A Symmetric Petri Net Model of Generic Publish-Subscribe Systems for Verification and Business Process Conformance Checking

Tom Meyer

Institute for Visual and Analytic Computing, University of Rostock, Albert-Einstein-Straße 22, Rostock, 18059, Germany

Abstract

The highly decoupled nature of distributed systems can greatly simplify modular development. However, it is easy to lose track of component interactions and emerging system behavior. To inspect the behavior of the aggregated system, an abstract model and analysis tools can be of great help. In this paper, we present such a model that follows the Event-Driven Architecture paradigm. We argue that a Petri net model that captures the event flow, without modeling event data, is a useful abstraction to give insight into a system composed of distributed components. We present a symmetric Petri net model with publish-subscribe middleware and a generic abstraction of components. The model was designed with the intention to produce a finite P/T net to be compatible with current Petri net model checking tools. To build such a model, information about component behavior is necessary. We show that this information is encoded in the system implementation, and we describe potential means for automatic extraction. With the formal Petri net model, we enable the use of established verification methods. We discuss ideas from business process conformance checking and other verification properties that may deepen the system understanding. The proposed methods can be used either for static analysis during design time or to automatically signal issues at runtime.

Keywords

Distributed System, Event-Driven Architecture, Publish-Subscribe, Petri Net Model, Conformance Checking, Verification

1. Introduction

A distributed system is a collaboration between multiple self-contained modules or components. Since these are usually individually developed separate code bases, this approach promotes modular development with a high focus on the individual parts.

However, hard program borders facilitate ignorance to other components and alienate developers from foreign code bases. Additionally, in the running system, components can connect and disconnect to the distributed system at any time, changing the system behavior continuously. It is hard to keep the overview in this context making predictions on the emergent system behavior exceedingly difficult. Can the required workflows be executed? Which event affects which component? A simple look at the implementation does not answer these questions and design documents and specifications also cannot consider every possible system adaptation.

PNSE'23: International Workshop on Petri Nets and Software Engineering, June 26–27, 2023, Lisbon, Portugal

tom.meyer@uni-rostock.de (T. Meyer)

However, a model of the implementation can be used in combination with a specification to answer such questions with formal verification. Yet verification can be complicated and increases development effort. To make these verification methods accessible, the implementation model should be extracted directly from the implementation. Additionally, if this model uses adequate abstractions, it stays humanly understandable, and formal methods can be used practically.

A common abstraction for distributed systems is the Event-Driven architecture paradigm (EDA). In an EDA system a central middleware connects distributed components using a standardized communication abstraction. Commonly a publish-subscribe interface is used here.

A benefit in using an EDA model to reason about the emergent system behavior is that message contents are usually not relevant. Just knowing the event flow already gives a lot of insight, while the precise forking conditions are often irrelevant for the general system capabilities. Conveniently, if messages are represented by type instead of their content, messages can be reduced to a token that is exchanged between communicating components. With this abstraction Petri nets [1] become a suitable modeling formalism for publish-subscribe based communication.

With a Petri net system model we can use different verification methods to compare our model with a given specification. The specification can take many forms such as logical formulas and other formalisms such as workflow nets (WF-nets) [2]. Particularly, WF-nets can again improve accessibility because they describe business processes that are already widely used to specify tasks for production targets or services. Additionally, the topic of business process conformance checking is under current research [3] as a verification method.

In this paper we present a Petri net model of a publish-subscribe system with a generic model of components for verification purposes. We describe what is necessary to build the model automatically from component implementations and give examples for usable verification methods. In particular, we describe how a behavioral specification in form of business processes can be integrated in the verification process by using methods from business process conformance checking [3].

2. Background

In the following we give an overview of our mainly used concepts to compile a verification approach for publish-subscribe systems.

2.1. Event-Driven Architecture (EDA)

In EDA, components *observe* events and notify other components about events using messages. An event is a “significant change in state” [4], while the corresponding *notification* message is only a representation of the event. Notifications are distributed by a central middleware, the *notification service (NoSe)* using a publish-subscribe (pub/sub) API [5]. Components use the API to publish messages on a topic. As a result all topic subscribers are informed. This results in a decoupling of communication in time, space, and synchronization [6], so that components have no information about each other and what publication effects they have. Only the notification service implicitly keeps track of component connections by storing subscriptions.

With EDA, the computation capacity of distributed hardware can be leveraged, e.g. to run interconnected web services, or to connect physically distributed devices as in IoT applications.

2.2. Petri Nets

Petri nets are a formalism for concurrent processes [1]. The basic place-transition nets (P/T net), are defined as a bipartite, directed graph connecting *places* with *transitions* and vice versa.

Transitions can *consume* and *produce* tokens on places, depending on the connecting arc direction and weight. Arcs that connect a place p with a transition t in the direction from p to t make p an input place of t. Similarly, if an arc connects a transition t with a place p, p is an output place of t. The set of input places of a transition t is denoted by $\bullet t$ and the set of output places is denoted by $t\bullet$. A similar notation is used for the pre- and post-sets of places.

A transition is *enabled* if all input places contain at least as many tokens as the arc weights specify. Enabled transitions can *fire* non-deterministically. A firing transition consumes tokens from the input places and produces tokens on the output places according to the arc weights.

In a system model, tokens on places represent the system state and transitions represent the state changes. P/T nets are well investigated, making them a good target for model checking.

However, in classical Petri nets, tokens are indistinguishable from each other. But, to model systems with practical complexity it is often useful to discriminate them. *Colored Petri nets* address this issue by extending Petri nets with a type (or color) for tokens [7]. To specify which token types are affected by a firing transition, transitions can be annotated with a guard. A guarded transition can only fire if the tokens in its input places satisfy the guard conditions. Colored Petri nets can be unfolded to P/T nets, although the unfolding may be infinite. However, a finite unfolding enables the use of P/T net model-checking methods.

A kind of colored Petri nets are *Symmetric Petri nets* (introduced as Stochastic Well-Formed Colored Nets) defined by Chiola et al. [8]. This net introduces modeling restrictions for custom analysis methods. In the paper, a grammar is introduced to define the allowed syntax of place classes, arc labels and transition guards with a corresponding semantics. The class of a place describes the color of the tokens that can be produced and consumed on that place. Token colors can be a composition of multiple classes described as a tuple. Arc labels then describe compositions of classes by an expression. Only tokens matching the expression are allowed as input for the connected transition. Additionally, the transition guards are predicates that define restrictions on classes and dependencies between input and output arcs.

Our model will follow the definition from Chiola et al. although we will abbreviate large transition patterns. In these cases we will show an example for the unabbreviated patterns. However, we observe that patterns can be unfolded into a P/T net for model-checking.

2.3. Business Processes

Business processes describe a collection of tasks that are used to build a product or that are involved in a service. A business process may define a series of tasks, parallel tasks, tasks that are executed conditionally or a combination of these options. There are a variety of business process models [9] including Petri net models.

Van der Aalst was instrumental to formalize business processes as Petri nets including the definition of a separate net class called workflow nets [2]. He defines a workflow net as follows:

"A Petri net $PN = (P; T; F)$ is a WF-net (Workflow net) if and only if:
(i) PN has two special places: i and o. Place i is a source place: $\bullet i = \emptyset$. Place o is a sink place: $o\bullet = \emptyset$.
(ii) If we add a transition t^* to PN which connects place o with i (i.e. $\bullet t^* = \{o\}$ and $t^*\bullet = \{i\}$), then the resulting Petri net is strongly connected."

He then defines important properties such as safeness and soundness for workflow nets.

To facilitate the understanding of business processes in WF-nets, other process models define translations back to Petri nets [10]. For example Dijkman et al. defined a translation [11] for the Business Process Modeling Notation (BPMN) [12] and Hinz et al. give a translation [13] for the Web Services Business Process Execution Language (BPEL) [14].

3. Related Work

Petri net models where used before in distributed systems. E.g.: Aldred et al. used Petri nets to model different decoupling approaches in distributed systems [15]. They analyze the implications for component interactions over the middleware.

Valero et al. and Gomez et al. modeled a publish-subscribe system in a timed colored Petri net specifically for verification purposes. Their model targets web services with a focus on message timing and timeouts [16, 17]. Hens et al. also used colored Petri nets to model a publish-subscribe system [18]. These models focus on the middleware and do not further constraint the component internals. However, the emergent system behavior dependents in large parts on the component behavior. For verification purposes it is crucial which states are reachable in every connected component and how the composed components can interact.

Unfortunately, the essence of software components is vague. Szyperski et al. summarize it as follows: "One thing can be stated with certainty: components are for composition. [...] Beyond this trivial observation, much is unclear." [19]. Becker also identifies composability in his component model as an important feature [20]. However, both Szyperski and Becker use the component abstraction in the software design process and not for verification. They capitalize on the reusability aspect that follows composability.

The EDA paradigm intrinsically exploits the composability by decoupling components [6]. This not only facilitates reusability but also enables components distribution. In EDA, components mainly process events and react to them. Lamport et al. [21], describe this pattern as a general feature in distributed computing:

"Underlying almost all models of concurrent systems is the assumption that an execution consists of a set of discrete events, each affecting only part of the system's state. Events are grouped into processes, each process being a more or less completely sequenced set of events sharing some common locality in terms of what part of the state they affect. For a collection of autonomous processes to act as a coherent system, the processes must be synchronized."

Based on the former descriptions, we define *components in an EDA system* as a set of processes that are executed as a reaction to incoming events. During the execution of a process, new events can be generated to synchronize component state with other components. With this definition, we can model a publish-subscribe system more precisely and use the model to improve verification accessibility and expressiveness.

An interesting verification branch uses runtime events to evaluate the system behavior. E.g.: Schmerl et al. built a colored Petri net from runtime observations [22]. This relates to process mining as described by van der Aalst [9]. He describes how processes can be discovered from log traces and how to check them against a specification. The specification takes the form of business processes, which are easy to understand in comparison to e.g.: formulas in a temporal logic. Carmona et al. describes this process in detail in their book on *conformance checking* [3]. Hens et al. also have a take on this idea and compare their model with business processes [18].

However, models always suffer a credibility gap. While the system implementations can deviate from manually created models, automatically discovered, log based models can always be incomplete. Also, models created by both approaches get outdated when the system evolves. This is why we like to explore system models that are close abstractions from the implementation. We aim at a model that can be automatically generated from code, to be used for verification. This approach would make model checking much more user-friendly and affordable.

4. Publish-Subscribe Model

In this section we introduce our Petri net model of a publish-subscribe system. We will first highlight our assumptions. After that, we break down the parts, explain their modeling and functionality. The parts are then extended with a synchronization structure for fairness constraints. At the end of this section, we describe what system information is needed to build the model and how we envision an automatic extraction from the implementation. The complete model is shown in Figure 9.

4.1. Assumptions

If we want to analyze the behavior of a complex distributed system, we cannot model every detail of the component implementations. Instead, we should find the right balance between abstraction and detail. Our assumptions try to balance model expressiveness and simplicity.

Given our system follows the EDA paradigm it is reasonable to orient our abstractions towards EDA principles. This means that the movement of events is a central concept. To analyze the system behavior it is not essential to know the event content, but which actions can follow. If the event content determines an event reaction, non-determinism can be used instead. Consequently, we do not model event contents, only event types.

Additionally, we use channels as event filtering mechanism [5][Chapter 2.3.1]. This is a pragmatic limitation to reduce the complexity of the Petri net model, but we believe that the model can be extended to use more complex filtering mechanisms i.e. subject-based filtering.

Our model use case is to verify a specification is covered; we want to know if a specified behavior is executable in the system. Therefore, we do not model unexpected failures like e.g. link failures or message loss and target the analysis of the expected system.

Also, we only consider the known components. If previously unknown components connect to the system, the model has to be rewritten.

```
EnvEvent = ee[1-m]
Notification = n[1-n]
SubState = [true, false]
ComponentEvent = EnvEvent, Notification
AllEvents = ComponentEvent, SubEvent
Component = cpt[1-o]
Process = p[1-p]
Action = a[1-p]_[1-q]
Channel = [envch, ch[1-r]]
```

Figure 1: Color classes used to build the color domains of places.

Finally, we exclude loops and recursive functions for publishing behavior. Although valid in EDA components, we presume that loops are used to compute values, not to communicate. With this assumption the reachability graph of component processes should be finite.

4.2. Color Classes

Tokens that are produced on places must satisfy a *color domain* which is composed of *color classes*, e.g.: the $ComponentEvents$ place (see next section) has a domain composed of two classes $Component \times ComponentEvent$. In a P/T net this composition is unfolded so that there is a place for every component-event combination.

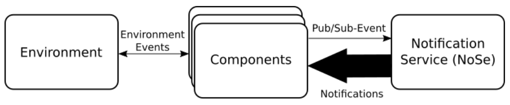
In the implementation the color domain represents required information to execute the behavior that is molded by the consuming transitions. Following the example this would be the affected component and what event type is received.

The color classes our model is based on are shown in Figure 1. Classes can be an indexed list of items (e.g. `p[1-p]` defines a list containing the processes p_1 to p_p), a list of named items (e.g. `SubState` class), or a combination of both (see [8] for the syntax definition). The indexed lists are placeholders for named identifiers that are used in an instantiated model e.g.: instead of an enumeration of event types, we assume a final model will use type names. In particular the elements of the action class are a placeholder for action instances. While the two indexes suggest each process (index p) has the same amount of tasks (index q), a model instance will have a varying number of tasks for each process. The next section gives an example how color classes are used to define place domains by introducing two central event places.

4.3. Events

Events are the central communication messages sent over the network. However, our usage of the term 'event' is not equivalent to the usage in an EDA since our environment exceeds the EDA borders. We distinguish between three types of events as depicted in Figure 2a:

1. *Environment events* are sent between components and entities that are not part of the communication via the NoSe.
2. *Subscription events* are sent from components to the NoSe to update subscription states.
3. *Notifications* are used for intercomponent communication. They are published on a specific channel and forwarded to all channel subscribers by the NoSe.



(a) Components communicate with three kinds of events: (1) environment events are used to communicate with the environment, (2) PubSubEvents are sent to the NoSe and can be a subscription update or a notification, and (3) the NoSe distributes notifications back to the components.

```
NoSeEvents = Component x Channel x AllEvents
ComponentEvents = Component x ComponentEvent
```

(b) All events leaving a component (including environment events) are produced on the NoSeEvents place. All incoming events are produced on the ComponentEvents place.

Figure 2: Event communication and event place domains.

Events are tokens with a given type. The event type is also a placeholder for the event contents. Places holding events will unfold to at least one place for every event type.

Components interact with two main event places (see Figure 4a) $ComponentEvents$ and $NoSeEvents$. $ComponentEvents$ cannot be subscription updates and are always directed at a specific component. *NoSeEvents* can be any of the three event types and are sent from a source component. Additionally, $NoSeEvents$ are sent on a specific channel. If the event is a subscription, the channel is the target for the subscription update. If the event is a notification, the channel is used by the NoSe to determine the affected subscribers. If the event is an environment event, the channel should always be the special type $envch$ to be consistent with the place domain. In practice this channel will not be used by the environment.

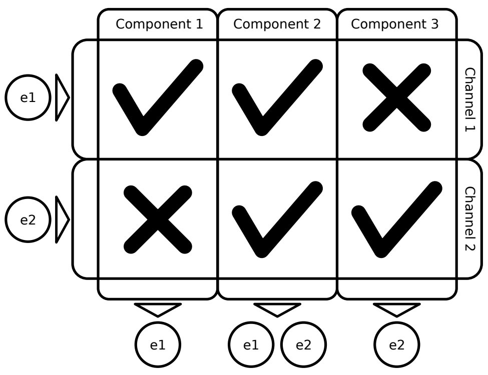
4.4. Notification Service

The fundamental purpose of the Notification Service (NoSe) is to forward events from one component to other components. In a publish-subscribe system, components can subscribe to a topic of events. The NoSe then filters all incoming events according to the subscriptions before forwarding. There are a variety of filtering mechanisms [5]. We use the simplest one: the concept of channels (see Figure 3a). Event producers publish all events with an associated channel, event consumers can subscribe on the different channels, and all components that are subscribed to the associated channel will be notified.

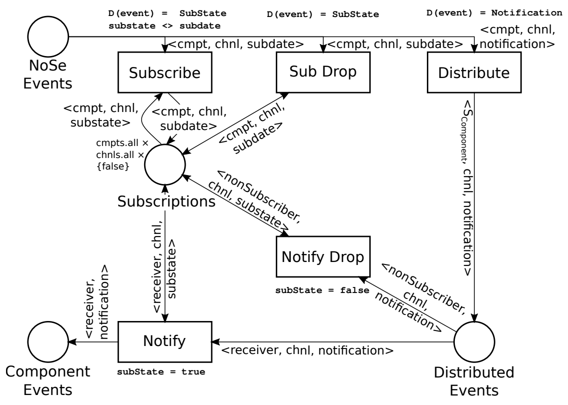
Internally the Nose has to

1. track subscription states,
2. forward events to all components where the subscription filter applies,
3. drop all events where the filter does not apply, and
4. update the subscription state according to special events.

We describe this behavior as depicted in Figure 3b and 3c. The *Distribute* transition forwards events of type *Notification* from *NoseEvents* to all components on the *DistributedEvents* place. The broadcast is modeled by using the $S_{Component}$ identifier which encodes the set of every component defined by the $Component$ class (see [8]). The distributed events are then either dropped or forwarded to the $ComponentEvents$ place, depending on the state of the $Subscriptions$ place.



(a) For each channel and each incoming event, the NoSe forwards the event to all components subscribed to the channel.



(b) The NoSe distributes notifications to subscribers or updates subscriptions. The subscription function is idempotent.

```
Subscriptions = Component x Channel x SubState
DistributedEvents = Component x Channel x Notification

Subscribe | D(event) = SubState AND subdate <> substate
NoSeEvents.Subscribe = <component, channel, subdate>
Subscriptions.Subscribe = <component, channel, substate>
Subscribe.Subscriptions = <component, channel, subdate>

SubDrop | D(event) = SubState
NoSeEvents.SubDrop = <component, channel, subdate>
Subscriptions.SubDrop = <component, channel, subdate>
SubDrop.Subscriptions = <component, channel, subdate>

Distribute | D(event) = Notification
NoSeEvents.Distribute = <component, channel, notification>
Distribute.DistributedEvents = <S_Component x channel x notification>

Notify | subState = true
Distribute.Notify = <receiver, channel, notification>
Notify.Subscriptions = <receiver, channel, substate>
Subscriptions.Notify = <receiver, channel, substate>
Notify.ComponentEvents = <receiver, notification>

NotifyDrop | subState = false
Distribute.NotifyDrop = <nonSubscriber, channel, notification>
Subscriptions.NotifyDrop = <nonSubscriber, channel, subState>
NotifyDrop.Subscriptions = <nonSubscriber, channel, subState>
```

(c) The NoSe transitions in a textual definition.

Figure 3: Behavior of the Notification Service (NoSe).

The distribution strongly benefits from a colored Petri net approach. For every combination of sender, channel and notification type, notifications can be sent to every receiver. In an unfolded P/T net every place-transition-combination has to be instantiated, resulting in a large structure.

The $Subscribe$ transition updates the subscription state of a component if the event is of type $SubState$. Additionally, the current subscription has to be different from the update. If so the guard is fulfilled, a token with the old subscription state is consumed for the given component and channel, and a new token with the new state is produced. If the subscription state is equal to the received subscription event, the update has to be ignored so that subscriptions are idempotent [5]. This case is handled by the $SubDrop$ transition.

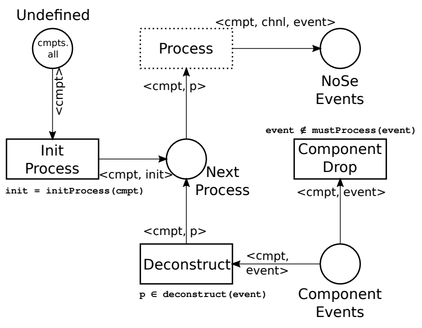
4.5. Components

Connected to the NoSe are the distributed components. We put forward a pattern of how components behave in publish-subscribe systems. This pattern can be explicit or implicit, but we argue an explicit implementation is beneficial. The pattern consists of the following parts (illustrated in Figure 4a, 5a and 5b):

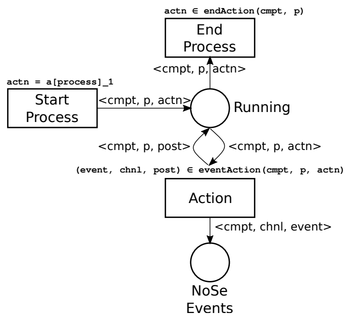
- An *initialization* routine, which may or may not produce events. Often, this routine will be used to subscribe to channels.
- *Receiving events* from other components, or the environment (e.g. a sensor or the OS).
- *Event deconstruction* of received events to select a process as reaction or to drop the event.
- And the *execution* of the selected Process, which may or may not *produce new events.*

Following this pattern a component is a collection of executable processes including one initialization routine. A component reacts to an incoming event by executing a process based on the incoming event type and content. The process execution is split into a sequence of actions that change the system state. For our Petri net model, we only consider actions that publish an event and the order in which actions can be sequenced.

Incoming events trigger a *deconstruction* mechanism. In the implementation this is usually a switch case equivalent. In our model the deconstruction explicitly either selects a process to react or ignores the event with two transitions.



(a) Transitions of a component. Events are received in the ComponentEvents place and send by a process to the NoSeEvents place. Processes are selected based on incoming events by the deconstruction transition. A special init process is executed on component startup.



(b) After it is started a process is in a running state until it finishes. A running process can change the system state with actions. Only actions that produce events are modeled.

Figure 4: Graphical definition of Components. The dotted Process transition is replaced by the process definition on the right.

```
NextProcess = Component x Process

Deconstruct | process ∈ deconstruct(event)
ComponentEvents.Deconstruct = <component, event>
Deconstruct.NextProcess = <component, process>

ComponentDrop | event ∉ mustProcess(event)
ComponentEvents.ComponentDrop = <component,
     event>
```

(a) Component deconstruction of incoming events. Each event can trigger a process, be dropped or both.

```
Undefined = Component

InitProcess | initProcess = initProcess(
     component)
Undefined.InitProcess = <component>
InitProcess.NextProcess = <component,
     initProcess>
```

(b) Every component is initialized with a specific process. Only initialized components can receive events.

```
Running = Component x Process x Action

StartProcess | action = a[process]_1
NextProcess.StartProcess = <component, process>
StartProcess.Running = <component, process, action>

Action | (event, channel, postAction) ∈ eventAction(component, process, action)
Running.Action = <component, process, action>
Action.Running = <component, process, postAction>
Action.NoSeEvents = <component, channel, event>

EndProcess | action ∈ endAction(component, process)
Running.EndProcess = <component, process, action>
```

(c) Processes are selected by the deconstruction mechanism. Each event received by a component can trigger a process as an event reaction.

Figure 5: Textual definitions of the component behavior.

The $ComponentDrop$ transition discards events that should be ignored. We abbreviate its guard predicate with the formula $event \notin mustProcess(ComponentEvent)$. To expand the formula, all cases that do not require a reaction are enumerated as shown in Figure 6b. The drop transition is enabled for all events that return $false$ for the $mustProcess$ relation.

The $Deconstruct$ transition is enabled for all events that may return a process. An event can result in multiple processes non-deterministically. We abbreviate this behavior with the formula $Process \in deconstruct(ComponentEvent)$. The formula expands to the pattern shown in Figure 6d. For every event where $mustProcess$ is $true$, $deconstruct$ cannot be $false$. Or $\forall e \in ComponentEvent : mustProcess(e) \Rightarrow deconstruct(e)$. However, an event can be dropped or deconstructed non-deterministically.

During *startup*, a component usually initializes its state, connects to the NoSe and often subscribes to a channel. We model this behavior with the $InitProcess$ transition. Before the $InitProcess$ transition has fired, the corresponding component state is undefined, and it cannot receive events (see Section 4.7). Only after initialization, more processes can be queued. We abbreviate the transition predicate with the formula $initProcess(Component) \rightarrow Process$ as shown in Figure 6c. The formula defines exactly one process for each component.

A component can produce new events in *processes*. Figure 1 shows that processes are uniquely

```
(action = a_x1 AND postAction = a_y1 AND event = e_z1 AND channel = type1) OR
(action = a_x2 AND postAction = a_y2 AND event = e_z2 AND channel = type2) OR
...
```

(a) Action transition pattern.

```
event != e1 AND
event != e2 AND
...
```

(b) Component drop transition pattern.

```
(component = c1 AND init = p1) OR
(component = c2 AND init = p2) OR
 ...
```

(c) Init process transition pattern.

```
(event = e1 AND process = p1) OR
(event = e1 AND process = p2) OR
(event = e2 AND process = p1) OR
(event = e2 AND process = p3) OR
...
```

(d) Deconstruct transition pattern.

```
action = ax1_y1 OR
action = ax1_y2 OR
action = ax1_y3 OR
...
```

(e) End Process transition pattern.

Figure 6: Component transition patterns

identified with an ID, indicated by the $process$ class. In the model a process is gated by a start and an end transition. Between these two transitions, the process is in its `running` state, where a sequence of actions is executed. In the component implementation, the *start transition* corresponds to the function call of the given process. The guard of the start transition ensures that the first action of the given process is produced.

Each *action* is executed by the $Action$ transition and always produces an event. Actions are indexed by a process ID and an action ID (see Figure 1). An action that produces a $Notification$ corresponds to a call to the $publish$ function in the publish-subscribe interface. An action that produces a $SubState$ event corresponds to a call to the $subscribe$ or $unsubscribe$ function in the publish-subscribe interface. An action that produces an $EnvEvent$ corresponds to the part of the code that interacts with the environment.

In a custom framework it might be useful to publish environment events with the publish-subscribe interface, on the special $envch$ channel, for easy parsing.

The *action transition* predicate generates all possible action sequences and is abbreviated with the formula $(AllEvents, Channel, Action) \in eventAction(Component, Process, Action)$. If a process includes a condition that produces multiple event sequences, an action can produce different followup actions. Thus, multiple transitions are enabled non-deterministically. Consequently, the guard maps the parameters from the $running$ place domain to at least one tuple of $(event, channel, postAction)$. The expanded pattern is shown in Figure 6a

The *end transition* corresponds to the return statement of the function previously called with the start transition. The guard of the end transition is an enumeration of end actions abbreviated by the formula $Action \in endAction(Component, Process)$. The pattern expansion is shown in Figure 6e. It is possible that $Action$ and end action are simultaneously enabled to allow conditional endings of a process. However, every action sequence requires an end action as last action to ensure process termination.

```
EnvEvents = Component x EnvEvent

Sense | - // no guard
EnvEvents.Sense = <component, event>
Sense.ComponentEvents = <component, event>

EnvironmentOut | D(event) = EnvEvent
          | channel = envch
NoSeEvents.EnvironmentOut = <component, channel, event>

EnvironmentIn | -
EnvironmentIn.EnvEvents = <component, event>
```

Figure 7: Environment interaction of Components

4.6. Environment Model

The system environment contains every entity that cannot use EDA abstractions to communicate with components, e.g.: human interaction, sensors or OS events. Because it is highly individual, we make no assumptions of its behavior. However, for a basic compatibility we define a default behavior with two transitions: $EnvironmentIn$ and $EnvironmentOut$. These transitions can produce and consume $EnvEvents$ in any order; at any time.

In general each component has a unique sensing capability (e.g. a specific combination of sensors). Therefore, produced environment events always target a component. Consequently, the domain of the $EnvEvents$ place is a tuple consisting of a produced event and a targeted component. Idle components can then sense environment events, deconstruct them and react with process actions.

For verification purposes it is useful to restrict the production of events to specific start events and collect produced events. This can be achieved by removing the $EnvironmentIn$ transition, putting the desired start event on the $EnvEvents$ place, and adding an output place to the $EnvironmentOut$ transition that collects the events consumed from the $NoseEvents$ place.

4.7. Synchronization

Modelling just the communication routes of our system is insufficient if events and processes have ordering constraints. To bind the arrival of an event to a process execution, we introduce a $ComponentIdle$ place. If an event arrives via $Notify$ or $Sense$, a token from the receivers $ComponentIdle$ place is consumed. Components cannot receive new events with an empty $ComponentIdle$ place. The $ComponentEvents$ place is therefore safe. The $StartProcess$ transition returns the $ComponentIdle$ token after deconstruction, freeing the event reception. As a result, component processes can run in parallel. The execution order depends on the event arrival order. Initially the $ComponentIdle$ place is not marked to ensure the $initProcess$ is always started first.

Events can only accumulate in an arbitrary order in the $NoSeEvents$ and $DistributedEvents$ places; the models only unbounded places (excluding environment places like $EnvEvents$ since the environment has no canonical form). However, often it is desirable to guarantee event ordering constraints. A similar problem is formulated in the literature for distributed event

```
Source = Component
Sync = Component
ComponentIdle = Component
NoSeIdle = {}

StartProcess.ComponentIdle = <component>
ComponentDrop.Idle = <component>
ComponentIdle.Sense = <component>
ComponentIdle.Notify = <receiver>

Sync.Action = <component>
EnvironmentOut.Sync = <component>

Distribute.Source = <component>
Subscribe.Source = <component>
SubDrop.Source = <component>
```

```
Done = {}

Notify.Done = <{}>
NotifyDrop.Done = <{}>
Subscribe.Done = <{}>
SubDrop.Done = <{}>

NoseIdle.Distribute = <{}>
NoseIdle.SubDrop = <{}>
NoseIdle.Subscribe = <{}>

Resume | - // no guard
Resume.NoseIdle = <{}>
Resume.Sync = <component>
Source.Resume = <component>
Done.Resume = <{}>
```

Figure 8: Synchronization places and arcs for FIFO ordering.

systems. Muehl et. al. [5][Chapter 2.5.3] distinguish four orderings:

1. *No ordering*: No guarantees are given on the event ordering.
2. *FIFO ordering*: "[...] the notifications that are published by a component C_1 should not be delivered to a component C_2 in an order different from the order in which they were published." All outgoing events from a single Component are ordered. Events can be shuffled, but events from a single source arrive in the order that they were sent.
3. *Causal ordering*: "[...] if there is a sequence of components $C_1, ..., C_k$ such that each component C_i publishes a notification n_i that is notified to component C_{i+1} if $i < k$ then a component Y should not be notified about n_1 after it was notified about n_k." In causal ordering, event chains that span multiple components are ordered.
4. *Total ordering*: "[...] if a component C_1 is notified about n_1 and eventually notified about n_2, then a component C_2 should not be notified about n_1 after it was notified about n_2." Events are ordered globally, but events can be omitted for single components.

We add a synchronization mechanism between components and NoSe to ensure event ordering constraints. The NoSe can only process events if $NoseIdle$ is marked. After the final transition has fired (i.e. $Notify$, $Subscribe$ or the corresponding drop transitions) a token is produced on the $Done$ place and the $Resume$ transition resets the event processing transitions.

The $Sync$ place then ensures synchronization between NoSe and Components. A $Sync$ token is required to enable the $Action$ transition. We can guarantee *FIFO ordering* with a $Sync$ place domain over all $Components$, or *Total ordering* with a single shared token for all components. Only after the previous event has been delivered and the $Resume$ transition has fired the next action can produce a new event.

To guarantee *Causal ordering* the notification service needs to store additional information. Firstly, the past notification order, and secondly, which event was already delivered to which consumer. This would be a complex addition to the model and is therefore excluded.

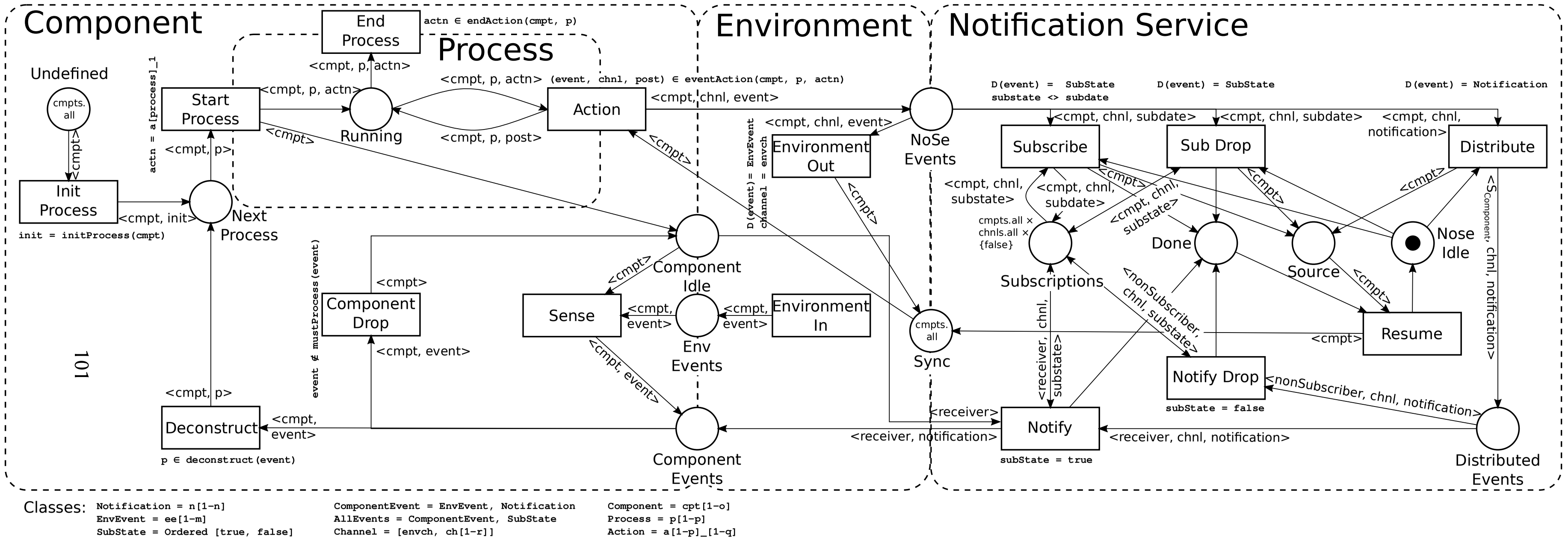


Figure 9: Graphical representation of the complete publish-subscribe model. A component can receive events on the $ComponentEvents$ place. Incoming events can either be *dropped* or start a process depending on the $Deconstroct$ transition. A process can publish new events in a sequence of $Actions$. Events are either send between components via the Notification Service (NoSe) or to the environment. The notification service forwards incoming events to all subscribers of the associated event-type. The $Sync$ place enforces a FIFO event ordering.

4.8. Building the Net

The complete model is shown in Figure 9. For assembly, all variables need to be instantiated, i.e.: the final enumeration for all color classes (Figure 1) and the final transition guards. We distinguish between the following kind of information:

Static information is encoded in the component implementations and known at compile time. For each compiled component the processes and action transitions are static. As a result the *Process* and *Action* classes can be enumerated completely at compile time. Additionally, initialization processes and event deconstruction is static. Consequently, the transition guards for the *initProcess*, *deconstruct* and *mustProcess* transition can be generated.

However, action sequences are still incomplete since the used channels and event types can depend on the component state. Hence, the remaining classes are dynamic.

Dynamic information For behavior that depends on the component state, we need assumptions or further knowledge to build the model. This knowledge might be known by the system designers or accumulated dynamically at runtime. However, newly discovered information then requires a rebuild of the model making all previous analysis obsolete.

The most characteristic dynamic information in the model are the involved components. During the system lifetime components can disconnect and entirely new components can connect. Thus, the enumeration of components in the *Component*-class changes. Newly introduced components also introduce new events and channels. This makes the *Notification*, *EnvEvent* and *Channel* classes also dynamic.

Furthermore, actions may change the channels on which events are published. While the domain of event types is static, channels are usually unrestricted i.e.: represented as strings. Creating an infinite number of new channels is entirely possible. Consequently, for newly introduced event-channel combination, a new action instance has to be added to the guard of the *Action* transition. If this action is also an end action, the *EndProcess* transition has to be updated as well. The new action instance will still have the same ID and post action.

Environment Information is what interacts with the system not using a publish-subscribe interface. However, the environment transitions introduced in Section 4.6 can be replaced with a more sophisticated environment model. In this case, the environment model has to be provided for the system composition.

The Initial marking depends on the use case. Yet, the *ComponentEvents* place is the most noteworthy to be marked. An initial set of events can be defined here. If desired components can be individually initialized by moving tokens from the *Undefined* place to the *ComponentIdle* place and setting the subscription state for a set of component-channel combinations to *true*. Additionally, the *NoSeIdle* place should be marked initially.

4.9. Automatic Extraction of Static Information

Manually collecting information that is already available in the component implementation is cumbersome and prone to errors. Instead, it is desirable to parse the implementation for information. We think this is possible with a static code analysis at compile time if the code:

1. uses a recognizable publish-subscribe-interface,
2. marks the initialization process,
3. has a deconstruction function of the form $deconstruct(event) \rightarrow Process$ with parsable event cases,
4. uses the same kind of event deconstruction for environmentally caused processes, and
5. marks environment interactions in a manner that is parsable, e.g.: with an interface, structured logs or comments.

This functionality could be integrated to an existing publish-subscribe client library. Besides the already existing publish-subscribe interface, this library could export a component interface with a deconstruction function and a distinct function to send environment events. Additionally, a basic process interface could be provided to encapsulate the event reactions. Then a static code analysis has to extract the following information from the implementation for each component:

1. The process that is used for initialization.
2. All possible event-process combinations in the deconstruction routine.
3. All possible action sequences.

The *initialization* process can be assigned to a known variable, so it can be extracted by reading this variable.

To parse the *event-process combinations* the return values (started processes) of the deconstruction function have to be related to a causing event. We assume this is usually done in a switch-case block, a matching statement or similar kinds of case distinction. Given the simple form that each event is handled by one case arm and each case returns at least one process, an extraction method traverses all case arms, stores the event for each case, and adds the returned processes to the stored event.

To extract the *action sequences* from a process, the process implementation has to be searched for all sequence paths. For this, we can build a reduced control flow graph, which only contains the event publishing statements. Occurring function calls can be replaced with the control flow of the called function. As long as no recursion is used, the graph will be finite.

In the reduced control flow graph every event publishing is an action and the possible successor actions as well as end actions are known. For actions with static channels, we can then build the complete transition guard. However, for actions with variable publishing information we can only build an action template at compile time. If no further input is given by the system designer, we require runtime information to generate the action instances.

5. Using the Publish-Subscribe Model for Verification

After model assembly we can use it to further our system understanding. This can be done with different verification methods at different stages in the software life cycle.

On the one hand we can run a *static analysis* during design time, before deployment. In this stage the system has no current state i.e. we do not know which components are connected. Instead, a reference state has to be defined for verification. This can be a default state or a scenario of interest. Since static analysis is not constrained by runtime requirements, it can be used with time or computation intensive verifications methods on dedicated hardware.

On the other hand, the model can be continuously updated for a *dynamic analysis* at runtime, to verify properties on the current system state. Although, dynamic analysis can be resource limited, it has the advantage of direct feedback. With our model we can reflect on the currently executable behavior to inform about errors or trigger required system adaptations. This is comparable to use cases envisioned in models@run.time by Blair et al. [23].

Additionally, not only the analysis phase influences the used verification methods, but also the unique system features. However, during modeling we collected properties that may indicate structural or semantic mistakes in the system design. Such properties can be defined once by experienced people and used anywhere in a general toolkit. Before we address custom business process specifications we like to give a short record of potentially general properties.

5.1. General Verification Problems

The model can be used with current Petri net model-checking tools for verification. During modeling we thought of this non-exhaustive list of properties that might be of interest to evaluate:

- *Live locks*: Usually the system reacts to an input event with a finite sequence of followup events. Afterwards the system returns into a listening state where no transition can fire before a new environment event is introduced into the system.
- *Dead components*: can this component be reached from an initial marking?
- *Distance*: Given an event produced by a component, how many other components are needed before a token is produced on the event input of a target component?
- *Dead messages*: is there an unproduced message type for an initial marking.
- *Short circuits*: is there a component that can send messages to itself for any incoming message type? This behavior can cause a live lock and message flooding.
- *Event cascades*: which events can be caused by a starting event (up to a maximum depth)? If all possible cascades are finite the system will always "terminate" into a listing state.
- *Event sequences*: can a sequence of events occur from a given initial marking?

However, a graphical workflow specification can make verification more accessible, since it is more comprehensive. But to use workflows, a mapping to the system model is needed.

5.1.1. Business Process Mapping

For the comparison of the publish-subscribe model with a business process, we first need a mapping relation between both. We assume the business process has the form of a WF-net and will address them as workflow from here on.

Finding a mapping is potentially the largest added human labor in our approach. Each task in the workflow needs to be mapped to a substructure in the implementation model. With a careful design the additional development overhead can be limited i.e. by mapping task names to matching event names. But this may not always be possible.

Every well-formed WF-net has three features that need to be mapped: a single start event a single end event and the tasks that describe the business process.

Mapping the start event: Van der Aalst makes the case that workflows [2] can be best analyzed on a single case basis to avoid case mixing. Therefore, the start event is mapped to a single token on the $ComponentEvents$ place. The place' color domain, $Component \times ComponentEvent$, gives a hint to the token semantics: it encodes an event sensing component and the sensed event, which is either an environment event or a notification. As a result, the corresponding marking has the form $EnvEvents(startcomponent, startevent)$.

Mapping the end event: An end event is mapped to a token on the $NoseEvents$ place with the color domain $Component \times Channel \times AllEvent$. The token must be contained in a set of possible end tokens where the event type fixed and $Component \times Channel$ is variable. A marking with such a token is equivalent to a workflow in its end state. The marking in our model has the form $EnvEvents(S_{Components}, S_{Channels}, endevent)$ where $S_{Components}$ are all connected components, $S_{Channels}$ are all known channels and $endevent$ is the fixed event type of the final event.

In contrast to a workflow in its end state, the execution of the publish-subscribe model can continue. Usually other events exist and other component processes can be executed.

Mapping Tasks: The task semantics in a workflow is vague. Van der Aalst states that it "corresponds to a generic piece of work" [2] and that transitions in its Petri net representation, "abstracts from the internal behavior of a task".

As a result an abstract workflow task transition may not map directly to a transition in our model, because they represent dissimilar concepts. However, it is reasonable to assume that a workflow task is constraint by a component process for the following reason: software components in general encapsulate some functionality for system composition [19]. In the context of our model, this is the 'work' component processes do. The work done by a workflow task is related to the same behavior, just from another point of view. Given this relation, the case that a workflow task is more general than a component process, seems unlikely.

Yet, tasks and processes are not equivalent. Indeed, multiple tasks can be executed in a single process. In a special case, every task is represented by a single action. However, since actions are derived from code while tasks are derived from business processes, this perfect fit is unreliable. Actions with and without an associated task usually take turns.

Thus, some actions can be executed without an associated task and tasks may be associated with multiple actions. The latter case is divided into two variants: either the actions represent the same work just on different paths in the implementation (e.g. a call to publish in a function that is called at different locations), or the task spans multiple actions. If the task spans multiple actions it begins with an initial action and ends with a succeeding action in the current process. We know that the task is done, when the last successor action has fired. If multiple actions mark the end of a task because of conditions in the path, executing one is sufficient to mark the end.

In an additional, undesirable relation there is no transition a task can be mapped to. This is the case for tasks that do not result in a significant state change in the publish-subscribe model. As a result no action transition is generated. Again, we assume this to be a rare case, because chances are, a system specification is reflected in the system implementation.

Nonetheless, this case needs to be resolved with a surrogate transition, marking task completion. A special surrogate is the execution of the end process transition: given a task is finished somewhere in a process, we know it is finished not later than the containing process. Alternatively, a dummy action, publishing a dummy event, added to the component implementation can be used as surrogate. Since a task completion is usually desired information, we can even consider this state change as significant and require a published event. Such events might also be desirable in other parts of the development, e.g. for logging. Otherwise, another existing action can act as surrogate if it is known to be executed after the task is done.

This leaves us with the following mapping possibilities:

1. Mapping the task to a single action.
2. Mapping the task to the last action(s) of a collection of connected actions.
3. Mapping the task to multiple actions that represent the same unit of work.
4. Mapping the task to a surrogate transition where the surrogate is one of the following:
   a) The Process End transition.
   b) A synthetic action added in the implementation, i.e.: publishing an additional event.
   c) A single or multiple actions on different paths, that are known to be executed directly after the given task has been finished.

The mapping relation then is a relation $task \rightarrow target$ where $target \subseteq Action \lor target \in Process$.

5.2. Business Process Verification

With a workflow specification, the publish-subscribe model and a mapping relation between the two, we can collect assurance that the model conforms to the workflow.

Compared to the conformance checking literature [3] we have a special case. Usually conformance checking is done on previously gathered event traces. Either to check if the business process models an observed trace or to check if a trace conforms to the business process. However, instead of an event trace, we have a Petri net model that is able to generate all possible event traces. We want to know if our implementation model conforms to a business process (represented as WF-net). In this section we like to sketch out how this can be achieved.

A minimal conformance requirement is the existence of the previously discussed mapping. In a running system, components can connect and disconnect to the broker at any time. However, disconnected components cannot react to events making all internal actions unreachable. Since unreachable parts cannot be executed, we cannot map tasks of a business process to them. As a result a first inexpensive test can *check the existence of a given mapping* to all connected components. If no complete mapping exists, the workflow cannot be executed in our system.

If a mapping exists, further methods can be applied. *Rule checking* is a conformance checking method that can be extended to our model. In rule checking, different rules are extracted from the workflow to confirm the rules are observed in the event trace [3]. For example, a certain activity always precedes another or two activities are never executed in the same case. The same rules should hold in the publish-subscribe model. If they are translated into a logical formula, a model checker can be used to search for proof that the rule holds, or generate counterexamples.

A more extensive, but also a more expensive, verification method can be based on a reachability analysis. For both the workflow and the publish-subscribe model we can build a reachability graph (given a finite state space). Having both graphs, we can check for similarity with different approaches. Hens et al. suggested [18] to show that the publish-subscribe net is branching bisimilar [24] to the business process, preserving its branching structure. This results in an observational equivalence so that the same event sequences can be observed (including silent intermediate events). Branching bisimilarity requires a symmetric relation that relates both graphs and fulfill some properties [24]. If it can be shown that our mapping relation fulfills these properties, our publish-subscribe model is observational equivalent to the workflow.

Without a pure equivalence we may show that all, or at least some possible workflow event sequences are observable in the publish-subscribe model. For this all transition sequences can be generated, and a model checking tool can verify the existence in the publish-subscribe net.

An even weaker equivalence can measure the *alignment* of transition sequences between workflow and implementation model [3]. Alignment of two transition sequences increases the more synchronous moves workflow and system model do, i.e. transitions related by our mapping fire in the same order. If related transitions fire in a different order, alignment decreases. Depending on the requirements, an imperfect alignments may suffice to show conformance.

Additionally, van der Aalst describes *workflow related properties* that could be applied to our model [2]. E.g. all transitions in a WF-net should be alive and on a path between start place to end place. These properties should also hold for the mapped tasks. A weaker version of this property can verify the reachability of an end marking from the initial marking.

During runtime, our model changes e.g. if previously unknown components connect to the NoSe. As a result previously evaluated properties have to be reevaluated. Therefore, runtime evaluations need to be adjusted to the model complexity and the available resources. For example checking the existence of a mapping should be cheap while reachability based methods can get impractical. The methods of conformance checking seem like a good middle way between computability and expressiveness as well. And they have the added benefit to be directly applicable on observed logs besides event traces that where sampled from our model.

6. Conclusion

In this paper we presented a symmetric Petri net model of a publish-subscribe system for verification. We explicitly included a generic component model to represent the state of the composed system. Since the behavior of an EDA system is largely characterized by components interchanging events, our abstractions reduce events to types without event contents.

A framework conforming to our component model could use static code analysis to automatically compose a model instance from component implementations. However, the resulting model may depend on runtime information like involved components and channel names. In this case the model can be managed and adapted by the central notification service at runtime.

We showed which part of the model is static and which is dynamic. Since components are modelled explicitly, a mapping of business process tasks to model transitions can be found. This enables a comparison of the system capabilities, with business processes behavior specified as workflow nets. We showed a variety of methods that can be used for the comparison partially inspired by business process conformance checking. Some of these methods are inexpensive enough to be tested at runtime to increase assurance.

A framework implementation that supports building the model with a high level of automation is desirable future work. The minimally constraining component model still allows expressive process behavior but may also make model checking highly accessible for EDA systems.

References

[1] C. A. Petri, Kommunikation mit Automaten, Ph.D. thesis, University of Bonn, 1962.

[2] W. M. P. Van Der Aalst, THE APPLICATION OF PETRI NETS TO WORKFLOW MANAGEMENT, Journal of Circuits, Systems and Computers 08 (1998) 21–66. doi:10.1142/S0218126698000043.

[3] J. Carmona, B. van Dongen, A. Solti, M. Weidlich, Conformance Checking: Relating Processes and Models, Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-99414-7.

[4] K. M. Chandy, Event-driven applications: Costs, benefits and design approaches, Gartner Application Integration and Web Services Summit 2006 (2006).

[5] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer-Verlag, Berlin, 2006.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2003) 114–131. doi:10.1145/857076.857078.

[7] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Springer Science & Business Media, 1996.

[8] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad, Stochastic well-formed colored nets and symmetric modeling applications, IEEE Transactions on Computers 42 (1993) 1343–1360. doi:10.1109/12.247838.

[9] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-19345-3.

[10] N. Lohmann, E. Verbeek, R. Dijkman, Petri Net Transformations for Business Processes – A Survey, in: K. Jensen, W. M. P. van der Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, volume 5460, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 46–63. doi:10.1007/978-3-642-00899-3_3.

[11] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Information and Software Technology 50 (2008) 1281–1294. doi:10.1016/j.infsof.2008.02.006.

[12] O. F. A. Specification, Business process modeling notation specification, février (2006).

[13] S. Hinz, K. Schmidt, C. Stahl, Transforming BPEL to Petri Nets, in: W. M. P. van der Aalst, B. Benatallah, F. Casati, F. Curbera (Eds.), Business Process Management, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2005, pp. 220–235. doi:10.1007/11538394_15.

[14] S. OASIS, Web services business process execution language version 2.0, http://www.oasis-open. org/committees/tc_home. php? wg_abbrev= wsbpel (2007).

[15] L. Aldred, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, On the Notion of Coupling in Communication Middleware, in: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2005, pp. 1015–1033. doi:10.1007/11575801_6.

[16] V. Valero, H. Macià, G. Díaz, M. E. Cambronero, Colored Petri Net Modeling of the Publish/Subscribe Paradigm in the Context of Web Services Resources, in: M. Núñez, M. Güdemann (Eds.), Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 81–95. doi:10.

1007/978-3-319-19458-5_6.

[17] A. Gómez, R. J. Rodríguez, M.-E. Cambronero, V. Valero, Profiling the publish/subscribe paradigm for automated analysis using colored Petri nets, Software & Systems Modeling 18 (2019) 2973–3003. doi:10.1007/s10270-019-00716-1.

[18] P. Hens, M. Snoeck, G. Poels, M. De Backer, A Petri Net Formalization of a Publish-Subscribe Process System, 2011. doi:10.2139/ssrn.1886198.

[19] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-oriented Programming, Pearson Education, 2002.

[20] S. Becker, The palladio component model, in: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering - WOSP/SIPEW '10, ACM Press, San Jose, California, USA, 2010, p. 257. doi:10.1145/1712605.1712651.

[21] L. Lamport, N. Lynch, Chapter on Distributed Computing:, Technical Report, Defense Technical Information Center, Fort Belvoir, VA, 1989. doi:10.21236/ADA208996.

[22] Schmerl, Bradley, Aldrich, Jonathan, Garlan, David, Kazman, Rick, Yan, Hong, DiscoTect: A System for Discovering the Architectures of Running Programs Using Colored Petri Nets, Technical Report, Carnegie-Mellon University Pittsburgh, 2006.

[23] G. Blair, N. Bencomo, R. B. France, Models@ run.time, Computer 42 (2009) 22–27. doi:10.1109/MC.2009.326.

[24] T. Basten, Branching bisimilarity is an equivalence indeed!, Information Processing Letters 58 (1996) 141–147. doi:10.1016/0020-0190(96)00034-8.