Fundamental Concepts in the Cyclus Fuel Cycle Simulator Framework and Modeling Ecosystem

Kathryn D. Huff[1],

Matthew J. Gidden[2],

Robert W. Carlsen[2],

Robert R. Flanagan[3],

Meghan B. McGarry[2],

Arrielle C. Opotowsky[2],

Erich A. Schneider[3],

Anthony M. Scopatz[2],

Paul P.H. Wilson[2]

[1]*University of California - Berkeley, Department of Nuclear Engineering Berkeley, CA, 94720*

[2]*University of Wisconsin - Madison, Department of Nuclear Engineering and Engineering Physics, Madison, WI 53706*

[3]*University of Texas - Austin, Department of Mechanical Engineering, Nuclear and Radiation Engineering Program, Austin, TX*

Send proofs to: Kathryn D. Huff

huff@berkeley.edu

2150 Shattuck Ave., Suite 230, Berkeley, CA 94704

Number of Pages: 43

Number of Tables: 3

Number of Figures: 9

Keywords: nuclear fuel cycle, simulation, software

Abstract

As nuclear power expands, technical, economic, political, and environmental analyses of nuclear fuel cycles by simulators increase in importance. To date, however, current tools are often rigid rather than flexible, fleet-based rather than discrete, or privately distributed rather than open source. Each of these choices presents a challenge to modeling fidelity, generality, efficiency, robustness, and scientific transparency. The CYCLUS nuclear fuel cycle simulator framework and its modeling ecosystem incorporate modern insights from simulation science and software architecture to solve these problems so that challenges in nuclear fuel cycle analysis can be better addressed. A summary of the CYCLUS fuel cycle simulator framework and its modeling ecosystem are presented. Additionally, the implementation of each is discussed in the context of motivating challenges in nuclear fuel cycle simulation. Finally, the current capabilities of CYCLUS are demonstrated for both an open and a closed fuel cycle.

I INTRODUCTION

As nuclear power expands, technical, economic, political, and environmental analyses of nuclear fuel cycles by simulators increase in importance. The merits of advanced nuclear technologies and fuel cycles are shaped by myriad physical, nuclear, chemical, industrial, and political factors. Nuclear fuel cycle simulators must therefore couple complex models of nuclear process physics, facility deployment, and material routing. However, current fuel cycle simulators rely on closed platforms and inflexible architectures which exhibit three main failure modes. First, they discourage targeted contribution and collaboration among experts. Next, they hobble efforts to directly compare modeling methodologies. Finally, they over-specialize, rendering most tools applicable to only a subset of desired simulation fidelities, scales, and applications.

The cardinal purpose of a dynamic nuclear fuel cycle simulator is to calculate the time- and facility-dependent mass flow through all or part the fuel cycle. Dynamic nuclear fuel cycle analysis more realistically supports a range of simulation goals than static analysis [1]. Historically, dynamic nuclear fuel cycle simulators have calculated fuel cycle mass balances and performance metrics derived from

them using software ranging from spreadsheet-driven flow calculators to highly specialized system dynamics modeling platforms. To date, current tools are often rigid rather than flexible, fleet-based rather than discrete, or privately distributed rather than open source. Each of these choices presents a challenge to modeling fidelity, generality, efficiency, robustness, and scientific transparency.

The CYCLUS nuclear fuel cycle simulator framework and its modeling ecosystem incorporate modern insights from simulation science and software architecture to solve these problems. These modern methods simultaneously enable more efficient, accurate, robust, and validated analysis. This next-generation fuel cycle simulator is the result of design choices made to:

- support access to the tool by fuel cycle analysts and other users,
- encourage developer extensions,
- enable plug-and-play comparison of modeling methodologies,
- and address a range of analysis types, levels of detail, and analyst sophistication.

CYCLUS, is a dynamic, agent-based model, which employs a modular architecture, an open development process, discrete agents, discrete time, and arbitrarily detailed isotopic resolution of materials. Experience in the broader field of systems analysis indicates that agent-based modeling enables more flexible simulation control, without loss of generality [2]. Furthermore, openness allows cross-institutional collaboration, increases software robustness [3], and cultivates an ecosystem of modeling options. This ecosystem is comprised of flexible, interchangeable models of fuel cycle components with varying scope, depth, and fidelity. It allows users and developers to customise CYCLUS to analyze the cases that are of interest to them, rather than those the simulator was originally developed to address. The fundamental concepts of the CYCLUS nuclear fuel cycle simulator capture these modern insights so that novel challenges in nuclear fuel cycle analysis can be better addressed.

IA Background

Nuclear fuel cycle simulators drive research development and design (RD&D) by calculating 'metrics', quantitative measures of performance that can be compared among fuel cycle options. The feasibility of the technology development and deployment strategies which comprise a fuel cycle option, the dynamics of transitions between fuel cycles, and many other measures of performance can be expressed in terms of these metrics. For example, economic feasibility is often measured in levelized cost of electricity (LCOE), requiring calculation of lifetime fuel costs and electricity generation, while environmental performance might be measured by spent fuel volume, toxicity, or mined uranium. A meta-analysis of fuel cycle systems studies identified over two dozen unique quantitative metrics spanning economics and cost, environmental sustainability and waste management impacts, safety, security and nonproliferation, resource adequacy and utilization, among others. With few exceptions, these metrics are derived from mass balances calculated by a fuel cycle simulator. [4].

However, methods for calculating those metrics vary among simulators. Some model the system of facilities, economics, and materials in static equilibrium, while other simulators capture the dynamics of the system. Similarly, while some simulators discretely model batches of material and individual facilities, others aggregate facilities into fleets and materials into streams. Simulators can model a single aspect of the fuel cycle in great detail while neglecting others. For example, a simulator created for policy modeling might have excellent capability in economics while capabilities for tracking transformations in material isotopics and the effects of isotopics on technology performance are neglected. The Code for Advanced Fuel Cycles Assessment (CAFCA) [5] simulator is problem-oriented in this way, having elected to neglect isotopic resolution in favor of integral effects.

Historically, domestic national laboratories have driven development and regulated the use of their own tools: the Verifiable Fuel Cycle Simulation Model (VISION) [6], Dynamic Model of Nuclear Development (DYMOND) [7], and Nuclear Fuel Cycle Simulator (NFCSim) [8,9]. Internationally, other laboratories have created their own as well, such as Commelini-Sicard (COSI) [10–13] and ORION [14]. Finally, some simulators initiated in a national lab setting have been continued as propriety, industry-

based simulators, such as Dynamic Analysis of Nuclear Energy System Strategies (DANESS) [15]. Outside the national laboratories, researchers have created new nuclear fuel cycle simulation tools when existing tools were not available or not sufficiently general to calculate their metrics of interest. With limited access to the national laboratory tools and a need to customize them for research purposes, universities and private industry researchers have "reinvented the wheel" by developing tools of their own from scratch and tailored to their own needs. Examples include CAFCA [16] and Dynamic Analysis of Nuclear Energy Systems Strategies (DESAE) [9, 17, 18].

CYCLUS emerged from a line of tools seeking to break this practice. Its precursor, Global Evaluation of Nuclear Infrastructure Utilization Scenarios (GENIUS) Version 1 [19, 20], originated within Idaho National Laboratory (INL) and sought to provide generic regional capability. Based on lessons learned from GENIUS Version 1, the GENIUS Version 2 [21, 22] simulator sought to provide more generality and an extensible interface to facilitate collaboration. The CYCLUS project then improved upon the GENIUS effort by implementing increased modularity and encapsulation. The result is a dynamic simulator that treats both materials and facilities discretely, with an architecture that permits multiple and variable levels of fidelity. Using an agent-based framework, the simulator tracks the transformation and trade of resources between autonomous regional and institutional entities with customizable behavior and objectives. This capability is an innovation not pursued by any existing fuel cycle simulator.

IB Motivation

The CYCLUS paradigm enables targeted contribution and collaboration within the nuclear fuel cycle analysis community to achieve two important goals: lower the barrier for innovative nuclear technologies to be included in fuel cycle analysis while improving the ability to compare simulations with and without those innovative concepts. This essential capability is absent in previous simulators where user customization and extensibility were not design objectives. While the *modular and open architecture* of CYCLUS is necessary to meet these goals, it is not sufficient. *Agent interchangeability* is required to facilitates direct comparison of alternative modeling methodologies and facility concepts. Finally,

CYCLUS is applicable to a broader range of fidelities, scales, and applications than other simulators, due to the flexibility and generality of its *agent-based modeling (ABM)* paradigm and *discrete, object-oriented approach*.

This structure recognizes that specialists should utilize their time and resources in modeling the specific process associated with their area of expertise (e.g., reprocessing and advanced fuel fabrication), without having to create a model of the entire fuel cycle to serve as its host. CYCLUS supports them by separating the problem of modeling a physics-dependent supply chain into two distinct components: a simulation kernel and archetypes that interact with it. The kernel is responsible for supporting the deployment and interaction logic of entities in the simulation. Physics calculations and customized behaviors of those entities are implemented within *archetype* classes.

Ultimately, modeling the evolution of a physics-dependent, international nuclear fuel supply chain is a multi-scale problem which existing tools cannot support. They have either focused on macro effects, e.g., the fleet-level stocks and flows of commodities, or micro effects, e.g., the used-fuel composition of fast reactor fuel. Each focus has driven the development of specialized tools, rendering the task of answering questions between the macro and micro levels challenging within a single tool. In contrast, the open, extensible architecture and discrete object tracking of CYCLUS allow the creation and interchangeability of custom archetypes at any level of fidelity and by any fuel cycle analyst.

IB.i Open Access and Development Practices

The proprietary concerns of research institutions and security constraints of data within fuel cycle simulators often restrict access. Use of a simulator is therefore often limited to its institution of origin, necessitating effort duplication at other institutions and thereby squandering broader human resources. License agreements and institutional approval are required for most current simulators (e.g. COSI6, DANESS, DESAE, EVOLCODE, FAMILY21, NFCSim) [23], including ORION, and VISION. Even when, as in the case of the MIT CAFCA software, the source code is unrestricted, the platform on which it relies is often restricted or costly. However, CYCLUS provides fully free and open access to all users and developers, foreign and domestic.

Moreover, both technical and institutional aspects of the software development practices employed by the CYCLUS community faciltate collaboration. Technically, CYCLUS employs a set of tools commonly used collaborative software development that reduce the effort required to comment on, test and ultimately merge individual contributions into the main development path. For many of the simulation platforms adopted by previous simulators, there were technical obstacles that impeded this kind of collaboration. Institutionally, CYCLUS invites all participants to propose, discuss and provide input to the final decision making for all important changes.

IB.ii Modularity and Extensibility

Modularity is a key enabler of extending the scope of fuel cycle analysis within the CYCLUS framework. Changes that are required to improve the fidelity of modeling a paticular agent, or to introduce entirely new agents, are narrowly confined and place no new requirements on the CYCLUS kernel. Furthermore, there are very few assumptions or heuristics that would otherwise restrict the algorithmic complexity that can be used to model the behavior of such agents.

For example, most current simulators describe a finite set of acceptable cycle constructions (once through, single-pass, multi-pass). That limits the capability to create novel material flows and economic scenarios. The CYCLUS simulation logic relies on a market paradigm, parameterized by the user, which flexibly simulates dynamic responses to pricing, availability, and other institutional preferences.

This minimal set of mutual dependencies between the kernel and the agents is expressed through the dynamic resource exchange (DRE) that provides a level of flexibility that does not exist in other fuel cycle simulators. It creates the potential for novel agent archetypes to interact with existing archetypes as they enter and leave the simulation over time and seek to trade materials whose specific composition may not be known *a priori*.

IB.iii Discrete Facilities and Materials

Many fuel cycle phenomena have aggregate system-level effects which can only be captured by discrete material tracking [24]. CYCLUS tracks materials as discrete objects. Some current fuel cycle

simulation tools such as COSI [16, 18, 25], DESAE [17], FAMILY21 [18], GENIUS version 1, GENIUS version 2, and NFCSim also possess the ability to model discrete materials.

Similarly, the ability to model disruptions (i.e. facility shutdowns due to insufficient feed material or insufficient processing and storage capacity) is most readily captured by software capable of tracking the operations status of discrete facilities [24]. Fleet-based models (e.g. VISION) are unable to capture this gracefully, perhaps representing it as a reduction in the capacity of the whole fleet. All of the software capable of discrete materials have a notion of discrete facilities, however not all handle disruption in the same manner. DESAE, for example, does not allow shutdown due to insufficient feedstock. In the event of insufficient fissile material during reprocessing, DESAE borrows material from storage, leaving a negative value [18]. The CYCLUS framework does not dictate such heuristics. Rather, it provides a flexible framework on which either method is possible.

II METHODOLOGY AND IMPLEMENTATION

A modular, agent-based modeling (ABM) approach is ideal for solving the coupled, physics-dependent supply chain problems involving material routing, facility deployment, and regional and institutional hierarchies which arise in CYCLUS. Additionally, the choice to build CYCLUS on open source libraries in modern programming languages enables both remote and multiprocess execution on a number of platforms. This section begins by describing the general design features that make CYCLUS both flexible and powerful: cluster-ready software and dynamic libraries. The ABM framework is then described, focusing on its implementation and benefits in a fuel cycle context. A discussion of the time-dependent treatment of discrete resources follows, focusing on the DRE. Support for users and developers via the CYCAMORE library of archetypes and the experimental toolkit are also presented. Lastly, the methods for quality assurance are outlined.

IIA Modular Software Architecture

The architecture of CYCLUS allows developers to define nuclear fuel cycle processes independent of the simulation logic. To achieve this, *agents* are developed which represent facilities, institutions, and regions comprising the nuclear fuel cycle. These agents are created using the CYCLUS framework application programming interface (API), a set of functions and protocols which assist in agent development and specify how agents should be defined. This encapsulated 'plug-in' design choice provides two major benefits. First, analysts can take advantage of the simulation logic API and archetype ecosystem when they apply CYCLUS to their specific problem. A modeler can focus on creating or customizing nuclear facility, institution, resource, and toolkit models within their specific area of technical expertise. Second, because CYCLUS uses a modular archetype approach, comparing two archetypes is straightforward. For example, if an analyst would like to compare the effect of using different models to determine the input fuel composition for fast reactors, fuel fabrication archetypes can be developed and interchanged while keeping the rest of the models used in the simulation fixed.

IIA.i Cluster-Ready Software

Innovative approaches to designing and optimizing robust fuel cycle strategies can become available by leveraging modern parallel computing resources. For example, large scale sensitivity analyses to quantify the dependence of fuel cycle outcomes on potentially hundreds of uncertain parameters would only be feasible in a massively parallelized environment. However, many fuel cycle simulators rely on commercial, off-the-shelf (COTS) and Windows-only software that limits performance on and compatibility with resource computing infrastructures. This constrains the possible scope of simulations and increases the wall-clock time necessary to conduct parameterized sensitivity analyses and other multi-simulation studies. CYCLUS, on the other hand, is primarily written in `C++` and relies on libraries supported by Linux and UNIX (including Ubuntu and OSX) platforms, which are flexible and support parallelization. Furthermore, the core infrastructure and related archetypes are free and open source, BSD-3-clause licensed. CYCLUS can therefore be easily deployed on large computer systems, such as

high-throughput computing (HTC) systems.

Cyclopts [26], a proof of principle design and implementation of a CYCLUS-enabled application on such a large computer system, uses UW-Madison's HTCondor HTC infrastructure to perform sensitivity studies. Cyclopts has run over 10^5 jobs, comprising more than 60,000 total compute hours. The HTC infrastructure has separately been utilized to run and collect information from full CYCLUS simulations running in parallel on 10^3 machines reliably for order 10^5 simulations.

IIA.ii Dynamically Loadable Libraries

The CYCLUS architecture encourages efficient, targeted contribution to the ecosystem of archetype libraries. With CYCLUS, a researcher can focus on generating an archetype model within their sphere of expertise while relying on the contributions of others to fill in the other technologies in the simulation. Similarly, individual developers may explore different levels of complexity within their archetypes, including wrapping other simulation tools as loadable libraries within CYCLUS.

CYCLUS achieves this behavior by implementing generic APIs and a modular architecture via a suite of dynamically loadable plug-in libraries (pictured in Figure 1). By anticipating the possible classes of information required by the simulation kernel, the CYCLUS APIs facilitate information passing between the plug-in agents and the core framework. Though common in modern software architecture, such a plug-in paradigm has not previously been implemented in a nuclear fuel cycle simulator. It allows the core CYCLUS framework to operate independently from the plug-in libraries, and the dynamically loadable plug-ins to be the primary mechanism for extending CYCLUS' capabilities independent of the core.

An additional benefit is the ability for contributors to choose different distribution and licensing strategies for their contributions. Users and modelers control the accessibility of their archetypes and data sets (See Figure 2). In particular, since the clean plug-in architecture loads libraries without any modifications to the CYCLUS kernel, closed-source archetypes can be used with the simulator alongside open source archetypes without transfer of sensitive information. This architecture allows closed-source libraries (e.g., those representing sensitive nuclear processes and subject to export control)

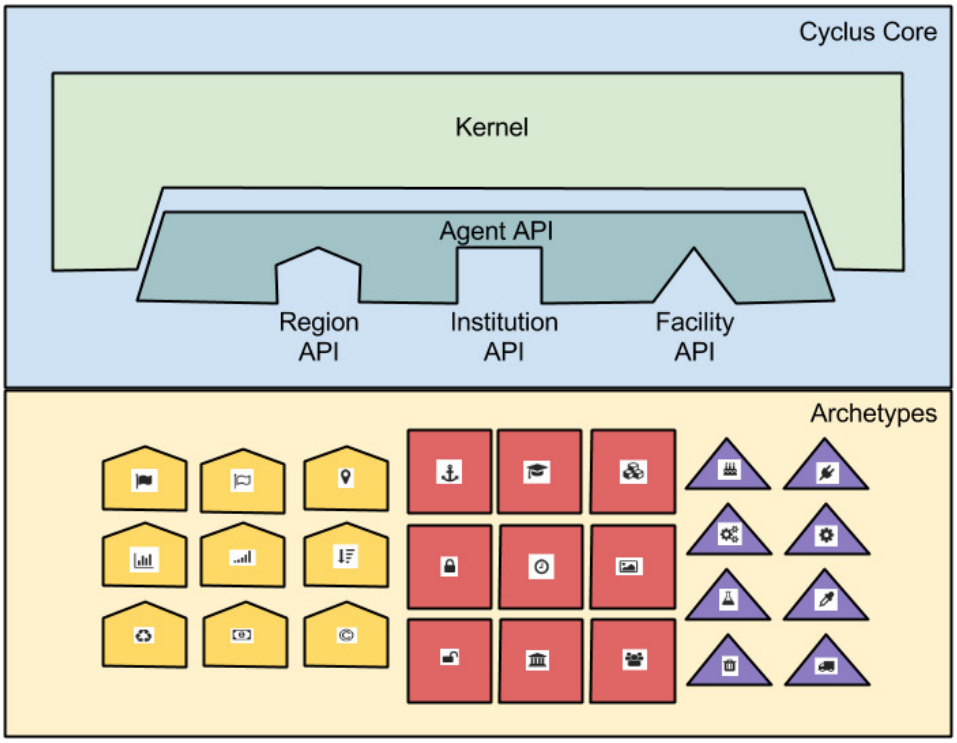

Fig. 1: The CYCLUS core provides APIs that abstract away the details in the kernel and allow the archetypes to be loaded into the simulation in a modular fashion.

to be developed and licensed privately.

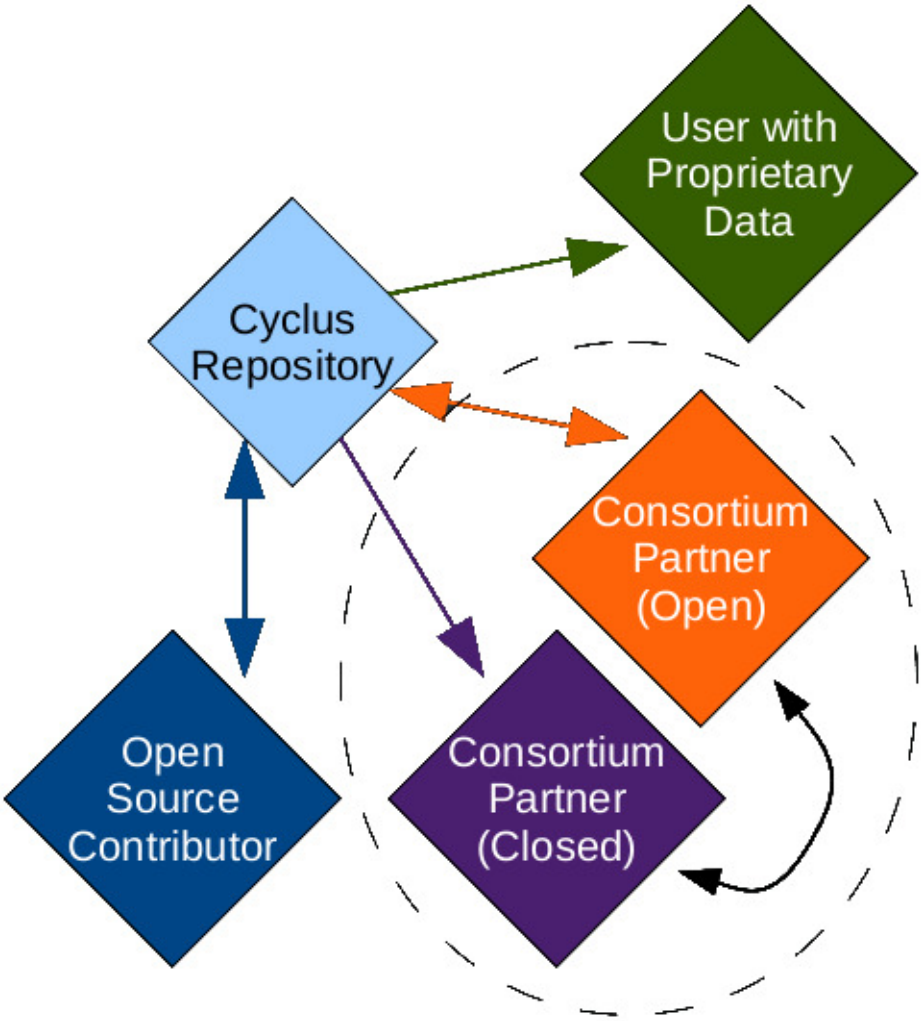

Fig. 2: The CYCLUS framework enables fully open, partially open, and fully closed collaborations [27].

Finally, dynamically loadable libraries enable CYCLUS to easily handle varying levels of simulation complexity. Hence a single simulation engine can be used by both users interested in big-picture policy questions as well as users focused on detailed technical analyses. They merely choose their preferred level of modeling depth from among the available libraries in the ecosystem.

IIB Agent-based Paradigm

CYCLUS implements an agent-based modeling paradigm. ABM enables model development to take place at an agent level rather than a system level. In the nuclear fuel cycle context, for example, an analyst can design a reactor agent that is entirely independent from a fuel fabrication agent. Defining the behavior of both agents according to the API contract is sufficient for them to interact with one another as bona fide agents in the simulation. The two archetype libraries can be used in the same simulation without any shared knowledge, allowing modelers to construct a simulation from building blocks of many types and origins.

Furthermore, the ABM paradigm is superior to the system dynamic approach used in current simulators. System dynamics is a popular approach for modeling nuclear fuel cycles [5, 15, 16, 28]. Formally however, system dynamics models are simply a strict subset of agent-based models [2]. That is, any system dynamics model can be translated into an agent-based model. ABM techniques therefore enable a broader range of simulations in a more generic fashion.

IIB.i Agent Interchangeability

ABM is inherently object-oriented because agents represent discrete, independently acting objects. Figure 1 illustrates the modular nature of CYCLUS archetypes. The core of the CYCLUS simulator creates a set of classes on which agent plug-ins are based. Agent plug-ins utilize the generic core API to interact with one another. For example, they use the resource exchange paradigm API for trading resources with one another.

Due to this plug-in architecture, implementations of agents in a given scenario can be easily interchanged. Critically, this novel functionality enables the comparison between agent implementations. For example, a low-physics-fidelity implementation of a reactor can be compared to an implementation with higher physics fidelity, allowing an analyst to discern the effect of reactor physical fidelity on a given fuel cycle.

Interchangeability is accomplished by providing inheritable APIs that define agent-to-agent interac-

tion and agent-to-environment interaction. For example, an archetype that inherits the `Region` interface, as in Figure 3, is interchangeable with any other Region agent. For the archetype developer, this interface provides enormous power very simply. The API abstracts away unnecessary detail while providing all necessary functionality for interacting with the CYCLUS simulation kernel.

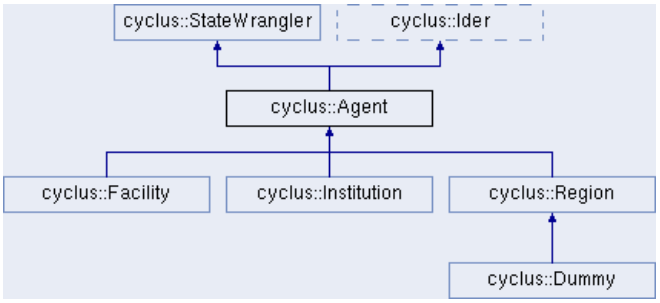

Fig. 3: Inheriting CYCLUS class interfaces, such as the `Agent`, `Facility`, `Institution`, and `Region` classes, abstracts away unnecessary details while exposing powerful functionality. In the above example, the `Dummy` archetype simply inherits from `Region` in order to become a bona fide Region-type `Agent`.

In this way, a researcher can directly compare two different reactor modeling implementations (perhaps the imaginary classes `DetailedReactor` and `SimpleReactor`) simply by exchanging the two corresponding archetypes. That is, two reactor archetypes both inheriting from the `Facility` class are indistinguishable from a simulation perspective. This can be done with any agent type, where agents can be "Regions," "Institutions," or "Facilities."

IIB.ii Regions, Institutions, and Facilities

CYCLUS provides a novel representation of entities in the nuclear fuel cycle that reflects the reality in international nuclear power: facilities implementing individual fuel cycle technologies, institutions managing those facilities, and regions providing geographical and political context for institutions and facilities. While few simulators have provided any notion of static regional effects [23, 24], CYCLUS allows for both regions and institutions to be first-class agents in simulated fuel cycles. The fundamental interactions for each entity are implemented in a corresponding archetype class in CYCLUS, i.e., the `Region` class, `Institution` class, and `Facility` class. Archetype developers can then build on

the provided functionality by inheriting from the appropriate class. CYCLUS implements a Region-Institution-Facility (RIF) relationship through the parent-child hierarchy described in §IIB.i, where regions are the parents of institutions which are, in turn, the parents of facilities. In other words, RIF hierarchies form a directed acyclic graph (DAG), with regions as root nodes and facilities as leaf nodes.

Two primary consequences arise from this structure. First, institutions are nominally responsible for deploying and decommissioning facilities. Accordingly, advanced logic regarding building and decommissioning can be implemented on top of those behaviors inherited from the `Institution` interface. Second, the `Facility` class implements the `Trader` interface, and thus institutions and regions, respectively, can adjust the resource flow preferences of their managed facilities. Importantly, this capability allows for the modeling of preferential regional trading of resources (e.g., tariffs) as well as preferential institutional trading (e.g., long-term contracts).

IIC Discrete Objects

CYCLUS models facilities, institutions, regions, and resources as discrete objects. A discrete resource model allows for a range of modeling granularity. In the macroscopic extreme, it is equivalent to time-stepped continuous flow. In the microscopic extreme, the model is capable of representing arbitrarily small material objects at isotopic resolution. In this way, CYCLUS is applicable across the full range of modeling fidelity.

Fleet-based, lumped-material models do not distinguish between discrete facility entities or materials. However, some questions require resolution at the level of individual facilities and materials. As a result, many detailed performance metrics cannot be captured with previously existing fleet-based models. For all of the reasons that the ABM paradigm in IIB enables novel simulations, multiple use cases require that these agents, such as the regions, institutions, and facilities in CYCLUS, must be represented as discrete objects. For instance, tracking of individual shipments is only viable if materials and resources are tracked as discrete objects.

IIC.i Resources and Materials

Another such use case seeks to capture system vulnerability to material diversion. Provenance and trade-history of distinct materials is the fundamental information unit in such studies, and so this type of analysis requires discrete simulation of a target facility and the individual materials modified within it. Material risk analysis, therefore, demands that both facilities and materials should be discretely modeled objects like those in CYCLUS.

In CYCLUS, agents can transfer discrete resource objects among one another. Cyclus supports two types of resources:

- materials: these represent typical nuclear materials with nuclide compositions;
- products: these can represent any user-defined measure: carbon credits, build permits, employees, etc..

All operations performed on every resource object (splitting, combining, decay, etc.) are tracked in detail as they are performed. This information includes the agent that created each resource when it was introduced into the simulation. The parentage of each resource is also tracked. This makes it possible to follow the history of resources as they are transferred between agents.

The CYCLUS kernel has built-in experimental support for decay calculations. Materials store the time since their last decay and agents are free to invoke the decay function on them as desired to decay them to the current simulation time. CYCLUS can currently operate in 3 decay modes, with 1 additional mode likely to be added in a future release:

- "manual" (currently implemented) is the default mode where agents decay materials when requested by an archetype,
- "never" (currently implemented) globally turns off all decay. The Material decay function does nothing,
- "lazy" (currently implemented) decays material automatically whenever its composition is observed (e.g. when an agent queries information about a material's ^{239}Pu content),

- "periodic" (future) automatically decays all materials in a simulation with some fixed frequency.

When decay is invoked, a material checks to see if it contains any nuclides with decay constants that are significant with respect to the time change since the last decay operation. If none of the decay constants are significant, no decay calculation is performed and the material remains unchanged. This error does not accumulate because the next time the material's decay function is invoked, the time change will be larger.

CYCLUS has no notion of "tracked" versus "untracked" nuclides. In CYCLUS, the composition of a material is represented by an arbitrarily large list (potentially thousands) of nuclides. Agents are free to treat nuclides present in materials any way they please - including ignoring them. It is the responsibility of archetype developers to choose how to handle potentially full-fidelity compositions.

In large simulations, many material objects may change frequently. Material decay can also contribute significantly to such changes. In order to help avoid unnecessary runtime performance and database size impacts, compositions in CYCLUS have some special features. In particular, compositions are immutable once created. This allows multiple material objects to hold references to the same composition safely. Additionally, new compositions resulting from decay are cached and used to avoid redundant decay calculations. Figure 4 illustrates how this decay history cache works. Composition immutability in concert with decay history caching help eliminate many redundant calculations in addition to reducing the total number of composition entries recorded in the database.

IIC.ii Dynamic Resource Exchange (DRE)

The CYCLUS simulation paradigm allows discrete agents, based on archetypes about which the kernel has no knowledge, to enter the simulation at arbitrary times and trade in discrete resources. These resources are not defined *a priori*. Therefore, the logic engine defining resource interaction mechanisms among agents is crucial. The DRE is that critical logic engine on which CYCLUS simulations are built. Supporting the general CYCLUS philosophy, facilities are treated as black boxes and a supply-demand communication framework is defined.

The DRE consists of three steps: supply-demand information gathering, resource exchange solution,

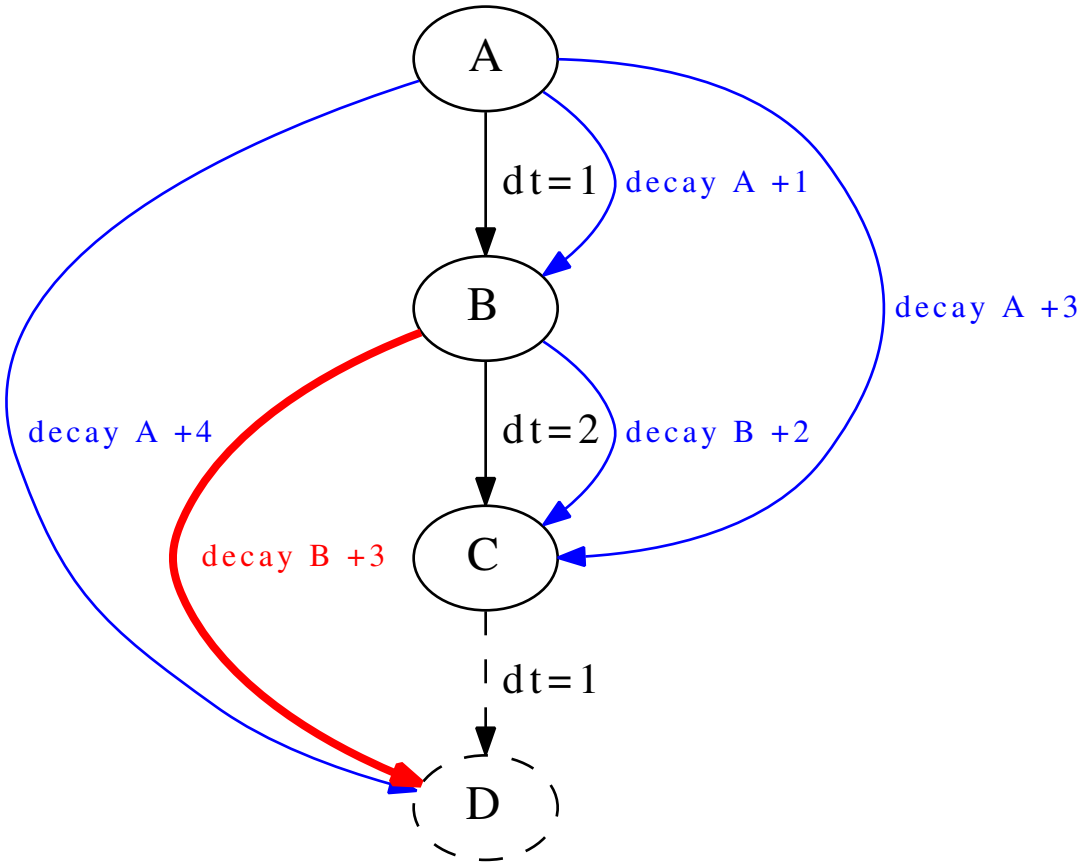

Fig. 4: A simple decay line cache holding compositions A, B, C, and a yet-uncomputed composition D. B comes from decaying A 1 time step. C comes from decaying B 2 time steps, etc. Black represents the cache for this particular composition chain. Blue indicates decay operations that can be satisfied by cache lookups. If A needs to be decayed 1 time step (A +1), a quick lookup returns the previously computed B. Decaying B 3 time steps requires a decay calculation to compute a new composition D, but subsequent requests such as decaying A 4 time steps will not require any recalculation.

and trade execution. Importantly, each step is agnostic with respect to the exchange of resources in question, i.e., the same procedure is used for both Materials and Products.

The information-gathering step begins by polling potential consumers. Agents define 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 as a series of *requests*. Users may optionally parameterize the agent to associate a collection of demand constraints with each collection of requests. Collections of requests may be grouped together, denoting *mutual* requests that represent demand for some common purpose. For example, a reactor may request uranium oxide (UOX) and mixed oxide (MOX) fuel mutually, indicating that either will satisfy its demand for fuel.

Suppliers then respond to the series of requests with a *bid*. A bid supplies a notion of the quantity and quality of a resource to match a request. Suppliers may add an arbitrary number of constraints to accompany bids. For example, an enriched UOX supplier may be constrained by its current inventory of natural uranium or its total capacity to provide enrichment (SWUs), and attach such constraints to its bids.

Any potential resource transfer, i.e., a bid or a request, may be denoted as *exclusive*. An exclusive transfer excludes partial fulfilment; it must either be met fully or not at all. This mode supports concepts such as the trading of individual reactor assemblies. In combination with the notion of mutual requests, complex instances of supply and demand are enabled.

Finally, requesting facilities, institutions and regions may apply *preferences* to each potential request-bid pairing based on the proposed resource transfer. Facilities can apply arbitrary complex logic to rank the bids that they have received, whether based on the quantity available in each bid or on the quality of each bid, and the consequent implications of the physics behavior of that facility. In addition, an institution can apply a higher preference to a partner to which it is congenial; similarly, a region may negate any transfers of material which have a higher uranium enrichment than is allowable.

Given a full definition of supply and demand, the DRE may be solved either optimally using a mathematical program or approximately by a simulation-based heuristic [29]. If any trade is denoted as exclusive, e.g., if an analyst desires an assembly-fidelity model, then either a heuristic must be used or

exchanges must be represented as a mixed-integer linear program (MILP). If no exclusive trades exist, a linear program (LP) model may be used. In practice, LPs solve much faster than MILPs. However, both capabilities exist in CYCLUS in order to provide users with the desired level of fidelity.

Trades between agents are initiated by the CYCLUS kernel after a solution to the DRE is found. For each trade, the supplying agent is notified of its matched request and provides an associated resource to the exchange. All supplied resources are then sent to the corresponding requesting agents.

In CYCLUS, the DRE is executed at each time step. Therefore, if a facility's request for a resource is not met at a given time step, it may offer a request in the following time step. Because agent behavior may change arbitrarily, the exchange executed at any given time step may be unique in a simulation.

The DRE is a novel simulation concept in the nuclear fuel cycle domain. It provides a flexible supply-demand infrastructure, supporting dynamic flows of resources between agents, even as those agents enter and leave the simulation, and even when those agents are defined by archetypes of arbitrary complexity. Trading between agents can be affected by both the proposed quality of a resource and agent relationships through the use of preferences. Accordingly, a wide range of possible effects can be modeled, from capacity-limited fuel supply to international trade agreements.

IID Simulation Support

So that users and developers can build working simulations in the shortest time possible, the CYCLUS ecosystem provides fundamental building blocks: basic archetypes and a toolkit of commonly needed functions. The CYCAMORE library provides a suite of fundamental Region, Institution, and Facility archetypes, while the CYCLUS toolkit provides assistance to developers.

IID.i Cycamore

CYCAMORE [30], the CYCLUS additional module repository, provides a fundamental set of agent archetypes for basic simulation functionality within CYCLUS. Since CYCLUS relies on external archetypes to represent the agents within a simulation, CYCAMORE provides the basic archetypes

a new user needs to get started running simple simulations. These archetypes support a minimal set of fuel cycle simulation goals and provide, by example, a guide to new developers who would seek to contribute their own archetypes outside of CYCAMORE.

As of version 1.0, CYCAMORE contains one region archetype, two institution archetypes, and four facility archetypes. Three additional facilities were added in version 1.3 . Short descriptions of these functions can be found in Table 1.

Entity	**Archetype**	**Functionality**
Facility	BatchReactor	A reactor model that handles batch refueling, based on pre-determined recipes of compositions.
Facility	EnrichmentFacility	This facility enriches uranium at a specified capacity.
Facility	Fab*	This facility fabricates fuel material based on separated streams.
Facility	Reactor*	This facility, based on BatchReactor, requests any number of input fuel types and transmutes them to static compositions upon discharge.
Facility	Separations*	This facility separates an incoming material into specified streams.
Facility	Sink	This facility is capable of accepting a finite or infinite quantity of some commodity produced in the simulation.
Facility	Source	This facility generates material of the composition and commodity type specified as input.
Institution	ManagerInst	The manager institution manages production of commodities among its facilities by building new ones as needed.
Institution	DeployInst	This institution deploys specific facilities as defined explicitly in the input file.
Region	GrowthRegion	This region determines whether there is a need to meet a certain capacity (as defined via input) at each time step.

TABLE 1: The Archetypes in CYCAMORE seek to cover a large range of simple simulation use cases [30]. Facilities added in version 1.3 are marked with *.

Section **??** will demonstrate how the current CYCAMORE release provides basic functionality enabling simple fuel cycle analyses. As future contributions are vetted, the capabilities in CYCAMORE may grow.

IID.ii Toolkit

In addition to the core functionality of the CYCLUS kernel, which is focused on the set of capabilities needed to implement an agent-based simulation with DRE, a toolkit is provided to assist developers and

users with related simulation and nuclear engineering tasks. The toolkit is an actively developed part of CYCLUS, with a primarily forward-looking focus on supporting interesting *in situ* metric analysis tools.

Simulation Tools A series of utility classes are provided to support demand-constrained agent facility deployment. For example, symbolic function representations of linear, exponential, and piecewise functions are supported via the `SymbFunctionFactory` class. Such functions are used with other toolkit classes to determine commodity demand (e.g., power demand) from user input. Four mix-in classes provide the basis for in-simulation deployment determination: `CommodityProducer`, `CommodityProducerManager`, `Builder`, `BuildingManager`. The `CommodityProducer` class provides an interface for querying the *prototypes* which have the capacity to produce a given commodity. The `CommodityProducerManager` provides an interface for registering `CommodityProducers` and querying the current capacity (supply) of a commodity. The `Builder` class provides an interface for querying which prototypes can be built and interacts with the `BuildingManager`, which orders prototypes to be built. The `BuildingManager` uses a simple minimum cost algorithm to determine how many of each prototype, y_i, to build given a demand (Φ), capacities (ϕ_i), and costs (c_i). Here i indexes I available prototypes which perform a similar function, and the demand, capacity and cost carry prototype-specific units which are defined by the developer.

$$\begin{aligned} &\min \sum_{i=1}^{N} c_i y_i \\ &s.t. \sum_{i=1}^{N} \phi_i y_i \geq \Phi \\ &\quad y_i \in [0, \infty) \ \forall i \in I, \ y_i \text{ integer} \end{aligned} \tag{1}$$

Nuclear Engineering Tools The CYCLUS toolkit provides two useful interfaces for querying physical parameters of `Material` objects. First, the `MatQuery` module provides a basic querying API, including the atom and mass fractions of nuclides, the number of moles of a nuclide in a material, etc. (i.e., a `Composition`), in a material. The `Enrichment` module provides an API for determining enrichment-related parameters of a material, including the separative work units (SWU) and natural uranium required

to enrich a material provided knowledge of feed, product, and tails assays.

Toolkit Extensions In addition to those that already exist, new tools will emerge from the archetypes developed by the community. As these tools gain adoption between projects and demonstrate their utility to the developer community, they will be considered for screening and adoption into the kernel as toolkit extensions. Likely extensions include

- fuel cycle metrics calculators,
- supportive data tables,
- policy models,
- and economic models.

IIE Quality Assurance

Simulation science - like experimental science - must distinguish trustworthy results from untrustworthy ones. Charles Babbage famously articulated this, "On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question." [31]. A simulator must assure correctness and guard against the *garbage in, garbage out* phenomenon.

Multiple strategies, collectively known as *quality assurance (QA)*, have been invented to mitigate structural and algorithmic errors in software. These include *verification and validation (V&V)* [32], testing, and others.

Nuclear engineering software quality is often governed by Nuclear Quality Assurance - 1 (NQA-1), an American Society of Mechanical Engineers (ASME) specification whose latest revision appeared in 2009 [33]. CYCLUS has adopted an *agile* development process [34], interpreting NQA-1 in a manner similar to the process adopted by the Department of Energy (DOE) within Nuclear Engineering Advanced Modeling and Simulation (NEAMS) [35] or by the PyNE toolkit [36].

Verification may be defined as the question, "Is CYCLUS being built correctly?" To answer this question, CYCLUS relies on software development best practices such as testing, documentation, version control, style guidelines, and continuous integration to ensure reliability and reproducibility. These verification techniques are discussed individually in the sections that follow.

Validation, in contrast, may be defined as the question, "Is CYCLUS the correct tool?" Since CYCLUS is alone in its class as an agent-based fuel cycle simulator, longitudinal validation is not possible. Nonetheless, code-to-code comparisons with fuel cycle simulators that use other modeling paradigms are underway [37]. However, such exercises are more likely to bring into relief the differences between the modeling paradigms than supply substantial QA and validation.

Sections IIE.i-IIE.v discuss in greater detail the software development components that comprise the CYCLUS verification strategy. Each of these on its own is a valuable addition to QA but is cannot be the entire answer to the requirements imposed by NQA-1. Taken together and strictly adhered to, they present a fortress to protect against poorly designed or otherwise undesirable code.

IIE.i Testing

Automated software *testing* is the first line of defense against errors in implementation. It also acts as an early warning sign if the simulator does not work as intended on a new system. Testing serves a critical role in QA because it directly compares the results of running software versus the expected behavior of the software. In CYCLUS, three categories of tests are defined: unit tests, integration tests, and regression tests. It is important to note that before a proposed code change is allowed into CYCLUS, the change must be covered by a test, either new or existing, and all tests must pass.

Unit Tests Unit tests compare the results of the smallest code *unit*, typically a single function or a class. CYCLUS uses the Google Test framework [38] as a harness for running unit tests. Sufficient unit tests are required for any proposed change to the CYCLUS code base. Currently, CYCLUS implements over 450 unit tests and CYCAMORE implements 85. These cover approximately 65% of their respective code bases, and these numbers are expected to grow over time.

Integration Tests Integration tests combine multiple elements of the CYCLUS interface and test that they work correctly with each other. By analogy, simply because the gears (units) are made correctly does not imply that the clock (integration) will run smoothly, run at all, or give the correct time. In CYCLUS and CYCAMORE, integration tests are performed by running sample simulations for scenarios that are sufficiently simple to predict the results and verifying that results match those predictions. A set of standard input files are run, then the output is inspected and compared via Nose [39], a Python test framework. In this way, CYCLUS code units are tested in the full context that they will be executed. This category of testing is especially useful for ensuring that major CYCLUS components are functioning as expected.

Regression Tests Regression tests ensure significant unintended changes do not occur over the course of CYCLUS development. Such a change is called a *regression* because the new version is almost always wrong. Regression tests are implemented similarly to integration tests. In this category, however, the comparison is done against the output of the same input file when run with a previous version of CYCLUS, typically the last released version. In some sense, regression tests are 'dumb' in that they do not care about the contents of a simulation being correct, only whether or not it changed.

IIE.ii Documentation

All of the public interface (the API) must be documented as required by the CYCLUS QA policy. This certifies that the intended goals of an API are communicated explicitly by the author in prose form. In CYCLUS, this information is aggregated together into static websites with the Doxygen [40] and Sphinx [41] tools, and can be accessed at `http://fuelcycle.org`.

IIE.iii Version Control

Version control preserves the development history, or *provenance*, of source code files. CYCLUS uses a well-established *distributed version control* tool called git [42] for this purpose. Distributed version control allows every user to have complete local copy of the *repository* (the version control term for

the collection of all possible histories). One feature that git performs exceptionally well is *branching*. Branches are distinct pathways in the history that diverge from a mainline source tree and then may be *merged* back in. Multiple simultaneous branches may exist at all points in time. Every change to the code is recorded in the history, along with metadata such as the author, a timestamp, and an accompanying message. Thus, it is possible for CYCLUS to accurately replay the entirety of who did what to the code when.

CYCLUS uses a strategy known as *git flow* [43] to manage topical branches where individuals develop code in parallel. All proposed software changes from topical branches must have sufficient successful tests and comprehensive documentation to be allowed into CYCLUS. As an example, a schematic of the development stages between reporting a bug and merging the fix is shown in Figure 5.

The act of proposing a change is known as a *pull request*. The main CYCLUS repository is hosted remotely on the GitHub website [44]. The online mechanism for pull requests allows for code review by a member of the CYCLUS core team, separate from the author, prior to any change being included. Non-members of the CYCLUS development team are allowed and encouraged to submit and review pull requests. However, only members of the CYCLUS core team are allowed to merge in any changes and only after the QA standards have been met. This step has been repeatedly shown to improve code quality [3].

IIE.iv Style Guide

In any multi-person software project, there is a tension between how individuals wish to write code. Every person tends to have their own custom style. To remedy this, coding style guides are the software analogy to those for written language, such as the Chicago Manual of Style. CYCLUS strictly enforces the use of the Google C++ Style Guide [45] for all software contributions. This means that all developers of CYCLUS nominally write CYCLUS code in the same way. This homogenization may be a hurdle to new developers but ultimately improves code legibility and, therefore, robustness [3].

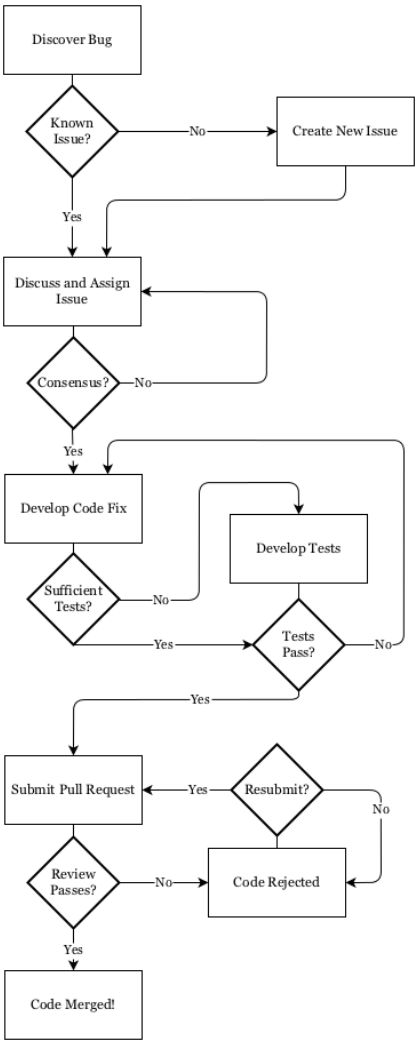

Fig. 5: A version-control enabled process for finding, reporting, and fixing a bug can take place within an issue tracker alongside multiple stages of software testing and review. In the CYCLUS case, the final review stage includes unit, regression, and integration testing as well as manual code inspection.

IIE.v Continuous Integration

Continuous integration (CI) is the idea that software should be tested and validated as it is being developed, rather than as a final stage in a longer development cycle. Under continuous integration (CI), every pull request (proposed code change) is tested independently immediately after it is proposed. Such tests are run on all officially supported platforms. CYCLUS uses a CI platform called Polyphemus [46]. Polyphemus serves as an intermediary between GitHub pull requests on the front-end and temporary CYCLUS servers on the back-end. These servers are hosted by the Build & Test Laboratory (BaTLab) [47] at the University of Wisconsin-Madison. Whenever a pull request is created, Polyphemus performs the following actions:

1. Copies CYCLUS with the proposed new code to BaTLab,
2. Initializes Linux and Mac OSX servers,
3. Builds CYCLUS on all platforms,
4. Runs the complete CYCLUS test suite,
5. Reports whether or not the above steps succeeded.

Since these steps are performed for all incoming code, it is easy for the CYCLUS core team to determine whether or not an individual pull request actually works. This helps identify build and test problems prior to broken code being allowed into the develop branch. CI is a necessary but not sufficient addition to the CYCLUS QA system: it keeps bad code out of CYCLUS. However, CYCLUS will always require human eyes and hands to let good code in.

III DEMONSTRATION

The success of the CYCLUS simulator can be measured in many ways. The most compelling are demonstrations of fundamental simulation capability and of the feasibility of the CYCLUS community-driven development model. Promising growth of the CYCLUS ecosystem at multiple institutions indicates

that a fuel cycle simulator can advance in this community-driven development paradigm. Additionally, simulation results for both once-through and more complex recycling scenarios demonstrate that CYCLUS possesses the fundamental fuel cycle simulation capabilities to contribute to the field.

IIIA Ecosystem

The CYCLUS community-driven software development model seeks to break the proliferation of specialized simulators. It instead leverages the collective expertise of fuel cycle researchers toward a single, more extensible, tool. Through the targeted contributions of those researchers, an ecosystem of capabilities should emerge. The CYCLUS 'Ecosystem' is the collection of tools, calculation libraries, archetypes, data, and input files intended for use with the CYCLUS simulator. Members of the ecosystem include:

- the archetypes provided in the CYCAMORE [30] repository
- the archetypes created by researchers
- isotopic composition data
- historical facility deployment data
- the CYCLUS graphical user interface (GUI) tool Cyclist
- fundamental analysis tools in the CYCLUS toolkit
- tools for CYCLUS optimization, parallelization, and development

Taken together, these form an ecosystem of capabilities. Over time, this ecosystem will grow as archetype developers, kernel developers, and even users contribute capabilities developed for their own needs. Indeed, the long-term vision for the CYCLUS framework predicts an ever-expanding ecosystem of both general and specialized capability extensions.

Already, the ecosystem is growing. Early cross-institutional contributions to the ecosystem demonstrate a significant achievement by the CYCLUS framework and provide the basis for a community-driven development model.

IIIA.i Supplementary Projects

A number of projects and tools outside of the core simulation kernel have been developed to improve the scope and the diversity of the capabilities in the CYCLUS ecosystem. Table 2 lists the tools and projects developed under close integration with the CYCLUS kernel. These tools are used to ease development and simulation design (Cycstub, Cycic, Ciclus), data visualization and analysis (Cyclist, Cyan), and remote execution (Cloudlus).

Name	Description	Citation
Cycic	Input control - embedded in Cyclist	[48]
Cyclist	Interactive data exploration environment	[49]
Ciclus	Continuous integration scripts for CYCLUS	[50]
Cycstub	Skeleton for clean slate module development	[51]
Cyan	CYCLUS analysis tool	[52]
Cloudlus	Tools for running CYCLUS in a cloud environment	[53]

TABLE 2: Many tools have been developed outside of the scope of the CYCLUS kernel for improved user, developer, and analyst experiences with CYCLUS.

IIIA.ii Archetype Contributions

It is expected that the most common type of contribution to CYCLUS will be contributions of new archetypes. Researchers will be driven to create a new archetype when a need arises, such as to improve the fidelity of simulation, or to represent a novel reactor type, an innovative reprocessing strategy, or a particular governmental or institutional behavior. The real-world utility of CYCLUS can in part be measured by the breadth and diversity of archetypes being developed in this way.

Early progress has been promising. Many archetypes external to the CYCAMORE library (Table 1) have been [54, 55] or are being [56–58] developed for contribution to the CYCLUS ecosystem. These archetypes provide the first examples of developer-contributed capabilities. They add to the fundamental

CYCAMORE archetypes by providing physics-based reactors, separations logic, fuel fabrication processing, storage facilities, and expanded institutional paradigms. The existence and diversity of these contributed archetypes illustrate the power and potential of the community-based development approach that CYCLUS has taken.

Name	Description	Citation
Bright-lite	A physics-based reactor archetype and fuel fabrication archetype	[56]
Nuclear Fuel Inventory Model	A flexible, ORIGEN-based, reactor analysis module	[57]
CommodConverter	A simple commodity converting storage facility archetype	[55]
MktDrivenInst	An institution that controls deployment based on commodity availability	[58]
SeparationsMatrix	A facility for elemental separations of used fuel	[54]
StreamBlender	A facility for fuel fabrication from reprocessed streams	[54]

TABLE 3: A diverse set of archetypes under development reflect the diverse needs of researchers at various institutions. These archetypes, contributed outside of the CYCLUS core and CYCAMORE libraries are the first demonstration of community-driven development in a fuel cycle simulator.

IIIB Simulations

This section will discuss the creation and results of a few simple fuel cycle simulations to help illustrate the flexibility of CYCLUS. These simulations are designed to illustrate Cyclus' capability. Many simplifying assumptions have been made with respect to material compositions, fuel transmutation, among others. The three fuel cycles examined are:

1. Once through (no recycle)
2. 1-pass MOX recycle
3. Infinite-pass MOX recycle

For each of these fuel cycles, a 1,100 month single-reactor CYCLUS simulation was run in addition to a 10-reactor simulation with staggered refueling times. As the number of staggered-cycle reactors

increases the system converges toward continuous material flow results. CYCLUS flexibility allows this transition to be examined. The facilities used in these simulations are:

- **Reactor** (`cycamore::Reactor`): This is a reactor model that requests any number of input fuel types and transmutes them to static compositions when they are burnt and discharged from the core. The reactor was configured to model a light water reactor with a 3 batch core (20,000 kg HM per batch) operating on an 18 month cycle with a 2 month refuel time. The reactor was also configured to take in either enriched UOX fuel or recycled MOX fuel.

- **Recycled Fuel Fabrication** (`cycamore::Fab`): This facility requests depleted uranium and separated fissile material and mixes them together into recycled MOX fuel that it offers to requesters. The two streams are mixed in a ratio in order to match simple neutronics properties of the target fuel as closely as possible. The method used is based on a variation "equivalence method" originally developed by Baker and Ross [59]. This technique has also been used in the COSI fuel cycle simulator developed by the French Commissariat a l'Energie Atomique et aux Energies Alternatives (CEA).

- **Separations** (`cycamore::Separations`): This facility takes in all kinds of spent fuel and separates it into plutonium and waste streams with some efficiency (0.99 was used for these simulations). Up to 60,000 kg of Pu can be separated per month.

- **Repository** (`cycamore::Sink`): This is an infinite-capacity facility that can take in all types of material including separations waste streams and spent reactor fuel.

- **UOX Fabrication** (`cycamore::Source`): This facility provides enriched UOX fuel as requested. This facility has infinite production capacity.

- **DU Source** (`cycamore::Source`): This facility provides depleted uranium as requested. This facility has infinite production capacity.

Modifying the input file specification to model each of the 3 fuel cycles was done by making simple adjustments to the commodities and trade preferences for each of the facilities. The reactor was

configured to request recycled MOX fuel with a higher preference than new UOX fuel. For the once through case, the `Reactor` was set to offer all spent fuel as waste. For the 1-pass recycle case, the `Reactor` offered spent UOX into a spent fuel market, but spent MOX is still offered as waste. In the infinite-pass recycle case, the `Reactor` offers all burned fuel into a spent fuel market. The separations facility requests spent fuel with a higher preference than the repository resulting in preferential recycle. All these preferences are easy to adjust and CYCLUS dynamically handles supply constraints and non-uniform preference resolution. It is notable that separations and recycle fuel fabrication capacity are deployed identically in all simulations. In the once through case, the recycling loop never acquires material, and so reactors always receive fresh UOX fuel. The DRE ensures everything operates smoothly in all cases.

Cyclus' discrete materials make single-pass recycle straightforward to implement. The reactors keep track of fuel as discrete batches. A reactor remembers where it received each batch from. If a batch was received from the recycled fuel fabrication facility, it does not offer it to separations and instead offers it as a waste commodity which is only requested by the repository. The material flows for the 1-pass recycle case are shown in Figure 6.

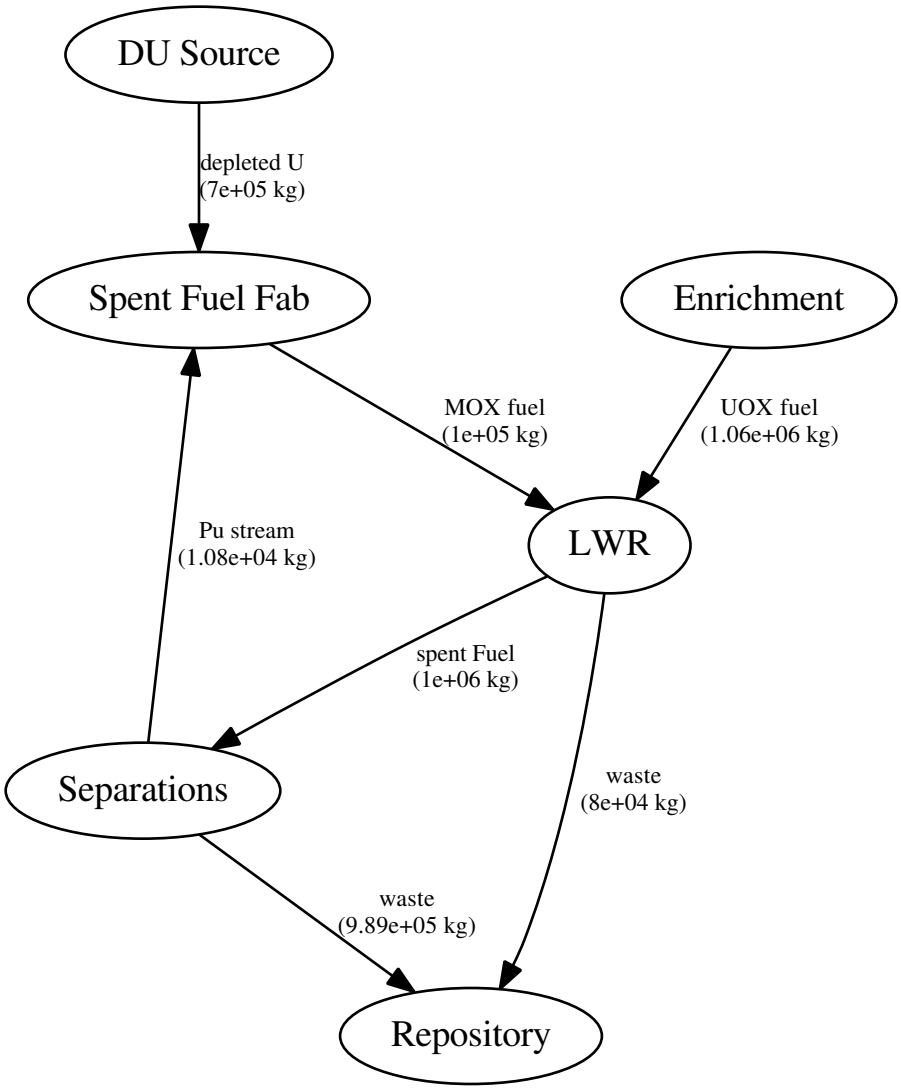

Fig. 6: 1-pass MOX recycle scenario material flows.

Changing the scenario from a 1-pass fuel cycle to an infinite-pass fuel cycle requires only a one-word change in the input file regarding the output commodity for the spent MOX fuel of the `Reactor`:

```
    <fuel>
      <incommodity>mox</incommodity>
-     <outcommodity>waste</outcommodity>
+     <outcommodity>spent_fuel</outcommodity>
      <inrecipe>mox_fresh_fuel</inrecipe>
```

```
        <outrecipe>mox_spent_fuel</outrecipe>
    </fuel>
```

This results in the material flows in Figure 7. A similarly trivial change was used to switch from the once through to a 1-pass fuel cycle. Note that because the reactors always transmute fuel into fixed compositions, the error in isotopic compositions is larger for the infinite-pass recycle case.

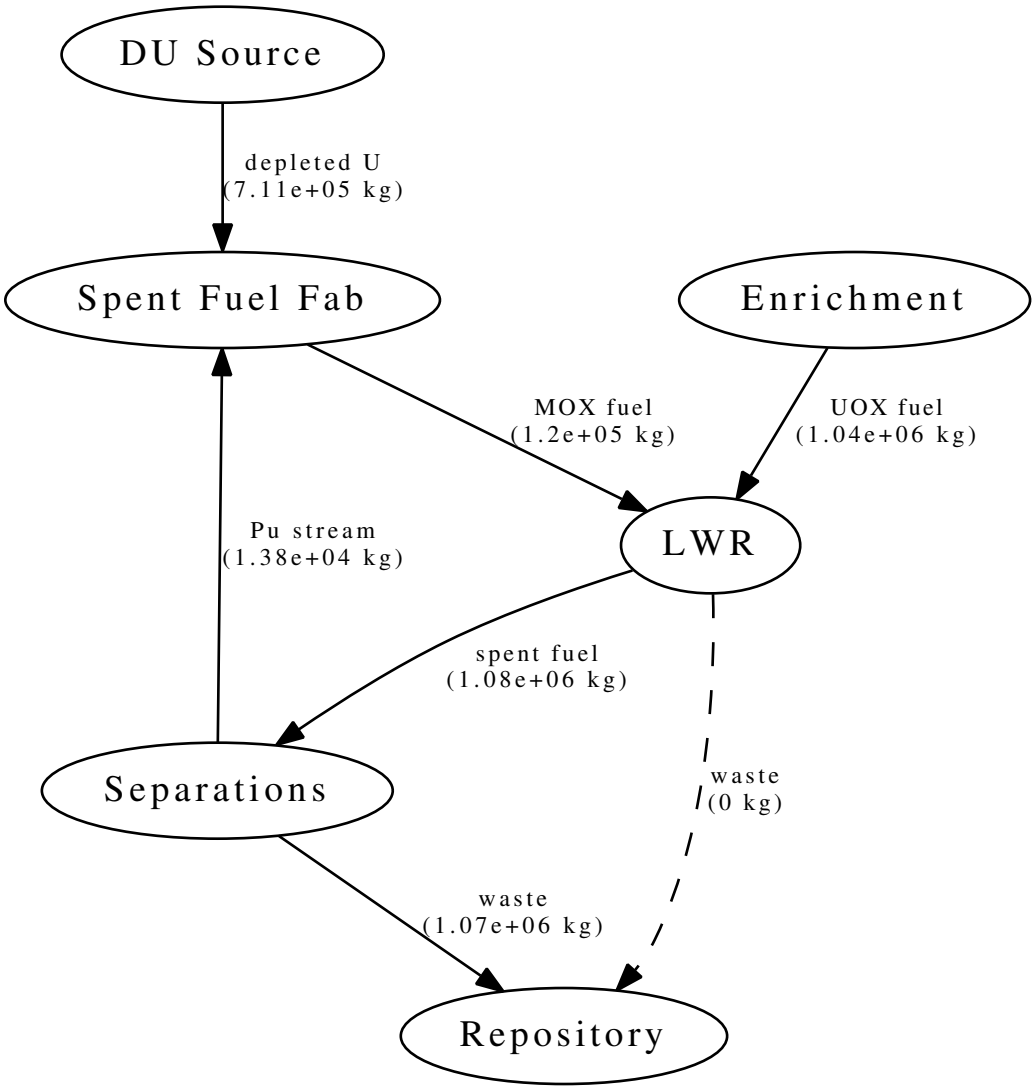

Fig. 7: Full MOX recycle (multi-pass) fuel cycle material flows.

Figure 8 shows the full-system plutonium buildup for once through (no recycle), 1-pass recycle, and infinite-pass recycle variations of the one-reactor scenario described above. The figure was generated directly from CYCLUS output data. After several batch cycles (near month 300) in the 1-pass and infinite-pass cases, enough separated fissile material accumulates in the fuel fabrication facility to generate a full recycled batch. When this batch is transmuted, more plutonium is burned than created. This results in a drop in the total fuel cycle system plutonium inventory. This pattern repeats roughly every 10 cycles (200 months) for the 1-pass recycle case and every 9 cycles (180 months) for the infinite-pass recycle case. Because the 1-pass recycle scenario does not re-recycle material, it takes the fabrication facility 2 cycles longer to accumulate a full batch of fissile material.

Because facilities are represented individually and transact discrete materials as discrete events, realistic non-uniform patterns in facility behavior that affect total system behavior are observed using CYCLUS.

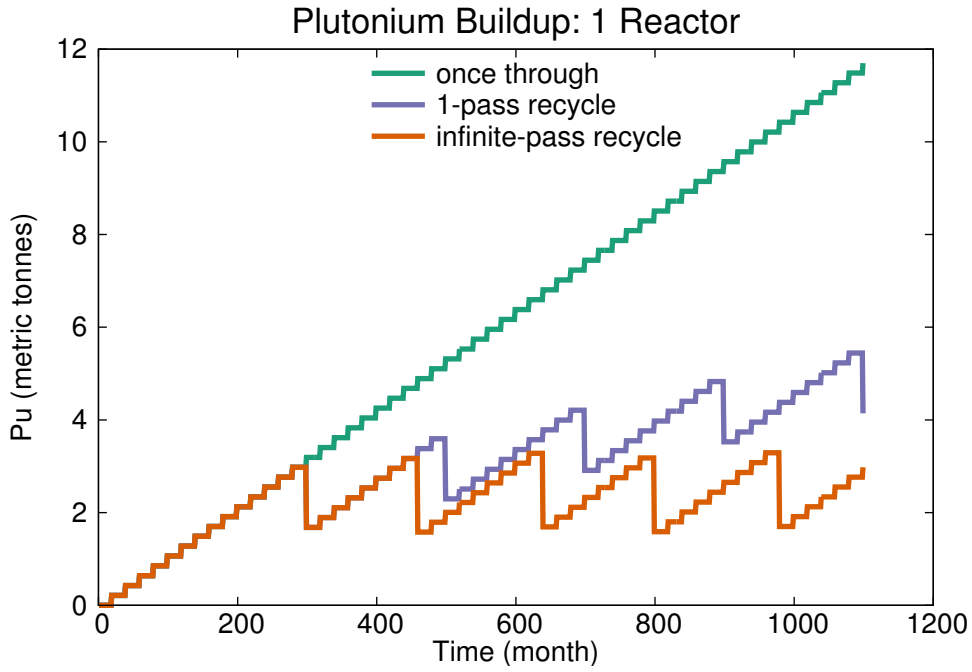

Fig. 8: System plutonium buildup with one reactor.

Figure 9 shows plutonium buildup for the 10-reactor simulations of the once through, 1-pass recycle, and infinite-pass recycle scenarios. As the number of reactors (with staggered refueling) increases, the behavior of the system approaches a more steady average reminiscent of continuous material flow models.

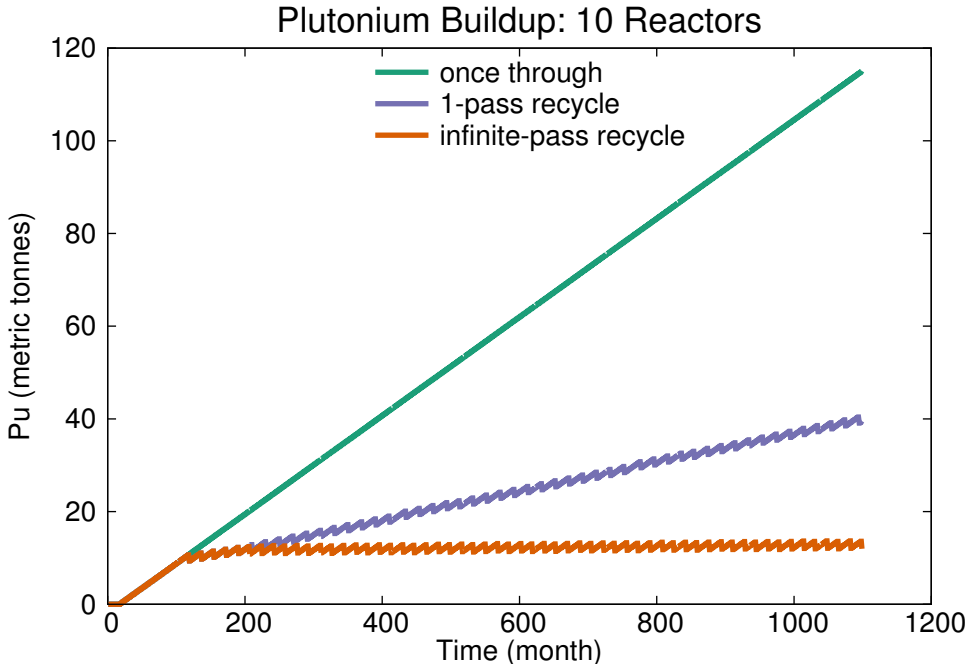

Fig. 9: System plutonium buildup with staggered refueling for many reactors.

The fundamental capabilities of demonstrated in these simulations qualify CYCLUS and its ecosystem to model the breadth of scenario types expected of nuclear fuel cycle simulator tools. These examples further show the flexibility provided by the DRE logic within the CYCLUS framework and provide an example of the resolution made possible by discrete-facility and discrete-material modeling in fuel cycle simulation.

IV CONCLUSIONS

The CYCLUS nuclear fuel cycle framework presents a more generic and flexible alternative to existing fuel cycle simulators. Where previous nuclear fuel cycle simulators have had limited distribution,

constrained simulation capabilities, and restricted customizability, CYCLUS emphasizes an open strategy for access and development. This open strategy not only improves accessibility, but also enables transparency and community oversight.

The object-oriented ABM simulation paradigm ensures more generic simulation capability. It allows CYCLUS to address common analyses in a more flexible fashion and enables analyses that are impossible with system dynamics simulators.

Similarly, the fidelity-agnostic, modular CYCLUS architecture facilitates simulations at every level of detail. Simulations relying on arbitrarily complex isotopic compositions are possible in CYCLUS, as are simulations not employing any physics at all. Indeed, agents of such varying fidelity can even exist in the same simulation. Researchers no longer need to reinvent the wheel in order to model a simulation focused on the aspects of the fuel cycle relevant to their research.

Furthermore, when the capabilities within CYCLUS, CYCAMORE, and the rest of the ecosystem are insufficient, adding custom functionality is simplified by a modular, plug-in architecture. A clean, modern API simplifies customization and independent archetype development so that researchers can create models within their domain of expertise without modifying the core simulation kernel. Throughout the CYCLUS infrastructure, architecture choices have sought to enable cross-institutional collaboration and sustainable, community-driven development. The ecosystem of capabilities, already growing, may someday reflect the full diversity of use cases in the nuclear fuel cycle simulation domain.

V ACKNOWLEDGEMENTS

This work has been supported by a number of people. The authors would like to thank software contributors Zach Welch and Olzhas Rakhimov. Additionally, the authors are grateful to the many enthusiastic members of the CYCLUS community and our Nuclear Energy University Programs (NEUP) collaboration partners at the University of Texas, University of Utah, and University of Idaho.

This work has received institutional support from Argonne National Laboratory (ANL), the NEUP, the Nuclear Regulatory Commission (NRC), the National Science Foundation (NSF), and the University

of Wisconsin (UW).

REFERENCES

1. S. J. PIET, B. W. DIXON, J. J. JACOBSON, G. E. MATTHERN, and D. E. SHROPSHIRE, "Dynamic Simulations of Advanced Fuel Cycles," *Nuclear Technology*, **173**, 227 (2011).
2. C. M. MACAL, "To agent-based simulation from system dynamics," *Proc. Simulation Conference (WSC), Proceedings of the 2010 Winter*, pages 371–382, IEEE, 2010.
3. J. COHEN, "Modern Code Review," *Making Software: What Really Works, and Why We Believe It* (2010).
4. M. FLICKER, E. A. SCHNEIDER, and P. CAMPBELL, "Evaluation Criteria for Analyses of Nuclear Fuel Cycles," *Proc. Transactions of the American Nuclear Society*, volume 111 of *Fuel Cycle Options Analysis – III*, pages 233–236, Anaheim, CA, United States, 2014, American Nuclear Society, La Grange Park, IL 60526, United States.
5. L. GUERIN and M. KAZIMI, "Impact of Alternative Nuclear Fuel Cycle Options on Infrastructure and Fuel Requirements, Actinide and Waste Inventories, and Economics," Technical Report MIT-NFC-TR-111, Massachusetts Institute of Technology (2009).
6. J. JACOBSON et al., "VERIFIABLE FUEL CYCLE SIMULATION MODEL (VISION): A TOOL FOR ANALYZING NUCLEAR FUEL CYCLE FUTURES," *Nuclear Technology*, **172**, 157 (2010).
7. A. M. YACOUT, J. J. JACOBSON, G. E. MATTHERN, S. J. PIET, and A. MOISSEYTSEV, "Modeling the Nuclear Fuel Cycle," *Proc. The 23rd International Conference of the System Dynamics Society," Boston*, 2005.
8. E. A. SCHNEIDER, C. G. BATHKE, and M. R. JAMES, "NFCSim: A Dynamic Fuel Burnup and Fuel Cycle Simulation Tool," *Nuclear Technology*, **151**, 35 (2005).
9. C. ALLAN, International Atomic Energy Agency, and International Project on Innovative Nuclear Reactors and Fuel Cycles, *Guidance for the application of an assessment methodology for Innovative Nuclear Energy Systems INPRO Manual: overview of the Methodology.*, Number IAEA-TECDOC-

1575 in IAEA-TECDOC, International Atomic Energy Agency, Vienna, rev.1 edition, (2008).

10. L. BOUCHER and J. P. GROUILLER, “"COSI" : A Simulation Software for a Pool of Reactors and Fuel Cycle Plants,” *Proc. Fuel Cycle and High Level Waste Management*, Beijing, China, 2005.
11. L. BOUCHER and J. P. GROUILLER, “"COSI": The Complete Renewal of the Simulation Software for The Fuel Cycle Analysis,” *Proc. Fuel Cycle and High Level Waste Management*, volume 1, Miami, FL, United states, 2006, ASME, New York, NY, USA.
12. M. MEYER and L. BOUCHER, “New Developments on COSI 6, the Simulation Software for Fuel Cycle Analysis,” *Proc. Proceedings of GLobal 2009*, Paris, France, 2009.
13. C. COQUELET-PASCAL et al., “Comparison of Different Scenarios for the Deployment of Fast Reactors in France - Results Obtained with COSI,” *Proc. Proceedings of GLobal 2011*, Makuhari, Japan, 2011.
14. A. WORRALL and R. GREGG, “Scenario Analyses of Future UK Fuel Cycle Options,” *Journal of Nuclear Science and Technology*, **44**, 249 (2007).
15. L. VAN DEN DURPEL, A. YACOUT, D. WADE, T. TAIWO, and U. LAUFERTS, “DANESS V4.2: Overview of Capabilities and Developments,” *Proc. Proceedings of Global 2009*, Paris, France, 2009.
16. L. GUERIN et al., “A Benchmark Study of Computer Codes for System Analysis of the Nuclear Fuel Cycle,” Technical Report, Massachusetts Institute of Technology. Center for Advanced Nuclear Energy Systems. Nuclear Fuel Cycle Program (2009), Electric Power Research Institute.
17. E. A. ANDRIANOVA, V. D. DAVIDENKO, and V. F. TSIBUL’SKII, “Desae program for systems studies of long-term growth of nuclear power,” *Atomic energy*, **105**, 385 (2008).
18. K. A. MCCARTHY et al., “Benchmark Study on Nuclear Fuel Cycle Transition Scenarios-Analysis Codes,” (2012).
19. M. L. DUNZIK-GOUGAR et al., “Global Evaluation of Nuclear Infrastructure Utilization Scenarios (GENIUS),” *Proc. GLOBAL 2007: Advanced Nuclear Fuel Cycles and Systems, September 9, 2007 - September 13, 2007*, GLOBAL 2007: Advanced Nuclear Fuel Cycles and Systems, pages 1604–1611, Boise, ID, United states, 2007, American Nuclear Society.

20. R. JAIN and P. P. H. WILSON, "Transitioning to global optimization in fuel cycle system study tools," *Proc. 2006 Winter Meeting of the American Nuclear Society, Nov 12 - 16 2006*, volume 95 of *Transactions of the American Nuclear Society*, pages 162–163, Albuquerque, NM, United states, 2006, American Nuclear Society.
21. K. M. OLIVER et al., "Studying international fuel cycle robustness with the GENIUSv2 discrete facilities/materials fuel cycle systems analysis tool," *Proc. Proceedings of GLOBAL 2009*, GLOBAL 2009: Advanced Nuclear Fuel Cycles and Systems, Paris, France, 2009.
22. K. D. HUFF et al., "GENIUSv2 Discrete Facilities/Materials Modeling of International Fuel Cycle Robustness," *Proc. Transactions of the American Nuclear Society*, volume 101 of *Nuclear Fuel Cycle Codes and Applications*, pages 239–240, Washington D.C., United States, 2009, American Nuclear Society.
23. C. A. JUCHAU, M. L. DUNZIK-GOUGAR, and J. J. JACOBSON, "Modeling the Nuclear Fuel Cycle," *Nuclear Technology*, **171**, 136 (2010).
24. K. HUFF and B. DIXON, "Next Generation Fuel Cycle Simulator Functions and Requirements Document," fcrd-sysa-2010-000110, Idaho National Laboratory (2010).
25. G. GRASSO, S. MONTI, and M. SUMINI, "NEA-WPFC/FCTS benchmark for fuel cycle scenarios study with COSI6," , Report RSE/2009/136 (2009).
26. M. GIDDEN, "Cyclopts 0.0.10," *Figshare* (2015), http://dx.doi.org/10.6084/m9.figshare.1288959.
27. R. W. CARLSEN et al., "Cyclus v1.0.0," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1041745.
28. J. J. JACOBSON et al., "VISION User Guide-VISION (Verifiable Fuel Cycle Simulation) Model," , Idaho National Laboratory (INL) (2009).
29. M. GIDDEN, "An Agent-Based Modeling Framework and Application for the Generic Nuclear Fuel Cycle," (2014).
30. R. W. CARLSEN et al., "Cycamore v1.0.0," *Figshare* (2014), http://figshare.com/articles/Cycamore_v1_0_0/1041829.
31. C. BABBAGE, *Passages from the Life of a Philosopher*, Cambridge University Press (2011).

32. B. BOEHM, *Software risk management*, Springer (1989).
33. ASME, *NQA-1a-2009, Addenda to ASME NQA-1-2008, Quality Assurance Requirements for Nuclear Facility Applications*, Number NQA-1a-2009 in Nuclear Quality Assurance, American Society of Mechanical Engineers, New York, NY (2009).
34. C. LARMAN, *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley Professional (2004).
35. NEAMS, "NUCLEAR ENERGY ADVANCED MODELING AND SIMULATION (NEAMS) SOFTWARE VERIFICATION AND VALIDATION (V&V) PLAN REQUIREMENTS," , U.S. Deptartment of Energy, Office of Nuclear Energy (2013).
36. E. BIONDO, A. SCOPATZ, M. GIDDEN, R. SLAYBAUGH, and P. P. BATES, CAMERON WILSON, "Quality Assurance within the PyNE Open Source Toolkit," *Proc. American Nuclear Society 2014 Winter Meeting*, ANS, 2014.
37. K. D. HUFF, M. FRATONI, and H. GREENBERG, "Extensions to the Cyclus Ecosystem In Support of Market-Driven Transition Capability," *Proc. Transactions of the American Nuclear Society*, Anaheim, CA, United States, 2014, American Nuclear Society.
38. G. INC, "googletest - Google C++ Testing Framework," Technical Report, Google Inc. (2008).
39. J. PELLERIN, "Nose is nicer testing for python," Technical Report, nose (2007).
40. D. VAN HEESCH, "Doxygen: Source code documentation generator tool," *URL: http://www.doxygen.org* (2008).
41. G. BRANDL, "Sphinx Documentation," (2014).
42. Software Freedom Conservancy, *Git*, Software Freedom Conservancy (2014).
43. E. KALLIAMVAKOU, D. DAMIAN, L. SINGER, and D. M. GERMAN, "The code-centric collaboration perspective: Evidence from github," , Technical Report DCS-352-IR, University of Victoria (2014).
44. L. DABBISH, C. STUART, J. TSAY, and J. HERBSLEB, "Social coding in GitHub: transparency and collaboration in an open software repository," *Proc. Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286, ACM, 2012.

45. B. WEINBERGER, C. SILVERSTEIN, G. EITZMANN, M. MENTOVAI, and T. LANDRAY, "Google C++ style guide," *h ttp://google-styleguide. googlecode. com/svn/trunk/cppguide. xml* (2008).
46. A. SCOPATZ, M. GIDDEN, and Z. WELCH, "Polyphemus v0.1," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1066058.
47. UW BaTLab Team, *BaTLab*, University of Wisconsin – Madison, Madison, WI, United States (2014).
48. R. FLANAGAN and E. A. SCHNEIDER, "Input Visualization for the Cyclus Nuclear Fuel Cycle Simulator: Cyclus Input Control," *Proc. Proceedings of GLOBAL 2013*, Salt Lake City, UT, United States, 2013.
49. Y. LIVNAT, H. GUR, K. POTTER, and R. FLANAGAN, "Cyclist," 2014, github.com/SCI-Utah/cnome.
50. A. SCOPATZ, Z. WELCH, and M. GIDDEN, "Ciclus," 2014, https://github.com/cyclus/ciclus.
51. R. W. CARLSEN et al., "CycStub," 2014, https://github.com/cyclus/cycstub.
52. R. W. CARLSEN, "CyAn," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1041836.
53. R. W. CARLSEN, "Cloudlus," 2014, https://github.com/rwcarlsen/cloudlus.
54. K. D. HUFF, "StreamBlender," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1134648.
55. K. D. HUFF, "CommodConverter," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1134647.
56. R. FLANAGAN, "Bright-lite," 2014, https://github.com/bright-dev/bright-lite.
57. S. SKUTNIK, "Development of an ORIGEN-based Reactor Analysis Module for Cyclus," *Figshare* (2015), http://dx.doi.org/10.6084/m9.figshare.1291144.
58. K. D. HUFF, "MktDrivenInst," *Figshare* (2014), http://dx.doi.org/10.6084/m9.figshare.1134650.
59. A. BAKER and R. ROSS, "Comparison of the Value of Plutonium and Uranium Isotopes in Fast Reactors," *Proc. Proc. Conf. on Breeding, Economics and Safety in Large Fast Breeder Reactors, ANL-6972*, pages 7–10, 1963.