Implementing ManPy, a Semantic-free Open-source Discrete Event Simulation Package, in a Job Shop

Oladipupo Olaitana, John Geraghty*, Paul Young#, Georgios Dagkakis#, Cathal Heavey#, Martin Bayer*, Jerome Perrin#, Sebastien Robind

*Enterprise Process Research Centre, School of Mechanical and Manufacturing Engineering, Dublin City University, Dublin, Ireland
#Leotech Rapid Prototyping and Tooling GmbH, Leonberg, Hifiingen, Stuttgart, Germany
Nexedi SA, Marcq-en-Baroeul, France

* Corresponding author. Tel.: +353-1-700-7739. E-mail address: john.geraghty@dcu.ie

Abstract

The cost of acquiring commercial simulation packages is considerably high and this might explain why organisations are often reluctant to make further investments on training and retraining of employees on simulation modelling. Ironically, the level of benefit derived from simulation is highly dependent on experimental, analytical and statistical skills of the user. These cost and skill requirements make simulation an unattractive decision support tool to SMEs and small multinational organisations. Proposed in this study is ManPy, a semantic-free open-source approach to discrete event simulation (DES), such that users with different levels of skills can derive considerable benefits from simulation. ManPy eradicates the high investments required for simulation modelling by making it possible for low skilled users to benefit from readily available generic modelling objects which are contributed to an open source platform by highly skilled simulation practitioners, statisticians and academics. Another benefit of ManPy is the ability to integrate with other enterprise planning tools for system knowledge extraction and real time simulation input data. Some of these benefits are demonstrated through the implementation of ManPy in a SME that specialises in rapid prototyping and rapid tooling.

1. Introduction

This work is a part of a larger project, titled simulation based application of Decision support in Real-time for Efficient Agile Manufacturing (DREAM), conceived to fill the skill void that often hinders the application of DES tools by industries, especially SMEs. The overall aim of the DREAM project is to develop an open source simulation platform to aid real time decision support in manufacturing. In this particular work, ManPy, the simulation modelling component of DREAM, is implemented in a rapid prototyping and tooling company. The extensibility of its generic modelling objects is demonstrated by extending them to meet specific modelling needs for the job shop system. Its results were then compared against those of well-established Commercial-Off-The-Shelf (COTS) and Open Source (OS) simulation packages. A modular simulation optimisation structure that allows the determination of the best schedule is also presented. The optimisation is based on an Ant Colony Optimisation (ACO) algorithm for determining the best combination of scheduling rules. The graphical user interface (GUI) features are also presented in relation to how they aid basic users of the software to achieve most of the modelling functions that would otherwise require considerable programming effort, especially in OS packages. The GUI tool is designed to be extensible in order to facilitate its currency with the modelling objects, when they are extended.
2. Background

Considerable efforts have been channelled in the past at providing OS alternatives to COTS packages. However, very few of them have lasted long enough to establish themselves as credible alternatives. The following section reviews some of the probable factors that could have been responsible for the inability of OS packages to challenge COTS ones. An overview of simulation modelling and optimisation of job shop systems is also presented.

2.1. Simulation modelling packages

COTS simulation software packages have contributed to making DES widely applicable, considerably reducing the programming effort of the modeller. In most of these packages the user’s experience is enhanced by modelling, execution, experimentation and interfacing tools [1]. However, COTS DES tools are often proven much more cumbersome to use than they originally promise. The requirement of coding may be, at least in simple cases, reduced, but still deep modelling skills are required in order to model all the peculiarities of an individual system [2]. Additionally, such software frequently becomes inflexible when it comes to modelling large and complex systems [3]. In order to allow customisation, most of them offer an internal programming language (for example, “visual logic” for Simult, “SimTalk” for Plant Simulation or “ModL” for ExtendSim). Nevertheless, this requires the practice of a new coding tool, which may be simplified, but on the other hand not widely used by programmers other than the users of the COTS product. Therefore, support is more difficult to be found.

High cost of COTS DES software is also a significant factor that deters organisations from adopting DES in their decision making processes. In addition to the substantial licensing cost, there are other associated costs to cover additional plug-ins and maintenance upgrades [4]. As a result, reusability of DES objects and models [5] is also hampered, because in case the license is not renewed then all past work is rendered useless.

The above problems indicate that there is room for an OS alternative in DES software. Firstly, this would eliminate the licensing costs, since OS software is freely distributed, and also ensure the reusability of models. Speed can also be improved, taking into consideration that computationally expensive DES based experiments can be deployed into different computers in the form of a parallel and web based simulation [6]. This is much more applicable in OS, because the installation of COTS software in a cluster of PCs may incur significant extra licensing costs.

However, most OS DES projects seem to have failed in gaining attention and remaining active [7]. This can be partly attributed to the fact that most of them require that the user is also an experienced programmer in order to build and execute even a simple model. These projects often underestimate the need to differentiate between programmers, modellers and end users. Furthermore, simplified installation procedures, and clearly defined project scope and licensing are other critical success factors which are underestimated in OS projects [8]. Another reason may be that DES requires a wide range of expertise which expert programmers/developers may not concurrently have at their disposal to develop a comprehensive package.

The above paragraph does not imply that there are not very interesting OS DES projects in the market. One example is OMNeT++, which is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators [9]. The OMNeT++ project seems to be one of the most established OS DES projects, which has already yielded numerous publications and retains a highly active OS community. Another interesting project is JaamSim [3] which was launched as open-source during the last year. JaamSim is heavily focused on providing a 3D graphical environment, influenced by lessons learnt from the computer games industry. It is not focused on manufacturing but it is generic enough to facilitate the modelling of such systems. Another reason for its attractiveness is that it is very easy to install and run, providing a standalone executable of less than 8 MBs.

Among the 23 OS DES projects reviewed was SimPy, which stands for “Simulation in Python”, and according to its authors, it is "a process-based discrete-event simulation framework based on standard Python". SimPy offers two major classes: The Process models an object which evolves in simulated time, for example a customer or a server. It is not a formal Python thread but it works as one. A Resource simulates a list where objects can be queued. SimPy also provides a short, but efficient, set of functions that help in handling the logic of the system in the simulated time.

SimPy is relatively active as a project and it is licensed under LGPL, which makes it feasible to be also used in proprietary projects. The requirements of DREAM dictate that the platform will be OS and also provide the possibility of being expanded to a tailored and closed solution for a company. Therefore, a LGPL licence, or other more permissive ones, was identified as being essential for DREAM needs.

2.2. Job shop modelling and optimisation

In a job shop, the uniqueness of jobs, and the fact that they are attributed to a specific customer order, makes it important to determine the best order in which jobs competing for resources are prioritised, so that customer orders are fulfilled in the earliest possible time. The process of determining the best order in which jobs should be processed is NP-hard because there is usually a wide range of possibilities in job scheduling. Scheduling rules are probably the most frequently applied techniques to solving shop scheduling problems, owing to their ease of implementation and low time complexity. Rules are used to assign processing priorities to jobs based on specific attributes that could relate to a job’s due date, material properties, order arrival date or its required processing time. Each scheduling rule represents a potential solution and, as such, they all have to be evaluated individually, possibly through simulation, in order to identify the one that delivers the optimal results. However, given that
single static rules often yield limited solutions, multiple rules are sometimes combined using evolutionary search techniques to identify the best rule combination for a single system. Combinatorial optimisation techniques, such as Tabu Search, Genetic Algorithm, Simulated Annealing and ACO, have also been separately used to find solutions to scheduling.

These search techniques require speedy, on-the-fly simulation of multiple scenarios, and by extension the ability to migrate from one rule’s scenario to the other using numerical or logical expressions to alter the model logic. For instance, ACO requires the creation and evaluation of successive generation of ants, with each of the ants corresponding to a potential solution scenario. ACO tries to emulate the information exchange mechanism that ants use in searching for the shortest path between their nest and location of food [10], where the information exchange is the pheromone laid on the travelled path. ACO has also been applied in solving travelling salesman, flow scheduling and assignment problems.

3. Implementation of ManPy in a job shop

3.1. ManPy overview

ManPy stands for “Manufacturing in Python” and is a new simulation engine implemented for the needs of DREAM. The work of ManPy development started inside DREAM, using insight gained from the requirements of the project. Like DREAM, ManPy is an ongoing project and it is regularly updated with DREAM research findings. Recently, the project was launched as public in GitHub, along with the rest of DREAM related code. In the root of this repository is a documentation of ManPy, where examples are also given to get users started.

The scope of the project is to provide simulation modellers with a collection of OS DES objects that can be connected like "black boxes" in order to form a model. This collection is designed to be expandable by providing means for developers to customise existing objects and add completely new objects to the list. As its name implies, ManPy’s initial focus has been the manufacturing domain, but the architecture is generic, so that users can make objects and models in other fields such as logistics, services, and generally in any procedure where flow of entities between objects exists.

ManPy architecture defines five generic classes of objects. These are:

- **CoreObject**: all the stations which are resident in the model. These can be servers or buffers of any type and also entry and exit points.
- **ObjectInterruption**: all the objects that can affect the availability of another object. For example failures, scheduled breaks, shifts etc.
- **Entity**: all transient objects that get processed by or wait in CoreObjects and they are not resident in a model. For example parts in a production line, customers in a shop, calls in a call centre etc.
- **ObjectResource**: all the resources that might be necessary for certain operation of a CoreObject. For example repairman, operator, electric power etc.
- **Auxiliary**: auxiliary scripts that are needed for different simulation functionalities. For example a main script to create the objects and run the simulation, global variables, methods for random number generation etc.

In order to achieve the interconnection of objects, which may potentially be built by remote developers in an OS environment, all ManPy classes have to follow a well-defined naming convention for methods and attributes. The efficiency of Python’s type casting plays a very important role in this. These elements define how DES objects such as stations, servers or buffers exchange entities such as materials or customers. Using the same names, an object can be connected with any other type of object. For example, a machine can exchange parts with another object no matter if this is another machine or a queue. This philosophy allows expandability since completely new objects may implement their versions of the methods and then be smoothly incorporated into the platform. Also, customised objects can inherit from existing ones and alter just part of its logic, e.g. how the object calculates its processing time can be changed by overriding one or more methods. More information on ManPy’s objects and methods are given in [11], and a more detailed and technical description is given in ManPy documentation on GitHub. The following are some of the methods relevant to the system studied in this work:

- **canAccept**: returns true if the object is in a state to receive an Entity.
- **haveToDispose**: returns true if the object is in a state to give an Entity.
- **canAcceptAndIsRequested**: returns true only when both conditions are satisfied, i.e. the object is in the state of accepting an Entity and also another object is requesting to give one Entity to it. Only when this method returns true the main simulation logic of the object is started.
- **getEntity**: gets the Entity from another CoreObject.
- **getGiverObject**: returns the giver in an Entity transaction.
- **getReceiverObject**: returns the receiver in an Entity transaction.
- **calculateProcessingTime**: calculates the processing time of the object.
- **sortEntities**: sorts the Entities that the object holds according to some rule.

3.2. Description of case study system

A benefit of the approach being followed in the development of ManPy is the availability of an array of industrial case studies within the DREAM project. These case studies pose multifaceted problems, and as such would ensure that the final version of ManPy is capable of solving a wide variety of problems through its generic simulation modelling objects. The case study for this particular work originates from a rapid prototyping and tooling company that manufactures a variety of products having varying processing
and routing requirements. The company combines mould design and manufacture with actual production of a small series of prototypes. The operations involved in the system are Engineering Design, CNC programming, Milling, Electrical Discharge Machining (EDM) and Injection moulding. Injection moulding is for the production of small series, so some customer orders for moulds do not necessarily get to this stage before they are delivered to the customer. The company’s multi-skilled personnel make it possible to practice a project based approach to production, such that a single project manager deals with the processing of an order from start to finish. The flexibility enjoyed from the practice of this approach is, however, tempered with the need to have a means of centrally coordinating the schedule of each one of the project managers.

The company has no previous simulation experience, but they appreciate the benefit that could be derived from using simulation modelling as a tool to evaluate different production scheduling plans before deciding on a final production programme. The need to evaluate alternative schedules also arises when rush orders are received from customers and there is need to quickly determine the overall impact this would have on pre-existing orders. Moreover, similar impact evaluation is necessary when resources become constrained, either through breakdown of machines or absenteeism of project managers, and there is a need to forecast the impact it would have on previous customer delivery commitments. Essentially, the main objective of the company, as is the case with most job shop systems, is to be able to instantly capture and all the information relating to orders that have to be processed and resources that are available for the processing, and simulate different scheduling scenarios in order to identify the best schedule with which to minimise delay. A scenario would correspond to a set of decisions taken on the jobs to process next at each of the machines when two or more jobs are waiting to be processed during simulation. It also covers the optimal allocation of the company’s pool of resources.

In this study, eight scheduling rules that are related to the attributes of the customer orders have been implemented and combined using ACO optimisation. The following are the implemented rules and their job attributes of preference:

- **SPT**: selects the job that requires the shortest processing time
- **LPT**: selects the job that requires the longest processing time
- **MS**: selects the job that has the shortest slack time
- **SRR**: selects the job that won’t require machine setup based on its material similarity to the previously processed job
- **ERD**: selects the job with the earliest order date
- **EDD**: selects the job with the earliest due date
- **WINQ**: selects the job with the shortest queue at its next destination
- **PCO**: selects the next job based on user specified priority

ManPy is able to accommodate these attributes as simulation inputs, along with other relevant information. This is one of the features, which are not readily available in most COTS and SimPy, but have been implemented in ManPy.

Information about a job is first instantiated when customer order arrives to the system, and it is continuously updated as the job progresses through the system. At any point in time when simulation is to be conducted, information about the status of jobs currently being processed in the system and job(s) to be introduced are required as inputs. Some of the inputs include order date, job due date, production route, processing time estimates, and operator and raw material requirements.

### 3.3. Extension of ManPy with job shop oriented objects

The DREAM platform is intended to be enhanced with objects that can model the specifics of different manufacturing disciplines, so that simulation practitioners, even those with the lowest level of skills, can easily find suitable blocks to represent their system entities. For a job shop system, it was necessary to have modelling objects that are able to achieve the following system representations:

- Job entities that are capable of holding job shop related information such as order arrival time, due dates, customer priority, routing and processing times.
- Resource objects, e.g. machine, operators, that are capable of reading and interpreting job attributes for application of scheduling rules.

The initial objects that were built in ManPy were mostly focused on the modelling of flow type production lines. So these objects had static predecessors that they could give entities to, and successors that they could obtain entities from. However, for a job shop system, custom objects were needed to override this logic. The logic of the system dictates that dedicated buffers are connected statically with stations, but stations can give entities to whichever buffer of the system or to the exit of the system, depending solely on the attributes of a job. Also the job itself indicates what the processing time in every station is.

The first object that had to be implemented was a new type of an Entity named Job. The main difference of Job and other types of Entities that already existed in ManPy is that it holds in its attributes the route that it has to follow and the processing time at each machine that it visits.

Then, there were three customised CoreObject types to be implemented:

- **MachineJobShop**: inherits from Machine and overrides the logic of methods such as or haveToDispose in order to be able to give to every CoreObject in the model. Also, it overrides canAccept and canAcceptAndIsRequested in order to be able to receive from every CoreObject in the model. The next CoreObject is read by the Entity’s attributes and this is done in getEntity. Finally, it overrides calculateProcessingTime in order to calculate the processing time according to the Entity’s attributes.
• QueueJobShop: inherits from Queue but overrides its logic in the same fashion as MachineJobShop does with Machine so that it can act in a job shop environment
• ExitJobShop: inherits from Exit but overrides the logic of methods, such as canAcceptAndIsRequested, in order to be able to receive from every CoreObject.

It is important to note that, in implementing the above objects, no change in the already existing objects was required; their code was treated as a black box. Nevertheless, there were two changes that were considered preferable to be implemented in the parent objects. These were:

• sortEntities method was updated in Queue so that it can implement different standard scheduling rules.
• priority, orderDate and dueDate where added as optional attributes in the Entity class.

ManPy is relatively new as a project and such changes are expected as more cases are considered. However, the change in already implemented objects was more of a strategic decision. The same results would have been achieved if the changes were only incorporated in the new objects. Also it should be noted, that in all of those changes, no SimPy related code had to be altered. It was merely Python code and this is essential because Python is a generic language that holds a huge community of users, thus finding documentation and support is much easier in Python, than in specific packages such as SimPy or ManPy.

4. Modelling and optimisation in ManPy

The complexity associated with the conventional approach of embedding data and logic into simulation models has been overcome by ManPy’s modular architecture. The conventional approach often renders simulation models inflexible to system dynamics. ManPy’s modular structure makes it possible for even basic users to be able to modify the systems logic through numerical inputs without having to delve into the intricacies of system modelling and optimisation.

**Fig. 1: Defining job shop system entities in ManPy.**

| J1 = Job('J1', 'Job1', ['Q1', 0], ['M1', 1], ['Q2', 0], ['M2', 7], ['Q3', 0], ['M3', 1], ['Q4', 0], ['M4', 7], ['E', 0], priority=1, dueDate=100), M1 = MachineJobShop('M1', 'Milling'), Q1 = QueueJobShop('Q1', Queue=1, capacity=Infinity, schedulingRule='FIFO') |

where J1 is a Job that has to go through a route from Machine M1, to M2 to M3 and M4 before finally proceeding to the Exit, E, of the system.

M1 = MachineJobShop('M1', 'Milling')

where M1 is a milling machine and is referenced as M1 elsewhere in the model.

Q1 = QueueJobShop('Q1', Queue=1, capacity=Infinity, schedulingRule='FIFO')

where Q1 is an infinite capacity queue with a First in First out (FIFO) scheduling rule.

**Fig. 2: ACO Algorithm for Scheduling Rule Combination.**

(A.) Initialize Optimiser

(B.) Loop for number of Ant generations

a) Read list of alternative rules for each machine

i. Create ant by randomly selecting rule per machine from their respective rules list

ii. Cancel ant if it has been previously simulated, and return to step i

iii. Set rules for the machines

iv. Simulate job shop, applying corresponding rule for sorting jobs queued for a machine

v. At end of simulation, calculate average delay of jobs

vi. Record average delay for ant

b) Rank ants based on average delay, and choose best performing n ants, i.e. ants with least average delay.

c) Update list of rule options, i.e. pheromone, with best performing n ants’ machine-rule combinations

c) Update list of rule options, i.e. pheromone, with best performing n ants’ machine-rule combinations

(C.) END simulation and present best performing schedules

For each of the machines, there is a list of alternative rules from which ants are created by selecting a rule from this list and assigning it to a machine. For example, each of the machines involved in the company could have the following rule options:

**Engineering Design** = [‘EDD’, ‘WINQ’, ‘LPT’, ‘SPT’, ‘SRR’, ‘ERD’, ‘PCO’]

**CNC Programming** = [‘EDD’, ‘WINQ’, ‘LPT’, ‘SPT’, ‘SRR’, ‘ERD’, ‘PCO’]

**Milling** = [‘EDD’, ‘WINQ’, ‘LPT’, ‘SPT’, ‘SRR’, ‘ERD’, ‘PCO’]

**EDM** = [‘EDD’, ‘WINQ’, ‘LPT’, ‘SPT’, ‘SRR’, ‘ERD’, ‘PCO’]

**Injection Moulding** = [‘EDD’, ‘WINQ’, ‘LPT’, ‘SPT’, ‘SRR’, ‘ERD’, ‘PCO’]

An ant created from the above machine rule options lists would be in the form:
4.1. Verification and validation of results

The verification and validation of applications that involve simulation modelling is a complex process. Validation is concerned with ensuring that a model accurately represents the real system in terms of the performance metrics of interest, while verification is aimed at ascertaining that a model is correctly translated into a computer program. Both are iterative processes that take place continuously throughout the lifecycle of a simulation study.

The approach used for validation in this study was to first conduct a workshop at the facility of the company. This provided the chance to define the scope of the study and gain a first understanding of the system and its peculiarities. After this, regular contacts through calls and meetings, a Request for Information (RFI) document and a questionnaire took place. The information obtained from these workshops was then used to determine the objects that would be required in ManPy’s job shop library. For the purpose of tractability during the verification phase, ManPy was first used to model a simplified case that exhibits most elements of the larger system. The correctness of its core modelling objects and their logic were then verified by comparing the outputs from its event trace feature with those from similar features in ExtendSim and SimPy. The trace feature shows all arrival and departure times for jobs at all machines, as well as the eventual completion times of jobs.

Once the correctness of the modelling objects had been established, a second workshop was conducted to obtain deeper technical details about the system. The outcomes of this workshop have also been used to improve the logic of the modelling objects. Other aspects of the system that are not exactly related to modelling objects have also been considered; the inputs and the outputs of the conceptual model have been demonstrated to the company to ensure that the outputs and their modes of presentation conformed to general expectations when analysing such systems.

Future steps will include conducting a structured walk through with company users, followed by the use of real life historical data as simulation inputs. The aims of these steps will be to validate the interaction of the components and to verify that the overall simulation model correctly represents the information provided during industrial workshops. The company does not have adequate records for this purpose, so templates have been developed to record experimental scenarios that will contain current system situations and the eventual outcomes.

4.2. Graphical User Interface (GUI)

The GUI is a vital component for user friendliness of software applications, and as such is a major component of the DREAM platform. However, it has been intentionally designed as a detachable module within the platform in order to avoid limiting the platform’s extensibility and openness. The GUI features are based on OS tools and are easily extensible so that users are able to customise them to match newly created modelling objects, as well as to meet specific company requirements. Specific requirements for companies could relate to features such as general simulation inputs, outputs and modelling object dialogue boxes.

A customised input interface allows block representation of machines, the production route of new jobs and those that are already in progress. As shown in Figure 3, the modelling objects are represented by graphical editor component, which enables the modification of objects and entering of objects attributes, in a similar way to dialogue boxes found in COTS. Most of the modelling functions achieved through ManPy codes can also be achieved in a simpler fashion using the GUI. For instance, machine objects can be interconnected to establish the routing of job objects.

![GUI Block representation of ManPy objects.](image)

However, this is not permissible in a job shop because of the uniqueness of the production route of successive jobs. Therefore, instead of the use of connectors, routings for the jobs are specified as GUI spreadsheet inputs. Other inputs entered for each of the jobs on the spreadsheet are job name, ID, order date, due date, priority and required material. The graphical user interface converts the inputs into JavaScript Object Notation (JSON) format which ManPy is able to interpret and execute during the run of simulation. A JSON format is similar to the format in which the inputs are specified without the use of the GUI. Since the entered inputs still have to undergo conversion within the GUI module into code like format before being transferred to ManPy, it is obvious that the essence of the GUI input feature is to aid users with entering simulation inputs, and this has been adequately considered in its design.

The GUI output will provide information about scheduling options and their respective degrees of attainment of objectives. The schedule information will be presented as a Gantt chart showing the timeline of progress of jobs through the different machines, as depicted in Figure 4. This provides a means by which senior managers can centrally coordinate the schedules of project managers.
The GUI output will also feature an utilisation bar chart, shown in Figure 5, to represent the utilisation of system resources. This is intended to make other underlying problems, which may be responsible for delays in meeting customer demands, visible to decision makers. Colour coding is also used to highlight outcomes that deviate from the desired target, e.g. when possible completion time would extend beyond due date. This is also important because of the uniqueness of jobs and customers from which they originated; performance of each one of the jobs is shown in the schedules and jobs that are likely to be delayed are highlighted for the user’s attention.

5. Conclusions and future work

As one of the industrial cases being researched in the DREAM project, a job shop system has been studied in this work for the purpose of implementing relevant modelling objects in ManPy. ManPy’s generic objects were extended to model components of a job shop and the model outputs were verified and validated against those of SimPy and well established COTS. Some customisations were also applied to DREAM platform’s GUI features to correspond to relevant job shop inputs and its required outputs. A modular structure that allows the incorporation of optimisation algorithms was also demonstrated. An ACO algorithm was implemented to identify the best combination of rules.

At the current phase of the wider DREAM project, simplified scenarios from the different cases studies have only been considered. A long term goal, for when the scenarios become larger and more complex, is to be able to deploy optimisation experiments over cluster of clients for parallel simulation. This approach has been proved to enable wider solution search space and also increase the speed at which an optimal solution is reached [12].

The DREAM platform will also have a Knowledge Extraction tool for automatic extraction of simulation inputs from different data sources. The K.E. tool will work with most MES, ERP and other data storage applications. A data conversion protocol to convert CMSD data to ManPy readable format has already been implemented in the K.E. tool [13], while other conversion protocols are also planned.

These future additions will ensure the DREAM platform has the advantages of being generic and easily extensible, owing to the size of the Python programming language’s community. It will also have the ability to interact with other manufacturing applications for its real time data needs.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 314364.

References

[1] Pidd, M., Carvalho, A. Simulation software: not the same yesterday, today or forever. Journal of Simulation; 2006, 1:1, p. 7-20.
[2] Kuljis, J., Paul, R-J. An appraisal of web-based simulation: whither we wander? Simulation Practice and Theory; 2001, 9:1-2, p. 37-54.
[3] King, D-H., Harvey, S-H. Discrete-event Simulation in Java: A Practitioner’s Experience. In: Proceedings of the Conference on Grand Challenges in Modeling & Simulation, Society for Modeling & Simulation, International Vim, CA; 2010, p. 436-441.
[4] McLean, C., Leong, S. The expanding role of simulation in future manufacturing. In: Peters BA, Smith JS, Madeiros DJ and Rohrer MW (eds). Proceedings of the Winter Simulation Conference, Arlington, Virginia; 2001, 2, p. 1478-1486.
[5] Paul, R-J., Taylor, S-J-E. What use is model reuse: Is there a crook at the end of the rainbow? In: Yucesan E, Chen CH, Snowdon JL and Charnes JM (eds). Proceedings of the Winter Simulation Conference, Washington DC; 2002, 1, p. 648-652.
[6] Byrne, J., Haavey, C., Byrne, P-J. A review of Web-based simulation and supporting tools. Simulation Modelling Practice and Theory; 2010, 18:3, p. 253-276.
[7] Dagkakis, G., Heavey, C., Papadopoulos C-T. A Review of Open Source Discrete Event Simulation Software, In: Proceedings of the 9th Conference on Stochastic Models of Manufacturing and Service Operations, Kloster Seeon, Germany; 2013, p. 27-35.
[8] Fogel, K. Producing Open Source Software: How to Run a Successful Free Software Project; 2010 [online]. http://producingoss.com/en/producingoss.pdf (Accessed 08/02/2013).
[9] Varga, A., Hornig, R. An Overview of the OMNeT++ Simulation Environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications SIMUTools, Marseille, France; 2008, 60:1.
[10] Colorni, A., Dorigo, M., Maniezzo, V. Distributed Optimization by Ant Colonies. In: Proceedings of European Conference on Artificial Life ECAL 91, Paris, France; 1991, p. 134-142.
[11] Dagkakis, G., et al. ManPy: An Open-Source Layer of DES Manufacturing Objects Implemented in SimPy, In: Proceedings of 8th EUROSIM Congress on Modelling and Simulation, Cardiff, Wales; 2013, p. 357-363.
[12] Syberfeldt, A., et al. A web-based platform for the simulation-optimization of industrial problems. Computers & Industrial Engineering; 2013, 64/4, p. 987-998.
[13] Barlas, P., Dagkakis, G., Heavey, C. A Prototype Integration of MANPY with CMSD, MANPY with CMSD, In: Proceedings of the 27th European Simulation and Modelling Conference - ESM 2013, Lancaster, UK; 2013.
[14] OMNeT++ http://www.omnetpp.org/
[15] JaamSim https://github.com/AusencoSimulation/JaamSim
[16] SimPy http://simpy.readthedocs.org/en/latest/
[17] DREAM (ManPy) https://github.com/nexedi/dream