Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

TRACE BASED ANALYSIS OF PROCESS INTERACTION MODELS

Peter Kemper
Carsten Tepper

Informatik IV, Universität Dortmund
D-44221 Dortmund, GERMANY

ABSTRACT

Simulation models of real world systems may have a complex dynamic behavior that has an impact on measures of interest but its causes are hard to detect by statistic measures commonly computed in a simulation run. Most simulators are able to document the details of a simulation run in a trace file. We propose a trace-based analysis and visualization method that allows us to identify reasons for large confidence intervals, high variances in lead times and correlations among delays. The technique is implemented in a stand alone tool for trace analysis that has been evaluated for the ProC/B modeling and simulation framework. We illustrate the visualization technique with the help of a ProC/B model of a warehouse.

1 INTRODUCTION

Simulation applies to a broad class of models and comes with relatively few constraints. In this paper, we consider discrete event simulation for stochastic models. We follow a process approach to simulation modeling. As in Law and Kelton (2000), a process is a time-ordered sequence of interrelated events that are separated by time intervals and that describe the entire experience of an entity that flows through a system. A system consists of resources that are used by entities. Resources can be of complex nature. We allow for resources that hide their internal complexity by the concept of services that are used by entities and whose internal description gives rise to a hierarchical description based on the refinement of actions and the inclusion of (internal) resources. We distinguish among different types of entities and denote them as process chains. A process chain describes the potential behavior of all entities of a certain type, while a process describes the events of a particular entity. Process interaction models capture many real world scenarios in a natural manner. For example, an entity can be a software process in a computer system, a message in a communication system, a part in a manufacturing system.

A typical goal of simulation modeling is to quantify how much resources are used by entities and how much the progress of an entity is delayed due to sharing of resources and dependencies among entities. Those goals are formulated as performance measures (or with slight variations as performability, dependability measures if resources may fail). We consider stochastic models, so if a simulator performs a simulation run, it usually computes estimates of mean and variance of measures. Those measures highly aggregate the detailed behavior of entities to more abstract terms like mean utilization of a resource or mean waiting time for entities at a resource.

This leads to the following situation: given that results of a simulation run are not acceptable, then there is a need to investigate the reasons for those results in more detail in order to either identify deficits in the model and/or the simulation software or gain more insight in the behavior of the system under study. This happens in the verification and validation phase of a simulation study as well as in the experimental phase.

Current simulation packages provide a number of features to support a modeler in this situation. Among others, Arena (Kelton et al. 2002) and Automod (Banks 2000) support a visual inspection of what happens in a simulation run by advanced 3D animations in particular for simulating manufacturing systems. A visualization can animate the procurement of products by 3D icons of parts that are moved by animated transportation means among animated 3D objects of machines. The visual inspection allows a modeler to observe a simulation run in terms of the modeled system. Another possibility is to animate a model description, e.g., by highlighting elements of the description that correspond to states and events of the simulator. A modeler observes a simulation run in terms of the model specification. This can be in a monitoring mode (either on the fly or from a precomputed run) or in an interactive mode where the modeler can select events to be performed (typically by ignoring the stochastic timing specification of a given model). A step further towards the simulator code is to consider the state of the simulator after each event.

This is called tracing if a simulation run is considered step by step, interactive debugging if in addition to monitoring the state, the modeler is able to make manually changes to the state of the simulator. Finally, a simulator can write states and events to a file, a batch trace, that can be used for subsequent analysis be it an animation or a step by step walk through of the simulator code.

All these approaches are valuable. However, we see the following difficulties. Any sequential observation requires the modeler to memorize what events lead to the current state in order to grasp the reasons for the current situation. It would be helpful if a visualization allows to track back how a simulation run evolved to the current state. Simulation runs tend to be lengthy such that it is crucial to select relevant time intervals that are worthwhile to be considered in detail in order to find the causes of a phenomenon. Finally, selection of an appropriate level of detail is a challenge, it ranges from a high-level 3D animation in terms of the modeled system to the most detailed technical view of a debugger that controls a stepwise execution of the simulator code.

We propose a visualization aid based on Message Sequence Charts (MSCs). MSCs (Recommendation 1996) and the much similar Sequence Diagrams known from the Unified Modeling Language (UML) also have a notion of process which we denote by MSC process for clarity. MSCs have been used for the visualization of the behavior of parallel programs, for instance in XPVM (Geist et al. 1996), and for the specification of software in UML and SDL (SDL Forum Society). MSCs are a partial order concept that is suitable to visualize a trace as a total order concept, if the simulation model follows a process interaction approach (Kemper and Tepper 2005). Since an MSC represents a trace completely, it naturally allows a navigation backwards from the current state to track down events that contributed to that state. If an operation like grouping of processes is supported, MSCs also allow a modeler to adjust the level of detail. The open issue is how to identify interesting, relevant or suspicious parts of a trace that have an impact on performance measures. This requires to consider the timing information contained in a trace. The contribution of this paper is to enhance MSCs with operations that visualize quantitative information, e.g., entities that contribute to a high variance for lead times of entities and service times of resources. We propose 3 operations and describe 5 scenarios where those operations help to identify the causes of unexpected, obscure, or undesired behavior of a simulation model as documented in a trace. The paper is structured as follows. In Section 2, we briefly comment on suitable modeling formalisms, consider the ProC/B formalism as a particular example and discuss which pieces of information a trace should contain for its analysis. In Section 3, we recall MSCs, interpret a trace as an MSC and discuss how MSCs are used for trouble shooting simulation models in Section 4. After some notes on implementation issues in Section 5, we conclude in Section 6.

2 MODELING FORMALISMS

In this paper, we follow a process approach to simulation modeling. We consider formalisms for open systems, where entities are created dynamically during a simulation run. Entities may communicate via shared memory or by synchronization over common actions. Entities acquire and release resources. Resources provide services that an entity can employ to perform an action. The notion of resource and service can be refined in a hierarchical manner such that using a service is like calling a function in a computer program.

The ProC/B formalism by Bause et al. (2002) is an example in this family of formalisms. It is a graphical modeling language where entities are created dynamically during a simulation run at source nodes, they perform a sequence of actions described by a process chain, and they terminate at sink nodes. A process chain describes the possible behavior for a whole class of entities. It is a description that allows for parameters, local variables, and a control flow description build upon common operators of programming languages, i.e., sequence, if-then-else with a state-dependent or probabilistic choice, a fork and join operator for internal parallelism, a loop operator and furthermore. Communication between entities is possible via shared variables and by synchronization operations. For each action of a process chain, it is possible to assign a service of a resource if the resource shall perform that action. Counter and server are two basic predefined resources. A counter is used to model space allocation and release, it is used to model a passive resource, a state variable of finite domain. A server is used to model time consumption, to model an active resource like a queue with a queuing discipline and particular speed. For complex actions, there is the notion of a Functional Unit (FU) that is a generic resource which offers services to an entity and which may internally have a process chain description for its detailed behavior. FUs encapsulate resources of their own and internal descriptions how a service is performed. This gives rise to a hierarchical model description based on action refinement (like function calls, but without recursion) and resource inclusion. We illustrate the ProC/B formalism by a small example model of a warehouse as given in Figure 1, see (Bause et al. 2002) for details on ProC/B. Trucks load and unload goods at a warehouse with a storage area of finite capacity and a number of staff and fork lifts. The ProC/B model of the warehouse is hierarchical and Figure 2 presents the part of FU *Store_WH*. FU *Store_WH* models the warehouse as a resource with two services, loading (*Store_out*) and unloading (*Store_in*) trucks that arrive. It contains FUs *ForkLift* and *Staff* each of which provides get

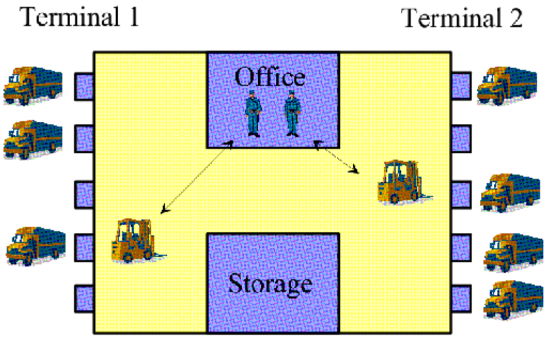

Figure 1: Sketch of a Warehouse

and put operations for the 2 fork lifts, resp. 2 persons it contains. A loading (unloading) service is a sequence of actions in Figure 2 which reads as follows: upon service call, e.g., for *Store_in*, it allocates 1 person from FU *Staff* at *Req_Staff1*. Afterwards, a probabilistic choice describes that the upper branch is taken with probability 0.9, the lower with probability 0.1. In the upper branch, the service allocates 1 fork lift from FU *ForkLift*, and finally it puts x units of goods into FU *Storage*, before it releases the fork lift and staff and returns. Loading goods into the store takes time but the put service at FU *Storage* is immediate; hence that duration is modeled by the delay node that has been inserted after the put operation and before the release of resource. The lower branch models a manual treatment of particular cases. The semantics of service *Store_out* is similar but describes loading goods from the store into the truck. The amount of goods and the time delays are described by random variables whose details on the selected distributions we omit. Given some additional specification on the arrival streams of trucks that request loading or unloading services, we can run a simulation.

The warehouse is used as a running example to illustrate various concepts. The ProC/B formalism serves as an example for a hierarchical formalism that is based on process interaction, resource usage and refinement of resources based on a call-semantics. Given such a model specification, a set of measures to observe and a starting state and seed, a simulator generates a sequence of events. The states that are reached and the actions that are performed at certain points in time give samples for the particular measures that the simulator shall evaluate. Frequent measures of interest are related to performance and measured by lead times of entities, the time they spend at resources for queuing or being blocked, and utilization of a resource. The aim is to identify bottlenecks, causes for high variances etc. In addition to this, performability, dependability, availability considers measures that distinguish whether a resource is up and running or down and not performing any service. In the warehouse example, there is interest in the utilizations of fork lifts and staff and the degree of filling for the storage capacity. Furthermore, trucks should not be delayed too much, so the total time used for loading or unloading operations gives rise to a performance measure as well. In discrete event simulation of stochastic models, measures are typically evaluated as estimates of mean and variance of random variables. From a conceptual point of view, a simulator generates a set of samples $\{X_1, \ldots, X_n\}$ for a random variable X that corresponds to a measure of interest. Those samples give rise to an empirical distribution $F(X)$. More precisely, let X be a continuous variable, let the sample values be ordered and $X_{(k)}$ denote the kth smallest of those values, so that $X_{(k)} \leq X_{(k+1)}$ for $k = 1, ..., n-1$. Then the empirical distribution is $F(x) = 0$ if $x < X_{(1)}$,

$$F(x) = \frac{k-1}{n-1} + \frac{x - X_{(k)}}{(n-1)(X_{(k+1)} - X_{(k)})}$$

if $X_{(k)} \leq x \leq X_{(k+1)}$ for $k = 1, 2, \ldots, n-1$, and $F(x) = 1$ if $X_{(n)} \leq x$. In case of a discrete variable, an empirical mass function is simply based on the proportion of the X_k's that are equal to x. Usually, a simulator estimates the mean value of X on-the-fly based on the sample mean $\bar{X}(n)$. In addition, $\bar{X}(n) \pm c\sqrt{S^2(n)/n}$ gives the confidence interval for the estimated mean of random variable X based on the sample variance $S^2(n)$ and c being either $c = z_{1-\alpha/2}$ or $c = t_{n-1,1-\alpha/2}$ taken from a normal or t distribution for confidence level α, for more details see (Law and Kelton 2000) pp. 254ff and Chapter 6.2.4. We make use of $F(X)$ to highlight particular elements of $\{X_1, \ldots, X_n\}$ in a trace, while a simulator usually only provides $\bar{X}(n)$ and its confidence interval. Simulator code can be annotated in many ways to reveal information on states and events for the generation of a trace. We make the following assumptions on the information that is present in a trace. Every entity has a unique identifier i taken from a finite set I and a class or type c of some finite set C, so that I can be partitioned according to types C. We choose I such that the corresponding value of c for an $i \in I$ (denoted by $c(i)$) can be deduced from the value of i. Every FU has a unique identifier f taken from a finite set F. Each state variable is either associated with a particular entity i or FU f or it is a global variable that is associated by definition with an additional (artificial) entity. An action $a = (ai, ac, w, O, i, ts, cr)$ is a 7-tuple where ai is a unique identifier, ac is the type or class of the action, e.g., the name of a service that is called. $w \in F \cup I$ and $O \in 2^{F \cup I}$ describe which entities and FUs are involved in a. a is a local action in entity or FU w if $O = \emptyset$. a is an action between w and elements of O if $O \neq \emptyset$. If appropriate according to the type ac, w is understood as initiator of an event, O as other entities or FUs that are involved, e.g., for a service call, w indicates the calling entity, $O = \{f\}$ indicates the FU f whose service is called. $i \in I$ denotes which entity is the cause for a, i.e., in case of nested service calls among FUs, i shows which entity initiated that cascade of service calls. This piece of information

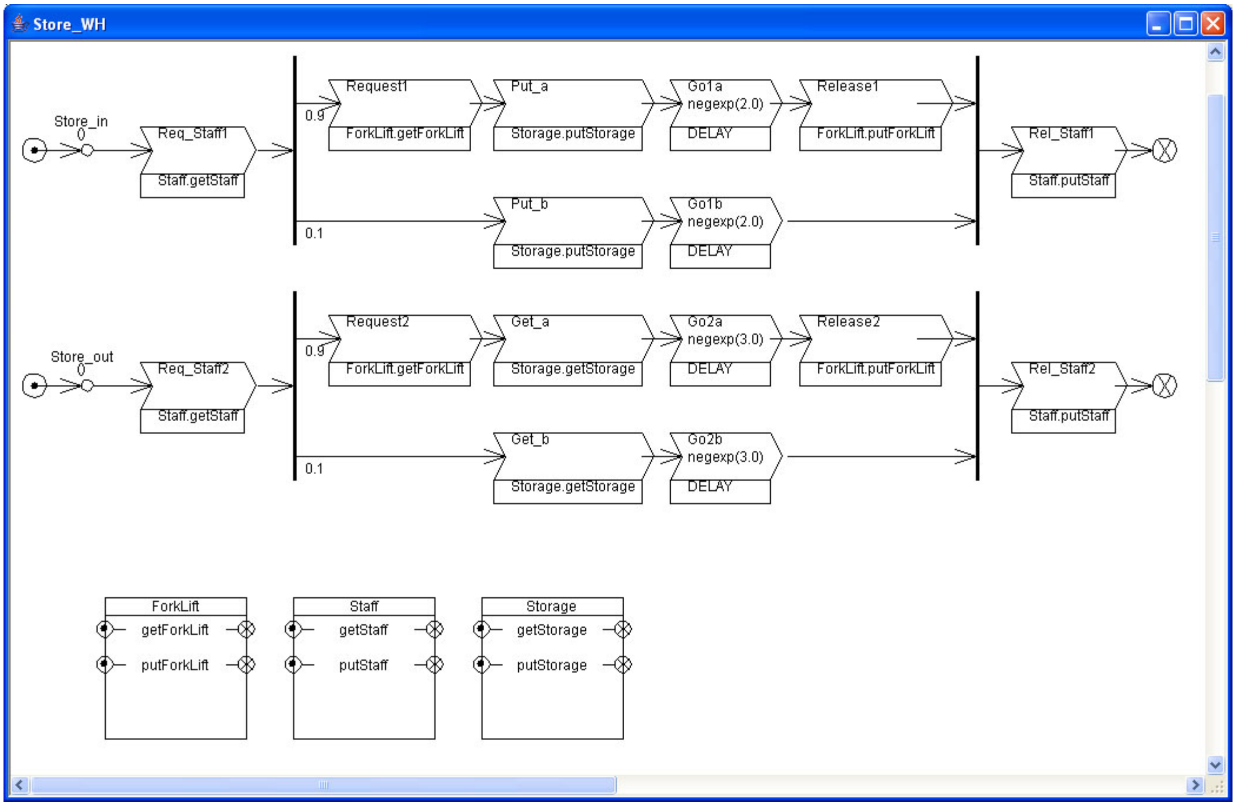

Figure 2: Functional Unit Store_WH in ProC/B Notation

allows us to associate every action with the entity it belongs to. $ts \in \mathbb{R}$ is a time stamp that describes at which point of time an action takes place. $cr \in \mathbb{N}$ is an identifier that can occur at most twice, it matches pairs of service calls and returns. It will be used to track the duration of service used by entity i at an FU f. Let $a = (ai, ac, w, \{f\}, i, ts, cr)$ be a service call of type ac of an FU w on behalf of entity i at time ts, then $a' = (ai', ac', f, \{w\}, i, ts', cr)$ with same id cr describes the return of that call at time ts'. We define $d_{ai}(ac) = ts' - ts$ as service time of ai. The service time of actions of type ac is a random variable $D(ac)$ with a set of samples given by $d_{ai}(ac)$ for all actions ai of type ac in a trace. $D(ac)$ has an empirical distribution $F(d(ac))$. Similarly, we can consider the life-span $l(i)$ of an entity $i \in I$ by the difference between time stamps of the first action (of a particular type ac for creation of an entity) and the last action (of a particular type ac for termination of an entity) that relate to i. Again, a trace yields a set of samples for entities of type c, such that we obtain an empirical distribution $F(l(c))$ for the life-span of entities of type c. Entities that do not terminate and service calls that do not return are excluded from considerations of $F(d(ac))$, respectively $F(l(c))$ and treated specially. We will make use of the empirical distributions to identify extremal behavior, i.e., entities with a long life-span and services with extreme service times in Section 4. Before we discuss which visualization is helpful for which type of question, we briefly recall the definition of MSCs.

3 MESSAGE SEQUENCE CHARTS

To track down particularities of the dynamic behavior of a complex model, we believe it is natural that one wants to see how entities proceed and how they interact with each other and available resources. MSCs set the focus on processes and their interactions.

Definition 1 *An MSC M is defined as a tuple $M = (V, <, P, M, K, T, N, m)$, where V is a finite set of events, $<\subseteq V \times V$ is an acyclic relation, P is a set of (MSC) processes, M is a set of message names, $L : V \rightarrow P$ is a mapping that associates each event with a process, $K : V \rightarrow s, r, b, l$ is a mapping that describes the kind of each event as send, receive, broadcast or local, $N : V \rightarrow M$ maps every event to a name, $m = m_{sr} \cup m_b$ is a relation called matching with $m_{sr} \subseteq V \times V$ that pairs send and receive events. Each send is paired with exactly one receive and vice versa. Events v_1 and v_2 can be paired, only if $N(v_1) = N(v_2)$. $M_b \subseteq 2^V$ relates up to $|P|$ broadcasting events.*

An MSC defines a labeled directed acyclic graph. Figure 3 visualizes part of a trace for the example of Figure 1 as an MSC. Each MSC process p is shown as a vertical line where events $v \in V$ with $L(v) = p$ are ordered from top to bottom according to the ordering relation < defined for events. Send, receive and broadcast events result in horizontal lines that connect all MSC processes that are involved. Sender and receivers are not distinguished for a broadcasting event. Figure 3 shows that MSC process *PC_Deliver* interacts with process *FU_Store_in*, which itself interacts with process *FU_Staff* and *FU_Storage*. Note that events can be arranged in the MSC proportional to a global clock based on the information of a time stamp per event. However, in the figures given in this paper, we switched to a representation that assumes a (virtual) grid with horizontal lines and has a least one event per line. This retains the order of events as given in the trace but distances are not proportional with respect to time stamps. More concise orderings are possible, we refer to (Kemper and Tepper 2005). The mapping from the ProC/B model that considers entities and services of FUs to MSCs needs further clarification and is considered next.

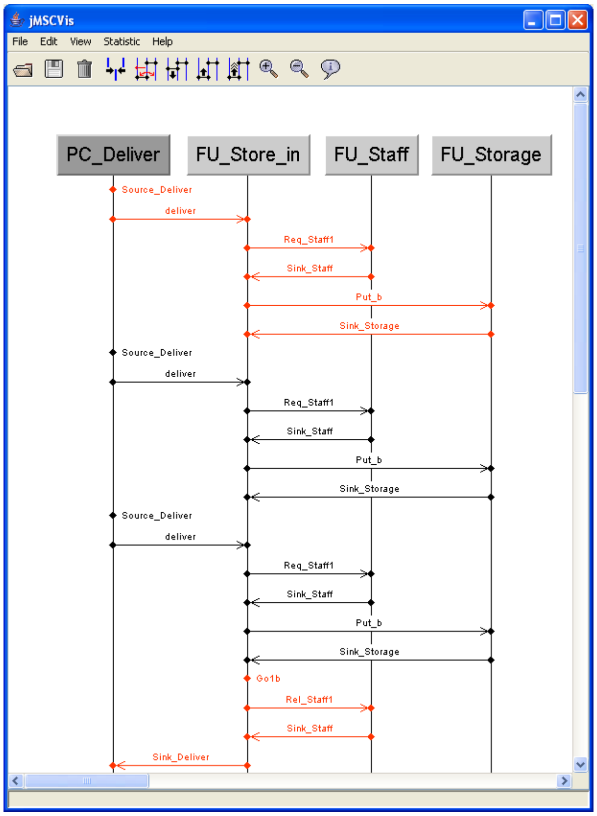

Figure 3: MSC with One Entity Highlighted

3.1 Mapping ProC/B Traces to MSCs

A trace of a ProC/B model contains states and actions of entities, their interactions and service calls to FUs. Among the many possible mappings, we selected one where each entity i and each service s of a functional unit gives an individual MSC process $p(i)$, resp. $p(s)$. If an entity calls a service of a functional unit, this results in a send event from process $p(i)$ to $p(s)$. If that service call terminates and returns, there is a send event from $p(s)$ to $p(i)$. Synchronization among entities (or services) results in broadcasting events, where the last entity that reaches a synchronization operator and releases other entities becomes the sender while waiting entities become the receivers of that broadcasting event. Actions other than those used for service calls and synchronizing entities become local events in an MSC process. The mapping results in a finite number of MSC processes for any finite trace. We expect the number of entities to be high and the lifetime of a single entity to be rather short compared to the length of the overall trace. Hence we seek to reduce the number of MSC processes that are visualized by joining (merging) processes. As a first step, we join all entities of same type into one MSC process, i.e., we obtain one MSC process per process chain. In order to be able to distinguish among entities, we attach the identifiers (a numerical integer value) of each involved entity to its corresponding events. Identifiers are optionally visualized in the MSC with their corresponding events.

For the warehouse example, Figure 3 contains one MSC process *PC_Deliver* that contains events of all entities of the process chain for unloading in the ProC/B model; same for *PC_Pick_up* and loading which is not visible and more to the right in the canvas shown in Figure 3. Each service of an FU results in a MSC process, which yields 8 processes, one *Store_in* and one *Store_out* process for FU *Store_WH*, one *get* and one *put* process per FU *FU_Staff*, *FU_ForkLift*, *FU_Storage*. In Figure 3, *Store_out* and processes of *FU_ForkLift* reside more to the right and are not visible, get and put services of *FU_Staff* have been merged for clarity, same for *FU_Storage*. The merging of MSC processes to retain a reasonably clear visualization is frequently used for non-trivial cases.

3.2 Visualizing Operations for MSCs

Traces tend to provide a lot of information and very detailed information. Since there is no intelligent way to automatically identify the most appropriate level of aggregation for a human to understand the dynamic behavior, a tool must provide a number of operations to dynamically aggregate or refine the visualized amount of data shown from the trace. In Kemper and Tepper (2005), we discussed the following set of operations: zooming and scrolling, reduction of data by projections, reduction of representation by merging processes, highlighting and coloring, horizontal permutations, switching between total and partial order. Merging MSC processes, e.g., for all entities of a process chain, is a fundamental operation that we frequently use to adjust the number of visualized MSC processes. All those operations neither take any timing information into account nor the pairing of MSC events for service call and return. In the following, we describe three additional visualization operations that are particularly designed to track down what happens in a trace for the considered modeling formalism and with respect to time.

3.3 Highlighting Individual Entities or Process Chains

For the given mapping of ProC/B models to MSCs, the information of which actions belong to an entity i gets distributed over FUs and their services. By coloring all actions $a = (ai, ac, w, O, i, ts, cr)$ with a particular value for entity i, we can inspect what actions belong to i over the set of FUs and the lifeline of the MSC process $p(i)$. For a tool support, this implies that sets I of entities and C of process chains of the ProC/B model are made available to a user for selection of elements $i \in I$, or $c \in C$. Furthermore, the natural generalization of this operation to subsets of I is helpful to visualize sets of entities with interesting properties.

3.4 Highlighting Extremal Behavior and Unfinished Activities

We use the empirical distributions of service times $F(d(ac))$ of type ac and life-spans $F(l(c))$ of entities of type c to identify extremal behavior, i.e., let $\alpha_1, \alpha_2 \in]0, 1[$ with $\alpha_1 \leq \alpha_2$ denote thresholds such that $F(l(c)) < \alpha_1$ gives the fraction of entities with short life-span, $\alpha_1 \leq F(l(c)) \leq \alpha_2$ give entities with a medium life-span and $\alpha_2 \leq F(l(c))$ indicate entities with a long life-span. For visualizing these three classes of entities, we can use different colors, e.g., green, black and red. We can do the same for services. This coloring scheme allows us by visual inspection to identify a number of suspects that cause mean values of life-span or service delays being high (or low) and correlations among them, e.g., long service times at one FU correlate with long service times at another. Service calls that do not return and entities that do not terminate before the simulation run ends obtain a different color. Figure 4 shows an MSC where processes are highlighted in red, black and green according to their life-spans.

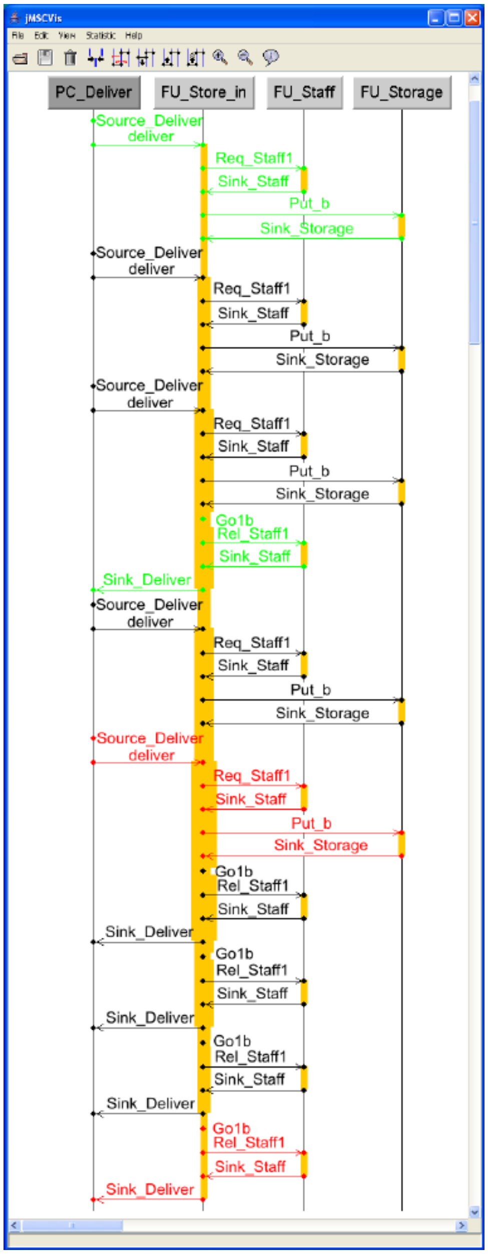

Figure 4: Busy FUs and Extreme Lifespans

3.5 Visualizing Congestion and Bottlenecks

By pairing actions for service calls with their returns, we can identify and count unfinished service calls at a FU f at any point of time and visualize its number x, for example by increasing the line thickness of the lifeline of f, respectively its service s proportional to x. This illustrates usage of FUs and congestion. We found two variants of this concept useful in practice. If an entity i calls a service s at f, this increments the value of x at $p(s)$. However, if i calls another service in a nested manner while residing at f, we can either retain the value of x at f (variant A) or decrement x (variant B). We use variant A to illustrate the amount of unfinished work per FU over time (if we join all MSC processes of services that belong to FU f). This view is accurate to see which resources are busy. It reflects that unfinished service calls are likely to make a difference to state variables of f. If one considers the service calls of i as a tree structure, then variant B focuses on the leaves of that tree. This focuses on the most recent and most deeply nested service calls and indicates which actions are currently performed. So, variant B indicates, where work is currently performed. Note that there are cases where internal parallelism of entity i is not appropriately visualized by variant B. For example, if i performs a fork operation at f, such that it is simultaneously active at f and calls another service at f', the visualization of variant B locates i being active either at f or f' but not at both. Within their limitations, variants A and B reveal which FUs are busy over time, which entities are involved and at which point in time this happens. Figure 4 shows an MSC where congestion is visualized according to variant A; obviously *FU_Store_in* is busy.

For a particular FU f let, b_j be the j-th period of time where f is busy, i.e., $x > 0$ for at least one service s of f following variant A and b'_j following variant B. The corresponding set of samples $\{b_1, \ldots, b_n\}$ (similar for b'_j) yields an empirical distribution again with mean $\bar{b}(n)$ and variance. Note that identification of most busy resources is the aim of traditional bottleneck analysis. In the literature, different ways for identification of bottlenecks are proposed. In Law and Kelton (2000), bottlenecks are identified either by measuring the waiting time in front of a resource or the percentage of time a resource is busy ($\sum_{i=1}^{n} b_j/T$ for a total time of T in our case). Roser et al. (2001) propose $\bar{b}(n)$ to measure bottlenecks, i.e., resources with highest average periods of being busy give bottlenecks. In both cases, highlighting those parts of the empirical distribution for b_j in an MSC that make $\sum_{i=1}^{n} b_j/T$ or $\bar{b}(n) = \sum_{i=1}^{n} b_j/n$ high in value gives guidance to a modeler why that FU is so busy. It is straightforward to make both variants A and B support the visualization of bottlenecks.

These operations are intended to support human perception and understanding of the dynamic behavior of a process oriented system that documents itself as a trace. In the following we discuss how to make good use of these operations.

4 TROUBLE SHOOTING BY TRACE VISUALIZATION

In this section, we discuss how traces and their MSC visualization can be used to understand more on what happens in a simulation run. The main benefit of the MSC visualization is that it allows to identify particular phases in a run where relevant phenomena occur and for those phases to give information on a very detailed level of state variables and actions. The challenge for a visualization is to guide a user to identify crucial situations in a simulation run. For a given simulation run, we propose to check a number of suspects.

4.1 Suspect 1: Non-terminating Activities

Service calls to FUs and entities that have not terminated can do so for different reasons: a) the non-termination may be intended by the modeler, b) termination is possible at a point of time beyond the end of the simulation run, and c) termination is not possible due to a permanent blocking or deadlocking situation that is not(!) intended by the modeler. The last case is the one that needs to be checked. Highlighting all non-terminated service calls and entities by a particular color helps us to identify those activities in the MSC. Clearly, a non-terminated activity has a duration from its beginning to the time at the end of the trace. Let d denote this duration. The larger the value of d is compared to the maximum value observed in the empirical distribution over values of same kind (life-span of entities or durations of service calls), the more likely it is, that the activity faces a deadlock or starvation. The MSC visualization has the option to use particular and different colors to make those activities getting more attention. We can track each involved entity and FU and check state variable information to make sure that no permanent blocking situation is present. We consult the model description to compare requirements for those actions that would yield termination and track if those are fulfilled or can be fulfilled. The trace in Figure 5 visualizes non-terminating activities and congestion at FUs following variant B. We observe immediately that service calls to *FU_Staff* increase in numbers, while service calls to *FU_ForkLift* and *FU_Storage* remain at a level of 2 till the end. We track entities i and i' from the calling events by highlighting all actions of i and i', we recognize that both are of the type that fill the buffer. By checking state variables at *FU_Storage*, we learn that the model reaches a partial deadlock because 2 entities that deliver goods get blocked due to a full storage area. The congestion at *FU_Staff* indicates that no entities that pick up goods can access the storage area because all staff at *FU_Staff* is allocated by the 2 blocked entities. It is straightforward to identify that situation in the MSC and to track its causes. There are several ways to modify the model to avoid deadlock situations: a) resource reservation or simultaneous allocation of resources, b) giving up fork lifts and staff if blocking occurs at *FU_Storage*, c) a scheduler reorders access to the store such that no blocking occurs, among others. We focus on the last variant since for a real system, trucks would sign up and identify their task when entering the parking lot of the facility. A controlling person would arrange for an order that works or would refuse access otherwise.

4.2 Suspect 2: Extremely Slow Entities

At this point, let a simulation run produce estimates of performance figures that are not satisfying. In particular, let lead times of entities be too high or too low on average, values show a variance that is considered too high, or utilization of resources be too high or too low. So, a modeler be interested in understanding what makes the observed figures being as they are. Given that a simulation model can incur variances in the delay of entities only by a) blocking effects, b) queuing delays at shared resources, and c) high variances of stochastic delays that allow for extreme

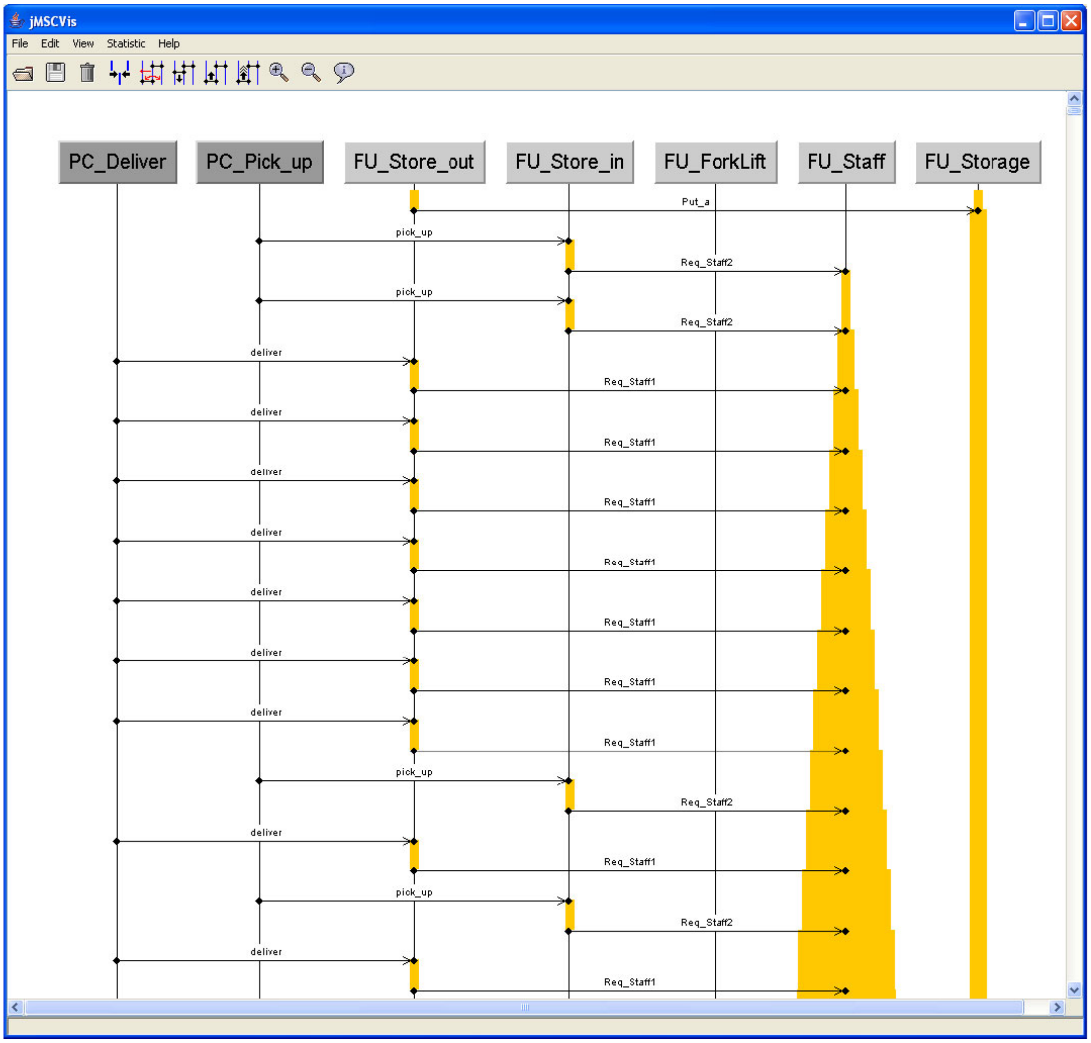

Figure 5: Deadlock Detection

values of random numbers, we discuss how to track down those cases with MSCs.

If the estimated mean life-span $l(c)$ of entities of type c is high in value or does have a large confidence interval this is often caused by the extreme cases that occur in the empirical distribution $F(l(c))$. In an MSC, we can highlight all actions $a = (..., i, ...)$ that correspond to an entity i where $l(c, i)$ is greater than a threshold x, or alternatively if $\alpha \leq F(l(c) = x)$. However, if i uses services with nested services calls to other FUs, it is helpful if we can easily identify those service calls that are served slowly. Served slowly can be understood as a service delay with a high value (absolute numbers) or as a service delay that is relatively long compared to other service calls of same kind that are present in the trace. We believe the latter is more likely to incur a high variance. Let i make use of a service ai of type ac provided by a FU f. Since the empirical distribution $F(d(ac))$ is known for f, we can quantify whether the delay $d(ai)$ is fast or slow on the minimum and maximum range given by $F(d(ac))$. Coloring ai according to being fast or slow wrt $F(d(ac))$ helps identifying the main contributors to the life-span of i if it is in the FUs that are used by i. In addition, we can visualize the number of unfinished service calls at each FU at each point of time (following variant A or B) to illustrate where that entity i relies on congested resources.

4.3 Suspect 3: Extremely Fast Entities

As for the slow entities, extremely fast entities contribute to a high variance and large confidence intervals for the estimated mean life-span $l(c)$. Clearly, fast usually is positive, e.g., in production systems, however, extremely fast may be due to errors in a model specification. So for both reasons, a high variance and functional deficiencies, fast entities require attention as well. We make use of the same MSC visualization techniques as for the extremely slow entities (but with obvious adaptations).

4.4 Suspect 4: Heavily Utilized Resources and Blocking

Identification of bottlenecks is classic issue in simulation based performance analysis. A bottleneck is a resource that is insufficiently dimensioned to cope with its workload. Bottleneck analysis aims at determining the location of the constraining resource(s). They are throttling the throughput and cause delays of entities that use them. Or in other words, there are few occasions where it is idle. Resources are FUs, counters and servers which are all mapped to individual MSC processes and their service call and return actions give rise to the notion of service delays and a visualization of congestion as discussed in Section 2. We visualize congestion at resources by the number of unfinished service calls over time for the lifeline of each FU/counter/server. Both variants, A and B, are useful; while A gives an impression how open service calls build up per FU, variant B identifies where processing takes place in a particular situation. Note that the latter is only true for sequential entities or services. If a service at FU f contains parallel activities a and a' individually calls a service of another FU f', variant B does not visualize activity a that is still processed at f.

We can make use of variant A (and variant B in case of sequential entities) for bottleneck analysis. As mentioned in Section 2, existing techniques for bottleneck identification consider the busy periods of resources (FUs, server and counter in our case). By highlighting busy periods b_j or b'_j that are located in the corresponding empirical distribution $F(b)$ with $F(b_j) > 1 - \alpha$ for some chosen $\alpha \in [0, 1]$ helps to pinpoint which periods make a particular f so high in value of $\bar{b}(n)$ or $\sum_{j=1}^{n} b_j / T$ to make f a bottleneck. In addition, each FU can be checked for the type of workload over time, e.g., if the work load is bursty, i.e., if few service calls occur that last extremely long (by visualization of extremely long service calls of $F(d(ac))$ or do services occur in batches of rather short calls (by visualization of the number of unfinished service calls).

In addition to bottlenecks, blocking may slow down entities as well. If an entity, its process chain, is the bottleneck due to synchronization, it can be identified as the one with little blocking at synchronization points. Hence there must be direct interaction among entities, and it is interesting whether at a particular action (namely the synchronization) always the same type of entities have to wait or not. In the MSC visualization, actions of a particular type can be highlighted and checked for which entities (resp. its process chain) where waiting for that action according the the time stamp of the preceding action. Coloring waiting phases towards a synchronization in red guides a user. Similar, a counter can be a bottleneck due to a limited value range and blocking at one bound of that range. This can be identified by a mean value close to that bound or again by highlighting waiting times for entities that access the counter.

4.5 Suspect 5: Transient Behavior, Identification of Startup-phase

For simulation experiments that are conducted to investigate in the long-term behavior of a model, the influence of the initial state is unwanted and typically removed from statistical evaluations by neglecting any measurements in an initial warmup or startup phase. For a modeler, the challenge is to identify the end of the startup phase. Several heuristics exists in the literature and in tools to support or automize this decision. Nevertheless, once a simulation run has been performed, it might be interesting to judge it for transient behavior. Our MSC visualization supports this in the following way. We observed that service times at FUs and life-spans of processes are likely to be extreme in an initial warmup phase (with respect to the corresponding empirical distribution) while in the rest of a trace, service times and life-spans have values from the whole range of values observed in the empirical distribution if the model shows a stationary behavior. If the trace is too short for a stationary behavior or the model fails to show a stationary behavior in general, it is likely that values of service times and life-spans follow a rather monotonous sequence, i.e., the value range of the empirical distribution is followed from one side to the other which results in a coloring of the MSC trace with 3 distinguishable phases, e.g. from green to red via black (or vice versa). We do not claim, that the MSC coloring is superior to other visualizations of a warmup phase, but it provides the potential to obtain a simple overview as well as the option to track down the details of the behavior.

5 IMPLEMENTATION ISSUES

Note that visualization of traces by MSCs is rather independent from the ProC/B formalism, respectively the simulation modeling formalism. Consequently, we developed an XML file format for traces that consists of some header information that defines the set of MSC processes P, state variables and names of events associated with each process as well as a static matching for any send, receive or broadcasting event to its processes. The trace as an ordered sequence of events (optionally an alternating ordered sequence of states and events) follows that header. The header avoids repetition of redundant information in the sequence of events that forms the trace.

6 CONCLUSIONS

We aim to improve the productivity in the verification, validation and testing phase of a simulation study by considering appropriate visualizations of simulation traces. We focus on process interaction models and propose a visualization by a variant of MSCs that highlights extremal delays for processes as well as resource usage. We discuss the value of that visualization to identify partial deadlocks as well as congestion, queuing and blocking situations, bottlenecks and an initial startup phase. An implementation of the approach in Java is evaluated for traces generated by the ProC/B simulation environment. Future work goes into handling large amounts of data which includes reduction techniques for its visualization by MSCs and implementation improvements that makes more effective use of available graphics hardware.

REFERENCES

Banks, J. 2000. *Getting started with AutoMod*. Bountiful, UT: AutoSimulations, Inc.

Bause, F. et al. 2002. The ProC/B toolset for the modelling and analysis of process chains. In *Computer Performance Evaluation, Modelling Techniques and Tools*, Springer LNCS 2324, 51–70.

Geist, G. A., J. Kohl, and P. Papadopoulos. 1996. Visualization, debugging, and performance in PVM, Proc. Visualization and Debugging Workshop (Oct 1994). In M.L. Simmons and et al, editors, *Debugging and Performance Tuning for Parallel Computing Systems*, 65–77. IEEE.

Kelton, W.D., R. P. Sadowski, and D. A. Sadowski. 2002. *Simulation with Arena*. Mc Graw Hill.

Kemper, P., and C. Tepper. 2005. Visualizing the dynamic behavior of ProC/B models. In T. Schulze et al., editor, *Simulation und Visualisierung*, 63–74. SCS Publishing House.

Law, A., and W.D. Kelton. 2000. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition.

Recommendation 1996. ITU-T Recommendation Z.120. *Message Sequence Charts (MSC'96)*.

Roser, C., M. Nakano, and M. Tamaka. 2001. A practical bottleneck detection method. In *Proceedings of the 2001 Winter Simulation Conference*, ed. B. A. Peters, J. S. Smith, D. J. Medeiros, M. W. Rohrer, 949–953. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

SDL Forum Society. <http://www.sdl-forum.org>.

AUTHOR BIOGRAPHIES

PETER KEMPER holds a Diploma degree in computer science (Dipl.-Inform., 1992) and a Doctoral degree (Dr.rer.nat., 1996), both from the Universität Dortmund, Germany. He performs research and lectures in the Department of Computer Science at Universität Dortmund, Germany. His main interests are in the quantitative evaluation of systems and formal aspects of software engineering. Since 1998, Dr. Kemper contributes to the collaborative research center on modeling large networks in logistics, SFB 559, funded by Deutsche Forschungsgesellschaft. His e-mail address is <peter.kemper@udo.edu>.

CARSTEN TEPPER received a Diploma degree in computer science (Dipl.-Inform., 2000) from the Universität Dortmund, Germany. He is a member of the group 'Modelling and Simulation' in the Department of Computer Science at the Universität Dortmund. His main interests are in the verification and validation of process-oriented systems. He is also a member of the collaborative research center on modeling large networks in logistics, SFB 559, funded by Deutsche Forschungsgesellschaft. His e-mail address is <carsten.tepper@udo.edu>.