nodes as shown in Figure 3.6. The Cyclopts entity, e.g., the family or species, associated with each action is also listed.

Label	**Action**	**Entity Responsible**
1	Translate a parameter space into all possible points.	Cyclopts Core
2	Convert a parameter point into a number of problem instances.	Problem Species
3	Execute a problem instance given a solver and record the solution.	Problem Family
4	Post-process a solution and instance, recording relevant information.	Problem Family & Species

3.2.3 Persistence Mechanisms

While the root node in Figure 3.6 is generated from a user-provided input file, each subsequent level in the hierarchy represents a stateful object: a point in parameter space, a problem instance, and a solution. Each stateful object can be written to and read from disc. Cyclopts also incorporates a post-processing step, during which all related objects may be analyzed and aggregate data may be collected and written to disc. While any input/output (I/O) persistence mechanism is valid, Cyclopts is currently implemented using the Hierarchical Data Format (HDF5) [33] via PyTables [5].

Data in HDF5 is stored hierarchically, similar to a file system. At the root node of the file-system-like structure, a *group* is defined for problem family and problem species data, named `Family` and `Species`, respectively. A *dataset* for aggregate results named `Results` is also defined. A path in HDF5 is designated in a UNIX-like manner. For example, the path to `Family` would be `/Family`, indicating that the group is directly under the root node, `/`. Further, groups are defined for each kind of family and species. The DRE problem family records data in the group `/Family/ResourceExchange`, front-end exchanges record data in the group `/Species/StructuredRequest`, and back-end exchanges record data in the group `/Species/StructuredSupply`.

Each stateful object is given a Universally Unique Identifier (UUID) by which it can be identified for future reading and analysis. The UUID is used in two distinct capacities: as a *primary key* in a dataset for future identification or as the name of a group. Whether to aggregate data in one large dataset or divide data into datasets for each object is a design decision informed by practical performance. A study of the trade-offs between each approach is presented in §3.2.3.1. As a result of that study, for objects that are both read and written, the latter approach is taken.

A description of all data gathered for each family and species for conversion, execution, and post-processing is detailed in Appendix A.

3.2.3.1 Performance Studies

Chunk size is a critical parameter of HDF5 datasets that affects I/O performance. HDF5's storage layout is not contiguous; rather, data is separated into equal-sized *chunks*. Any reading or writing occurs on a chunk of data, rather than accessing an entire dataset. Accordingly, choosing a reasonable chunk size can greatly increase performance for known data access operations. In PyTables, the *compression level* of a dataset is also a tune-able parameter that affects I/O performance. Compression, of course, reduces overall database size. Therefore, an ideal compression is the largest possible that retains acceptable performance.

Originally, all Cyclopts datasets used a UUID-as-primary-key layout. For instance, rather having a tables with a layout described in Table A.6, a single table with an extra column naming the instance UUID was used. However, extremely long read times were encountered when post processing data. The basic procedure for performing a post-process operation included reading all rows associated with a UUID in an exchange species dataset, reading all rows associated with the same UUID in an exchange family dataset, selecting a value from each row (resulting in two vectors), and performing a dot product operation.

In order to investigate possible chunk size and compression optimizations, a small ($\sim$ MBs) dataset and a large ($\sim$ GBs) dataset were created. The post-processing step was then run on 25 instances in each dataset. The operation was timed using the UNIX `time` command. An initial chunksize for each dataset was chosen to be proportional to the ratio of a normal L2 cache to row size and a compression level of four was selected per suggestions from the PyTables documentation [2]. For the performance study, chunk size and compression level were varied around these recommended values in order to determine if any tuning was available. The results of the study on the small dataset is shown in Figure 3.7. The large dataset results is shown in Figure 3.8.

Assuming some level of compression, an ideal chunksize range is identified for the small database of between $\sim 10^3 - 10^5$ bytes. Further the small database example confirms that the study's methodology is well founded: an ideal chunksize range is established. A similar optimal chunk size range is found for the large database. However, note that in this exercise, only $\sim 0.25\%$ of instances are post-processed. An optimal performance of > 80 seconds per instance is unacceptable.

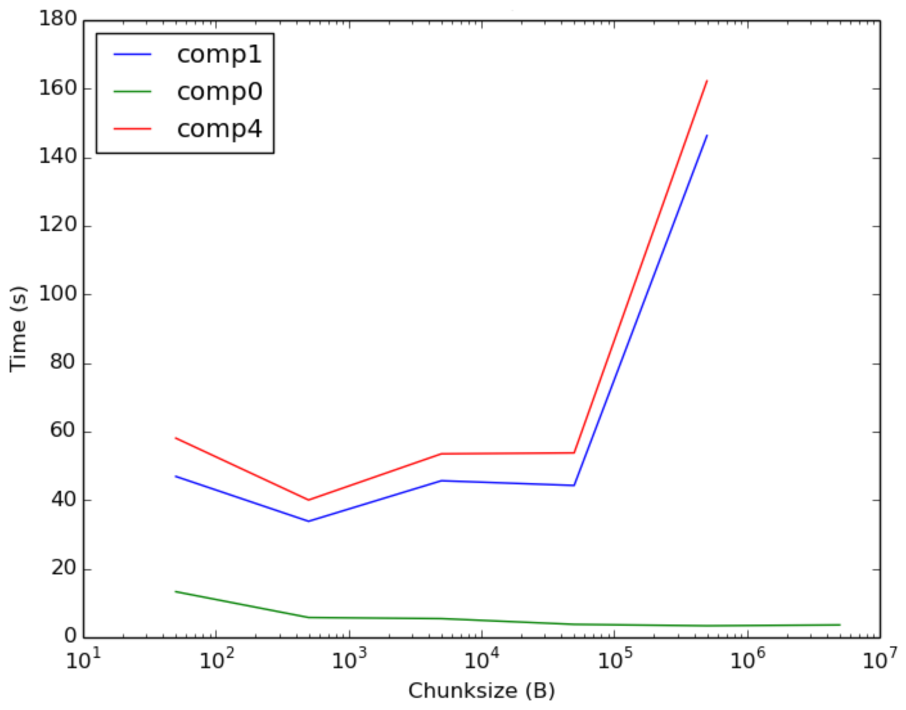

Figure 3.7: Post-processing performance for 25 entries of a small-sized database for a variety of compression levels and chunk sizes.

A number of strategies exist for trying to increase performance. A classic strategy is pivoting the group-dataset structure such that data queries are made upon an entire group rather than rows in a dataset. In this example, such a pivot involves dividing the single, large dataset into n datasets, where n is the number of unique primary keys, i.e., UUIDs.

Accordingly, an additional performance test was conducted with a new database layout. All datasets on which queries are made were pivoted such that new group nodes were added for each UUID, and all data for that UUID was appended to a dataset under the associated group. The post processing step was divided into the read and vector-population operations associated with the exchange family and the the read and vector-population operations associated with a species. The exact same operations were applied to a large database with the column-based layout and a large database with the group-based layout. Specific instances, increasing in size, were identified to be post-processed. The group-based results were compared with the column-based results and are shown in Figure 3.9. The speed of each operation was

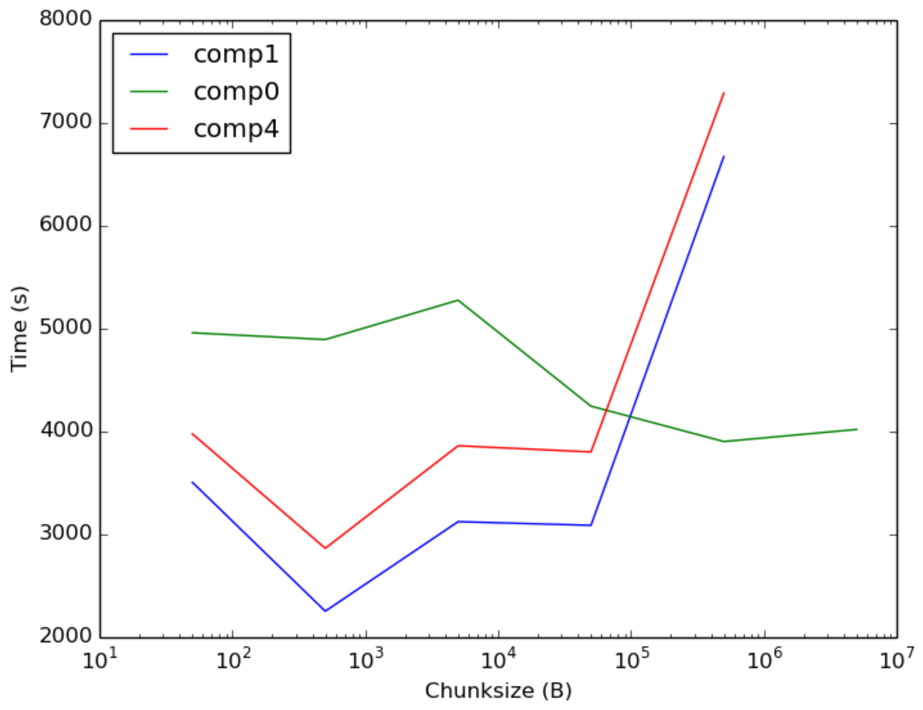

Figure 3.8: Post-processing performance for 25 entries of a large-sized database for a variety of compression levels and chunk sizes.

compared directly for both layout strategies. The ratio of the group-strategy running time to the column strategy running time was then plotted. Therefore, a low ratio implies a large time savings, and a ratio close to unity implies almost no time savings. Times were calculated using the IPython magic `%timeit` command.

As can be seen, the group-based strategy performs quite well, over an order of magnitude better than the column-based strategy for species operations. Furthermore, species operations are shown to have a much larger speedup relative to family operations. This artifact is due to the fact that at the time of this analysis, solution values were stored *only if* they were nonzero. When read, a data structure must be allocated and populated for each non-zero index rather than simply copying a block on data on disc. The writing of family-based solution values has since been updated to also write zero values to avoid this issue.

For the purposes of this study, the dataset-group pivot served the required purpose. Post-processing

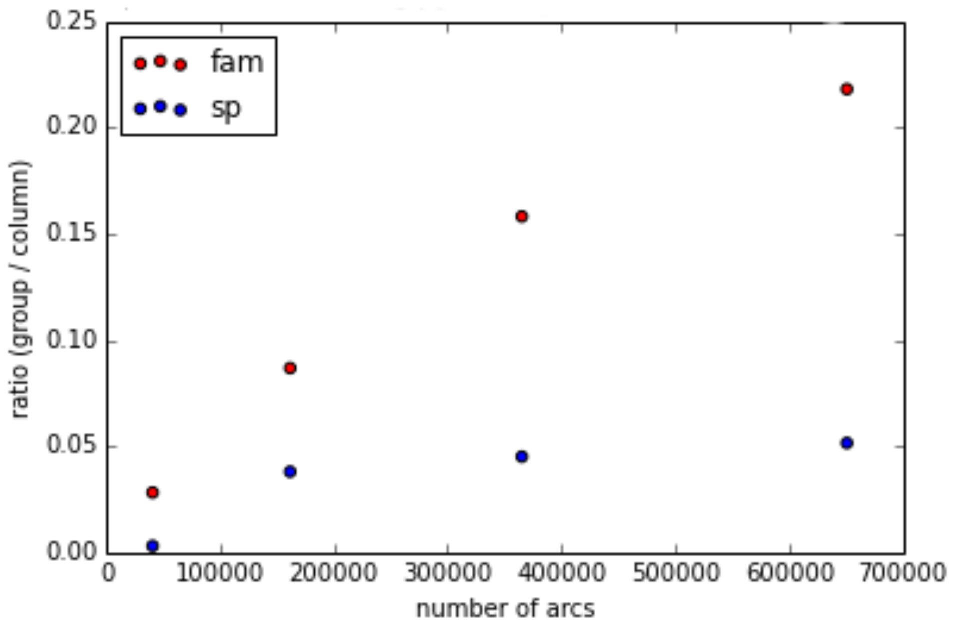

Figure 3.9: The ratio of group-based queries to column-based queries as a function of problem size. A lower ratio indicates a faster process time for the group strategy over the column strategy.

now performs satisfactorily for the operations needed and the database sizes experienced. However, if future performance issues arise, other strategies may be investigated. Perhaps the most fruitful of these will be returning to the single dataset layout and using PyTable's indexing feature.

3.2.4 Implementation

Cyclopts defines abstract application programming interfaces (APIs) for both families and species in the `ProblemFamily` and `ProblemSpecies` classes, respectively. While many parts of an API are related to the workflow discussed in §3.2.2, others are related to the persistence mechanisms discussed in §3.2.3. The full API is described in detail in the Cyclopts documentation [14]. The `ExchangeFamily` class implements a concrete, NFCTP-specific `ProblemFamily` interface. The `StructuredRequest` class implements a concrete, front-end-exchange interface of the `ProblemSpecies` class. Similarly, the `StructuredSupply` implements a back-end interface to the class.

Given a point in parameter space, both the `StructuredSupply` and `StructuredRequest` generate an instance of an `ExchangeGraph` per the rules described in §3.1. Cyclopts is nominally written in Python and

Cyclus is written in C++. In order to construct objects that can interact with Cyclus, an interoperability layer is required.

A series of C++ wrapper objects, namely arc, node, and group objects, are defined which mirror the constituents of an `ExchangeGraph`, as described in §2.3.4. These objects are then translated into Cython [7] by use of the XDress software package [30]. Python can directly call into Cython libraries, similarly, Cython can directly call C and C++ libraries. Hence, an interoperability layer is established.

3.2.4.1 Solvers and Performance Timing

Once an instance of an `ExchangeGraph` has been generated, it can be solved. Cyclopts supports three types of solvers: CoinCLP, CoinCBC, and the `GreedySolver`, an implementation of the Greedy Heuristic in Cyclus. If either the Greedy or CBC solvers are invoked, an appropriate instance of a `ExchangeSolver` is constructed with the *exclusive orders* flag turned on. If the CLP solve is invoked, an associated `ExchangeSolver` instance is constructed with the *exclusive orders* flag turned off. In short, CBC and Greedy solvers solve the MILP formulation of the NFCTP, and the CLP solver solves the LP formulation.

Given an instance of an `ExchangeGraph` and `ExchangeSolver`, the `Solve` method of the `ExchangeSolver` is invoked. Before and after the `Solve` function call, the `CoinCpuTime()` function is called and the result is stored. The difference between the two resulting values is recorded as the time required to determine a solution. The implementation of the `CoinCpuTime()` function is open and easily available [12]. It simply adds the seconds and microseconds fields of the `ru_utime` structure populated by the standard UNIX `getrusage()` function.

3.2.5 High Throughput Computing

Cyclopts can be executed locally using the `cyclopts exec` command-line interface (CLI) described in Appendix B. When exploring a large parameter space, of which each point can generate a large number of unique instances, local execution on a single machine is insufficient. In order to overcome this limitation, support for Condor-based systems has been implemented in Cyclopts and available using the `cyclopts`↩
`condor-submit` CLI. Condor [32] is a high throughput computing (HTC) framework that supports sophisticated job scheduling over a very large, distributed network of individual and clustered computers.

HTC systems are ideal for analyses in which many independent executions must be performed. Upon completion, the results may be aggregated and analyzed. The resource-exchange use case fits such a design specification with a single caveat: because it is a first-of-a-kind performance analysis, timing results are crucial. Therefore, the systems on which instances are executed must be equivalent in order to compare different timing results. Support is provided in Cyclopts for identifying execute nodes that conform to a series of architecture and related constraints in order to support this analysis limitation.

3.2.5.1 Remote Execution and Operation

In order to efficiently schedule a large number of optimization problems, the WorkQueue framework [9] is utilized. WorkQueue is a Condor-aware master-worker implementation. A master process exists at some location and manages the scheduling jobs to be run. Workers, in the form of persistent Condor jobs, ask the master for the next job to be run after a previous job has been completed. A master-worker system is especially useful in Condor environments in which resources are limited and must be specifically targeted, as is the case with the aforementioned timing studies.

Cyclopts launches a master process that requires a series of execute nodes to target, a problem instance database, and a list of solvers to execute on each instance. A copy of the instance database is sent to every targeted execute node. Note that an execute node may have many execution threads, each of which can be used to execute instances individually. The master manages an instance queue from which jobs are provided to workers. Upon completion, a worker will request a new instance of the master. The full set of instances in a database are hence efficiently executed.

3.2.5.2 Packaging and Environment

A common issue in remote execution environments is package dependency. When access to an execution node is provided, a user must assume that only the barest of environments exist. For example, if a user is provided an Ubuntu-based execution node, the user generally must assume that it is a fresh Ubuntu installation. Accordingly, package management in a highly distributed, heterogeneous environment is a difficult problem.

Luckily, solutions exist for distributed package management. Cyclopts utilizes the Code, Data, and Environment (CDE) [18] tool to manage its execution environment. CDE provides a virtualized environment based on the local execution of a command. Using CDE with the given command, all libraries and utilities used during the process execution are monitored. Upon process exit, every object in the filesystem that was invoked is copied into a virtual environment. That virtual environment can then be packaged and distributed. Upon landing on a foreign system, a user can enter the CDE environment and execute the supported command.

Cyclopts provides a CLI, `cyclopts cde`, that will package Cyclopts itself into a virtual environment and ship the environment to a Condor submit node. As part of the Cyclopts Condor job execution, the CDE environment is copied to each execute node. It is therefore easy to incorporate changes in a local copy of Cyclopts to the corresponding remote execution.

4 EXPERIMENTS & RESULTS

4.1 Experimental Setup

Talk about specifics of computers used, how results are calculated, etc. Expand on specifics from prev chapter.

Specifically, the solution time required to solve exchange instances is analyzed because this study is of both a novel solution methodology to a novel problem, and because the problem is by definition NP-Hard as described in §1.3.3.

The solvers used to determine solutions to the DRE instances are then described in §**??**. Three solvers are supported: COIN-CLP, COIN-CBC, and the Greedy Hueristic described in §2.3.3.6.

4.1.1 Solvers

4.2 Experiments Executed

4.3 Summary & Observations

5 SUMMARY

5.1 Statement of Work

5.2 Recommendations

5.3 Suggested Future Work

A CYCLOPTS HDF5 DATABASE LAYOUT

This appendix details the exact database layout used by Cyclopts for the `ExchangeFamily`, `StructuredRequest` species, and `StructuredSupply` species.

A.1 Parameter Space

Both front-end and back-end species record the state of every point in a given parameter space in a data set called `/Species/<species type>/Points`, where `<species type>` is either `StructuredRequest` or `StructuredSupply`. Each point incorporates both fundamental and instance parameters as described in §3.1. The tables associated with parameter spaces are described in Tables A.1-A.2.

Table A.1: Data-type description of the `/Species/StructuredRequest/Points` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
f_fc	1-byte integer	As described in §3.1
f_loc	1-byte integer	As described in §3.1
f_mox	4-byte float	As described in §3.1
f_rxtr	1-byte integer	As described in §3.1
n_reg	4-byte unsigned integer	As described in §3.1
n_rxtr	4-byte unsigned integer	As described in §3.1
r_inv_proc	4-byte float	As described in §3.1
r_l_c	4-byte float	As described in §3.1
r_s_mox	4-byte float	As described in §3.1
r_s_mox_uox	4-byte float	As described in §3.1
r_s_th	4-byte float	As described in §3.1
r_s_thox	4-byte float	As described in §3.1
r_t_f	4-byte float	As described in §3.1
r_th_pu	4-byte float	As described in §3.1
seed	8-byte integer	The random seed used to generate an instance.

A.2 Problem Instances

Problem instances are generated by problem species and are executed by problem families. Accordingly, both species and families can record information about instances. Front and back-end exchange species each record two types of information: details about each arc in an instance and a summary of species-specific information. The exchange family records information regarding each of the entities that comprise an instance: nodes, groups of nodes (having been translated from portfolios), and arcs. Further, aggregate summary information is also recorded.

Table A.2: Datatype description of the `/Species/StructuredSupply/Points` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
d_f_mox	4-length array of 8-byte floats	As described in §3.1
d_f_thox	4-length array of 8-byte floats	As described in §3.1
d_th	3-length array of 8-byte floats	As described in §3.1
f_fc	1-byte integer	As described in §3.1
f_loc	1-byte integer	As described in §3.1
f_mox	4-byte float	As described in §3.1
f_rxtr	1-byte integer	As described in §3.1
n_reg	4-byte unsigned integer	As described in §3.1
n_rxtr	4-byte unsigned integer	As described in §3.1
r_inv_proc	4-byte float	As described in §3.1
r_l_c	4-byte float	As described in §3.1
r_repo	4-byte float	As described in §3.1
r_s_mox	4-byte float	As described in §3.1
r_s_mox_uox	4-byte float	As described in §3.1
r_s_th	4-byte float	As described in §3.1
r_s_thox	4-byte float	As described in §3.1
r_t_f	4-byte float	As described in §3.1
r_th_pu	4-byte float	As described in §3.1
seed	8-byte integer	The random seed used to generate an instance.

A.2.1 Exchange Family

The exchange family records information regarding all major constructs in an exchange: nodes, groups, and arcs. A summary table is written to `/Family/ResourceExchange/ExchangeInstProperties`. Nodes and group data are recorded in an aggregate dataset located at `/Family/ResourceExchange/ExchangeNodes`, node group data is located at `/Family/ResourceExchange/ExchangeGroups`, and arc data is collected in the `/Family/ResourceExchange/ExchangeArcs` group. A dataset per instance UUID is used because it has been found to have better performance in the post-processing phase. A summary of family-specific instance data are detailed in Tables A.3-A.6.

A.2.2 Exchange Species

Both exchange species record information about each arc in an exchange instance. A parent group for arc data is defined under each species group. A group for each instance, whose name is the hex string of the UUID, is defined under the associated arc group. Finally, arc information associated with each instance is stored as a dataset in that instance's group. For example, the arc data for a given UUID of a front-end exchange is located as a dataset in the group `/Species/StructuredRequest/Arcs/<UUID hex>`. Summary information related to each species is also recorded in a data set for each species type located in the group `/Species/<species type>/Summary`. Tables describing species-specific instance

Table A.3: Datatype description of the `/Family/ResourceExchange/ExchangeInstProperties` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
species	30-character string	A description of a problem species.
n_arcs	8-byte integer	The number of arcs in an NFCTP instance.
n_u_grps	8-byte integer	The number of supply groups in an NFCTP instance.
n_v_grps	8-byte integer	The number of demand groups in an NFCTP instance.
n_u_nodes	8-byte integer	The number of supply nodes in an NFCTP instance.
n_v_nodes	8-byte integer	The number of demand nodes in an NFCTP instance.
n_constrs	8-byte integer	The number of constraints in an NFCTP instance.
excl_frac	8-byte float	The fraction of arcs in a NFCTP graph that are exclusive.

Table A.4: Datatype description of the `/Family/ResourceExchange/ExchangeNodes` dataset.

Name	Data Type	Description
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
id	8-byte integer	A uniquely identifying value.
gid	8-byte integer	A unique value identifying an ExchangeGroup
kind	1-byte integer bitfield	Whether an object is associated with supply or demand.
qty	8-byte float	A quantity.
excl	1-byte integer bitfield	Whether or not an arc is exclusive.
excl_id	8-byte integer	A unique value identifying the mutually exclusive group an arc belongs to.

Table A.5: Datatype description of the `/Family/ResourceExchange/ExchangeGroups` dataset.

Name	Data Type	Description
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
id	8-byte integer	A uniquely identifying value.
kind	1-byte integer bitfield	Whether an object is associated with supply or demand.
caps	4-length array of 8-byte floats	Capacity RHS values.
cap_dirs	4-length array of 1-byte integer bitfields	Whether a constraint is greater or less-than
qty	8-byte float	A quantity.

data are detailed in Tables A.7-A.9.

A.3 Solutions

For every solution, data is added to the Cyclopts `/Results` dataset. Problem solutions are determined from problem instances, and are thus managed by a problem family. Aggregate solution information is provided in a family dataset `/Family/ResourceExchange/ExchangeSolutionProperties`. The full results of each solve, i.e., the amount of resources flowing across each arc, are recorded in a group specific to each solution UUID. Tables related to instance solutions are described in Tables A.10-A.12.

Table A.6: Datatype description of the `/Family/ResourceExchange/ExchangeArcs/<Instance UUID>` dataset.

Name	Data Type	Description
id	8-byte integer	A uniquely identifying value.
uid	8-byte integer	Supply node for an arc.
ucaps	4-length array of 8-byte floats	Capacity coefficients for a supply node.
vid	8-byte integer	Request node for an arc.
vcaps	4-length array of 8-byte floats	Capacity coefficients for a request node.
pref	8-byte float	Preference value of an arc.

Table A.7: Datatype description of the `/Species/<Species Type>/Arcs/<Instance UUID>` dataset.

Name	Data Type	Description
arcid	4-byte unsigned integer	The hex value of a UUID for an arc.
commod	4-byte unsigned integer	The commodity associated with an arc.
pref_c	4-byte float	Commodity-based preference of an arc.
pref_l	4-byte float	Location-based preference of an arc.

Table A.8: Datatype description of the `/Species/StructuredRequest/Summary` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
n_r_th	4-byte unsigned integer	As described in §3.1
n_r_f_mox	4-byte unsigned integer	As described in §3.1
n_r_f_thox	4-byte unsigned integer	As described in §3.1
n_s_uox	4-byte unsigned integer	As described in §3.1
n_s_th_mox	4-byte unsigned integer	As described in §3.1
n_s_f_mox	4-byte unsigned integer	As described in §3.1
n_s_f_thox	4-byte unsigned integer	As described in §3.1

Table A.9: Datatype description of the `/Species/StructuredSupply/Summary` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
n_r_th	4-byte unsigned integer	As described in §3.1
n_r_f_mox	4-byte unsigned integer	As described in §3.1
n_r_f_thox	4-byte unsigned integer	As described in §3.1
n_s_uox	4-byte unsigned integer	As described in §3.1
n_s_th_mox	4-byte unsigned integer	As described in §3.1
n_s_f_mox	4-byte unsigned integer	As described in §3.1
n_s_f_thox	4-byte unsigned integer	As described in §3.1
n_s_repo	4-byte unsigned integer	As described in §3.1

A.4 Post-Processing

Post-processing may be applied parameter, instance, and solution data. The exchange family, front-end species, and back-end species each contain a `PostProcess` dataset. Dataset layouts associated with post

Table A.10: Datatype description of the `/Results` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an Exchange-Graph instance.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
solver	30-character string	A description of the solver used.
problem	30-character string	A description of the problem family.
time	8-byte float	How long a solution took.
objective	8-byte float	The objective value associated with a solution.
cyclopts_version	12-character string	The version of Cyclopts used to generate a solution.
timestamp	26-character string	A timestamp of when a solution was ran.

Table A.11: Datatype description of the `/Family/ResourceExchange/`↩ `ExchangeInstSolutionProperties` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
pref_flow	8-byte float	The value of the product of preference and flow for arcs.
cyclus_version	20-character string	The version of Cyclus used to generate a solution.

Table A.12: Datatype description of the `/Family/ResourceExchange/ExchangeInstSolutions/<`↩ `Solution UUID>` dataset.

Name	Data Type	Description
arc_id	8-byte integer	
flow	8-byte float	

processing are described Tables A.13-A.15.

Table A.13: Datatype description of the `/Family/ResourceExchange/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
pref_flow	8-byte float	The value of the product of preference and flow for arcs.

Table A.14: Datatype description of the `/Species/StructuredRequest/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
c_pref_flow	8-byte float	The value of the product of commodity-based preference and flow for arcs.
l_pref_flow	8-byte float	The value of the product of location-based preference and flow for arcs.

Table A.15: Datatype description of the `/Species/StructuredSupply/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
c_pref_flow	8-byte float	The value of the product of commodity-based preference and flow for arcs.
l_pref_flow	8-byte float	The value of the product of location-based preference and flow for arcs.

B CYCLOPTS COMMAND LINE INTERFACE

Cyclopts provides a rich command line interface (CLI) for instance generation, local execution, and remote execution. The CLI includes a number of useful utilities, however this section will only present those required for running the full Cyclopts workflow, both local and remote. The full set of CLI options is presented in Listing B.1.

Listing B.1: All available Cyclopts CLI options (the result of `cyclopts -h`).

```
usage: Cyclopts [-h]
                {convert,exec,pp,condor-submit,condor-collect,
                condor-rm,cde,combine,col2grp,dump}
                ...

positional arguments:
  {convert,exec,pp,condor-submit,condor-collect,condor-rm,cde,
   combine,col2grp,dump}
    convert Convert a parameter space defined by an input run
                      control file into an HDF5 database for a Cyclopts
                      execution run.
    exec Executes a parameter sweep as defined by the input
                      database and other command line arguments.
    pp Post process input and output.
    condor-submit Submits a job to condor, retrieves output when it has
                      completed, and cleans up the condor user space after.
    condor-collect Collects a condor submissions output.
    condor-rm Removes processes on condor for a user.
    cde Updates the Cyclopts CDE tarfile on a Condor submit
                      node.
    combine Combines a collection of databases, merging their
                      content.
    col2grp Moves input and output databases from id-column form
                      to id-group form.
    dump Dumps information about an instance database.

optional arguments:
  -h, --help show this help message and exit
```

B.1 Local Execution

When working locally, the primary workflow is `cyclopts convert`, followed by `cyclopts exec`, finishing with `cyclopts pp`. `cyclopts convert` converts a user-provided definition of a parameter space into an instance database. `cyclopts exec` then executes some or all of those instances, resulting in a solution database. Finally, `cyclopts pp` post-processes the instance and solution data. The options for each are described in Listings B.2, B.3, and B.4, respectively.

Listing B.2: CLI options for `cyclopts convert`.

```
usage: Cyclopts convert [-h] [--cycrc CYCRC] [--profile] [--proffile ↩
    PROFFILE]
                     [--species_module SPECIES_MODULE]
                     [--species_class SPECIES_CLASS] [--rc RC] [--db DB]
                     [-n NINST] [--count] [-v] [--debug] [-u UPDATE_FREQ]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                     $HOME/.cyclopts.rc useful for declaring global
                     family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --species_module SPECIES_MODULE
                     The module for the problem species
  --species_class SPECIES_CLASS
                     The problem species class
  --rc RC The run control file to use that defines a continguous
                     parameter space.
  --db DB The HDF5 file to dump converted parameter space points
                     to. This file can later be used an input to an execute
                     run.
  -n NINST, --ninstances NINST
                     The number of problem instances to generate per point
                     in parameter space.
  --count Only read in the run control file and count the number
                     of possible samplers that will be created.
  -v, --verbose Print verbose output during the conversion process.
  --debug Use objgraph and pdb to debug the conversion process.
  -u UPDATE_FREQ, --update-freq UPDATE_FREQ
                     The instance frequency with which to update stdout.
```

Listing B.3: CLI options for cyclopts exec.

```
usage: Cyclopts exec [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                     [--family_module FAMILY_MODULE]
                     [--family_class FAMILY_CLASS] [--db DB]
                     [--solvers [SOLVERS [SOLVERS ...]]]
                     [--instids [INSTIDS [INSTIDS ...]]] [--rc RC]
                     [--outdb OUTDB] [--conds CONDS] [-v]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                       $HOME/.cyclopts.rc useful for declaring global
                       family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                       The module for the problem family
  --family_class FAMILY_CLASS
                       The problem family class
  --db DB An HDF5 Cyclopts database (e.g., the result of
                       'cyclopts convert').
  --solvers [SOLVERS [SOLVERS ...]]
                       A list of which solvers to use.
  --instids [INSTIDS [INSTIDS ...]]
                       A list of instids (as UUID hex strings) to run.
  --rc RC The run control file, which allows idetification of a
                       subset of input to run.
  --outdb OUTDB An optional database to write results to. By default,
                       the database given by the --db flag is use.
  --conds CONDS A dictionary representation of execution conditions.
                       This CLI argument can be used instead of placing them
                       in an RC file.
  -v, --verbose Print verbose output during execution.
```

Listing B.4: CLI options for cyclopts pp.

```
usage: Cyclopts pp [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                   [--family_module FAMILY_MODULE]
                   [--family_class FAMILY_CLASS]
                   [--species_module SPECIES_MODULE]
                   [--species_class SPECIES_CLASS] [--indb INDB]
```

```
                 [--outdb OUTDB] [--ppdb PPDB] [--verbose_freq VERBOSE_FREQ↩
                     ]
                 [--limit LIMIT]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                     $HOME/.cyclopts.rc useful for declaring global
                     family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                     The module for the problem family
  --family_class FAMILY_CLASS
                     The problem family class
  --species_module SPECIES_MODULE
                     The module for the problem species
  --species_class SPECIES_CLASS
                     The problem species class
  --indb INDB An HDF5 Cyclopts input database (e.g., the result of
                     'cyclopts convert').
  --outdb OUTDB An HDF5 Cyclopts output database (e.g., the result of
                     'cyclopts exec').
  --ppdb PPDB An HDF5 Cyclopts post processed database (can be
                     combined with others via 'cyclopts combine'.
  --verbose_freq VERBOSE_FREQ
                     Stdout is informed of progress at the given processed
                     instance frequency.
  --limit LIMIT Post process only X instances (used for
                     profiling/testing).
```

A diagram explaining the role of the CLI workflow with respect to Cyclopts object tree (as seen in Figure 3.6) is shown below in Figure B.1.

B.2 Remote Execution

In order to execute Cyclopts on a Condor system, the submit node must contain the Cyclopts environment. That operation is supported by the `cyclopts cde` CLI, presented in Listing B.5. A job, the input of which is an instance database, can be submitted using `cyclopts condor-submit`. Upon completion, results can be collected with `cyclopts condor-collect`. The arguments for both are shown in Listings B.6 and B.7.

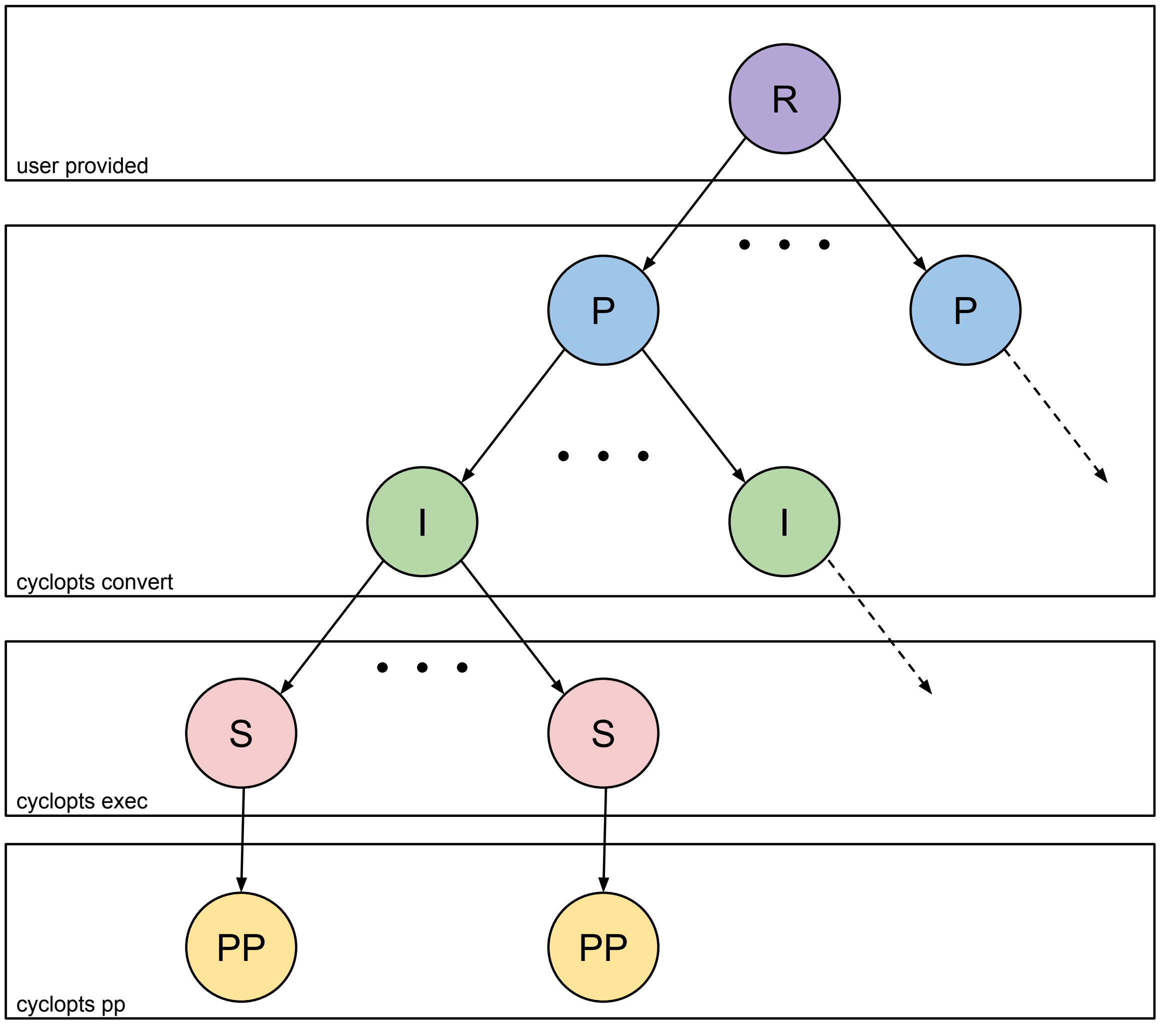

Figure B.1: The Cyclopts object tree structure is shown with boxes around each group of objects that are created given a CLI call. Note that the root node is determined from user-provided input.

Listing B.5: CLI options for `cyclopts cde`.

```
usage: Cyclopts cde [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                  [--family_module FAMILY_MODULE]
                  [--family_class FAMILY_CLASS] [--source-path PREFIX]
                  [-u USER] [-t HOST] [--no-clean] [--keyfile KEYFILE]
                  [--fname FNAME]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                      The module for the problem family
  --family_class FAMILY_CLASS
                      The problem family class
  --source-path PREFIX The path to cyclopts source.
  -u USER, --user USER The cde user name.
  -t HOST, --host HOST The remote cde submit host.
  --no-clean Do not clean up files.
  --keyfile KEYFILE An ssh public key file.
  --fname FNAME The function to wrap with cde.
```

Listing B.6: CLI options for `cyclopts condor-submit`.

```
usage: Cyclopts condor-submit [-h] [--cycrc CYCRC] [--profile]
                           [--proffile PROFFILE]
                           [--family_module FAMILY_MODULE]
                           [--family_class FAMILY_CLASS] [--rc RC]
                           [--db DB] [--instids [INSTIDS [INSTIDS ...]]]
                           [--solvers [SOLVERS [SOLVERS ...]]] [--count]
                           [-u USER] [-t HOST] [--keyfile KEYFILE]
                           [-d REMOTEDIR] [-k {dag,queue}] [--log]
                           [-p PORT] [--nodes [NODES [NODES ...]]] [-v]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
```

```
                     family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                     The module for the problem family
  --family_class FAMILY_CLASS
                     The problem family class
  --rc RC The run control file, which allows idetification of a
                     subset of input to run.
  --db DB An HDF5 Cyclopts database (e.g., the result of
                     'cyclopts convert').
  --instids [INSTIDS [INSTIDS ...]]
                     A list of instids (as UUID hex strings) to run.
  --solvers [SOLVERS [SOLVERS ...]]
                     A list of which solvers to use.
  --count Only count instances to be run.
  -u USER, --user USER The condor user name.
  -t HOST, --host HOST The remote condor submit host.
  --keyfile KEYFILE An ssh public key file.
  -d REMOTEDIR, --remotedir REMOTEDIR
                     The remote directory (relative to ~/cyclopts-runs) on
                     the submit node in which to run cyclopts jobs.
  -k {dag,queue}, --kind {dag,queue}
                     The kind of condor submission to use.
  --log Whether to keep a log of worker queue data.
  -p PORT, --port PORT The port to use for a condor queue submission.
  --nodes [NODES [NODES ...]]
                     The execute nodes to target.
  -v, --verbose Print output during the submisison process.
```

Listing B.7: CLI options for `cyclopts condor-collect`.

```
usage: Cyclopts condor-collect [-h] [--cycrc CYCRC] [--profile]
                            [--proffile PROFFILE] [--outdb OUTDB] [-u USER]
                            [-t HOST] [--keyfile KEYFILE] [-l LOCALDIR]
                            [-d REMOTEDIR] [--clean]


optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                     $HOME/.cyclopts.rc useful for declaring global
```

```
                    family/species information.
--profile Enable profiling.
--proffile PROFFILE Name of profiling filename if profile is set.
--outdb OUTDB An HDF5 Cyclopts output database (e.g., the result of
                    'cyclopts exec').
-u USER, --user USER The condor user name.
-t HOST, --host HOST The remote condor submit host.
--keyfile KEYFILE An ssh public key file.
-l LOCALDIR, --localdir LOCALDIR
                    The local directory in which to place resulting files.
-d REMOTEDIR, --remotedir REMOTEDIR
                    The remote directory (relative to the users home
                    directory) in which output files from a run are
                    located.
--clean Clean up the submit node after.
```

REFERENCES

[1] Advanced reactors information system (aris). https://aris.iaea.org/sites/core.html. Accessed: 2014-02-02.

[2] Optimization Tips for pytables. http://pytables.github.io/usersguide/optimization.html. Accessed: 2014-12-18.

[3] 2009. International project on innovative nuclear reactors and fuel cycles (INPRO). Progress Report, IAEA.

[4] Ahuja, Ravindra K, Thomas L Magnanti, and James B Orlin. 1993. Network flows: theory, algorithms, and applications.

[5] Alted, Francesc, Ivan Vilata, et al. 2002–. PyTables: Hierarchical datasets in Python.

[6] Bairiot, H, P Blanpain, D Farrant, T Ohtani, V Onoufriev, D Porsch, R Stratton, C Brown, P Deramaix, I Golovnin, et al. 2003. Status and advances in mox fuel technology. *Technical Report Series-International Atomic Energy Agency* 415:1–179.

[7] Behnel, S., R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. 2011. Cython: The best of both worlds. *Computing in Science Engineering* 13(2):31 –39.

[8] Bertel, Evelyne, and Thierry Dujardin. 2007. Management of recyclable fissile and fertile materials. *NEA News* 25(2).

[9] Bui, Peter, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, and Douglas Thain. 2011. Work queue+ python: A framework for scalable scientific ensemble applications. In *Workshop on python for high performance and scientific computing at SC11*.

[10] Cantor, Georg. 1890. Ueber eine elementare frage der mannigfaltigketislehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1:72–78.

[11] Carlsen, Robert. 2014. CyAn - Cyclus Analysis Tools.

[12] Forrest, John, et al. 2014. COIN-OR Open Solver Interface. https://projects.coin-or.org/Osi.

[13] Gidden, M. 2013. An agent-based modeling framework and application for the generic nuclear fuel cycle. Prelim, University of Wisconsin, Madison.

[14] Gidden, Matthew. 2014. Cyclopts. http://mattgidden.com/cyclopts/.

[15] Gidden, Matthew, R. Carlsen, A. Opotowsky, O. Rakhimov, A. Scopatz, and P. Wilson. 2014. Agent-based dynamic resource exchange in cyclus. In *Proceedings of PHYSOR*. Kyoto, Japan.

[16] Gidden, Matthew, and Paul Wilson. 2013. An agent-based framework for fuel cycle simulation with recycling. In *Proceedings of GLOBAL*. Salt Lake City, UT, United States.

[17] Guerin, L., L. Van Den Durpel, B. Dixon, L. Boucher, and M. Kazimi. 2009. A benchmark study of computer codes for system analysis of the nuclear fuel cycle. Tech. Rep. MIT-NFC-TR-105, MIT.

[18] Guo, Philip J. 2011. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th international conference on large installation system administration*. LISA'11, Berkeley, CA, USA: USENIX Association.

[19] Heinonen, Olli J. 2010. Safeguards in action: Iaea at rokkasho, japan. In *International atomic energy agency*.

[20] International Atomic Energy Agency (IAEA). 2007. *Liquid Metal Cooled Reactors: Experience in Design and Operation*.

[21] Jacobson, J. J., et al. 2009. *Vision user guide - vision (verifiable fuel cycle simulation) model*. Idaho National Lab, inl/ext-09-16645 ed.

[22] Julka, N., R. Srinivasan, and I. Karimi. 2002. Agent-based supply chain management-1: framework. *Computers & Chemical Engineering* 26(12):1755–1769.

[23] Klee, Victor, and George J Minty. 1970. How good is the simplex algorithm. Tech. Rep., DTIC Document.

[24] Kok, Kenneth D. 2009. *Nuclear engineering handbook*. CRC Press.

[25] Law, Averill M., and David M. Kelton. 1999. *Simulation modeling and analysis*. 3rd ed. McGraw-Hill Higher Education.

[26] Merriam-Webster Online. 2014. Merriam-Webster Online Dictionary.

[27] Murray, Paul. Personal communication, NWTRB meeting, Arlington, VA.

[28] Oliver, Kyle M. 2009. Geniusv2: Software design and mathematical formulations for multi-region discrete nuclear fuel cycle simulation and analysis. Ph.D. thesis, University of Wisconsin-Madison.

[29] Rineiski, Andrei, Makoto Ishikawa, Jinwook Jang, Prabhakaran Mohanakrishnan, Tim Newton, Gerald Rimpault, Alexander Stanculescu, and Victor Stogov. 2011. Reactivity coefficients in bn-600 core with minor actinides. *Journal of nuclear science and technology* 48(4):635–645.

[30] Scopatz, Anthony. 2013. XDress. `https://s3.amazonaws.com/xdress/index.html`.

[31] Shropshire, D. E., K. A. Williams, W. B. Boore, J. D. Smith, B. W. Dixon, M. Dunzik-Gougar, R. D. Adams, and D. Gombert. 2009. Advanced fuel cycle cost basis. Tech. Rep.

[32] Thain, Douglas, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience* 17(2-4):323–356.

[33] The HDF Group. 1997-NNNN. Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5/.

[34] Vlissides, John, R Helm, R Johnson, and E Gamma. 1995. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley* 49.

[35] Wilson, P.P.H., M. Gidden, K. Huff, and R. Carlsen. 2013. Cycamore : The cyclus additional modules repository. Http://cyclus.github.com/.