A Petri Net Approach to Verify and Debug Simulation Models

Peter Kemper[1] and Carsten Tepper[2]

[1] College of William and Mary, Department of Computer Science
Williamsburg, Virginia 23187-8795, USA
kemper@cs.wm.edu
[2] Universität Dortmund, LS Informatik IV
D-44221 Dortmund, Germany
tepper@udo.edu

Abstract. Verification and Simulation share many issues, one is that simulation models require validation and verification. In the context of simulation, verification is understood as the task to ensure that an executable simulation model matches its conceptual counterpart while validation is the task to ensure that a simulation model represents the system under study well enough with respect to the goals of the simulation study. Both, validation and verification, are treated in the literature at a rather high level and seem to be more an art than engineering. This paper considers discrete event simulation of stochastic models that are formulated in a process-oriented language. The ProC/B paradigm is used as a particular example of a class of simulation languages which follow the common process interaction approach and show common concepts used in performance modeling, namely a) layered systems of virtual machines that contain resources and provide services and b) concurrent processes that interact by message passing and shared memory. We describe how Petri net analysis techniques help to verify and debug a large and detailed simulation model of airport logistics. We automatically derive a Petri net that models the control flow of a Proc/B model and we make use of invariant analysis and modelchecking to shed light on the allocation of resources, constraints among entities and causes for deadlocks. This paper is a revised version of [1].

Keywords. Discrete event simulation, verification, debugging, process interaction, Petri net analysis

1 Introduction

Simulation is a very popular and common analysis technique to obtain quantitative results for the dynamic behavior of discrete event dynamic systems. It allows to analyse very detailed and fine grain models if necessary, which makes it the method of choice for instance in application areas like manufacturing systems and logistics.

However, large and detailed models are a challenge for verification, validation, and debugging of simulator code. The price to pay for a powerful simulation language is that debugging a model is about as difficult as debugging software in general, and if the simulation model adopts the common and popular process-based approach then the complexity is that of debugging multithreaded programs. In this paper, we try to get beyond what can be achieved by syntax checking and interactive step-by-step debugging of simulator code as it is provided by state-of-the-art simulation software. Our research has been stimulated by a simulation study of an air cargo network. The resulting simulation model is formulated in the ProC/B notation, a process-based graphical simulation language. The goal of that case study is to quantify the effect of different bundling strategies that are applied at two hubs in a network of airlines and airports for the transportation of packets. Our interest is in the lessons learnt in the process of creating an executable simulation model that is supposed to describe the behavior of a conceptual model. We recognized that once simple errors, which are detected by a syntax and consistency check, have been corrected, there is a class of errors that yield partial deadlocks among interacting processes and that has its roots in constraints due to the finite capacity of passive resources and simultaneous allocation of multiple resources. We propose to map such a model to a Petri net and use invariant analysis and modelchecking to identify constraints of a model or particular submodels. The technique applies to submodels in isolation as well as to a hierarchically structured overall model.

The paper is structured as follows. In Section 2, we briefly recall main concepts of the ProC/B notation and describe a ProC/B model that is used in a case study of air cargo traffic and that has stimulated this research. In Section 3, we precisely specify the particular issues of model verification that we address. Section 4 recalls Petri nets and describes how the control flow of a ProC/B model is mapped to a Petri Net. Section 5 considers analysis issues, in Section 5.1 we recall invariant analysis and propose its use for finding constraints and for identifying the role of resources in a ProC/B model. Section 5.2 shows to what extend modelchecking and liveness analysis of a Petri net helps to find deadlocks in a submodel. We conclude in Section 6.

2 Modeling Air Cargo Networks with Process Chains

An air cargo network can be organized in many ways. In [2], the potential economic impact of various strategies for bundling traffic at hubs is evaluated with the help of a large and detailed simulation model.

That model follows a process-based approach, where packets are entities that travel through a network of resources from a starting airport towards a destination airport via two hubs. The model incorporates a flight schedule with exact dates and planes of different kinds, in particular planes of different capacities. Packets are entities that require storage space at their current location and time to handle them with the necessary resources (personnel, fork lifts etc). The chosen level of detail implied a large model that is structured into a hierarchy of 3

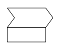
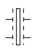
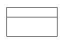
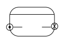
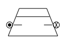
ProC/B element	Symbol	Semantic for Dynamic Behavior
`Source`	$\odot$	process creation
`Sink`	$\otimes$	process termination
`Process Chain` `Element` (PCE)		activity in a process chain, semantics: (stochastic) delay, service call
`Sequence`	$\longrightarrow$	sequential order of activities, from left to right
`AND-Connector`		synchronization of incoming activities (left) with joint start of outgoing activities (right)
`OR-Connector`		probab. /bool. branching, selects single activity (right), triggered by termination of any activity from left
`PC-Connector`		synchronization of processes (left side) continuation or creation of processes (right side)
`Functional Unit` (FU)		structuring element, a form of container, provides services, contains resources
`Server`		familiar queuing station
`Counter`		familiar passive resource (semaphore etc.)

Table 1. Core ProC/B modeling elements: graphical denotation and semantics

levels and consists of a number of submodels. The model is open in the sense that packets are generated at airports and terminate at their individual destination airports, so the number of packets in the system is not constant and not limited (although storage areas are). In this section, we will give more details on the model and use it as a running example to illustrate aspects of the ProC/B modeling formalism [3]. The ProC/B formalism has been developed in the collaborative research center "Modeling of Large Logistic Networks" (SFB 559)[1] at the university of Dortmund and it is dedicated towards modeling logistic networks in a process-based manner. A set of software tools complement the formalism to allow for a graphical model specification and simulative analysis of ProC/B models. From a conceptual point of view, the ProC/B notation yields simulation models that are service networks with active resources like queueing servers and passive resources like storage areas where a dynamic number of entities are generated at some source nodes, experience a life-cycle that follows a given set of possibly complex patterns of behavior (called process chains, PCs), and terminate at some sink node. Table 1 shows core ProC/B modelling elements. A Process Chain Element (PCE) is an elementary operation that can describe a delay, or a service call to a functional unit, which is analogous to a function call in programming, or some native code directly executed by the

[1] see http://www.sfb559.uni-dortmund.de/index.php?lang=eng

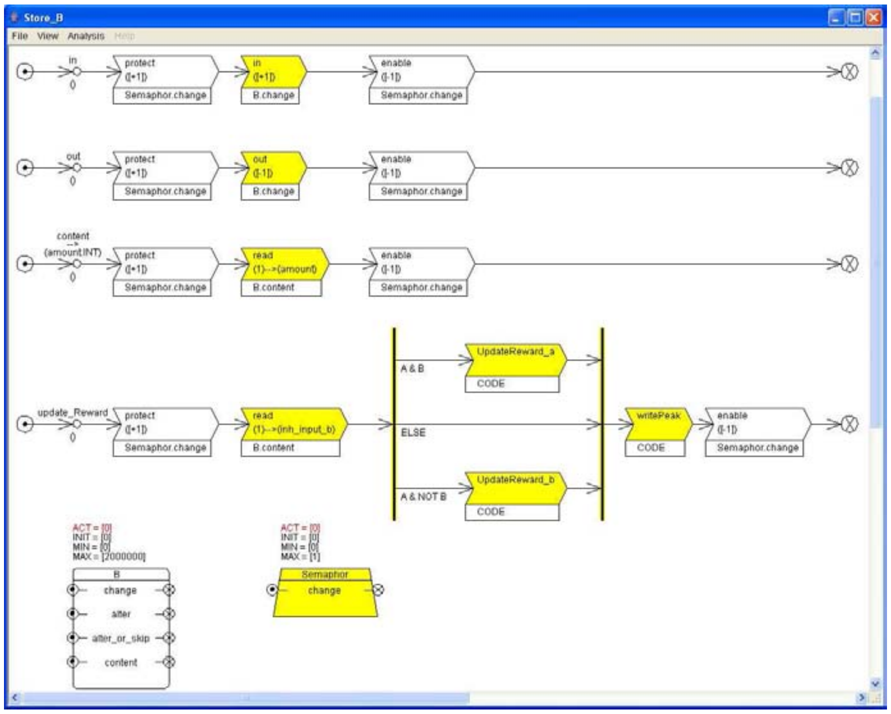
simulation engine. Arrows connect PCEs and in this way they also define an order among PCEs and other nodes of a ProC/B graph. Other elements as for branching, fork and join, and dynamic creation of entities are familiar concepts known from progamming languages.

Entities can carry parameters, so do service / function calls. Two types of resources are given, counters (like local variables of finite domain) correspond to passive resources, servers correspond to queueing stations and are used to model load-dependent delays and queueing effects. An entity performs steps as specified by its process chain. An entity may experience delays at delay PCEs and at servers; it may be stopped for some time or forever while waiting for a sufficient number of other entities to arrive at an AND-connector, at a PC-connector or at a sequence with a given cardinality that carries only batches of entities. In addition, it may also be blocked at an access operation to a counter that would give a range violation, e.g., a counter with range $\{0, 1, , \ldots, 5\}$ would block a subtraction that results in a value less than 0 and an addition that would results in a value greater than 5.

The ProC/B formalism has two notions of hierarchy, namely a hierarchy based on refinement of actions and a hierarchy based on inclusion of resources, and it integrates both into a single concept, namely that of a functional unit (FU). Fig. 1 shows a FU for a storage area which is employed in several slightly larger versions in the air cargo model. A FU provides services that can be called from their environment, and whose detailed behavior is defined inside its FU. In Fig. 1, there are 4 services named *in*, *out*, *content*, and *updateReward*. The behavior of a service is expressed again by PCs, i.e., process patterns made up of activities. For example, service *content* has an (output) integer parameter that specifies how many packets are currently on stock; it consists of 3 PCEs, where the first allocates a semaphore, the second name *read* calls a service *content* of a contained FU named B to obtain a value for the amount of packets on stock and assign it to parameter *amount*, and the third releases the semaphore. Note, that we had to simplifiy the store FU of the real model to match space requirements of this paper. The motivation for using a semaphore is to ensure atomicity over several steps performed by a particular service. Service *out* removes packets from stock, *content* checks the number of packets on stock and *updateReward* updates performance measures that shall be evaluated in a simulation run. Note that *updateReward* contains PCEs with native simulator code and a state-dependent branching connector to model a *case-switch* construct. Fig. 1 illustrates as well that the hierarchy is also based on an inclusion of resources, i.e., the store FU contains another FU B and a counter *semaphore*. Note that this inclusion hierarchy must by acyclic, it does not allow for recursion.

The air cargo model has a hierarchy with 3 levels. At the top level, entities model packets with a source and destination airport. Those entities call services of 4 FUs. Two FUs model flight schedules for flights between hubs B and C and between hubs and feeder airports. Furthermore, there is one FU for a hub B and one for a hub C and all packets/entities call a service at each hub to make the connection from hub B to hub C. Packets are transported in so-called *unit load* *devices* (UIDs), which means that it is necessary to break down a incoming UID of packets, to reorganize packets for connecting flights and to build up new UIDs for outgoing flights. The model focuses on those procedures, hence the hub FUs are rather complex and contain sub-FUs to model the bundling of packets in further detail.

The goal of the simulation study was to evaluate how those processes are organized in a cost-efficient and reliable manner. The air cargo model is evaluated by simulation. The ProC/B toolset makes use of HIT [4] as a backend simulation engine for the quantitative analysis of ProC/B models which resulted in performance figures and cost estimates for different configurations of the model, see [2] for results.



ACT = [0]
INIT = [0]
MIN = [0]
MAX = [2000000]

ACT = [0]
INIT = [0]
MIN = [0]
MAX = [1]

Fig. 1. Definition of FU Store at hub B, invariants are highlighted

3 Problem Statement

ProC/B models formulate open systems in the sense that sources can generate an unlimited number of entities over time. This point of view also holds for FUs which experience an in principle unlimited sequence of calls to the set of services they provide. A main issue for the correctness of a model is that each entity will eventually reach a sink node - terminate in case of a process chain, return in case of a service call. At least for a common simulative analysis of the steady state behavior of a model, this would be considered a normal requirement.

The crux is that a performance model is used to study the impact of blocking situations on performance measures, however that blocking should not be permanent. ProC/B has a number of constructs that may cause blocking: a) change operations at counters which would lead to a range violation, e.g., removing objects from an empty buffer would be blocked till the buffer is sufficiently filled, b) join operations given by AND-connectors and PC-connectors with several incoming arcs, c) multiplicities at arcs that require certain batch sizes to proceed.

Hence, we want to analyze a model whether it allows for (partial) deadlocks among entities in a simulation run. Note that such liveness properties are difficult to establish in general, hence our more modest goal is to assist a modeller in the following way: 1) identify the role of counters, whether they are used to model reusable or consumable resources (hammer vs nails), 2) for each process chain and each service of a FU identify whether it can be repeated arbitrarily often or whether there are constraints on ratios between service calls of a FU or process chains of a model, 3) which parts of a process chain or service have limited access due to reusable resources (like semaphores), 4) if there are parts in a process chain or service that require simultaneous allocation of multiple resources.

In the following section, we describe how to map the control flow of a ProC/B model to a Petri net, such that Petri net analysis techniques can help us to achieve those goals.

4 Mapping a ProC/B model to a Petri Net

Before we describe how a ProC/B model can be mapped to a Petri net, we briefly recall some Petri net related terminology, see e.g., [5] for details.

PNs are directed bipartite graphs with two types of nodes called places and transitions which are connected by arcs. A place represents a state variable with range IN_0. Its graphical notation is a circle. Its current value is considered as the number of tokens that reside at that place. A transition represents a rule for changing those state variables that are connected to it. Its graphical notation is a rectangle. State changes are based on addition and subtraction of constant values. Formally, a Petri Net (PN) is defined as a 5 tuple $PN = (P, T, I^-, I^+, M_0)$, where $P = \{p_1, \ldots, p_n\}$ is a finite and non-empty set of places, $T = \{t_1, \ldots, t_m\}$ is a finite and non-empty set of transitions ($P \cap T = \emptyset$), I^-, $I^+ : P \times T \rightarrow IN_0$ are backward, forward incidence functions, and $M_0 : P \rightarrow IN_0$ is the initial marking. The initial marking M_0 is a special case of a marking $M : P \rightarrow IN_0$. A marking M denotes that state of a Petri net, it can be interpreted as an integer (row) vector which includes per place p one integer value describing the number of tokens on place p. Let $\bullet t := \{p \in P | I^-(p,t) > 0\}$, $t\bullet := \{p \in P | I^+(p,t) > 0\}$ denote the input (output) places of a transition t, analogously for $\bullet p := \{t \in T | I^+(p,t) > 0\}$, $p\bullet := \{t \in T | I^-(p,t) > 0\}$ for place p. A Petri net describes a dynamic behavior, i.e., changes to the current marking that are performed by the firing of transitions. A transition $t \in T$ is enabled at marking M, denoted by $M[t >$, iff $M(p) \geq I^-(p,t)$, $\forall p \in P$. A transition $t \in T$ that is enabled at marking M may fire and yield a new marking $M`$ where $M`(p) = M(p) - I^-(p,t) + I^+(p,t)$, $\forall p \in P$, denoted by $M[t > M`$. The incidence matrix I is defined as a $n \times m$ matrix with $\mid P \mid= n$, $\mid T \mid= m$ and $I(p,t) = I^+(p,t) - I^-(p,t)$.

Figure 2 shows a Petri net that corresponds to the FU in Fig. 1. Each place and each transition has a unique identifier that is not given in the figure, non-unique names are displayed instead that match corresponding elements of the ProC/B model. The part on the top of the figure matches service *in*, it starts on the left side with transition *insource* that has an empty preset, is thus always enabled and its firing creates a token at place *protect*. Place *protect* is the input to transition *protect*, so is place *semaphoreC*. Note that the places *semaphore*, *semaphoreC*, *B*, and *BC*, are shared places, so called fusion sets, which means there are multiple graphical representations of the same place in the graph in order to avoid cluttering the presentation. Firing a sequence of transitions *insource*, *protect*, *Bin*, *enable*, *insink* would give the initial marking again but with one token less at place *BC* and one more at *B*.

For the analysis of ProC/B models, we propose a mapping to Petri nets which we briefly discuss in the following. The mapping can be done for the complete ProC/B model or for FUs. Analysis results of the generated PN model can be transfered back to a ProC/B model, such that a modeler need not be familar with PN theory.

The key idea of the mapping is to translate a PCE into a place p and a transition t such that $p \in \bullet t$. From that building brick, a sequence of two PCEs A and B is mapped to a sequence of mappings of A and B where the transition of A outputs a token to the input place of B. A comparison of service *in* in Fig. 1 with the upper part of Fig. 2 will make the approach more clear. The amount of entities at a PCE is expressed by the marking of the place that corresponds to that PCE. Firing of the transition describes moving of one ore more entities to the successor PCE. Similarly, other core elements of Table 1 are mapped, e.g., place and transition *protect* correspond to PCE protect, places *semaphore* and *semaphoreC* correspond to counter *Semaphore*, and the change operation with value 1 is described by $I^+(semaphore, protect) = 1$. Note that the limited range of the counter variable is modeled by a complementary place *semaphoreC* that gives the "free capacity" in the range of that variable. There are established means to model Branching, Fork, Join etc operations by Petri nets, the lower part in Fig. 2 models the state-dependent branching at the OR-connector by a non-deterministic choice with 3 concurrently enabled transition *or*. Since access to a server can only result in delays but no blocking, we map service calls to a server just like an delay PCE by a simple place-transition sequence. Service calls to FUs can be handled as a macro expansion,i.e., by replacing the PCE of the call by the PC of the service and then by mapping the extended ProC/B model.

Note that internal resources of a FU are not multiplied that way but remain shared. In our case, the service *change* of FU B resulted into a simple access to a counter B, which is shared among services *in* and *out*. Due to lack of space, we refer to [6] for more details on mapping ProC/B models to Petri nets.

The limits of the mapping are that 1) state-dependent decisions are modeled by non-determinism, 2) variables and parameters are ignored, respectively require a constant value, 3) PCEs with native code are treated by a simple transition, 4) entities are not unique objects, entities becomes tokens and thus tokens at the same place cannot be distinguished, and 5) timing aspects are ignored. Since we omitted any effect and impact on data structures like variables, rewards etc, and timing, we denote this mapping a mapping of control flow rather than a mapping of a full ProC/B model. The main motivation for the mapping is to make use of invariant analysis which applies to Petri nets and which we discuss next.

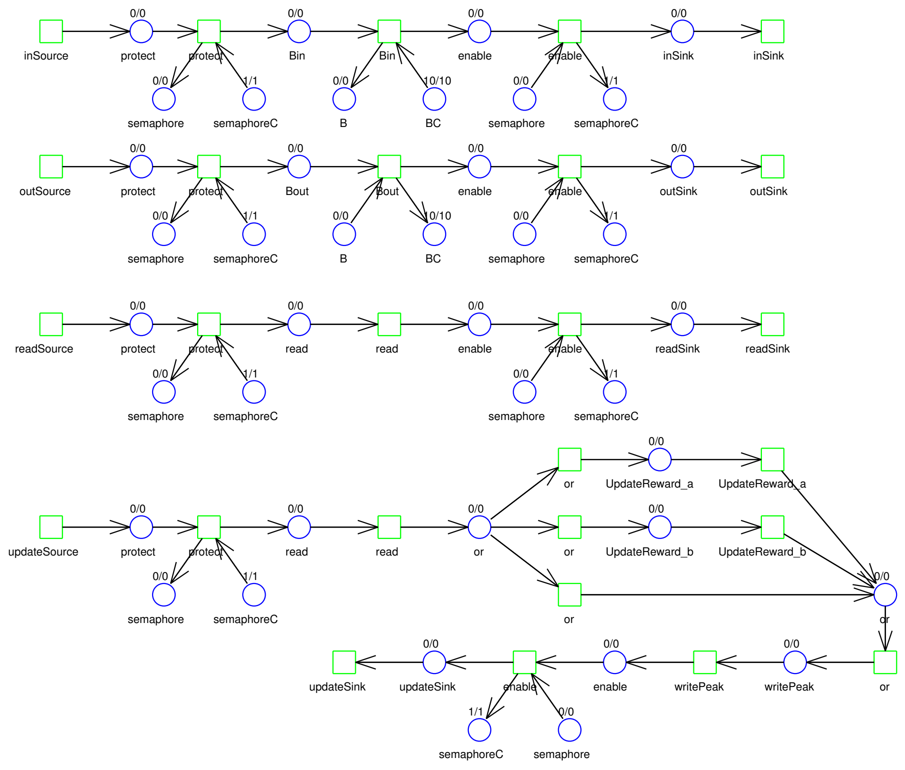


Fig. 2. PN model of FU 'StoreB'

5 Analysis

We present a two-step approach. The first step (model-based analysis) focuses on the static structure of a Petri net. In the latter step (state-based analysis), the state space of the Petri net is generated and a liveness check or model checking is performed. In the following, we assume a reasonably well specified model which succeeds a simple syntax and consistency check, for instance, there is a minimal connectivity between source and sink nodes, and we assume that services of FUs and process chains have been tested to work at least once for a given model.

5.1 Invariant Analysis

The notion of invariants is known for long in Petri net theory, so we only briefly recall definitions and properties. A vector $v \in \mathbb{Z}^n, v \neq 0$ is called P-Invariant if $v^T \cdot I = 0$ with $n = \mid P \mid$ and I denotes the incidence matrix. A vector $w \in \mathbb{Z}^m, w \neq 0$ is called T-Invariant if $I \cdot w = 0$ with $m = \mid T \mid$. We are only interested in semi-positive invariants, i.e., $v \in I\!N_0^n, w \in I\!N_0^m$, and those places (transitions) that have a positive value in v (w) give the support of an invariant. Invariants are additive - the sum of invariants is an invariant again - so if a net has at least one semi-positive invariant, then the set of invariants is infinite, which motivates consideration of a generating set of minimal invariants. Algorithms for the computation of that set are known, in particular [7]. The algorithm has an exponential worst case time and space complexity but in practice, it works extremely well and efficient in most cases.

A semi-positive P-invariant v means that the weighted number of tokens on the support of v is constant for all reachable markings M of the net, formally, $v^T \cdot M = v^T \cdot M_0 = const$. A semi-positive T-invariant w means that if a sequence of transitions is fired that contains each transition t as often as w_t, then the change of markings in total is zero, which means that the firing sequence is a loop that leads back to the marking where it started. Note that a t-invariant considers only the effect but does neither check for the enabling of transitions nor does it give information of a possible order of transitions in that sequence. Nevertheless, the good news is that invariants can be computed independently from an initial marking and independently from the size of the state space, even for unbounded nets with an infinite number of markings.

We apply invariant analysis to different parts of a Petri net that results from mapping a ProC/B model or one of its FUs. We describe the procedure by help of a net for our example FU as shown in Fig. 2, the analogous treatment of process chains for a full ProC/B model is immediate.

1st step: t-invariants of a service without consideration of resources. We start with t-invariants of a net that results from mapping a single service without places that correspond to resources. For that service, we expect one invariant w (several invariants in case of branching/non-deterministic choice) that covers the net. If the net of a single service is not covered, it is likely that there is a modeling error since that service call would at least partially retain inside the FU. We check the values of the source and sink transitions to obtain ratios for the input/output behavior of that service. A 1:1 ratio is considered the normal case, however other ratios are possible. The existence or non-existence of invariants and the ratio is reported to the modeller.

In the example, the upper most service *in* would yield a t-invariant with a value 1 for transitions *insource*, *protect*, *Bin*, *enable*, and *insink*. Note that places like *semaphore* and B are not part of the net in the 1st step.

2nd step: t-invariants of services with consideration of resources. We check the invariants of the first step, say an invariant w, with respect to the net of the complete FU. Let $d = I \cdot w$, this vector denotes the effect of the service and we check if there is a non-zero effect on counters which are accessed by that service. In the example, the invariant of the upper most service obtained in the first step is not an invariant of the complete net. It would not change the marking of *semaphore* but the marking of B (and BC).

We classify those counters with a non-zero effect as *consumable resources* since (depending on the sign of the entry in d) the service either produces or consumes tokens from the place of that counter. Counters that are accessed by a service but are effectively unchanged at termination of that service are classified as *reusable resources*. In the example, we obtain that *semaphore* is a reusable resource, B is a consumable resource. The terminology follows the classical considerations for deadlock detection in operating systems [8]. The classification of counters is expected to be consistent among all services and is reported to the modeller.

If consumable resources are present, we compute t-invariants for the complete net. In the example, this is the net as shown in Fig. 2 and for instance there is a t-invariant that follows the flow of packets from *insource* to *outsink*. We check if all transitions for the access to consumable resources are covered by t-invariants. If this is not the case, there is likely an error in the model and by reporting corresponding transitions respectively PCE elements we can give guidance to the modeller for further investigations.

Furthermore, the net construction implies that t-invariants always need to include some source and sink transitions. Hence, if a t-invariant w covers a consumable resource, at least two services are involved and we can report input output ratios for those services from the values of w. By help of the ratios obtained in step 1, we can transform those ratios into input:input ratios between at least two services and report those to the modeller. Those ratios formulate a constraint on the usage of services in the long run.

The motivation behind is that consumable resources can be seen as a communication between processes in a producer-consumer relationship and with a buffer of finite capacity. The ratio among service calls for different services shows how to use the FU without blocking effects, i.e., to retain a buffer in a reasonable state under the assumption that it starts in a non-blocking state (not empty, not full). The buffer capacity is able to let the producer run ahead, but not arbitrarily far, in the long run the ratio has to hold due to blocking effects.

3rd step: check p-invariants.
We check the p-invariants of the complete net. For each counter, there is trivial

p-invariant on the 2 places (the counter and its complementary place) that are generated for each counter by construction. If not, the mapping is not correctly implemented.

In addition, each counter classified as a reusable resource will yield at least one p-invariant that covers partly all services that access that counter. If the counter is binary, this invariant indicates parts of services that are used in mutual exclusion. Those areas across services can be visually highlighted to a modeller as shown in Fig. 1 and a modeller can check whether this is intended.

In particular, this is interesting if simultaneous resource allocation takes place. If that is the case for reusable resources, areas covered by p-invariants of several counters will overlap. There is a classic solution for deadlock prevention - reusable resources should be allocated in a given order to avoid a circular waiting situation - which can be checked automatically with the available information on areas covered by p-invariants. It is straightforward to implement a check for existence of an order and to highlight corresponding nodes in the ProC/B model.

However, the presence of consumable resources adds to the complexity of deadlock detection. If an area that is covered by a p-invariant for a reusable resource contains access operations to a consumable resource, the model need to be checked if the case of blocking for access to the consumable resource is handled appropriately. This is the problem that is present in the example FU, access to the consumable resource B takes place inside the area that is protected by the *semaphore*. Fig. 1 highlights the area of the semaphore. In this respect, we got to the point that we can highlight areas with simultaneous resource allocation of consumable and reusable resources.

Note that this approach works for single FUs and for complete ProC/B models. Invariant analysis focuses on critical structures due to resource allocation. Servers can be ignored, since their scheduling strategies only impose delays but no permanent blocking (critical cases like Last-come-first-serve and priority servers that can yield starvation are known and simple to localize). Other potential causes of blocking are state-dependent behavior based on variables which is not treated yet.

5.2 Statebased Analysis: Modelchecking for Liveness

Modelchecking techniques and the Liveness check for Petri nets takes place at the level of the reachability set or state space of the model. Those techniques apply to finite state spaces only.

So in our case - we have an open system of potentially infinite population - we need to limit the number of entities to achieve a finite population and finite state space. We obtain this at the level of a Petri net by adding an extra place per service of an FU (or process chain in case of a complete model), source transitions take tokens away from that place, sink transitions produce tokens to it. The initial marking of those places limits the number of entities considered. Arc weights for transitions can be derived from the input output ratio computed in the 1st step of the invariant analysis in Section 4.

The crux of state-based analysis is the size of state spaces, the well-known state space explosion. In our case, we can apply some net-level reductions upfront to reduce the state space that we need to consider for checking liveness. We observe for instance that mapping ProC/B models generates many simple transitions with one input, one output place that move tokens around and that are not critical. We make use of two rules from a set of rules defined by Berthelot [9,10] that reduce a Petri net and retain liveness (the reduced and the original net are both live or none). Those rules are known as *Prefusion* and *Postfusion* rules for two transitions. We apply those rules to reduce a Petri net ahead of a liveness computation to reduce the state space. The approach is worthwhile, since the application of rules is computationally inexpensive and the effect on the state space is significant. For example, if we limit the number of entities for each service call by 2 and the capacity of the store FU by 5 for the FU in Fig. 2, we observe a reduction from 377136 to 55 states.

For the resulting net, we can generate the state space either symbolically or explicitly and check the liveness condition for each transition. Due to lack of space, we refer to the literature for a detailed description on Petri net analysis techniques and only briefly comment on the results obtained for the considered FU. The liveness check identifies a deadlock for all considered limits and generates a sequence of transitions that starts at the initial marking and leads to the deadlock (if it is a total deadlock). In case of a partial deadlock, the firing sequence ends at a marking from which it is impossible to enable certain transitions again. If that sequence was obtained from a reduced net, it requires an expansion to relate to the corresponding firing sequence in the original net. The latter sequence can be used to animate the ProC/B model [11]. In addition to animation, a trace visualization by message sequence charts can be employed as well; we refer to [11] for a comparison of both approaches.

In our example, that trace leads to a deadlock state where the store FU has an internal value 0 (it is empty) and a service call *out* occured and blocks within the part that is protected by the semaphore counter, hence no subsequent service call *in* can allocate the semaphore and proceed to the point where the store FU gets filled to resolve the blocking situation.

So even in the case of a limited population of entities with a small limit, we could identify the reason for a deadlock. One solution is to carefully check what requires protection of a semaphore and remove that protection to resolve the deadlock problem. Note, that by checking finite subsets of the in principle infinite state space can but need not guide us to deadlocks, however it is not sufficient to guarantee their absence.

6 Conclusion

Simulation modeling of real world systems may yield large and complex models of interacting processes as it is the case for model of an air cargo network that we consider in this paper. It is formulated in the Proc/B notation and describes an open system with infinite state space. Motivated by that model, we propose

Petri net analysis techniques to support debugging of ProC/B models of logistic networks. In particular invariant analysis is useful to shed light on resource usage and identify constraints, in addition state based analysis techniques for liveness and modelchecking also support a modeller in finding reasons for encountering partial deadlocks among processes in simulation runs.

References

1. Kemper, P., Tepper, C.: A Petri net approach to debug simulation models of logistic networks. In Troch, I., Breitenecker, F., eds.: Proc. 5th Mathmod Vienna. Number 30 in Series ARGESIM Reports, CD-ROM (2006)
2. Völker, M., Sieke, H.: Process-oriented simulation of air cargo flows within an airport network. In: Proc. ASIM 2005, Erlangen (2005)
3. Bause, F., Beilner, H., Fischer, M., Kemper, P., Völker, M.: The Proc/B Toolset for the Modelling and Analysis of Process Chains. In T. Field, P. G. Harrison, J. Bradley, U. Harder (eds.) Computer Performance Evaluation Modelling Techniques and Tools (Proc. Performance TOOLS 2002) **Springer, LNCS 2324** (2002) 51–70
4. H. Beilner, J. Mäter, C. Wysocki: The Hierarchical Evaluation Tool HIT. Short Papers and Tool Descriptions of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Vienna (Austria) (1994)
5. Murata, T.: Petri nets: Properties, Analysis and Applications. Proc. of the IEEE 77 (1989) **4** (April 1989) 541–580
6. Fischer, M., Kemper, P., Tepper, C., Wu, Z.: Abbildung von ProC/B nach Petri Netzen - Version 2. Technical Report 03011, ISSN 1612-1376, Sonderforschungsbereich 559, Modellierung grosser Netze der Logistik, Universität Dortmund (2003)
7. Martinez, J., Silva, M.: A simple and fast algorithm to obtain all invariants of a generalized Petri net. In C. Girault and W. Reisig, editors, Application and Theory of Petri Nets **Informatik Fachberichte 52** (1982)
8. Holt, R.: Some deadlock properties of computer systems. ACM Computer Surveys **4** (1972)
9. Berthelot, G.: Checking properties of nets using transformations. in: G. Rozenberg (Ed.) Advances in Petri Nets 1985 **Springer, LNCS 222** (1986)
10. Berthelot, G.: Transformations and decompositions of nets. in: G. Rozenberg (Ed.) Advances in Petri Nets 1986 **Springer, LNCS 254** (1987)
11. Kemper, P., C.Tepper: Visualizing the Dynamic Behavior of ProC/B Models. In Schulze, T., Horton, G., Preim, B., Schlechtweg, S., eds.: Proc. of the 16th Conference on Simulation and Visualization, Magdeburg, Germany, SCS Publishing House (2005) 63–74