EUROSIM2013

8th EUROSIM Congress on Modelling and Simulation

Cardiff, Wales,
United Kingdom

10-13 September 2013

Edited by:
Khalid Al-Begain
David Al-Dabass
Alessandra Orsoni
Richard Cant
Richard Zobel

Published by CPS Conference Publishing Services

Product Number E5073
ISBN 978-0-7695-5073-2
BMS Number CFP1397U-CDR

Copyright © 2013 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

2013 8th EUROSIM Congress on Modelling and Simulation

EUROSIM 2013

Table of Contents

Track 02. B. Fuzzy Systems

Track 03. C. Evolutionary Computation

Reverse Engineering Hospital Processes Out of Visited Nodes

Barbara Glock[1], Gabriel Wurzer[2], Felix Breitenecker[1]
[1]Institute for Analysis and Scientific Computing
[2]Institute of Architectural Sciences
Vienna University of Technology
Vienna, Austria
barbara.glock@student.tuwien.ac.at
gabriel.wurzer@tuwien.ac.at

Niki Popper
dwh simulation services
dwh GmbH
Vienna, Austria
niki.popper@dwh.at

***Abstract* – During hospital planning, "flow-chart"-like depictions of processes (nodes and edges) are often overlaid over the floor plan of a clinic, in order to model patient pathways. However, such static definitions may not be accurate, as patients freely cross from one node into another, depending on individual treatment. Modelling each possible sequence of nodes is neither practical nor intelligible. This means graphical complexity due to overlapping edges. In our work, we thus present an edgeless process model in which each node acts as "dispatcher": It carries the logic for choosing a next node, thus enabling us to omit edges and introduce dynamic distribution of simulated patients. "Processes" (i.e. often-perceived sequences of activities) can nevertheless be inferred in an a posteriori step with an algorithm that will be presented, utilizing the actual patient pathways as indicator. Furthermore, the visualization of how processes are generated out of activity chains by reverse engineering will be discussed too.**

Keywords - hospital planning; process simulation; treatment chains;

I. INTRODUCTION

Processes act as fundamental basis and constraint for the planning of a clinic: Due to the complexity of the work processes that the building has to support in a 24/7 fashion, it is not possible to plan such a building without knowing all occurring activities and activity-holders in advance. Modelling of processes is usually done in graph-like fashion, i.e. we see the activities as nodes and their temporal and causal ordering as edges. However, having to draw edges might be suboptimal from a modeling point of view, as is shown in the example depicted in Figure 1(a). Assume that we have a standard process for an emergency department, in which a patient either enters via the main entrance (walking or sitting patient) or is brought in by an ambulance (lying or sitting patient). If we have to draw the process on top of the clinic's floor plan, already some visual cluttering occurs. However, and more importantly, the process drawn does not reflect the chain of activities that are carried through in the course of treatment accurately; for example, some patients are immediately treated without undergoing triage (when arriving with an ambulance), others go through the registration process and eventually end up in the treatment area twice, with an x-ray in between. Figure 1(b) shows the real decomposition of the patient volume into the different activities encountered, which cannot be easily modeled in the process fashion seen in Figure 1(a).

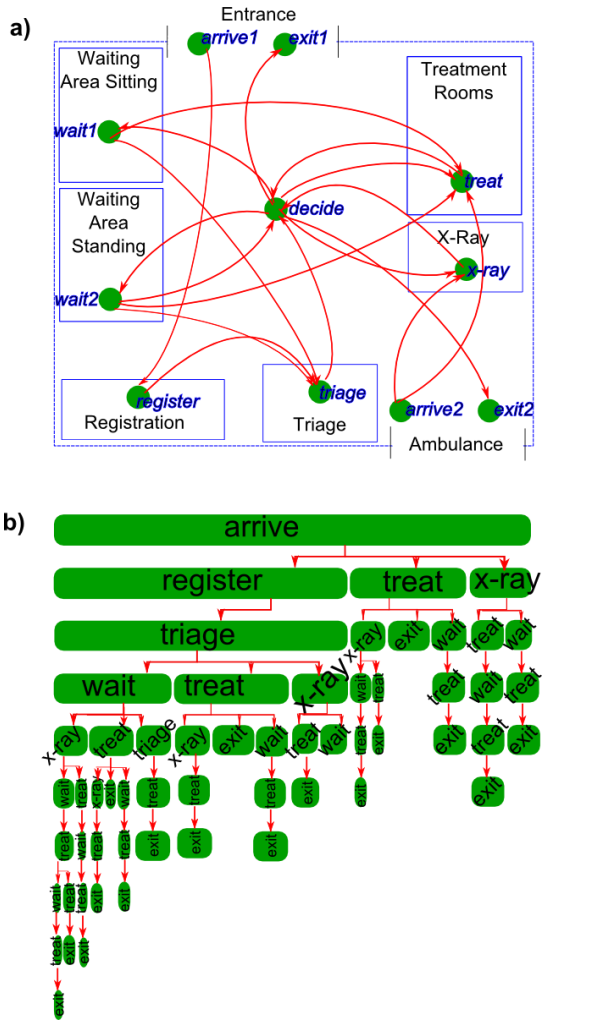

Figure 1: (a) Example of an emergency department process modeled as spatial flowchart. (b) Activity decomposition for the whole patient volume.

However, as stated before, this would lead to visual cluttering and is furthermore not practical (too much work for too little benefit). In fact, why model processes statically at all, if a dynamic sequence of activities is to be reflected?

The idea to give up the concept of static edges connecting nodes lies at the core of what we want to present in this paper. More specifically,

- we propose an approach in which nodes act as dispatching units, which enables us to model dynamic chains of activities in a comfortable manner (see Section III.A),
- we give up the notions of edges in favor of an alternative way with which agents are moved between the nodes: A dispatching algorithm written by the user acts within the nodes, which in turn moves the agents through the floor plan (see Section III.B),
- we reverse-engineer "processes" from the occurring activity chains, taking into account the utilization of each activity chain during the simulation. As result, we are able to depict the activity chains that occur most frequently (see Section III.C).

To sum up, we can eliminate unnecessary clutter, when depicting processes visually and ensure that activity chains of patients can be modeled accurately and efficiently. The mining of often occurring processes in these activity chains is yet another contribution which we bring forward in our work. From the simulation side, using the described modelling approach required nodes have to be seen as active entities which dispatch agents to the required treatment nodes. The details of how this is done conceptually will be brought in a rather long description under section III, for which we make no apologies: Some concepts used there were less obvious to us, and required quite a bit of elaboration.

II. Background on process modeling

Process modelling typically includes "flowchart-like" process descriptions showing activities as boxes connected by arrows. Several flavors of activities and connectors exist, both from the representation as well as the semantic side: For example, flowcharts are standardized under ISO 5807:1985, however, there is the more recent Business Process Model Notation (BPMN) that has gained widespread usage in computer science and business administration. Apart from notation, some models explicitly aim at semantic aspects of these models – such as Event-driven Process Chains (EPCs) or Activity Diagrams within the Unified Modeling Language (UML). Simulating such process definitions uses Graph-Oriented Programming (GOP, see [1]) in conjunction with a discrete simulation [2] that handles aspects such as resource allocation, queuing and so on. From a more mathematical side, these approaches can also be seen as Petri Nets [3], in which a process token representing a process instance moves along a graph of nodes representing transitions. The latter aspect is a difference to the aforementioned notations, in which nodes stand for the states and edges for the transitions. Even though Petri Nets offer the advantage of being formally analyzable, they are hardly used in the context of business administration and building planning, which is our context. We thus omit the discussion of this subject and concentrate on our main idea, the simplification of process descriptions and subsequent simulation within these.

III. Elaboration

In the following sections we will define what we exactly mean by a process model consisting only of dispatching nodes and tokens that flow between them. In the first part, we will outline the structure of our model and define the basic terminology that will be used throughout the paper. We then go on to elaborate how a process simulation can execute the model, in order to obtain a flow (utilization of each node and sequence of nodes visited). In the last part, we analyze this information to infer "processes", i.e. often-recurring activity chains.

A. Basic Process Simulation Model

Our process model is composed of the following two basic constituents:

- The process token, which represents an instance of an executed process.
- The nodes, which are visited by the process tokens and act as dispatcher, in the following fashion: A token that enters a node executes code contained therein, resulting in a next node to visit. This node is set as next goal for the token (transition). There are two special cases of nodes, in the form of start- and end-nodes. Start-nodes generate a process token, i.e. they cannot be set as goals. End-nodes terminate a token, which means removing the process instance from the simulation. There is also the case of process composition, in which a node passes the token on to another process being called on its behalf. In that case, the start-node of the called process becomes the entry point and the end-nodes the exit point for the sub-process invocation.

A token can be visualized as an agent, when in the floor plan. The transitions between nodes can be animated, yielding a "patient-flow"-like depiction intended for end-users (architects, planners).

B. Simulation within a Node

A token having arrived in a node executes the code contained therein, in order to arrive at the conclusion what next node to take. In our case, this "code" is again graph-based, which allows us to utilize almost the same methods for spatial and non-spatial simulation:

- A program has one entry point (start-node) and many exit points (end-nodes).
- Between start- and end-nodes come a variety of intermediate nodes that read data contained in the token, perform calculations on that data and finally decide what next goal to select. The description of this programming language is deferred until later, for now we just wish to express that the "code" is yet another network through which the token needs to travel.

- The token itself becomes a data container, similar to what programming languages call context: For example, a node executing "*c=a+b*" would reach into the token to retrieve the values for *a* and *b* stored there, storing the result in a new data entry *c*.

Figure 2 shows an example of the whole model, both outside and inside nodes: Initially, we start the process on the start-node *arrive1* (Figure 2: *a)*), which generates the process token and executes the contained code graph (Figure 2: *b)*). Typically, in this step, we will read the intended activity chain for that process instance from a spread sheet and store that within the token (Figure 2: *b*) in *goals*). Let us assume that this would be *register* > *triage* > *treat* > *exit*, for the sake of the example. The code would set the goal of the token to the first entry, in this case *register*. The token (Figure 2: *c)*), is then animated spatially as it transitions to the registration node *register*. Upon entering that node (Figure 2: *d)*), a waiting time of three minutes is introduced by the contained code (Figure 2: *e)*). Behind the scenes, the token is moved into a passivated list, in which it stays for the given duration (Figure 2: *f)*). A scheduler has to be implemented such that it can activate processes again, also using events as criterion (to be discussed in due course). As next step, the goal, and therefore the token, is advanced to *triage* (Figure 2: *g)*). The contained code in that node (Figure 2: *h)*), assigns one of three priorities randomly to the token. It examines the next node, which is *treat*. Each node on the floor plan is implemented as a server of limited capacity (*treat*: three places, of which none is currently available). *triage* determines that instead of advancing to the next goal, the token has to go to *wait1*, until *treat* becomes free. The token walks over to *wait1* (Figure 2: *i)*) and becomes passivated waiting for *treat* (Figure 2: *j)* → f)). As has been mentioned before, the scheduler has to reactivate that process token, when *treat*'s capacity is below its maximum. Technically, treat broadcasts a "freed event" which the scheduler utilizes to that end. Once the token has entered *treat* (Figure 2: *k)*) and waited for the treatment time (Figure 2: *l)*) again by being sent to the passivated list, it advances to the end-node *exit* (Figure 2: *m)*), where the process ends.

It is crucial to mention that several dynamic actions can happen within a node. For example, treat could have determined that additional intermediate steps (*x-ray* > *treat*) would be necessary, based on some probabilistic criterion. Vice versa, some elements of the process chain could be removed, in case the treatment concludes that those steps are obsolete. This could be again the case for probabilistic model.

As a next step, we describe the inner mechanics of the simulation core from an implementation standpoint. We have two parts that are combined: First, the code that is executed within nodes (which we call code graph for reasons we will explain in due course), second, the algorithm that takes care of transitioning between nodes. In both cases, it is the scheduler that handles all of these types of executions and, furthermore, is responsible for passivation/activation (based on time and events).

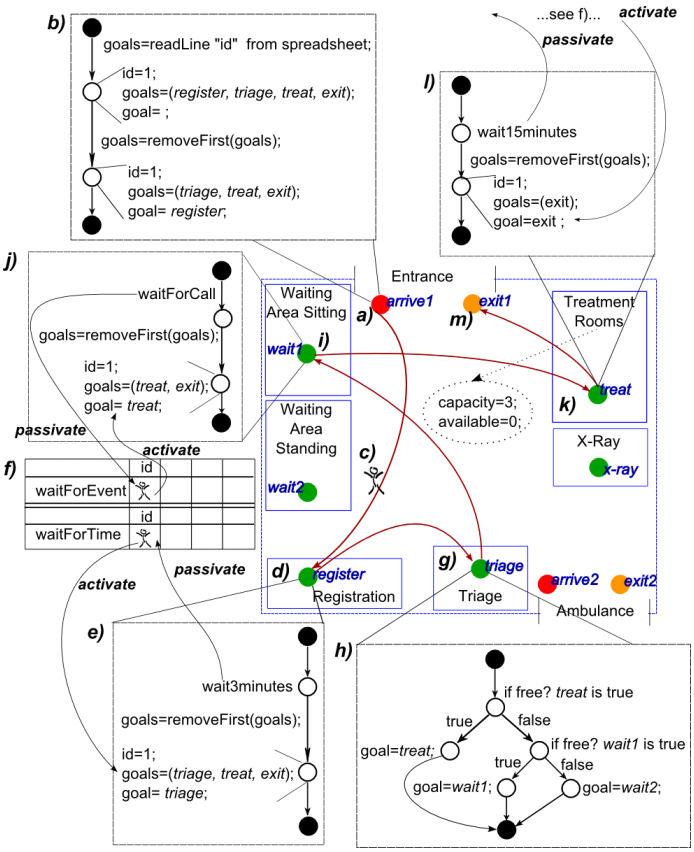

Figure 2: The activity chain of a token through the floor plan.

1) *Code Graph*

In contrast to a variety of programs that let the user write textual code, we have a graphical editor for the same sake (see Figure 3). Even though this does not free us of the programming language itself, it does provide the benefit of a "drag and drop" development of the dispatching code, in which an instruction can only be dropped if it can be accommodated by the target block. Such an avoidance of errors is significant for our intended audience, which consists of hospital planners (i.e. architects, process designers) and not programmers. It is based on a similar approach by the CMU's Alice Team [4].

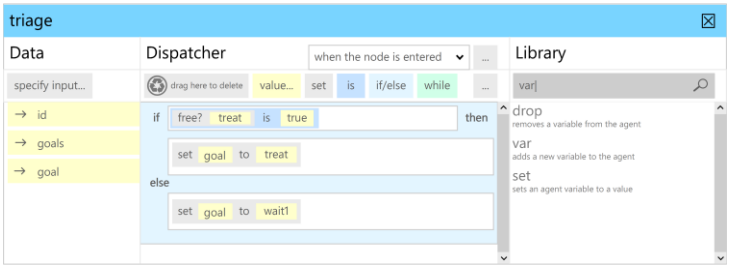

Figure 3: Dispatcher code graph, visualized as hierarchy of "blocks within blocks"

Interestingly, a block-based program forms a hierarchy, which can also be interpreted as graph. For example, the program in Figure 3 is analogous to the graph shown under Figure 2h. We use a graph engine acting on that level to actually execute us the code. Internally, each token carries an *execution pointer* to the current node of this code graph. The token itself furthermore contains the necessary data (values the computation in the nodes need). The scheduler has the task of advancing the execution pointer, based on the result of the current node. For example, the expression "if free? *treat* is *true* then" can return either true or false, which is also the name of the transition in the code graph used for getting to the node of the code.

The overall responsibility for advancing the execution pointer lies with the scheduler. More precisely, this manages all tokens and their respective state (data, execution pointer, execution history). It encompasses two waiting lists - one for tokens waiting for a timestamp, one for tokens waiting for an event to occur. Once the execution arrives in a node that requires passivation (based on one of the two the mentioned criteria), the token is moved to these lists, to be activated in the future. In our model, we use a fixed-step progression of one second, plus events that are fired when resources become free, nodes are entered and left and so on; however, this is no necessity - we could have also used a scheduler with a variable time step, i.e. future event list for time, plus an additional one for events. Since events can only be generated by tokens that wait for a certain time, we could have first activated time-based tokens, then event-based tokens until all signals are consumed. However, we have not done so, because we want to be able to animate the passage from one node in the floor plan to another - hence we progress in seconds.

2) Transitioning between Nodes of the Floor Plan

The transitioning algorithm is, in the simplest case, a linear interpolation between the source and target node of a token. However, at this level, we might also explicitly deal with the floor plan and physical movement of each token (shown as an agent), in the following fashion:

- Using the layout of the hospital exemplified in Figure 1, we can perform a physical simulation [5] that simulates each individual's passage through the floor plan for high-density situations. To a certain extent, this approach is useful for simulating crowded waiting areas, entry zones and the like. It must be mentioned, however, that these algorithms are more targeted at egress planning than early hospital design, which is our context.
- Even if we do not model the physical transition of each token through the floor plan, we can make it walk along the circulation (basically a graph giving walkable pathways). An approach that automatically calculates such a transition sequence along a circulation graph (also taking walls and entrances that lie along the way into account) has previously been presented in [6].

C. Reverse Engeneering Hospital Processes out of visited Nodes

After having simulated the patient pathways in the described manner, we want to reverse-engineer processes out of the occurring activity chains.

Simply assigning a utilization to each node would not serve the purpose, because only the capacity of each node would be displayed (by a counter that increases by one for each token (agent) passing by). Our goal is to display the whole processes and not only the visited nodes. Therefore, agent histories are used. An agent history is a string storing each of the patient visited nodes. Those histories are then (mentally) laid on top of each other to get a look on the occurring pathways.

An algorithm is used to extract the "real" processes out of these pathways. To get a clearer look what is meant by "real" processes, an example is discussed shortly: The activity chain (each visited node) of an agent with for example *id*=2 stored in the agent's history is: *arrive1* > *register* > *triage* > *wait1* > *treat* > x-*ray* > *treat* > *exit*. The agent with the *id*=3 has an activity chain such as: *arrive1* > *register* > *triage* > *treat* > x-*ray* > *treat* > *exit*. The difference of these two activity chains lies in one waiting node; the agent with *id*=3 advances from *triage* directly to *treat* and does not has to wait. Due to the fact that those two activity chains have different strings stored in the agent's history, they would further count as different processes, although their (for the modeler) important pathway is the same.

Therefore, the nodes are distinguished between *significant* and *insignificant* nodes. The latter one is a node representing waiting in a waiting area, because no significant activity is done here. Usually waiting areas in hospitals are not from particular importance to the modeler due to the fact that they exist in every hospital. In Figure 2 these insignificant nodes would be *wait1* and *wait2*. Furthermore, there are some nodes for which the user can decide himself if they are significant or not, like *triage*. On the one hand *triage* is a waiting node too, but on the other hand it requires additional items (such as staff that is triaging the patient). The modeler can decide of its importance and therefore of its significance.

Furthermore, the user has to define a threshold that determines the amount of patients having used this pathway, used for qualifying the pathway as a being a "process" or not. That can be a real number of total patients, or a percentage of patients. This is important, because otherwise each (for insignificant nodes reduced) activity chain would qualify as a process. An activity chain that represents a specific rare behavior would then be a process too, i.e. the very rare case of a patient suffering from ruptured appendix after being treated and therefore not ending his pathway in an exit, but in being hospitalized.

After having set the threshold and distinguished between the significant and insignificant nodes, the insignificant nodes are eliminated by the algorithm from the agent history. The remaining activity chains now qualify for a "real" process if their utilization is above the threshold. To determine the cardinality for each group of remaining

activity chains, they need to be sorted from longest to smallest by length and also by content, so that they can be grouped by the equal ones. Now it is easy to count the amount of grouped activity chains (sequentially executing the list of sorted activity chains and increasing a counter) and assigning each group of chains its cardinality. Those which are above the threshold are the required processes.

Next the visualization of these processes is discussed shortly: To avoid cluttering these processes have to be laid on top of each other, which makes it difficult to distinguish between processes that are part of other processes together with depicting their cardinality. The pathways of processes are depicted by edges connecting the significant nodes of the activity chain on the floor plan. To display the cardinality of these processes a simple chart of color range is used, for example by depicting often occurring process by a darker color.

The main problem arises if there are processes that are part of other processes. Let's assume there is *process1*: *arrive1* > *register* > *triage* > *treat* > *x-ray* > *treat* > *exit*, seen in Figure 4 (a) as yellow line, because its cardinality (how often the process occurs) is not that high, and *process2*: *arrive1* > *register* > *triage* > *treat* > *exit* seen as purple line in Figure 4 (a) having a higher cardinality than that of *process1*.

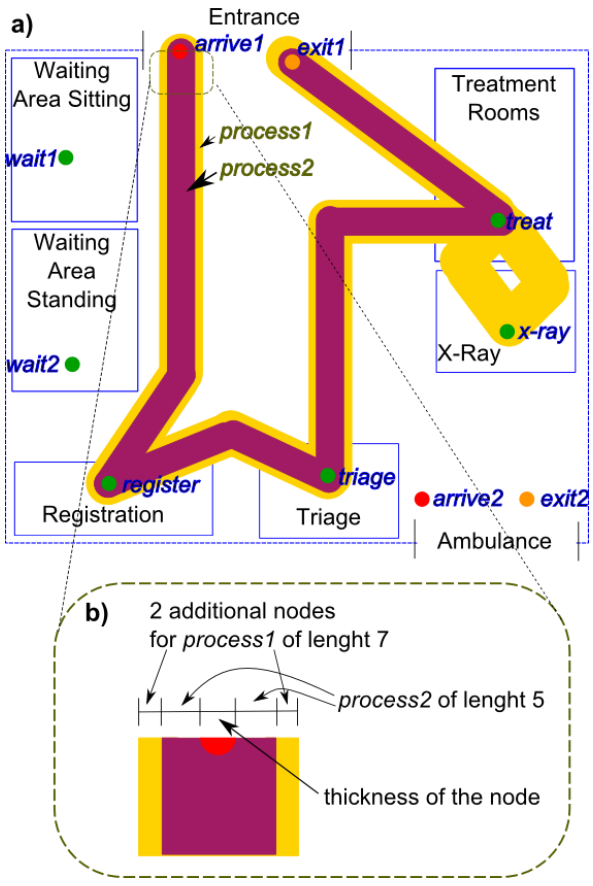

Figure 4: a) Visualization of reverse engineering processes. (b) Detailed view of overlaid processes with different length of activity chains

The length of those two processes differ (and for the sake of the example also their cardinality), but *process2* is (although being an independent process) a part of *process1*. To lay these two processes on top of each other would lead to not visualizing *process2*, because it would be totally overlaid. Therefore, we display longer processes (more visited nodes) different from shorter processes (less visited nodes) by adding additional space to the width of displayed activity chain for each additional visited node. In Figure 4 (b) this circumstance can be seen: The process initially gets the width of the start-node and for each additional visited node of the activity chain, the width of the visualized process gets additional space resulting in longer activity chains having a thicker displayed width. Therefore, the yellow line of process 1, inheriting 7 visited nodes appears thicker than that of *process2*, only inheriting 5 nodes.

Cluttering can be avoided with this kind of visualization of processes, although the cardinality (how often a process is simulated) and every process, no matter if it is part of another process or not, can be displayed accurately.

IV. Discussion

At the core of our approach lies the concept that sequences of treatment steps are dynamic, in contrast to processes, which are static (at least when it comes to the choice of a next node). Characterizing further, our model builds up processes based on the observed data (treatment sequences), while traditional processes are prescriptive and "state what shall be done". We have argued that, even process definitions in the latter sense are produced during planning, these may fail to capture the "real-world" behavior of patients undergoing treatment, and may thus be less effectively employed in simulation.

Technically, we see our contribution in the dispatcher facility, implemented as a scheduler acting on a "code graph": Apart from distributing tokens to next nodes (which is trivial), we have closely looked at the mechanisms for supporting passivations/activation *within a node*. This step represents a challenge, as (in our case: Java) does not support passivating a program and activating it again, lest we turn to more techniques such as one thread per agent (a no-go, considering performance). If we represent the "code" as graph and the execution pointer, we can get around this constraint: The token containing both the agent's data (i.e. the context, in programming lingo) and the execution pointer (the state) is written into a list (passivation). The scheduler can then activate the program again based on either time-based or event-based criteria, by simply removing the token from the list - in which case the execution resumes exactly where it has left off.

A further contribution lies within the actual reverse-engineering of static processes out of the dynamic activity chains during treatment. On the surface, it would look as though this step could also be done quite easily without our approach, if all treatment steps are known. For example, one might bring up a spreadsheet processor apply the steps listed under Section III.C. However, as has been said earlier, the code can alter the treatment chain as it sees fit - inserting

additional steps or removing them. A typical case for that might be the refurbishment of a hospital, in which the newly planned operational concept differs from the old one: Given the patients of the old hospital and their treatment chains therein, a node code can perform a mapping onto the new operational concept, by inserting, altering or deleting parts of the chain. Such an approach was successfully used by the authors for the case of an 800-bed clinic near Vienna [7], which is also our test-bed for the simulation. Because of the possibility for mapping and other dynamic behavior within nodes, we can safely state that this is not possible lest we resort to an approach as presented herein.

V. CONCLUSIONS

We have presented a model in the context of hospital planning, for situations in which a "flowchart-like" process description is to be simulated on top of a floor plan. Trivially, such depictions suffer from visual cluttering caused by overlapping edges. But, an even more serious issue is that such static process graphs fail to capture the dynamic nature of treatment, in which patients cross individually from node to node, instead following a preset route. To cope with these two problems, our model gives up the notion of edges altogether, replacing them with nodes that can dispatch patients to next nodes dynamically. "Processes", i.e. static sequences of activities, are derived by a-posteriori analysis of all agent histories. As result, we are able to model treatment chains adequately, while at the same time making it easier for planners to work on the according simulation models (only nodes and dispatcher code is entered).

ACKNOWLEDGMENT

Special thanks to ZIT – The Technology Agency of the City of Vienna, a subsidiary of the Vienna Business Agency for supporting us on the project *Modular Dynamic Planning and Simulation for Health Care Facilities (MODYPLAN).*

REFERENCES

[1] "Graph Oriented Programming, JBoss JBPM Documentation". http://docs.jboss.com/jbpm/v3/userguide/graphorientedprogramming.html, accessed 17.06.2013.

[2] B. Rücker. "Building an open source Business Process Simulation tool with JBoss jBPM", Master Thesis, Stuttgart University, 2008.

[3] G. Music, I. Hafner, St. Winkler, I. Skrjanc. "A Matlab based Petri Net Tool for E-learing: Examples for timed Simulation and Scheduling", Full Paper, Vienna, MathMod 2012.

[4] St. Cooper, W. Dann, R. Pausch. "Teaching Objects-first In Introductory Computer Science". SIGCSE 2003.

[5] G. Wurzer, M. Ausserer, H. Hinneberg, C. Illera, A. Rosic: "Sensitivity Visualization of Circulation under Congestion and Blockage", Poster: PED 10 The Fifth International Conference on Pedestrian and Evacuation Dynamics, Gaithersburg, MD; 2010-03-08 – 2010-03-10; Proceedings of PED 2010, Springer, (2010), 899 – 902.

[6] G. Wurzer: "Schematic Systems – Constraining Functions Through Processes (and Vice Versa)", International Journal of Architectural Computing (invited), 08 (2010), 02; 197 – 213.

[7] G. Wurzer, W. Lorenz, M. Pferzinger. "Pre-Tender Hospital Simulation Using Naive Diagrams As Models", Talk: IWISH 2012, Vienna; 2012-09-19 – 2012-09-21; Proceedings of the International Workshop on Innovative Simulation for Health Care 2012, Dime Università Di Genova, PDF on CD (2012), ISBN: 978-88-97999-13-3; Paper ID 35, 6 pages.