Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

MODULAR PYTHON LIBRARY FOR SIMULATIONS OF SEMICONDUCTOR ASSEMBLY AND TEST PROCESS EQUIPMENT

Robert Dodge[1], Zachary Eyde[2], and Giulia Pedrielli[1]

[1]School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA
[2]Intel Corporation, Chandler, AZ, USA

ABSTRACT

Increasing global demand has led to calls for better methods of process improvement for semiconductor wafer manufacturing. Of these methods, digital twins have emerged as a natural extension of already existing simulation techniques. We argue that despite their extensive use in literature, the current tools used to construct semiconductor simulations are underdeveloped. Without a standardized tool to build these simulations, their modularity and capacity for growth are heavily limited. In this paper, we propose and implement a library of classes in the Python language designed to build on top of the already existing SimPy library. These classes are designed to automatically handle specific common logical features of semiconductor burn-in processes. This design allows users to easily create modular, adaptable, digital twin-ready simulations. Preliminary results demonstrate the library's efficacy in predicting against benchmark data provided by the Intel Corporation and encourage further development.

1 INTRODUCTION

With the increasing demands for semiconductors to power automotive, HPC and AI applications, advanced methods for improving facility output and efficiency are needed to meet the demand. Prior research suggests that the use of a Digital Twin (DT) is an effective method to achieve this increase in productivity and equipment utilization in semiconductor fabrication facilities (Sivasubramanian et al. 2023). Although semiconductor wafer fabrication has received important research focus and contributions, limited research has been conducted on DTs, particularly inline process DTs in semiconductor Assembly and Testing (A/T). Key challenges in utilizing DT in the semiconductor A/T are: (i) large resource commitments to generate/sustain models, (ii) computational speed for real-time decision making, (iii) domain experience on process and equipment, (iv) and lack of access to data and/or physical twin for bi-directional data flow implementation. Despite these challenges, we argue that inline process DTs hold significant untapped potential in semiconductor A/T processes. Specifically, we argue that a DT of a semiconductor A/T process can be leveraged to improve (i) the purchasing and allocation of limited resources (e.g., test cells, pick heads) and (ii) mid-process decision-making (e.g., dispatching, maintenance scheduling). Furthermore, we propose integrating novel machine learning (ML) techniques within a DT. In particular, we aim to explore the potential of using a trained ML model as a decision-making agent within a DT.

In this paper, we focus on establishing the foundational framework for such a DT and setting the stage for future research involving ML methodologies. To this end, we present the modular Python library that will be used to construct the simulation engine that will power the inline DT in future work. In addition, we present an example discrete event simulation of a piece of burn-in test equipment used by Intel constructed using our library. We have created common classes of components such as robotic arms and testing sites which automatically handle the internal logic of A/T testing processes to allow the end-user to focus on the overall simulation performance and policy creation. For example, the user will not have to code how units, TBOTs, and test sites interact, but rather focus on connecting these components together through testing policies and resource requests. With this Python library, industry partners can quickly and easily develop new A/T simulation models of existing equipment without requiring extensive simulation expertise. The paper has the following structure: A review of the current literature is presented in Section 2, Section 4 explains the approach to address the problem highlighted in Section 3. The development of the modular Python library and methods used are explained in Section 5, with Section 6 showing the results of the validation performed on a discrete event simulation model of the Intel Burn-In module created using our library. Finally, Section 7 provides conclusions and potential future work.

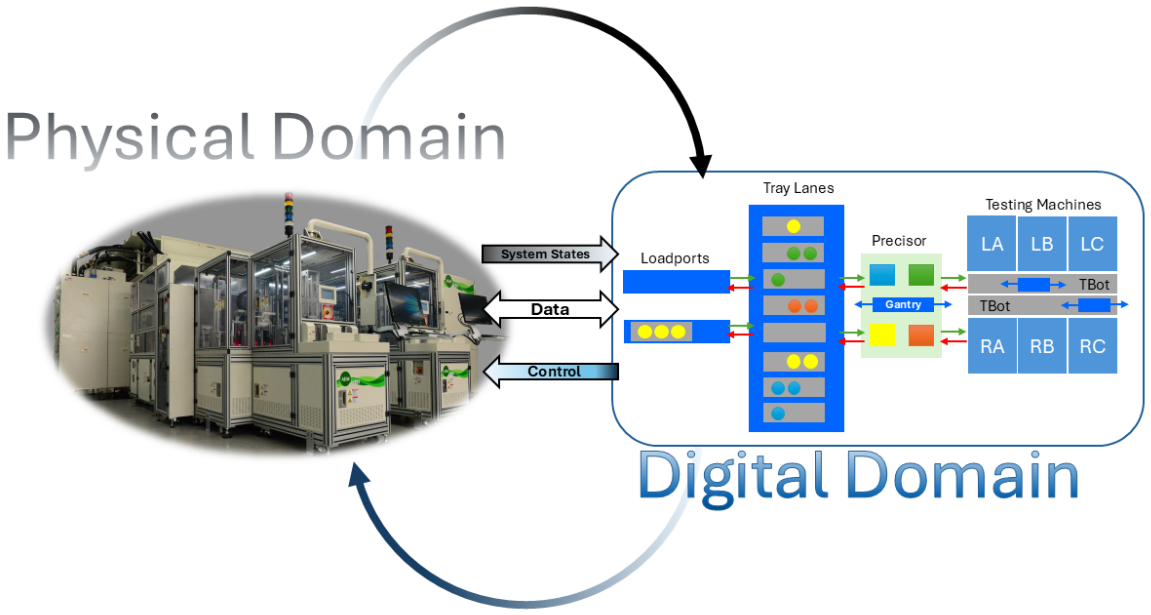


Figure 1: Conceptual design for a Digital Twin of the Intel Burn-In test equipment.

2 LITERATURE REVIEW

Pre-existing literature regarding simulation of semiconductor manufacturing is extensive. As such, we will present examples of semiconductor simulations used in prior literature with a focus on identifying the purpose of the simulation as well as the tool used to create it. Ahn et al. (2019) broadly divides the semiconductor manufacturing process into two sub-processes, a front-end photolithography process and a back-end A/T process. Our literature review will follow a similar structure presenting photolithography simulations first, followed by A/T simulations.

Beginning with the more commonly examined photolithography sub-process, simulation literature primarily focuses on scheduling and reticle management. Hickie (1999), De Díaz et al. (2005), and Park et al. (1999) utilize a simulation model of a wafer fabrication photolithography process created in FORTRAN to examine the impact of factors such as forecasting and product homogeneity on the overall performance of the system. Hung and Chen (1998) examine the performance of several common dispatching rules using a simulation of a photoligography system coded in the C++ language. Li et al. (2014) on the other hand, propose an entirely new dispatching rule for photolithography processes. This rule is then implemented and evaluated using a simulation model created in the real-time scheduling simulation platform developed by Shanghai Jiao Tong University. Siebert et al. (2018) implement a photolithography lot dispatching policy that iteratively optimizes lot selection based on current state of the system.

While photolithography simulation literature has a rich history dating back 50 years (Mack 2005), simulation with respect to the back-end subprocess is comparatively new. One of the first examples of simulation used in semiconductor A/T is seen in the work by Sivakumar (1999). In this paper, the authors construct a discrete event simulation model of a semiconductor back-end process using TestSim/X$^{\text{tm}}$ to perform a cycle time optimization experiment. Weigert et al. (2009) on the other hand, utilize a discrete event simulation model in MODELLER to construct an optimal production schedule using heuristic methods. A more recent example of simulation within A/T simulation can be seen in the work by Chiu et al. (2023). In this paper, the authors implement a novel simulation-optimization algorithm for minimizing the cycle time of a predetermined list of products. Kwon et al. (2024) on the other hand, utilize reinforcement learning optimization with a simulation model of a semiconductor burn-in process to solve a scheduling problem.

Despite the extensive history of semiconductor simulation, the tools used to create these simulations are relatively underdeveloped. Khemiri et al. (2021) notes that despite their widespread use, a generic wafer fabrication simulation tool is non-existent. A common trend across all papers presented here is a lack of standardization regarding the simulation tool used. Tools used range from pre-existing models used by a partner company to commercial software such as Arena or AnyLogic to custom implemented models using a coding language such as C++. Khemiri et al. (2021) specifically note that this lack of a standardized tool for semiconductor simulation modeling results in a high level of difficulty when creating or expanding models. Suggesting that the logical complexity of semiconductor simulations places a very high time sink on the act of creating one. Beyond time commitment, they also note that such implementation requires a high level of knowledge regarding both the process and modeling tool. Furthermore, since no standardized tool exists, model makers are required to manually implement the components and their interactions. To solve this, Khemiri et al. (2021) propose a generic simulation model for a semiconductor photolithography process that is capable of solving these shortcomings while still maintaining the efficacy of a custom model. While effective, the model produced by Khemiri et al. (2021) lacks support for the creation of back-end semiconductor simulations. As such, we propose the creation of a library within the Python coding language that can be used to produce accurate, DT ready, semiconductor A/T simulations quickly and modularity, while still preserving the customization and adaptability of a custom built model.

3 PROBLEM DESCRIPTION

A major component of the DT framework of our research is the simulation engine. This component is the core of the virtual representation of the physical equipment and is held to an exacting standard. Our stakeholder identified two primary problems that our DT needed to solve. First, the DT needs to be able to quickly identify approximately optimal decisions with respect to lot dispatching and tool changeovers. Second, the DT needs to be able to serve as a guide for equipment requisition by providing process performance estimates that can be used to predict the number of required tools given a demand forecast. Therefore, we have outlined several attributes that we wish to design our simulation around. To start, the computational speed must be high enough to keep pace with the physical asset. While immediate response times are not needed for uses such as equipment requisition forecasting (decisions made months to years in advance), tool change decisions and lot dispatching decisions require fast responses, (minutes for the former and seconds for the latter) as they are performed continuously at runtime. As we intend for our simulation to provide support online, having slow simulation times runs the risk of slowing down the physical half of the system as these live decisions are made. Second, the DT must have sufficiently high fidelity such that it is able to replicate policies down to individual component level. The closer a DT can replicate the behavior of the physical asset, the better predictions it is able to make. Additionally, we wish for the DT to update in real-time when discrepancies are detected between virtual and physical representations.

During investigation of possible approaches for the simulation engine, current modeling solutions of Intel test equipment were evaluated. Although current Intel approaches are well suited for capacity analysis and planning activities, these approaches do not meet all the requirements listed above. For instance, the A/T capacity models capture the macro-level performance of the equipment run quickly but do not capture the intra-equipment dependencies needed for a digital twin implementation. Other emulation-type solutions exist. However, these emulators are required to run in real-time to capture the SW interactions with the programmable logic controls (PLC). The simulation of a single scenario could take multiple hours to complete; violating requirement of fast computational speed for a DT. For this reason, we determined that an alternative solution approach was needed.

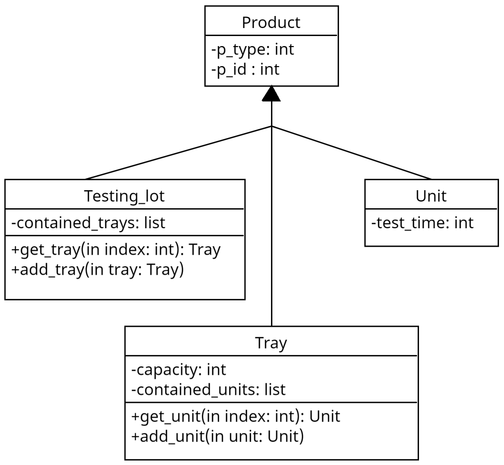


Figure 2: UML diagram for product classes implemented in our library.

4 PROPOSED APPROACH

With the problem and stakeholder needs identified, we propose the creation of a library of classes that can be used to design digital twins and standard discrete-event simulation models for semiconductor testing processes. To accomplish this, we have chosen to build on top of the existing SimPy library (SimPy 2002) in the Python language.

Our package makes use of the SimPy DES library offered within Python, thus guaranteeing minimal development efforts and complementarity/interoperability with any existing Python simulation packages with minimal integration efforts. That said, while SimPy offers a great deal of customization with respect to process logic, its implementation of resources lacks features commonly needed for semiconductor testing systems. Specifically, how devices under test (commonly referred to as DUTs or units) interact with resources within the simulation. First, we identified that SimPy lacked accommodation for multiple nonidentical resources with a shared queue, a common occurrence in semiconductor manufacturing. Another feature SimPy lacks native support for, is part-type restrictions for resources. Lastly, SimPy resources lack built-in support for batching (e.g., a test cell waiting for all of its sockets to be full before beginning testing).

These features were identified as important for our library due to being common functions of a semiconductor test process, while also being cumbersome to implement using the base SimPy library. Therefore, our library aims to supplement the SimPy library with additional classes and methods that can be used in conjunction with the base SimPy library to accommodate for these features. To this end, we have implemented two families of classes designed to work within the SimPy framework. These families cover the product side of the burn-in process (i.e., lots, trays, and units) and the resource side (i.e., test cells and robotic arms). The implementation and usage of these classes is discussed in Section 4.2.

4.1 Policies

Beyond the features outlined above, another shortcoming of the SimPy package is that the logic of SimPy base resources is more primitive than what is required by semiconductor manufacturing. Standard resource requests are served according to a first in, first out (FIFO) policy and all resources are treated as identical.

SimPy does offer advanced resources in the *FilterStore* and the *PriorityResource* classes, but these classes are still insufficient for our application. *FilterStore* resources can only filter items by binary categories while *PriorityResources* only classify priority when the request is created, not when a request is served. What if a user wanted to prioritize requests by a LIFO policy? What if the state of the model changes drastically after the request is created? Using the default SimPy package, situations such as these require custom logic coded into the process functions.

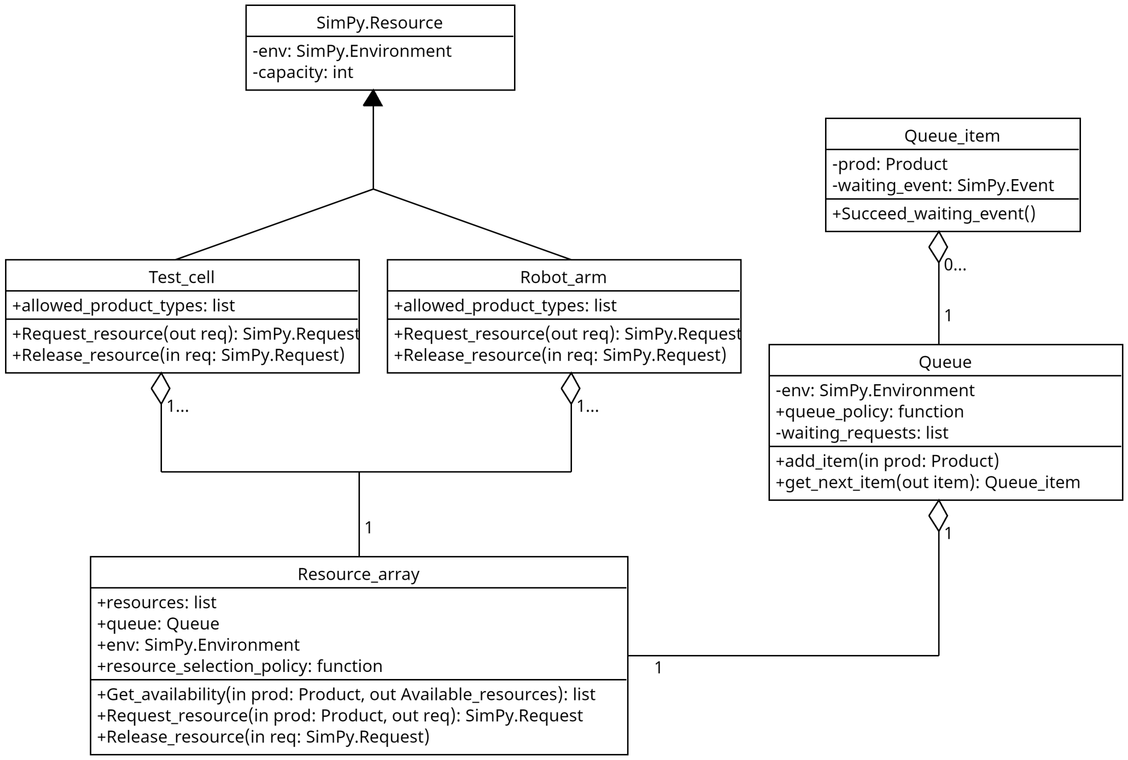
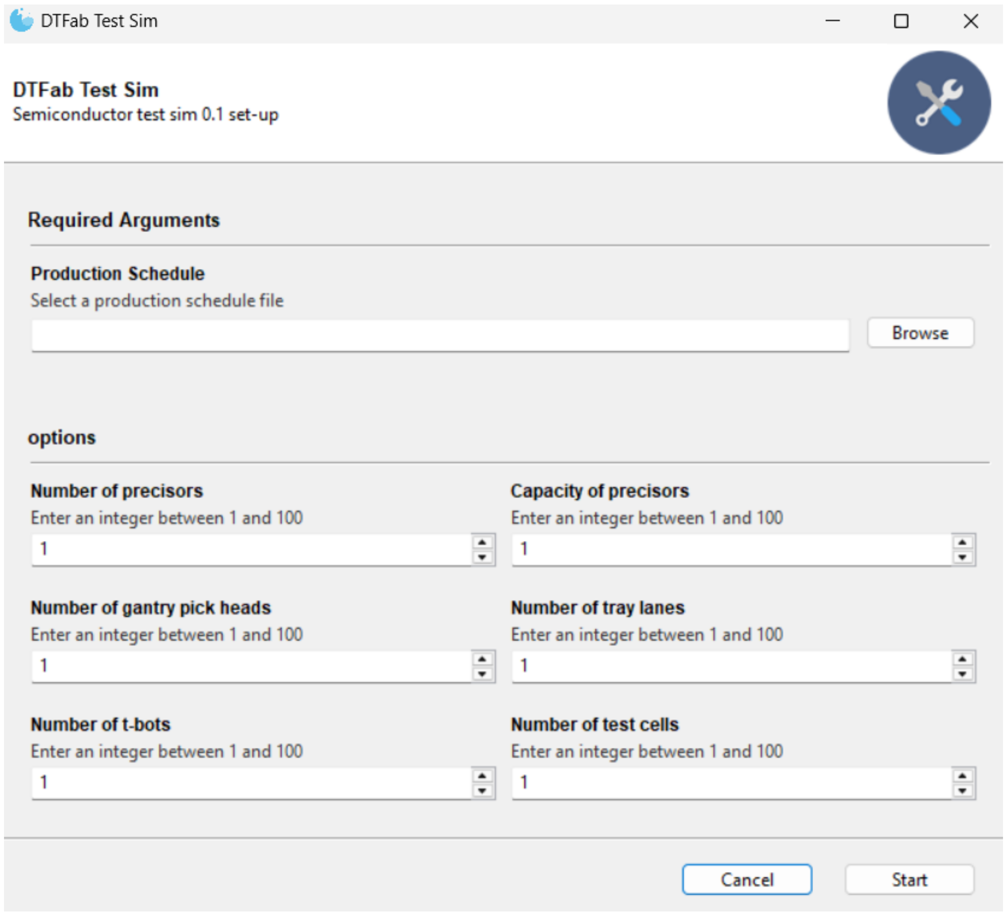

SimPy.Resource
-env: SimPy.Environment
-capacity: int
Queue_item
-prod: Product
-waiting_event: SimPy.Event
+Succeed_waiting_event()
0...
1
Test_cell
+allowed_product_types: list
+Request_resource(out req): SimPy.Request
+Release_resource(in req: SimPy.Request)
Robot_arm
+allowed_product_types: list
+Request_resource(out req): SimPy.Request
+Release_resource(in req: SimPy.Request)
1...
1...
1
Queue
-env: SimPy.Environment
+queue_policy: function
-waiting_requests: list
+add_item(in prod: Product)
+get_next_item(out item): Queue_item
1
Resource_array
+resources: list
+queue: Queue
+env: SimPy.Environment
+resource_selection_policy: function
+Get_availability(in prod: Product, out Available_resources): list
+Request_resource(in prod: Product, out req): SimPy.Request
+Release_resource(in req: SimPy.Request)
1

Figure 3: UML diagram for resource classes defined in our library. Note that the SimPy Resource block is incomplete and only shows the most relevant attributes of the class.

Therefore, to accommodate for complex policies such as these, we have designed each *Resource_array* and *Queue* object (see Section 4.2) to have an associated *Policy* attribute that determines how the object prioritizes requests and resources. Letting x be the current state of the system, we define these policies as some function $u(x)$ that returns a decision regarding which request the queue will dispatch next or which available resource will serve the next request. In summary, these policies serve as a modular way to guide behavior on which product should be serviced next and which resource should service it. By implementing policies in this manner, users of our package are able to create policies that directly replicate the behavior of the real system without needing to code the logic directly into the process function. Beyond increased flexibility, as these policies are isolated functions, this implementation allows a user to compare the behavior of the system under different policies without extensively rewriting the logic of the system.

4.2 Class Implementation

The classes in our package can be broadly divided into products and resources. Beginning with the product side, we have defined classes for *Lots*, *Trays*, and *Units*. Each of these classes is designed to carry the necessary attributes that inform the policy decisions within the simulation and inherit a *Product* superclass. The UML class diagram for these classes is shown in Figure 2.


DTFab Test Sim

DTFab Test Sim
Semiconductor test sim 0.1 set-up

Required Arguments

Production Schedule
Select a production schedule file
Browse

options

Number of precisors
Enter an integer between 1 and 100
1

Capacity of precisors
Enter an integer between 1 and 100
1

Number of gantry pick heads
Enter an integer between 1 and 100
1

Number of tray lanes
Enter an integer between 1 and 100
1

Number of t-bots
Enter an integer between 1 and 100
1

Number of test cells
Enter an integer between 1 and 100
1

Figure 4: GUI created for user input.

With respect to the resource classes, we have implemented 5 new classes to be used in SimPy simulations. These are the *Test_cell*, *Robot_arm*, *Queue*, *Queue_item*, and *Resource_array* classes. The *Resource_array* class is designed to replicate the behavior of parallel resources (e.g., groups of test cells) and is the primary class that end users will interact with after object creation. Regarding attributes, each instance of the *Resource_array* class possesses a list of resources available to it (either *Robot_arm* or *Test_cell* objects), a *Queue* object that handles its requests, and a *Policy* object (see Section 4.1) that determines how requests are allocated to its available resources. Methods of the *Resource_array* class are designed to mimic the behavior of the default SimPy *Resource* class with respect to how resources are requested and released. When a process wants to request a resource from a *Resource_array*, the *Request_resource* method is used. This method takes a *Product* as input and adds a *Queue_item* containing the passed *Product* to the *Resource_array's* associated *Queue*. *Products* in queue are dispatched according to the *Queue's Policy* attribute. Once the *Product* has been dispatched and allocated a resource according to the *Resource_array's Policy*, the method returns the SimPy *Request* object. Once the process has finished with the resource, the *Release_resource* method is called, passing the *Request* object returned by the prior *Request_resource* method. The UML diagram for the resource classes are shown in Figure 3.

4.3 Parameter Input

To allow an end user to design simulation runs using our tool, we have additionally provided a GUI that prompts the user for parameters and a production schedule. Shown below in Figure 4 is a sample GUI with input parameters expected for a standard burn-in process.

```python
def start(env):
        #Establish initial SimPy processes.
        for i in units:
            env.process(example_process(i))
        yield env.timeout(1)

def example_process(unit):
    #Request a test cell from the resource array.
    req = yield env.process(test_cells.Request_resource(unit))
    #Time out the process for the duration of the test once a test cell is assigned.
    yield env.timeout(unit.test_time)
    #Release the resource once testing is complete.
    env.process(test_cells.Release_resource(req))
```

Figure 5: Example of a simple simulation process created using our library. In this example, units are tested in a FIFO order by a group of 3 identical test cells. The object creation code for this example is shown below in Figure 6.

```python
#Create SimPy enviornment variable
env = simpy.Environment()

#Create unit objects
units = []
for i in range(5):
    new_unit = Unit(p_type = 0, p_id = i+1, t_time = 100)
    units.append(new_unit)

#Create 3 test cells that can test one unit of type 0 at a time, with a FIFO queue,
    and are prioritized equally.
test_cells_queue = Queue(env, queue_policy = resource_fifo)
test_cells_resources = []
for i in range(3):
    new_test_cell = Test_cell(env,allowed_product_types = [0],capacity= 1)
    test_cells_resources.append(new_test_cell)
test_cells = Resource_array(env, resources = test_cells_resources, queue_object =
    test_cells_queue, resource_selection_policy = Sample_policy)

#Start process
env.process(start(env))
env.run()
```

Figure 6: Code segment demonstrating the creation of the objects that are used by the process defined in Figure 5.

4.4 Example Implementation

Shown below is an example of how the classes in our package can be used to create simulation process functions. In this example, 5 units are created and tested by a group of 3 test cells each capable of testing 1 unit at a time. This example can be seen in Figures 5 and 6, the former of which defines the simulation process functions while the latter creates the objects and starts the simulation.

5 EXPERIMENTAL METHODOLOGY

To evaluate the efficacy of our proposed solution, we used a semiconductor burn-in test process used by our partner Intel as a base for a simulation model created using SimPy and our library. This model served as a test bed for our library to ensure that it was functional for both a developer interested in producing a model as well as an end user.

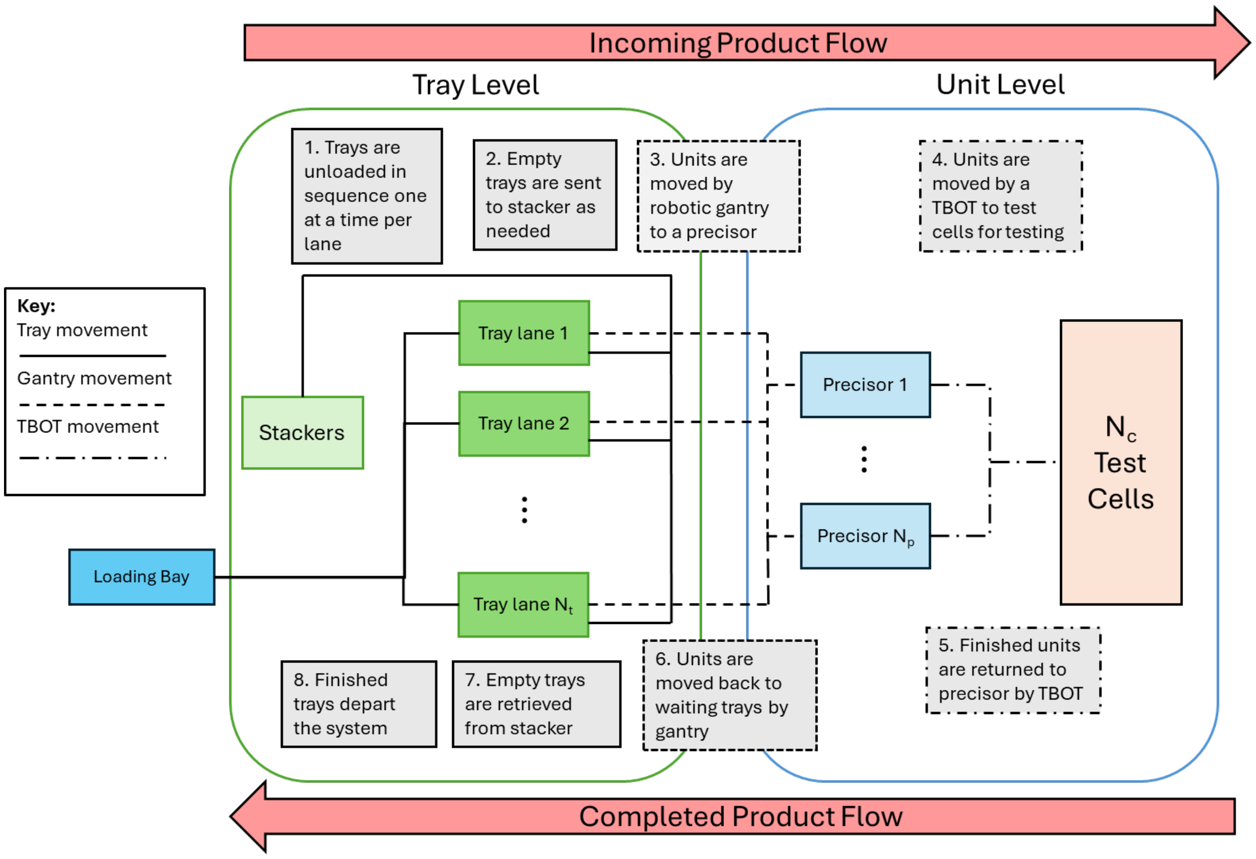

Incoming Product Flow
Tray Level
Unit Level
1. Trays are unloaded in sequence one at a time per lane
2. Empty trays are sent to stacker as needed
3. Units are moved by robotic gantry to a precisor
4. Units are moved by a TBOT to test cells for testing
Key:
Tray movement
Gantry movement
TBOT movement
Stackers
Tray lane 1
Tray lane 2
Tray lane N_t
Loading Bay
Precisor 1
Precisor N_p
N_c Test Cells
5. Finished units are returned to precisor by TBOT
6. Units are moved back to waiting trays by gantry
7. Empty trays are retrieved from stacker
8. Finished trays depart the system
Completed Product Flow

Figure 7: Diagram showing how units and trays move through the burn in process. Steps numbered in ascending order in sequence they are completed. Movements are indicated by lines with the type indicating which part of the system is performing the movement.

5.1 The Burn-In Process

The process in question is the semiconductor standard burn-in process. In back-end semiconductor testing, DUTs are batched into groupings, called lots, with homogeneous product types that are processed sequentially. Each lot is comprised of a predetermined number of trays, each containing a set number of individual units depending on product type. During burn-in testing, the general process flow consists of lots entering the system where individual units are unloaded from trays by a robotic gantry and placed into one of N_p waiting precisors. From the precisor, the waiting units are batched and moved by a TBOT to an available test cell where the quality test is performed. Once a unit is tested, it moves through the same steps in reverse. It is first returned to the precisor by the TBOT before the gantry places it back into a waiting tray. The only exception to this is when a unit is selected for retest with probability p. In that case, after the unit fails its initial test, it is placed into the precisor where it once again requests a test cell and the TBOT (causing additional test cycles to be needed). A unit can only be tested a maximum of two times. Units are removed from the process after the second test regardless of outcome. Therefore, we have broken down this system into three broad processes, a lot process, a tray process, and a unit process. These processes are defined as a series of operations performed by the system's resources. Namely, a series of N_t tray lanes, a set of N_p precisors, a robotic gantry, a TBOT, and a group of N_c test cells. The general flow of these processes is shown in Figure 7, while a visualization of the physical layout of the system is shown in Figure 8.

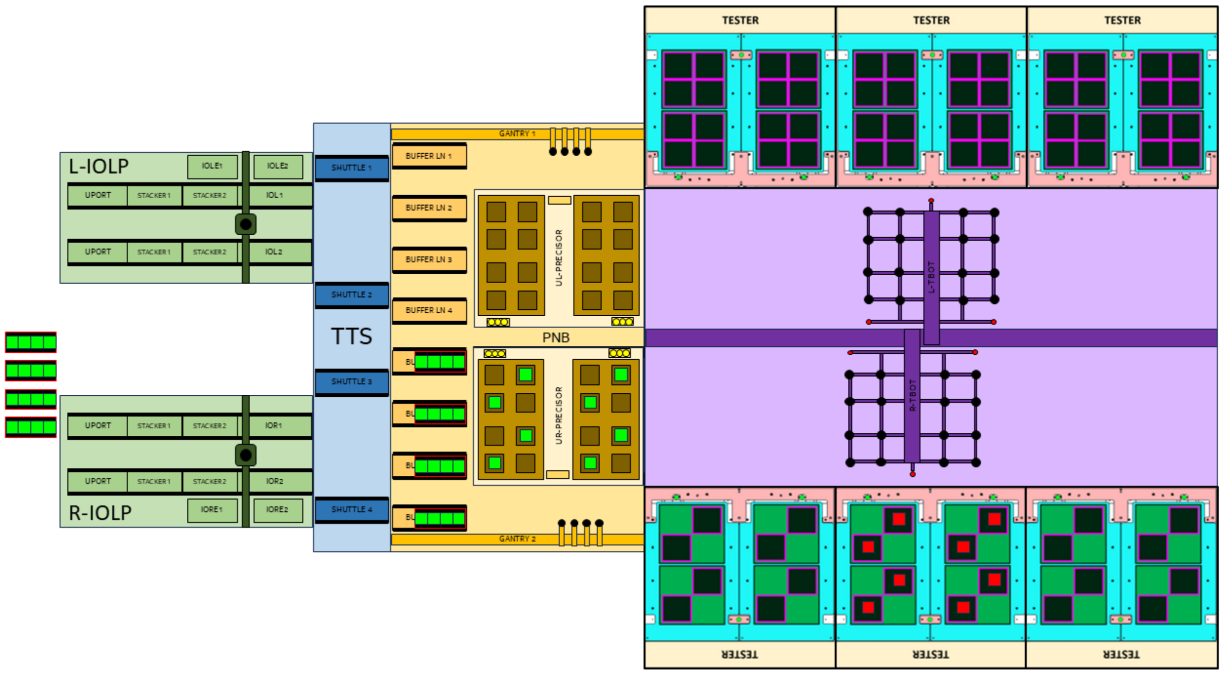


Figure 8: Physical layout of the Intel burn-in test equipment that the simulation is based on.

5.2 Assumptions

Before we began validation, we established a set of assumptions regarding what our model currently would not consider. First, this validation model would not consider hardware failures. Although these are a reality of any manufacturing process, we first wanted to establish that the model produced realistic results before complicating the process with hardware failures. Second, the model would not consider the exact spatial relation of the test cells, precisor plates, and tray lanes. I.e., with respect to the time taken for movement, each test cell is considered the same distance from the precisor and each precisor is the same distance from each tray lane. In reality, the time taken for the gantry or TBOT to move between these resources is dependent on exactly which resources the trip is between. E.g., a movement from the precisor to test cell 30 may take longer than a movement from the precisor to test cell 1. Finally, we assumed that the duration of time for each test cycle was deterministic, leaving the retest process (see 5.1) as the only source of stochasticity within these experiments. This implementation was selected as we specifically wanted to guarantee the logic within the model behaved properly under varying retest parameters. Although this is a limitation of the validation model, the variations caused by test duration and robot movements are marginal compared to the additional test cycles introduced by retesting. Ensuring that the retest process was accurately replicated was therefore a higher priority.

5.3 Notation and Experimental Set-up

To ensure that the model was functioning properly, we began by testing its performance by running simulations of single lots selected from pre-existing production data. To do this, we reviewed the production data that Intel had recently collected for benchmarking and selected several lots for evaluation. Our selection criteria for this aimed to select lots that were from a variety of product types with a range of tray densities, test times, and unit counts. Additionally, we also wanted to select lots that had no hardware failures or other anomalies that would skew the data. The parameters for the selected lots are shown in Table 1. Note that the values shown in the table represent placeholders to communicate where the values are shared across lots while maintaining information privacy.

Table 1: Lots from production data selected for validation. The same value across multiple lots is interpreted as the lots sharing parameter value. E.g., α and β have the same tray density x_1.

Lot	Product Type	Tray Density (units/tray)	Tray Count	Unit Test Time (s)	Retest Probability	Test Cell Count
α	1	x_1	y_1	z_1	p_1	n^t_1
β	1	x_1	y_1	z_1	p_2	n^t_2
γ	2	x_2	y_2	z_2	p_3	n^t_3
δ	3	x_3	y_3	z_3	p_4	n^t_4
ε	3	x_3	y_3	z_3	p_5	n^t_4

We then devised a series of experiments in which each lot was simulated from a cold start using our simulation tool for 100 replications. Each of these experiments was set up such that the simulation parameters matched the configuration of the test machine from the benchmark data, ensuring a valid comparison.

6 RESULTS AND DISCUSSION

These experiments were performed on an AMD Ryzen 7950x 16-core desktop processor. Each set of replications took 6.4 s, 6.2 s, 5.2 s, 19.8 s, and 20.3 s for experiments 1-5, respectively. To evaluate the performance of the model, we chose to measure the cycle time of each lot and the number of trips the TBOT took to complete the lot. Of these two, cycle time was chosen because it gives a concise comparison to the performance of the real system. TBOT trips on the other hand, was chosen as a means to gauge whether internal logic of the system was behaving as expected. Given that resources and delays in the system match the real system, large deviations in the number of trips that the TBOT takes would indicate that something is wrong with the internal logic. Additionally, TBOT trips serves as a substitute measure of the overall utilization of the test cells since TBOT trips are directly correlated with the number of test cycles. The results for these experiments are shown in Figure 9

With respect to the cycle time, the median of the predicted cycle times for each of these lots are all within 7% of the target number. The highest median percent error being 6.6% and the lowest being 1.1%. The number of TBOT trips is predicted with similar accuracy. Across all 5 experiments, The median predicted number deviates from the actual by 7.8% at most and 0% at the lowest. While the majority of our replications accurately predicted the performance of the lot, as shown in the plot, a small percentage of our data points fell quite far from median prediction.

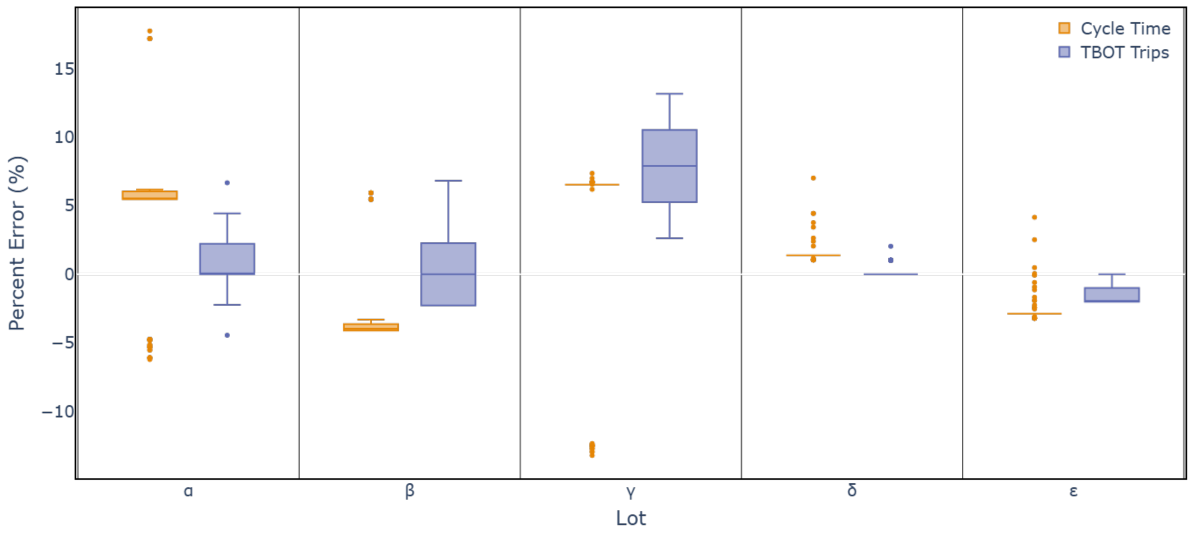


Figure 9: Box plot showing the percentage error of the predictions produced by our simulation model for each lot. Note that a negative percent error means that the result deviated that many percentage points below the target. E.g., -2% error means the prediction was 2 percentage points below the target.

Our investigation revealed that these deviations are due to the retest process. As noted by our stakeholder, individual units requiring retest can have a large impact on the overall cycle time, especially in cases where the overall number of test cycles is low. Meaning that the variance shown in our results is in-line with the expected variance of the retest process and not indicative of a logical flaw of the model. In terms of run-time, the speed of model evaluations represents a significant improvement over existing tools implemented by the stakeholder. That said, while the times are sufficient for evaluations such as equipment purchasing and tool maintenance (minutes, hours, or years in advance), these times are still too slow for in-the-loop DT decision making needed for lot dispatching (seconds in advance).

7 CONCLUSION AND FUTURE WORK

In this paper we presented the initial work on a Python library designed to allow for easier creation of modular, DT ready, simulations of semiconductor burn-in processes. This library is designed build on the features of the already existing SimPy library, preserving its strengths, while addressing its flaws.

While the results presented in this paper are promising, more validation is certainly required. The validation experiments presented in this paper only evaluated the performance of our model in simulating a single lot. In reality, it is common for a burn-in process to have several lots queued back to back, often with different product types. Additionally, as discussed in Section 5, our current model does not consider hardware failures or stochastic test times. A more complete model would consider both of these in its predictions. As such, removing these assumptions as well as validating our model against situations where multiple lots are cascaded back to back make natural extensions of the work presented in this paper.

Beyond further validation, we would also like to expand our library to handle tool management systems using the same modular policy system implemented in our resources and queues. This would allow models created using our library to capture the full dynamics of a running semiconductor burn-in system in extended simulations, paving the way for the planned full DT integration with ML techniques.

REFERENCES

Ahn, G., M. Park, Y.-J. Park, and S. Hur. 2019. "Interactive Q-Learning Approach for Pick-and-Place Optimization of the Die Attach Process in the Semiconductor Industry". *Mathematical Problems in Engineering* 2019(1):4602052.

Chiu, C.-C., C.-M. Lai, and C.-M. Chen. 2023. "An Evolutionary Simulation-Optimization Approach for the Problem of Order Allocation with Flexible Splitting Rule in Semiconductor Assembly". *Applied Intelligence* 53(3):2593–2615.

De Díaz, S. L. M., J. W. Fowler, M. E. Pfund, G. T. Mackulak, and M. Hickie. 2005. "Evaluating the Impacts of Reticle Requirements in Semiconductor Wafer Fabrication". *IEEE Transactions on Semiconductor Manufacturing* 18(4):622–632.

Hickie, M. 1999. *Improving Photolithography Reticle Management with Network Modeling and Discrete Event Simulation*. Ph. D. thesis, Arizona State University.

Hung, Y.-F., and I.-R. Chen. 1998. "A Simulation Study of Dispatch Rules for Reducing Flow Times in Semiconductor Wafer Fabrication". *Production Planning & Control* 9(7):714–722.

Khemiri, A., C. Yugma, and S. Dauzère-Pérès. 2021. "Towards a generic semiconductor manufacturing simulation model". In *2021 Winter Simulation Conference (WSC)*, 1–10 https://doi.org/10.1109/WSC52266.2021.9715349.

Kwon, S.-W., W.-J. Oh, S.-H. Ahn, H.-S. Lee, H. Lee, and I.-B. Park. 2024. "Scheduling of Wafer Burn-In Test Process Using Simulation and Reinforcement Learning". *Journal of the Semiconductor & Display Technology* 23(2):107–113.

Li, Y., Z. Jiang, and W. Jia. 2014. "An Integrated Release and Dispatch Policy for Semiconductor Wafer Fabrication". *International Journal of Production Research* 52(8):2275–2292.

Mack, C. A. 2005. "Lithography Simulation in Semiconductor Manufacturing". *Advanced Microlithography Technologies* 5645:63–83.

Park, S., J. Fowler, M. Carlyle, and M. Hickie. 1999. "Assessment of Potential Gains in Productivity due to Proactive Reticle Management Using Discrete Event Simulation". In *1999 Winter Simultaion Conference (WSC)*, 856–864 https://doi.org/10.1109/wsc.1999.823298.

Siebert, M., K. Bartlett, H. Kim, S. Ahmed, J. Lee, D. Nazzal, *et al*. 2018. "Lot Targeting and Lot Dispatching Decision Policies for Semiconductor Manufacturing: Optimisation Under Uncertainty with Simulation Validation". *International Journal of Production Research* 56(1-2):629–641.

SimPy 2002. "SimPy: Simulation framework in Python". https://simpy.readthedocs.io/en/latest/contents.html. Accessed 1st April 2025.

Sivakumar, A. I. 1999. "Optimization of a Cycle Time and Utilization in Semiconductor Test Manufacturing Using Simulation Based, On-Line, Near-Real-Time Scheduling System". In *1999 Winter Simulation Conference (WSC)*, 727–735 https://doi.org/10.1109/WSC.1999.823204.

Sivasubramanian, C. K., R. Dodge, A. Ramani, D. Bayba, M. Janakiram, E. Butcher, *et al*. 2023. "DTFab: A Digital Twin based Approach for Optimal Reticle Management in Semiconductor Photolithography". *Journal of Systems Science and Systems Engineering* 32(3):320–351.

Weigert, G., A. Klemmt, and S. Horn. 2009. "Design and Validation of Heuristic Algorithms for Simulation-Based Scheduling of a Semiconductor Backend Facility". *International Journal of Production Research* 47(8):2165–2184.

AUTHOR BIOGRAPHIES

ROBERT DODGE is a is a researcher at Arizona State University working under Dr. Giulia Pedrielli. His research focuses on stochastic simulation and optimization within semiconductor manufacturing processes. His email address is rwdodge@asu.edu.

GIULIA PEDRIELLI is currently Associate professor for the School of Computing and Augmented Intelligence (SCAI) at Arizona State University. She graduated from the Department of Mechanical Engineering of Politecnico di Milano. Giulia develops her research in design and analysis of random algorithms for global optimization, with focus on improving finite time performance and scalability of these approaches. Her research is funded by the NSF, DHS, DARPA, Intel, Lockheed Martin. Her email address is giulia.pedrielli@asu.edu and her website is https://www.gpedriel.com/.

ZACHARY EYDE is currently a Test Research and Development engineer in the Intel Sort Test Technology Development department. He holds Master's degrees in Mechanical and Electrical Engineering from University of Arizona, as well as a Master's degree in Industrial Engineering from Arizona State University. His work at Intel involves improving process and efficiency of the Test modules from New Production Introduction to High-Volume Manufacturing. His email address is zach.s.eyde@intel.com.