MODELLING, SIMULATION AND OPTIMIZATION OF LOGISTIC SYSTEMS

Pasquale Legato [a], Daniel Gullì [b], Roberto Trunfio [c]

[a] DEIS, Università della Calabria, Via P. Bucci 41C, 87036, Rende (CS), Italy
[b] CESIC – NEC Italia S.r.l., Via P.Bucci 22B, 87036, Rende (CS), Italy
[c] CESIC – NEC Italia S.r.l., Via P.Bucci 22B, 87036, Rende (CS), Italy

[a] legato@deis.unical.it, [b] daniel.gullì@eu.nec.com, [c] roberto.trunfio@eu.nec.com

ABSTRACT
In the last two decades, discrete simulation has become the most powerful tool in modelling logistic systems in a dynamic stochastic environment. An open challenge is trigged by the need to devise ways of developing easy-to-read and expressive visual modelling paradigms. Most modelling paradigms, as Event Graphs and Petri Nets, currently adopted mainly in an academic context, have been generally supplanted by simulation packages in real system applications. Nevertheless, our belief is that these are more expressive and powerful in modelling logistic systems. We rely on an innovative visual modelling paradigm based on the process interaction conceptual framework and on the holistic modelling approach, which we are presenting in this paper. The paper focuses on the way of representing processes, resources and entities that compose our simulation modelling paradigm. Modelling capabilities of our modelling paradigm are compared to those of Event Graphs and Petri Nets within a real case study.

Keywords: modelling, simulation, optimization, logistics, container terminal

1. INTRODUCTION

In a stochastic dynamic environment, discrete event simulation (DES) models are well capable of representing the behaviour of large and complex logistic systems. Thus DES models are widely adopted as planning and control tools to estimate system performances under uncertainty and conduct scenario analysis.

Modern, commercial DES simulation packages based on a "point & click" logic – that is what Pidd (2004) calls VIMS – are devoted to minimize the system modelling effort by hiding the behavior of the components adopted to construct the model, but at a loss of model readability, understanding and customization. In opposition, a *modelling paradigm* (MP) is a visual modelling approach based on a formalism designed on a worldview which requires a lot of modelling ability, and which provides superior representational capabilities.

In the past, a lot of interesting and powerful DES modelling paradigms (MPs) have been developed based on the three classical worldviews (or *conceptual frameworks*), i.e. *Event Scheduling* (ES), *Activity Scanning* (AS) and *Process Interaction* (PI) (Derrick et al. 1989). The most notable MPs developed using these conceptual frameworks are respectively: *Event Graphs* (Schruben 1983), *Petri Nets* (Petri 1962) and *Hierarchical Control Flow Graphs* (Fritz and Sargent 1995). Derrick et al. stated that a worldview is an underlying structure and organization of ideas which constitute the outline and basic frame that guide a modeller in representing a system in the form of a model.

The PI conceptual framework better suits the needs for model readability and understanding. In fact, while ES and AS modelling approaches are based respectively on events and activities, the PI is based on the concept of *process*. A process is a complex concept which represents a flow of events and activities through which a particular model object moves; therefore, in a model specification, it describes the life cycle of a model object. Whilst a model object moves through its process, it may experience certain delays and be hold in its movement. Thus, the first thing to do in the PI worldview is to identify all the model objects which are involved in the model specification. Subsequently, the modeller must specify the sequence of events and activities for each model object. For this reason, a process is often represented with a schematic representation known as *flow-chart*, where the events and activities that compose the process are the nodes of the chart. A flow-chart is usually represented as a directed graph. For each activity included in a process routine there is a stretch of simulated time (even null). The PI qualities are a moderate burden for the modeller, a high maintainability, an excellent natural representation capability. The most notable drawback is the high burden on execution, even if modern object-oriented approaches better fit the implementation needs of this conceptual framework, and the effort required for the developmental time, which is quite high.

Modelling a stochastic system using a MP or, in alternative, a commercial VIMS (e.g., GPSS and Arena) implies the use of a specific modelling approach. The two classical dichotomous modelling approaches are *reductionism* and *holism*. Reductionism relies on the

belief that a complex system may be decomposed into its constituent parts without any loss of information, predictive power or meaning (Pidd and Castro 1998). The totality of MPs and VIMS are developed using this modelling approach. Unfortunately reductionism does not consider that large and complex logistic systems cannot be decomposed into their constituent parts with the guarantee of all the above conditions. This statement can be practically proved by trying to analyse real systems, e.g. *supply chains* and *container terminals*: in the latter case, it is easy to verify that it is not possible to decompose the system into its constituent parts (logistic processes) and analyze them separately, because they are partially or entirely related: some significant loss of information, predictive power and meaning will occur. Pidd and Castro (op cit) have shown that the best approach for the management of a complex system is based on holism. Holism assumes that systems possess some properties that are meaningful only in the context of the whole and not in the parts (e.g., in a maritime container terminal, shuttle vehicles must be assigned to the quay cranes considering the whole work-load). Therefore, using a holistic approach to model a whole system achieves better results in terms of the replication of the real system behaviour. Thus, developing an MP or VIMS by using the holistic approach should be appropriate. However, considering that both alternatives are affected by an intrinsic difficulty in model understanding, which makes simulation models unpopular at the model end-users, here we propose an MP which provides: *i)* a high representational capability from the conceptual point of view, as well as *ii)* a common language for both modellers and end users from the model understanding sight.

We believe that a stimulating possibility is to define an innovative MP for modelling logistic systems under uncertainty using a holistic approach, having in mind the goal of developing a simulation based optimisation platform. Besides, the considerations about strong and weak points of the ES, AS and PI worldviews provided by Derrick et al. (1989) convince us that we must define our MP using the PI conceptual framework. Thus, we dedicate the next section describing our MP, based on a previous work (Legato and Trunfio 2007). Successively, to compare our MP with two different, successful MPs, EG and PN, we model the same real logistic process in a container terminal, showing the different way of modelling this reality by using these MPs. Finally, we discuss some specific issues when developing a friendly tool for the integration of optimisation techniques within simulation models.

2. WHOLISTIC DISCRETE EVENT SIMULATION WITH PROCESS INTERACTION

In this paper we propose a holistic MP for DES modelling based upon the PI, that we call *Holistic Modular Process* (HMP) simulation models. In the following, we define the main concepts of our simulation MP. In the subsequent section we illustrate its potentiality by modelling a typical logistic process that arises in a maritime container terminal and comparing it with the EG and PN models.

2.1. Holistic Modular Process Simulation Models

The MP proposed in this paper is aimed to be flexible and expressive in the modelling of complex systems. It tries to achieve three primary objectives: model readability, reusability and customizability.

Model readability is a property which allows a model to be simple-to-read for a non-modeller. This property has a special importance when top managers are directly involved in scenario analysis: in our MP readability is achieved by describing the components' behaviour within a simulation model by a sort of flow-chart. As for reusage property, our experience at the Gioia Tauro Container Terminal confirms the requirement that a specialised simulation tool has to be reused in some of its forms (model reuse, component reuse, function reuse and code scavenging). Model reuse, under calibration and repeated tuning, occurs as soon as traffic conditions change over time. Furthermore, component and function reuse are both required to give the operational manager the possibility of quickly implementing a first order model of some emerging situations, before the structured intervention of external expertise. According to our concept of holistic MP, we provide model reusability by means of hierarchical, modular model definition and redefinition of simulation parameters.

Model customizability is the base of an MP for the effective modelling of complex systems. It relies on the user-definition of process properties that allow describing uncommon situations, as it is the case when the modeller is asked to represent local, best practices in logistics organisation and management.

An outline of the HMP simulation models follows now. An HMP model includes a set of *model objects*, or *objects* for short. For each model object an inner and outer view is defined. The outer view is depicted as a box equipped of input and output ports (see Figure 1). The inner view depicts a sequence of activities and events. The sequence is also called the *model object process routine*, or simply *process*.

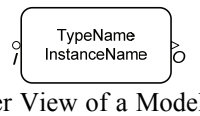


Figure 1: The Outer View of a Model Object in a HMP Simulation Model.

In a similar way as described for HCFG models, relationships between model objects are depicted using directed edges (*channels*). Model objects are equipped with *input* and *output ports* (depicted as shown in Figure 1) that can, eventually, be renamed to improve model readability. Two model objects are linked by means of only one channel from an output port of the first object to an input port of the second object and vice

versa, i.e. no more than two channels connect directly two model objects. Two model objects interact by message passing via channel, despite a channel has its own direction (the head of the edge is connected to an output port, while the tail to an input port). Multiple types of messages flow forward and backward along a channel, whilst *entities* (or *jobs*) of the simulation model can flow only through the proper channel direction.

In this way, by adopting a modular approach, an HMP simulation model is a net of model objects. Therefore, hierarchical modelling is pursued by coupling different processes and grouping the resulting net of model objects into a *sub-model* (which is depicted in a similar to a model object, as proposed in Figure 2). In this case, the sub-model can be used to be coupled together with other model objects or sub-models. A requirement when constructing sub-models is that at least one input or output port must be defined and linked to the inner model-objects, otherwise, the depicted model is a *super-model* or *final model*. The inner view of a sub-model is just another HMP model where some channels link the inner part to the outer part of the model through its boundaries.

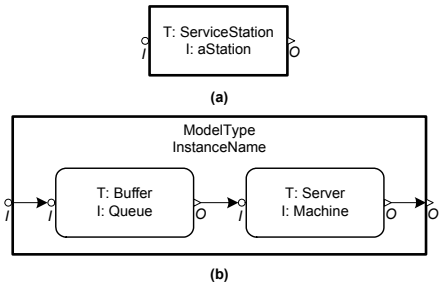


Figure 2: (a) The Outer View of a Sub-Model depicting a Service Station. (b) The Inner View of a Service Station, Composed by a Buffer and a Server.

As stated above, the inner view of a model object is a process, or rather a sequence of activities and events that define the model object behavior. The process is represented as a particular hierarchical flow-chart, called *Event-Activity Diagram* (EAD), where activities and events are nodes and edges fix the logical and temporal sequence between nodes (other node types will be introduced in the following). A process can be constructed hierarchically by grouping events and activities to compose a sub-process. Thus, a sub-process is an EAD itself. A sub-process we provide component reuse and model readability.

A detailed model definition is obtained by the EAD. An EAD provides the process definition. In fact, an EAD defines the structure of a process by means of a sort of flow-chart. Because of the possible use of sub-processes for the process definition, an EAD could be a hierarchical structure (composite nodes are expanded revealing a new EAD) and a hierarchical tree may be used to explore the EAD structure.

2.2. Model Objects

Following the PI worldview, the first step consists in identifying all the model objects which are involved in system modelling and detect all common features. Then, class-dependent features are described for the few classes of model objects introduced in our MP. The objects are contextual, so it is necessary to specify the model in which they are defined. The *model name* parameter is used to declare which model an object belongs to. This parameter serves two reasons: first of all, the holistic approach says that such objects can only be used in some contexts; furthermore, the use of sub-models could cause a lot of confusion, especially if a sub-model is exploded (i.e. deleting model boundaries) and the same object is used in a model and in its sub-models. Objects of the same type are identified by the *type name* parameter. If the model name is a parameter that depends on the model, the type name is a non-changeable parameter. Each instance of a certain type of object is also identified by the *instance name* parameter. The set composed by these three parameters univocally identifies a model object within an HMP simulation model.

There is at least another important parameter that allows managing heterogeneous objects, known as *category name*. The use of the category name allows us to make associations between objects that are apparently disjoined. A possible use can be seen in the development of a simulation package, in the packages for statistical analysis and optimization of system performance measures. As a matter of fact, in this context generic rules and algorithms can be defined over a class of mixed objects.

Model objects have a set of variables and data structures used to support the logical representation of the process behavior. These properties are not explicitly depicted and refer to the code implementation of each model object type. Model objects variables and data structures are accessed by a set of public functions which allow their manipulation. A function is called by message passing, or rather sending an explicit message to call a specific model object function.

Model objects are illustrated in detail in the following.

There are two basic classes of objects: *resources* and *resource managers*. As stated above, entities of the simulation model are depicted as messages that are able to envelop properties, data structures and other entities (e.g., a ship that carries thousands of containers loaded in different holds).

Resources are *active* or *passive* depending on their role in the simulation model. Passive resources are not depicted explicitly and are not able to execute action/events or process entities. Nevertheless, a passive resource is able to execute incoming requests and actions of other model objects (as declared above, a passive resource shows a set of public functions that can be used to manipulate the resource). Passive resources are generally managed by an active resource or a resource manager. Whenever a passive resource is

managed by a resource manager, active resources linked to the resource manager are able to overwork it only under the conditions specified by the resource manager. A passive resource must be managed by a resource manager if more than one resource may request it for use during a simulation (e.g., items storage into a shelf using forklifts). An active resource can possess passive resources and it can offer a service to one or more entities per time. It can also make queries to other objects to which it is linked by message passing via input/output ports. The behavior of an active resource is described using an EAD.

By means of resources one can only represent just a system governed by a few simple rules. As matter of fact, the need of modelling complex systems in a holistic approach leads us to introduce the resource managers. A resource manager is a high-level object, which is able to interact with a set of model objects (also heterogeneous). Resource managers can take decisions (e.g., solve a scheduling or an assignment problem, negotiating the use of a sub-system, etc.) by applying rules and policies and making queries to other model objects. They have free access to modify the behaviour of all the resources in their own model to which they are linked. In a holistic approach, as demonstrated by Pidd and Castro (op cit), the use of that kind of model objects avoids the explosion of object links and therefore it represents a more easy-to-use modelling tool.

2.3. Processes and Event-Activity Diagrams

The role of a process in model object specification is analysed here and process representation is shown. The interaction of processes during the simulation is briefly described.

According to the definition of process given in the PI worldview overview, a process is a series of temporally related events and activities. Usually flow-charts may be used to represent processes. In our flow-charting methodology, called *Event-Activity Diagram* (EAD), events and activities are nodes, while directed edges define one or more paths that can be covered by a process. Other useful elements compose a process, namely the *logical nodes*.

Activities and events are not intended to perform actions, but to show to non-modellers, in a friendly-way, how a process can work. The role of executing requests, performing actions and introducing time delays between activities and/or events is assigned to directed edges. Therefore, the use of a flow-charting graphical methodology to depict a process allows us to achieve at a good extent the readability objective. Bearing in mind the list of the process components, let us start an in-depth discussion about these components.

In our MP, activities are classified focusing on the simulation duration of the activity; hence activities are partitioned in *timed* and *instantaneous*. The first type of activities are those able to start operations at a simulated time instant and finish operations in a future simulated time instant. For instance, timed activities are those that perform operations characterized by a variable simulated time length, e.g. waiting or servicing activities. The second type refers to activities that start and end operations at the same simulated time instant, e.g. activities representing a choice or check by a resource or resource manager. Nevertheless, also timed activities can starts and ends operations at the same simulated time instant.

A process needs the specification of an initial activity that is enabled, or rather the activity that possesses the process checkpoint (more details about checkpoints are provided in the following). The initial activity is the activity from which the process starts when it becomes active (initial process state). In an EAD one or more activities per time can be enabled, i.e. when the process flow has been forked in different logical paths. The set of currently enabled activities is called the *process state*.

An event is a fact that forces one out of a set of possible changes of the current state. It may precede or follow an activity, thus representing something that is just happened or that is going to happen. If an event precedes an activity, then it is processed at the same simulated time of the activity start; if an event follows an activity, then it is processed at the same simulated time of the activity end.

As stated before, EADs have a hierarchical process structure. In fact, nodes are activities and events and even sub-processes. A sub-process is itself an EAD, therefore it can be zoomed revealing the included flow-chart. The EAD of a sub-process can refer to input and output ports of the including process, i.e. a sub-process is only a convenient arrangement for grouping an EAD sub-net and depicting it as a single node (this solution aims to improve model readability).

The different shapes for the main nodes of an EAD are depicted in Figure 3.

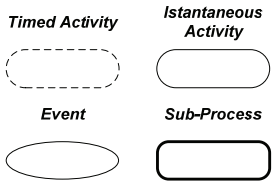


Figure 3: EAD Activities, Events and Sub-Processes Nodes.

The core of the process behavior is designed adding directed edges. Edges are used *i)* for message passing and the evaluation of conditions through *functions* call, and *ii)* for the introduction of timing into process activity flows. We classified edges in four types: *null edges* (or *true edges*), *inner edges*, *incoming edges*, *outgoing edges*. A null edge can be used only to connect events to activities or activities to activities. An inner edge connects an activity to another activity or to an event. These edges are depicted as arcs (see Figure 4). Incoming edges are used to depict incoming messages from an input/output port, while outgoing edges depict outgoing messages to an input/output port. Incoming edges connect only activities to activities or

activities to events, while outgoing edges also link events to activities. Edges may not have any node linked to the head, thus they work as a termination arc (i.e, the process becomes definitely terminated when one of these nodes is traversed).

Edges produce a transition from the current enabled node (connected to the edge tail) to the future enabled node (that is linked to the edge head). To avoid deadlock, edge tail and head must be different nodes. While the transition rule for a null edge is always true, for the other edge types the edge traversal is allowed only if certain conditions are met. To understand the nature of these conditions, we introduce the following edge functions: a *Timer* and a *Request*.

A Timer is a time function that generates a delay time using a distribution (e.g., exponential) and the appropriate set of parameters. Only inner and outgoing edges may have a Timer. Whenever an inner/outgoing edge is selected by an activity to be eventually traversed, the Timer generates the simulated delay time and an "alarm" is scheduled to warn the edge after the delay time. When an edge has been warned, it is allowed to enable its head node (i.e., the process state has been changed), or rather it is traversed.

The transition rule of an edge is composed by a set of Requests. A Request is a (public or private) function of a process that can be called to check conditions, to set up model objects and entities parameters and (in case) exchange entities among model objects. If the conditions are met, the function returns true; false, otherwise. A set of primary Requests can be used on the same edge by the formulation of a logical expression using boolean operators (*and*, *or*, *xor*) and negation (*not*), that we call the *primary rule*; moreover, the verification of the primary rule may depend on a set of conditioning Requests, or what we call the *conditioning rule*. Both conditioning and primary rules compose the transition rule in the following way: "primary rule | conditioning rule". Only if the conditioning rule is evaluated as true, then the primary rule is evaluated.

All the edge types (excepted null edges) may have a transition rule. A considerable difference exists between inner and incoming/outgoing edges. In fact, by using incoming/outgoing edges, a process can: *i)* make a call to a public function of an external process (*receiver process*) linked to an input or output port; *ii)* receive a call to its own public functions by an outer process (*sender process*) linked to an input or output port. In this way we enable communication among processes (e.g., exchanging entities and assigning work). In particular, the primary rule of an incoming (outgoing) edge must only include the call of a Request by (of) another process linked to an input/output port. The symbolism used to call a Request of an outer object is "PortName<Request", while to depict the call of a Request by an outer object "PortName>Request" is used. For incoming/outgoing edges, the conditioning rule may also include calls to outer functions.

If a sender process calls a function of a receiver process, and this occurs only if at least an incoming edge starting from the current node of the receiver has the called Request as primary rule, then the call to the Request function may return true; otherwise, false is automatically returned (i.e. the call is not accepted). Indeed, incoming edges work as triggers for processes that are waiting for an external input.

With the exception of null edges, which are always traversed, if an inner edge (or an outgoing edge) is selected to be traversed by the current node, then it is traversed once the traversal rule is true.

Timers and Requests are the condition functions that may be involved to cause an edge traversal. Nonetheless, inner and outgoing edges may have both a Timer and Request function. In this case, the execution priority states that the Timer must be executed before a Request, i.e. the transition rule is verified when the edge has been warned.

All the edges may also have a list of *Action* functions. An Action is a function that performs such operation, e.g. changing the parameters of the process or of an owned entity. If a Timer and/or a transition rule have been defined for the same edge, the Action must always be executed at last, i.e. after the edge has been warned and if true has been returned by the transition rule.

Each Request and Action can explicitly receive a set of parameters; therefore, the behavior of a Request/Action may change in function of the received parameters.

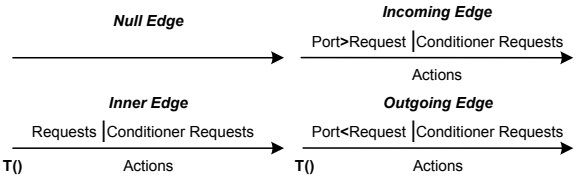


Figure 4: Edges of an EAD.

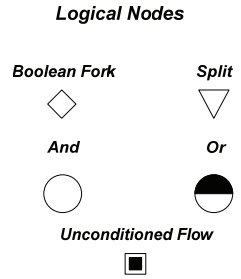


Figure 5: Logical Nodes of an EAD.

To improve MP scaling and model object reusability, as proposed in Sargent (1997), each process may use *multiports*. A multiport is an indexed set of *k* ports named *portname[1],..., portname[k]*. A multiport is explicitly depicted as different ports in the external view of a model object; in the inner view, multiport may be used either in an explicit or implicit way by incoming/outgoing edges. For instance, an explicit use of a multiport may occur when the process needs to call a Request function of an external process linked to the port "*portname[i]*" (where the suffix *[i]* stands for the

index of a port that is included in the multiport *portname[1,…,k]*), then the name of the port is explicitly depicted in the transition rule of the function as follows: "*portname[i]<Request*". Another example is introduced to show the implicit use of a multiport. If the process needs to call the same Request function of the external processes linked to the multiport, the following notation must be used in the transition rule of the outgoing edge: "*portname[i:1,…,k]<Request*". In this way, for each *portname[i]*, where index *i* varies between 1 and k, the edge must check the associated transition rule and the edge is traversed whenever at least one transition rule is true. The implicit use of a multiport allows our MP to be compact and to easily implement the behavior of dynamic processes. Moreover, by using implicitly a multiport, it is possible to specify a sub-set of port indexes that *i)* must be called through an outgoing edge or *ii)* allow an external process to call an inner Request through an incoming edge. The sub-set of port indexes can also be a list name generated at runtime using an Action function.

Our MP allows defining a process behavior via EAD using activities, events, sub-processes and directed edges. Another type of nodes, called *logical nodes*, has been defined in order to support the definition of process paths. These nodes are: *i) boolean fork*; *ii) split*; *iii) and; iv) or*; *v) unconditional flow* (Figure 5). A logical node may be used to represent alternative paths to be chosen under specified conditions.

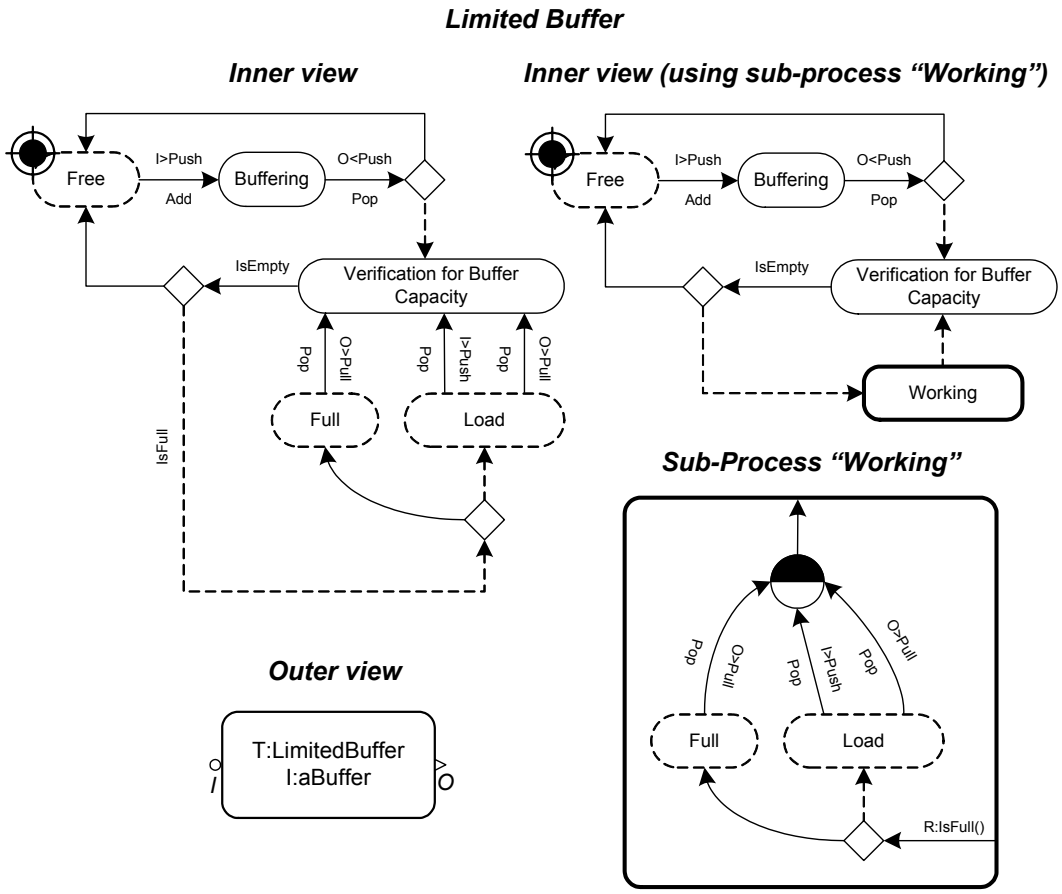


Figure 6: An Example of Use of a Sub-Process for a Model Object that is Designed as a Buffer with a Limited Capacity.

A boolean fork can be connected to the head of a conditional edge, or rather those edges that have a transition rule. Using this node, if the transition rule is true, than the edge is traversed and the process control flows through an edge starting by the boolean fork node; otherwise, the edge is also traversed, but the process control flows through a special edge, depicted with a dashed line, which starts by the boolean fork node. Therefore, using the boolean fork node after a conditional edge, the edge is always traversed. The outgoing edges from a boolean fork node are of the null or inner type. In case inner edges are used, another boolean fork must be used for each inner edge to catch the result of the transition rule.

The unconditional flow node acts in a similar way of a boolean fork, but whatever be (true or false) the transition rule of the incoming edge, only one edge must leave this node.

The remaining logical nodes may be connected to the head of any edge type. Once the edge is traversed, the process control passes at the logical node. A split node is used to separate the process path in two or more alternative paths. When the process path is separated in more paths, to recombine two or more paths, an and/or node is required. The and node, becomes enabled at the time all the edges incoming to the node have been traversed. The or node become enabled at the time at least one of the edges incoming to the node has been traversed.

The and, or and split nodes can be part of the process state. Typically the process state has only one active activity per process state. However, anytime the process flow is separated by using a split, many activities and the split/and/or nodes may be enabled, thus they may be part of the process state.

Some explanations are required about the use of sub-processes as nodes of an EAD. As they are EAD nodes, edges that start and arrive to these nodes are of the types defined in the previous. The only restriction is for edges that start by a sub-process node. If an edge that starts from a sub-process node is not a null edge type, a boolean fork must catch the edge transition results (otherwise the process checkpoint may stay on an edge included in the sub-process). The inner view of a sub-process is a particular EAD which consists of at the most one edge that links the outer view to an inner node (or rather from the inner boundary to a node) and/or at the most one edge that links an inner node to the outer view (or rather from a node to the inner boundary). Also if the edge that starts from the sub-process inner boundary to a sub-process node is not of null edge type, a boolean fork must catch the edge transition results (otherwise the process checkpoint may stay on an edge included in the sub-process). An example about a limited buffer is shown in Figure 6 (details of the Request and Action functions used in this model object are provided in a PhD thesis).

Processes are usually executed during more simulation time periods. For this reason the nodes of a process can be visited during different time periods. To track this possibility, we adopt the *process checkpoint*. A process checkpoint is a property of the processes that shows the set of nodes from which a process starts or is reactivated, i.e. the checkpoint locates the enabled nodes (the process state). A process checkpoint is also called *reactivation point*. For each process, the reactivation point must be explicitly shown to depict the initial active states as done in Figure 6 for the "Free" node (that is the initial state of the buffer model object): the reactivation point is depicted as a "target" and is usually placed on the upper-left corner of a node.

In HMP simulation models, a process can have the following status in the sense of simulation execution: *active*, *passive* or *terminated*. Focusing on a non-concurrent simulation, only one process can be active at a time. A process is active in the sense that it is moving through its paths, until it enters the passive state or is terminated. A process enters the passive state when a Timer schedules a delay time or a message is sent to an input/output port. In a similar way, a process becomes active by means of an external trigger (a message from an input/output port) or a previously scheduled warning time. To start model simulation, at least one process must receive a message to be activated for the first time. An active process may become terminated whenever an outgoing or transfer edge is traversed and the edge head is not linked to any node. A terminated process returns false to each incoming message and will never be active during the simulation.

3. A MODELLING CASE STUDY

In this section, a real case study of modelling and optimization of a complex logistic process is presented, with the aim of comparing the expressiveness of our MP with other well-known approaches.

The system that we are intended to model concerns the management of quay crane operations and it includes some key logistic processes, such as the "vessel discharge/loading" and the "container transfer from ship to yard (and vice versa)". This is a typical operative management problem that requires both simulation and optimization to be successfully approached. We refer to the Gioia Tauro terminal, one of the major container terminals in the Mediterranean Sea, located in Southern Italy.

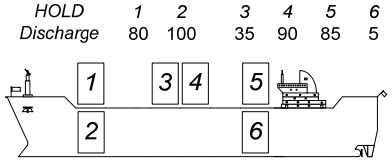


Figure 7: Map with Discharge/Loading Info per Vessel Hold

To fix ideas, we give a brief description of the system which includes the quay crane operations problem. Once a vessel is moored in a maritime container terminal, a certain number of containers must be discharged/loaded from/into the vessel's holds according to a pre-established operation plan (an example is shown in Figure 7).

Container discharge/loading can be initiated only if mechanical (and human) resources are allocated; if not, the ship waits in its berth position until resource assignment. Discharge/loading operations are performed by Rail Mounted Gantry Cranes (RMGCs) placed along the berth: one or multiple cranes move containers between the ship and the quay area. When multiple cranes are assigned to the same ship, crane interference has to be avoided and a complex scheduling problem arises to manage the relationships (precedence and non-simultaneity) existing among the holds of the same ship (this is the "Quay Crane Scheduling Problem" - Legato et al. 2008).

Table 1: Events.

Event	*Description*
StartDO	Cranes start discharging operations
EndDO[i]	Crane i-*th* ends assigned discharging operations
StartHD[i]	Crane i-*th* starts discharging of *current hold*
EndHD[i]	Crane i-*th* ends discharging of *current hold*
StartD[i]	Crane i-*th* starts discharge of a container from the current hold
EndD[i]	Crane i-*th* ends discharge of a container from the current hold
WaitNH[i]	Crane i-*th* waiting to acquire the next hold

InQ[i]	Straddle carrier arrival in quay
StartL[i]	SC starts container loading
EndL[i]	SC completes container loading

Table 2: Enabling conditions.

Edge Condition	*Description*
Rule 1	Buffer area under crane capacity is less than 6
Rule 2	Hold has not been completely discharged
Rule 3	No violation of precedence and non-simultaneity constraints
Rule 4	Crane has more holds to be discharged
Rule 5	Current hold is completely discharged
Rule 6	At least one SC waiting in quay
Rule 7	At least one container under the crane

The transport of containers from the quay side to the yard side and the reverse movement depends on the transfer mode of the container terminal. At Gioia Tauro, containers are transferred according to the direct transfer mode, i.e. the transport of containers and the container stacking processes are performed by a fleet of straddle carriers (SCs). SCs take in charge containers and cycle between the berth area and the assigned storage positions within the yard. SCs are guided to a slot within the yard structure to stack/retrieve containers. They are also able to cover the distance from the assigned RMGC to the yard slot and back.

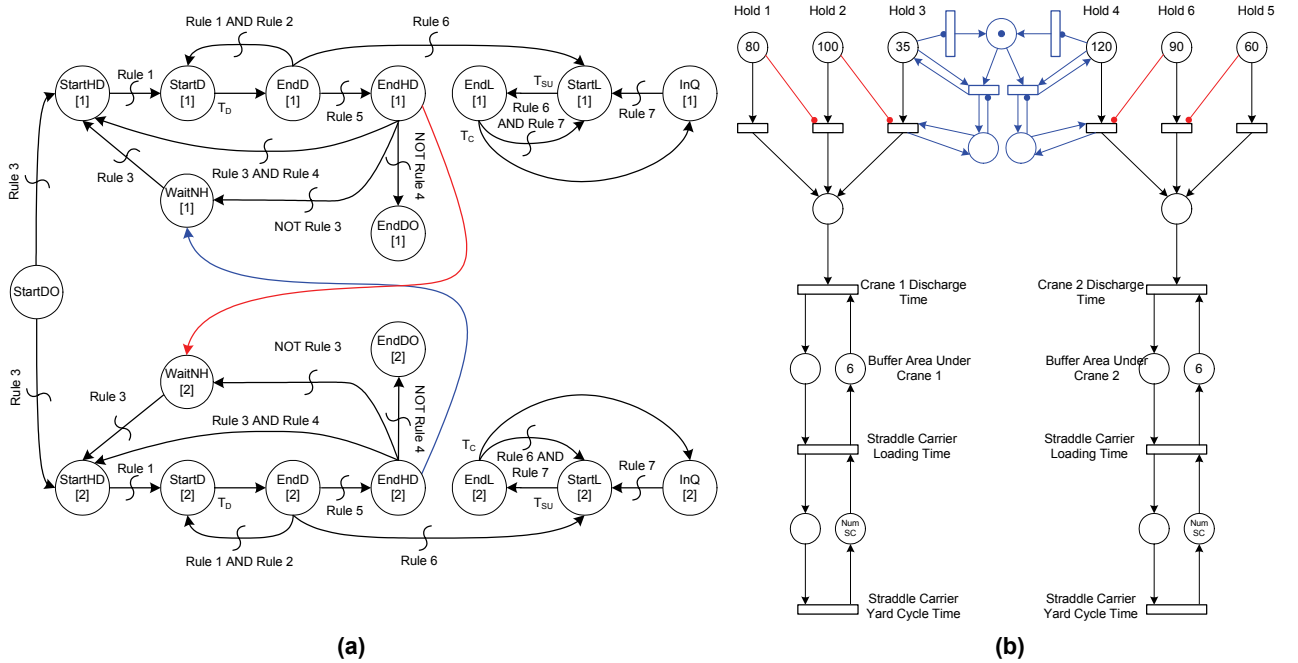
Here, we are interested in depicting the container discharge process of a certain vessel and depicting the container transfer from the limited buffer area (max 6 containers) under a RMG quay crane to the yard side by means of SCs.

Since the process of transferring containers from the quayside to the yardside is performed by SCs that are used to cycle among these areas with (outward path) and without containers (backward path), this process may also be depicted as a unique service time, or rather "the yard cycle time", which includes the outward time, the container unloading time and the backward time.

In the following, we suppose for simplicity that two RMGCs must simultaneously perform the discharging operations of the vessel depicted in Figure 7. The sequence of holds that must be respectively discharged by the two cranes is the following: {*Hold 1, Hold 2, Hold 3*}, and {*Hold 6, Hold 5, Hold 4*}.

Figure 8(a) shows the Event Graph model of the logistic processes described above; events are labeled with numbers in squared brackets that refer to a specific crane.

Tables 1 and 2 report the descriptions of the events (the vertices of the graph) and the edge conditions. The edge delay time functions T_D, T_{SU}, T_C, are respectively: *i)* container discharging time for each RMGC, *ii)* container set-up time for a generic SC, and *iii)* cycle-time for an SC (i.e. the time to transfer a container from the quay to the assigned yard-slot, to set-down the container and, finally, to move back to the assigned RMG quay crane). As a matter of fact, the possibility to discharge a hold by a crane is due to the respect of the precedence and non-simultaneity constraints. In this way, the availability of a hold depends on the current task that is assigned to the next crane. In our example, if crane 1 is discharging hold 3, then crane 2 to discharge hold 3 must wait the completion of hold 1 (and vice versa). To depict this process, every time a crane completes its discharging operation on a hold, then it warns the next crane (edges in blue and red).



(a) **(b)**

Figure 8: (a) The Event Graph of the Quay Crane Operations Model. (b) The Petri Net for the Quay Crane Operations Model.

Using Petri Nets, we can model the system as depicted in Figure 8(b). In this model, seeing that the relationships between the model components are very clear. This is true especially if the model end-user looks

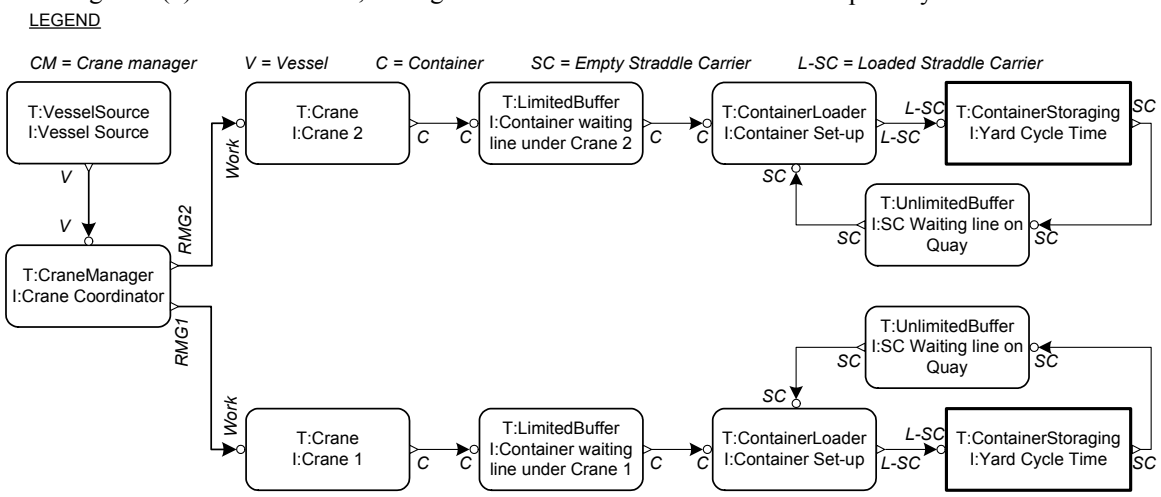


Figure 9: The HMP for the Quay Crane Operations Model.

number of tokens in some places were too large to be explicitly depicted (e.g. the tokens which stand for the containers to be discharged) we have directly put in each place the number of tokens. Using Petri Nets, we are able to design the non-simultaneity constraint between Hold 3 and Hold 6 (edges and nodes in blue), as well as precedence between holds to be discharged (inhibitor arcs in red). Nevertheless, we do not provide reasonable model readability, and the net explosion is inevitable every time the number of holds and constraints rise up, as in real cases.

Finally, we provide a representation of the quay crane operations model using an HMP model, as shown in Figure 9, based on the coupling of six types of model objects and a sub-system. The model objects are: *i)* a *VesselSource* for the generation of vessels, *ii)* a *Crane* for depicting the quay RMGC resource, *iii)* a *CraneManager* to manage the interactions among RMGCs working on the same vessel and assign the sequence of holds of the berthed vessels, *iv)* a *ContainerLoader* (Figure 10) to perform container set-up operations for empty SCs, *v)* an *UnlimitedBuffer* (Figure 6) to depict the waiting line of empty SCs that wait for load discharged containers and *vi)* a *LimitedBuffer* to depict the buffer area under a crane (for no more than 6 TEUs). The EADs for the *UnlimitedBuffer*, *VesselSource*, *Crane* and *CraneManager* model objects will be provided in a companion paper.

With this approach, a modeller can design the behavior of a set of context specific model objects (e.g., cranes and queues), even unrelated, and then he/she can reproduce a certain system by linking these model objects and simply specifying suitable parameters (e.g., the distribution function for the crane discharge time). If the designed model objects are collected in a library in a "point & click" simulation environment, they are close at hand to be reused to design a new system.

It is clear that by using this approach we provide a compact way of representing the system logic: in fact, entities flow through the model objects and the at name of the ports, which explicitly indentifies which are the input(s) and the output(s) of each model object.

Moreover, if one wishes to further simplify the readability of the model by hiding a given part of the system, this can be performed in a simple way by grouping and depicting the selected system part as a sub-model. For instance, in Figure 9 the *ContainerStoraging* is a sub-model that receives in input a loaded SC and returns an empty SC after a while (the yard cycle time); focusing our attention on the quayside operations, we are not interested in understanding how an SC has been unloaded and why it has returned after a certain time period, therefore the use of a sub-model in this context appears convenient.

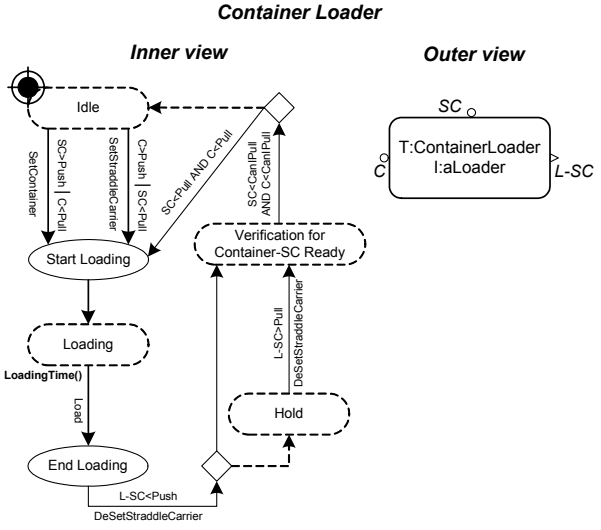


Figure 10: EAD for the ContainerLoader model object.

The behavior of each model object is designed by using an EAD. Each EAD is composed by *i)* default components (events, activities, logical nodes, edges) and *ii)* context specific timers and functions. Both timers and functions are developed around a set of global variables of each process (e.g., a list of entities for the infinite queue depicted by the model object

UnlimitedBuffer) and using a scripting language like TCL (www.tcl.tk).

This approach has its strong point on the use of functions to check conditions and execute algorithms devoted to take decisions. For instance, once a vessel arrives at berth, the CraneManager uses a function that solves a mathematical model (the Quay Crane Scheduling model proposed by Legato, Mazza and Trunfio – 2008) to dynamically assign the holds of the vessel to a specific set of cranes. Obviously, this approach is possible using our MP or EGs and HCFGs. Petri Nets are a powerful tool for modelling a system at a very low level, describing also conditions and complex constraints by means of its intuitive formalism, but a PN cannot be re-designed at runtime in case of need (e.g., is not possible to use the PN in Figure 8(b) to simulate the discharge process of a vessel with a different structure).

4. SIMULATION-BASED OPTIMIZATION

A modern simulation platform for logistic systems must provide an easy-to-understand language for system modelling, but also a tool for designing of optimization problems. We are developing a platform for the optimal management of logistic systems management based on the following three phases: *i) system modelling*, *ii) model analysis* and *iii) system optimization*. In the first phase, the real system is modelled by using our MP, as discussed in-depth in the second and third section.

Numerical results from the simulation model obtained in the system modelling phase are analyzed in the second phase by means of a statistical analysis tool. Performance measures are evaluated on the outputs from the simulation runs of the simulation model. In the model analysis phase, indices and parameters are defined on the simulation model by means of some tool or language. The evaluation of these indices and parameters provides the set of performance measures necessary for the simulation-optimization platform.

After that, in the third phase an optimization problem is defined. General problem setting is made of input and output variables, objective function and constraints. The nature of the optimization problem is intrinsically stochastic, due to the nature of the simulation output variable). Output variables are simulation model performance measures; input "variables" are quantitative or qualitative in nature. In this context, a language to easily develop simulation-based optimization algorithms must be planned, e.g. by using the TCL language.

5. CONCLUSIONS

We have proposed a Modelling Paradigm to support the development of simulation models oriented to the optimal management of logistic activities. The MP uses a holistic approach to capture the complex relationships among sub-systems and allows for a direct representation of the hierarchical structure of the decision making process for system management. System modelling is completed by a flow-chart based definition of processes involved in the model at hand. Real case examples of modelling have been presented, comparing our approach to those provided by Event Graph models and Petri Nets, with the aim of supporting the future design of simulation-based optimisation techniques. Currently, a Java tool is under development.

REFERENCES

Derrick, E. J., Balci, O., Nance, R.E., 1989. A Comparison of Selected Conceptual Frameworks for Simulation Modelling. *Proceedings of the 1989 Winter Simulation Conference*. 711-718, Blacksburg (VI, USA).

Fritz, D.G., Sargent, R.G., 1995. An Overview of Hierarchical Control Flow Graph Models. Proceedings of the 1995 Winter Simulation Conference. 1347-1355, Arlington (Virginia, USA).

Legato, P., Mazza, R.M., Trunfio, R., 2008. Simulation-based optimization for the quay crane scheduling problem. *Proceedings of the 2008 Winter Simulation Conference*. Miami (FL, USA).

Legato, P., Trunfio, R., 2007. A Simulation Modelling Paradigm for the Optimal Management of Logistics in Container Terminals. *Proceedings of the 21th European Conference on Modelling and Simulation*. 479-488 , Prague (Czech Republic).

Petri, C.A., 1962. Kommunikation mit Automaten, Ph.D. Thesis, Schriften des Institutes füt Instrumentelle Matematik, Bonn.

Pidd, M., 2004. *Computer simulation in management science* (5th edition). John Wiley & Sons: Chichester.

Pidd, M., Castro, R.B., 1998. Hierarchical Modular Modelling in Discrete Simulation. *Proceedings of the 1998 Winter Simulation Conference*. 383-389, Washington (DC, USA).

Sargent, R., 1997. Modelling queuing systems using hierarchical control flow graph models. *Mathematics and Computers in Simulation* 44(3): 233-249.

Schruben, L.W., 1983. Simulation Modelling With Event Graphs. *Communications of the ACM* 26(11):957-963.

AUTHORS BIOGRAPHY

PASQUALE LEGATO is an Assistant Professor of Operations Research at the Faculty of Engineering (University of Calabria). His home-page is http://www.deis.unical.it/legato.

DANIEL GULLÌ is devoted to research in numerical simulation at the Center for High-Performance Computing and Computational Engineering (CESIC) in NEC Italy.

ROBERTO TRUNFIO is currently pursuing a Ph.D. degree in Operations Research from the University of Calabria and is a logistics engineer at the CESIC in NEC Italy.