Data and abstraction for scenario-based modeling with Petri nets

Citation for published version (APA):

Document status and date:
Published: 01/01/2012

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.
Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 15. Oct. 2019
Data and Abstraction for Scenario-Based Modeling with Petri Nets

Dirk Fahland and Robert Prüfer
Data and Abstraction for Scenario-Based Modeling with Petri Nets *

Dirk Fahland and Robert Prüfer

Eindhoven University of Technology, The Netherlands
Humboldt-Universität zu Berlin, Germany
d.fahland@tue.nl, pruefer@informatik.hu-berlin.de

Abstract. Scenario-based modeling is an approach for describing behaviors of a distributed system in terms of partial runs, called scenarios. Deriving an operational system from a set of scenarios is the main challenge that is typically addressed by either synthesizing system components or by providing operational semantics. Over the last years, several established scenario-based techniques have been adopted to Petri nets. Their adaptation allows for verifying scenario-based models and for synthesizing individual components from scenarios within one formal technique, by building on Petri net theory. However, current adaptations of scenarios face two limitations: a system modeler (1) cannot abstract from concrete behavior, and (2) cannot explicitly describe data in scenarios. This paper lifts these limitations for scenarios in the style of Live Sequence Charts (LSCs). We extend an existing model for scenarios, that features Petri net-based semantics, verification and synthesis techniques, and close the gap between LSCs and Petri nets further.

Keywords: scenario-based modeling, data, abstraction, Petri nets

1 Introduction

Designing and implementing a distributed system of multiple components is a complex task. Its complexity originates in the component interactions. Established scenario-based methods such as (Hierarchical) Message Sequence Charts ((H)MSCs) [26] and Live Sequence Charts (LSCs) [6] alleviate this complexity: A system designer specifies the system’s behaviors as a set of scenarios. Each scenario is a self-contained, partial execution usually given in a graphical notation. Then system components are synthesized (preferably automatically) that together interact as described in the scenarios. Alternatively, a specification becomes a system model by equipping scenarios with operational semantics.

A significant drawback of established techniques is that components have to be synthesized in a different formal theory than the one in which the scenarios

are given, e.g., HMSCs or LSCs are synthesized into Petri nets or statecharts [7, 19]. Also operational semantics for MSCs and LSCs require a translation into another formalism like automata [31], process algebras [30], or require involved formal techniques such as graph grammars [24] or model-checking [20]. Many HMSC and LSC specifications cannot be distributed into components but require centralized control [7, 4]. This renders turning scenarios into systems surprisingly technical while scenarios appear to be very intuitive.

Approaches which express scenarios, system behaviors and system model in the same formal theory face less problems. In particular, approaches which describe scenarios in terms of Petri nets and their partially ordered runs, e.g., [9], have been successful. The approach in [2] presents a general solution for synthesizing a Petri net from HMSC-style specifications in a Petri net-based model. [8] shows how to compose complex system behaviors from single Petri net events with preconditions. The model of oclets [12, 13] adapts ideas from LSCs to Petri nets: a scenario is a partial run with a distinguished precondition; system behavior emerges from composing scenarios based on their preconditions. This idea allows oclets to adapt existing Petri net techniques for a general solution to the synthesis problem for LSC-style scenarios [13].

The Petri net-based scenario techniques [9, 2, 8, 12, 13] in their current form only describe control-flow and provide no means for abstracting behavior in a complex specification. Any practically applicable specification technique needs some notion of abstraction as well as some explicit notion of data, and means to describe several components of the same kind.

This paper addresses these problems of practical applicability of Petri-net based scenarios. We show for the model of oclets how to extend scenarios by abstract causal dependencies (abstracting from a number of possibly unknown actions between two dependent actions), and how to express data in scenarios by adapting notions of Algebraic Petri nets [32]. Our contribution is two-fold: First, abstraction and data are two key features of LSCs, so our extension of oclets closes the gap between LSCs and Petri nets further. Second, all our extensions are simple generalizations of existing concepts from Petri net theory, giving rise to the hope that existing verification and synthesis results, e.g., [13], can be transferred to the more expressive model proposed in this paper.

We proceed as follows. Section 2 recalls the scenario-based approach in more detail and explains the basic ideas of oclets by an example. In Sections 3 and 4, abstract causal dependencies and data are introduced into the model, respectively. Section 5 discusses the relation of oclets to Petri nets. We conclude and discuss related and future work in Section 6.

2 Specifying with Scenarios

This section recalls the scenario-based approach by the help of an example and discusses features and limitations of scenario-based specification techniques. Two of these limitations will be addressed in the remainder of this paper.
2.1 Running example and requirements for capturing it

Our running example is a gas station (adapted from [23]) that allows customers to refuel their cars using one of the available pumps as follows. When a customer arrives with his car at a pump, he asks the operator to activate that pump for a certain amount of fuel for which he pays in advance or after he finished pumping the gas. The customer can start an activated pump to refuel his car, pumping gas one unit at a time. The pump stops when all requested gas has been pumped, or when stopped by the customer. The pump then signals the operator the pumped amount and the operator returns corresponding change to the customer. Each customer gets a free snack that he may pick up after starting the pump and before leaving the gas station.

A specification technique capable to express this gas station has to describe (R1) distributed components (e.g., pump, customer, operator), (R2) interaction between components, (R3) sequential, independent, and alternative ordering of actions, (R4) preconditions of actions (e.g., “when all requested gas has been pumped”), (R5) actions that depend on data (e.g., returned change, amount of pumped gas), (R6) multiple instances of the same kind of component (e.g., multiple customers and pumps). Furthermore, a specification technique also should allow a system designer to keep an overview of larger specifications by (R7) means of abstraction. Finally, the specification technique should allow to (R8) derive the specified behavior in an intuitive way, that is, the derived behavior should be “correct by construction” and not require additional verification.

2.2 Principles of scenario-based specifications

In the scenario-based approach, a system designer obtains a system model of a distributed system (e.g., our gas station example) in two steps. First she describes the system behavior as component interactions. One scenario describes how several components interact with each other in a particular situation; a specification is a set of scenarios. When she completed the specifications, components are synthesized (preferably automatically) such that all components together interact as described in the scenarios.

The most valued feature of this approach is that scenarios tangibly decompose complex system behavior into smaller, self-contained stories of component interactions (scenarios) which are easy to understand. Fig. 1 shows 3 scenarios (M0, M1, M2) of the running example in the well-established syntax of MSCs. In each MSC, a vertical lifeline describes one component, arrows between
components describe interactions, boxes at components describe local actions; \( M_0 \) is a special case as it describes the creation of a new instance of a customer. Established scenario-based techniques adhere to a few simple principles that allow to derive system behavior from scenarios in a comprehensible way as follows.

S1 A scenario is partial order of actions, understood as a partial run of the system. A specification is a set of scenarios.
S2 System behavior follows from composing (appending) scenarios.
S3 Each scenario distinguishes a prefix as a precondition describing when the scenario can occur; the remainder of the scenario is called contribution.
S4 When a system run ends with a scenario’s precondition, the run can continue by appending the scenario’s contribution. Scenarios with the same precondition and different contributions lead to alternative runs.

S1 is the most generally agreed upon principle for scenarios and usually expressed in an MSC-like notation as in Fig. 1; other notations are possible [6, 9, 12, 2]. To specify practically relevant systems, more principles are needed. The probably most established scenario-based techniques – (H)MSCs, LSCs and UML Sequence Diagrams – realize these principles differently as we discuss next.

**HMSCs** proposed principle S2 first, where the order of scenario composition is described by a finite automaton [26]. For instance, HMSH of Fig. 1 describes that \( M_0 \) is followed by \( M_1 \) or alternatively by \( M_2 \), and then \( M_0 \) can occur again. This way, HMSCs are capable to express the requirements R1-R3 of Sect. 2.1, but not R4. In the HMSC standard, notions of data (R5) are only provided on a syntactical, but not on a semantical level [26]. Also multiple instances of the same component cannot be expressed: the HMSC of Fig. 1 allows only one customer to be served at a time. Means of abstraction (R7) are provided by the possibility of nesting one MSC inside another MSC. UML Sequence Diagrams express scenario composition entirely by nesting scenarios in each other.

It has been repeatedly observed that this approach to scenarios requires a global understanding of the entire system as the ordering of scenarios is described in a global automaton [17, 13]. Moreover, as MSCs of a HMSC cannot overlap, a specification may have to be refactored when a new scenario shall be included [34] and specifications tend to consist of many small-scale scenarios composed in complex ways, which is counter-intuitive to the idea of one scenario describing a “self-contained story” of the system [38, 13].

**LSCs** extend MSCs in a different way to provide enough expressive power [6]. Altogether, LSCs provide notions for preconditions of scenarios, data, multiple instances and abstraction on component lifelines, satisfying R1-R7 of Sect. 2.1 [21]. LSCs first proposed principle S3 that a scenario is triggered by a precondition; this idea has been adopted in other approach as well [17, 35, 12]. Fig. 2 shows 3 LSCs corresponding to the MSCs of Fig. 1. However, LSCs specify system behavior not by scenario composition, but each LSC denotes a linear-time temporal logic formula: when a system run ends with an LSC’s precondition, then the run must continue with the LSC’s contribution, i.e., the given events occur in the run eventually in the given order.
For instance, L1 expresses that after arrive occurred (precondition), the customer pays, starts and stops the pump, and gets his change (contribution). Additionally, LSCs specify particular events as implicitly forbidden at particular stages of an LSC. For instance, in L2, event pay is forbidden to occur before event stop. At the same time, L1 requires pay to occur before right after arrived (before stop) and forbids occurrences of pay after stop. In contrast to intuition, L1 and L2 are contradictory and no system satisfies both LSCs. This contradiction arises because of the linear-time semantics of LSCs as both L1 and L2 have to occur in the same run. The contradiction vanishes when using a branching-time semantics for LSCs as proposed in the model of epLSCs [37]: when the pre-condition occurs, some run continues with the contribution (principle S4). However, also in this model, deriving system behavior from a specification is cumbersome: whether two epLSCs have to occur in different runs or may occur overlappingly in the same run still requires to check their temporal logic formulae, possibly requiring verification on large parts of the specified state-space [20].

Between LSCs and HMSCs. To summarize, HMSCs have a simple semantics that allows to derive specified behaviors by composing scenarios. Though, HMSCs suffer from the global automaton and that scenarios cannot overlap. LSCs allow for local preconditions and overlapping scenarios, but are based on an intricate semantics that makes it hard to understand behavior specified by a set of LSCs. That simplicity of semantics influences the way how components can be synthesized from scenarios can be seen when comparing available techniques. While synthesis is generally infeasible from both HMSCs and LSCs, synthesis from feasible subclasses of scenarios to Petri nets is straight forward for HMSCs [31, 2] yielding components that are correct by construction, whereas synthesis from LSCs [1, 28] requires to verify the synthesis result to ensure correctness.

In the following, we derive a scenario-based technique that inherits the advantages of HMSCs and LSCs without their disadvantages: a LSC-style syntax of scenarios with local precondition is given a HMSC-style semantics where system behavior follows from scenario composition. The hope is that a simpler semantic model allows to synthesize from LSC-style scenarios components that are correct by construction. Indeed, this already has been proven to be successful for a simple model of scenarios that we present next.
2.3 A simple model for scenarios based on Petri nets

We derive a simple semantic model for LSC-style scenarios by applying principles of Petri net theory. Petri net-based scenarios benefit from expressing scenarios, behavior, and system in the same formal model, which allows to create a Petri-net based operational semantics for scenarios and supports the crucial step from specification to system model. For HMSC-style specifications, corresponding semantics and synthesis techniques are already available [2].

For LSC-style scenarios, a simple model that adopts the semantics of epLSCs [37] to Petri nets has been proposed in the model of oclets [12, 13] that we recall next. Oclets realize all principles S1-S4 in the following way. Fig. 3 shows four scenarios of the gas station example of Sect. 2.1 in the notation of oclets. The partial order of actions is expressed as a so called labeled causal net. A transition (place) of a causal net is called event (condition); the flow relation defines a partial order over events and conditions s.t. the net is conflict-free. The grey-filled (white-filled) nodes indicate the precondition (contribution) of an oclet.

The four oclets describe some behavior of the gas station example. Oclet prepay: after a customer arrived at the gas station, he asks to activate a pump and pre-pays his gas (event pay). Alternatively (oclet pay later), the customer may just ask the operator to activate the pump (order). Oclet activate: after receiving the order, the operator activates the pump. Oclet start: When a pump is activated, the customer may start it.

Oclets generalize the semantics of Petri net transitions to scenarios. Whenever a run ends with an oclet’s precondition, the oclet is enabled and the run can continue by appending the oclet’s contribution. Two oclets with the same precondition and different contributions are alternatives and hence yield alternative continuations.

For example, consider the run $\pi_0$ indicated in Fig. 4. In $\pi_0$, oclets pay and use card are enabled. Continuing $\pi_0$ with pay yields the run $\pi_1$ indicated in Fig. 4(top), that was obtained by appending pay’s contribution to $\pi_0$. Appending use card yields the run $\pi'_1$ of Fig. 4(bottom) that is alternative to $\pi_1$. In $\pi_1$, oclet activate is enabled; appending its contribution yields $\pi_2$. This way, oclets derive the specified behavior of the gas station by composition.

Properties and limitations. In their current form, oclets allow to express properties (R1-R4) of Sect. 2.1 and allow to analyze scenarios, and synthesize a system by reusing and extending Petri net techniques [13]. Yet, oclets cannot express data (e.g., how much gas to pump) or distinguish instances (e.g.,
two different pumps). Furthermore, the events in an oclet currently describe a “contiguous piece of behavior.” In the worst case, the specification consists of many short scenarios, only. Abstraction would allow a system designer to also specify longer scenarios of corresponding, non-contiguous pieces of behavior. In the remainder of this paper, we show how to introduce abstraction and data to oclets, thus providing a scenario-based technique between HMSCs and LSCs. We stay close to the spirit of Petri nets and define an extension in terms of simpler, existing principles.

3 Adding Abstraction: Abstract Dependencies

In this section, we introduce means to abstract from behavior in a scenario. We sketch the idea by our running example before we present formal definitions.

3.1 Abstracting causal dependencies

As stated in Sect. 2.3, the flow relation of an oclet as described in [12, 13] denotes direct causal dependencies which may restrict how a particular system behavior can be specified. Abstract causal dependencies allow other events to occur between two events of a scenario.

Fig. 5 shows examples of abstract dependencies; an abstract dependency is drawn as a dashed arrow. The oclet main describes the main interaction between customer and the gas station’s operator and the pump (see Sect. 2.1). The oclet abstracts from other behavior taking place at the gas station, some of that behavior is needed to make the customer’s interaction happen. For instance, oclet main only abstractly describes the dependency of the two Pump conditions. This allows events which are not depicted to occur between these conditions. In particular event start of oclet start of Fig. 3 can occur here. In other words, oclet start refines this abstract dependency of oclet main. There are further abstract dependencies in main that have to be detailed by other oclets. Yet, the main scenario clearly describes that once the pump has been activated, it eventually completes pumping, which will lead to the operator returning change to the customer. We complete the specification in Sect. 4.5.
Abstract dependencies are also useful in a scenario’s precondition. Here, they allow to specify that an oclet is enabled if a specific behavior occurred “some time” in the past, instead of immediately. For instance, oclet main[completed] in Fig. 5 expresses that activate must have occurred some time in the past in order to enable event completed, including the possibility that other events occurred in between.

Introducing abstract dependencies in oclets comes at a price: we cannot continue a run with an enabled oclet by appending its contribution. The principle solution is to decompose an oclet into basic oclets such as main[completed] in Fig. 5(top right). Each basic oclet contributes exactly one event, its precondition consists of all transitive predecessors in the original oclet. This effectively moves abstract dependencies into preconditions and system behavior emerges from concatenating well-defined single events.

In the remainder of this section, we formalize these ideas by extending syntax and semantics of oclets [12, 13] with abstract dependencies. We assume the reader to be familiar with the basic concepts of Petri Nets and their distributed runs; see [33] for an introduction.

### 3.2 Basic Notions

First, we recall some basic notation. A partial order over a set $A$ is a binary relation $\leq \subseteq A \times A$ that is reflexive (i.e. $\forall a \in A : a \leq a$), transitive (i.e.
∀a, a', a'' ∈ A : a ≤ a' ∧ a' ≤ a'' ⇒ a ≤ a''), and antisymmetric (i.e. ∀a, a' ∈ A : a ≤ a' ∧ a' ≤ a ⇒ a = a'). Let a↓ := {a' ∈ A | a' ≤ a} and a↑ := {a' ∈ A | a ≤ a'}. Denote the transitive predecessors and successors of a ∈ A, respectively. As usual, for a relation R ⊆ (A × A), R* and R~ denote the transitive, and reflexive-transitive closures of R.

We write a Petri Net as N = (P, T, F), with places P, transitions T (P ∩ T = ∅), and arcs F ⊆ (P × T) ∪ (T × P). Notation-wise, introducing N, N1, N2 implicitly introduces their components P_N, P_1, P_2 etc. For each node x ∈ X_N := P ∪ T, \*x = {y ∈ X | (y, x) ∈ F} and x* = {y ∈ X | (x, y) ∈ F} are the pre- and post-set of x, respectively. A causal net π = (B, E, F) is a Petri net where (1) \preceq := F* is a partial order over X_π, (2) for each x ∈ X_π, x↓ is finite, and (3) for each b ∈ B, |b| ≤ 1 and |b*| ≤ 1. An element of B (E) is called condition (event). The arcs of a causal net denote direct causal dependencies: x depends on y iff x ≤ y, and x and y are concurrent iff neither x ≤ y nor y ≤ x. We write min π = {x | x* = ∅} and max π = {x | x* = ∅} for the nodes without predecessor and successor, respectively.

In the following, we consider labeled causal nets π = (B, E, F, ℓ) where each node x ∈ X_π is assigned a label ℓ(x) ∈ L from some given set L. We will interpret a labeled causal net as a partially ordered run (of a possibly unknown system); in analogy to Petri nets, an event e describes an occurrence of an action (or transition) ℓ(e), and a condition b describes an occurrence of a local state (a token on a place) ℓ(b).

### 3.3 Runs with abstract dependencies

We formally introduce abstract dependencies by generalizing the notion of a partially ordered run to an abstract run. This definition then canonically lifts to oclets with abstract dependencies.

**Definition 1 (Partially ordered run).** An abstract partially ordered run (run for short) π = (B, E, F, A, ℓ) is a labeled causal net (B, E, F, ℓ) with abstract dependencies A ⊆ X_π × X_π s.t. \preceq := (F ∪ A)* is a partial order over the nodes X_π. π is concrete iff A = ∅.

We write πα for any run that is isomorphic to π by the isomorphism α : π → πα. Run π occurs in run ρ, written π ⊆ ρ, iff B_π ⊆ B_ρ, E_π ⊆ E_ρ, F_π ⊆ F_ρ, A_π ⊆ A_ρ, ℓ_π = ℓ_ρ|X_π. We will work with two important relations on runs: prefixes and refinement.

**Definition 2 (Prefix).** A run π is a prefix of a run ρ, written π ⊆ ρ, iff (1) min ρ ⊆ X_π ⊆ X_ρ (2) F_π = F_ρ|X_π × X_π, A_π = A_ρ|X_π × X_π (π contains all predecessors), (3) for each e ∈ E_ρ and all (e, b) ∈ F_ρ holds (e, b) ∈ F_π (events have all post-conditions). The set of all prefixes of ρ is Pre(ρ) := {π | π ⊆ ρ}.

Fig. 5(bottom right) shows an abstract run; Fig. 4 shows concrete runs; π_1 is a prefix of π_2; π'_1 is not a prefix of π_1. Each abstract run describes a set of concrete runs (without abstract dependencies) that refine the abstract run. Intuitively, an
abstract dependency in a run \( \pi \) can be refined by a number of nodes that respect the partial order of \( \pi \). The refinement can exclude nodes with a particular label, which we need for oclet semantics.

**Definition 3 (Refine an abstract run).** Let \( \pi \) and \( \rho \) be abstract distributed runs. Let \( \kappa : A_\pi \rightarrow 2^L \) assign each abstract dependency in \( \pi \) a (possibly empty) set of forbidden labels. \( \rho \) refines \( \pi \) w.r.t. \( \kappa \), written \( \rho \preceq \kappa \pi \), iff

- \( B_\pi \subseteq B_\rho, E_\pi \subseteq E_\rho, \forall x \in X_\pi : \ell_\pi(x) = \ell_\rho(x) \),
- \( F_\pi \subseteq F_\rho \) and \( (F_\pi \cup A_\pi)^+ \subseteq (F_\rho \cup A_\rho)^+ \), and
- \( \forall (x,y) \in A_\pi : \exists e \in E_\rho : x \leq_\rho e \leq_\rho y \land \ell_\rho(e) \in \kappa(x,y) \).

We write \( \pi \preceq \rho \) if \( \kappa(x,y) = \emptyset \) for all \( (x,y) \in A_\pi \). Every run \( \pi \) describes the set \( J_\pi = \{ \rho \mid \rho \preceq \pi, A_\rho = \emptyset \} \) of all concrete runs that refine \( \rho \).

Run \( \pi_2 \) of Fig. 4 refines the abstract run \( \pi'_2 \) of Fig. 5.

### 3.4 Scenarios with abstract dependencies

**Syntax.** Abstract dependencies canonically lift from abstract runs to oclets. As already sketched in Sect. 3.1, an oclet is an abstract run with a distinguished prefix.

**Definition 4 (Oclet).** An oclet \( o = (\pi, \text{pre}) \) consists of an abstract distributed run \( \pi \) and a prefix \( \text{pre} \subseteq \pi \) of \( \pi \).

Fig. 5 shows oclet main. Def. 4 generalizes “classical” oclets as introduced in [12, 13] by abstract dependencies of the underlying run. We call \( \text{pre} \) the precondition of \( o \) and the suffix \( \text{con}(o) := (B_\pi \setminus B_{\text{pre}}, E_\pi \setminus E_{\text{pre}}, F_\pi \setminus F_{\text{pre}}, A_\pi \setminus A_{\text{pre}}, \ell_\pi|_{X_{\text{con}(o)}}) \) its contribution; technically \( \text{con}(o) \) is not a net as it contains arcs adjacent to nodes of the precondition.

A specification is a set of oclets together with an initial run that describes how system behavior starts.

**Definition 5 (Specification).** A specification \( \Omega = (O, \pi_0) \) is a set \( O \) of oclets together with an abstract run \( \pi \) called initial run.

A specification usually consists of a finite set of oclets; we allow the infinite case for technical reasons. This is all syntax that we need.

**Semantics.** The semantics of oclets is straightforward. An oclet \( o \) describes a scenario with a necessary precondition: the contribution of \( o \) can occur whenever its precondition occurred. Then, we call \( o \) enabled.

**Definition 6 (Enabled Oclet).** Let \( o = (\pi, \text{pre}) \) be an oclet and let \( \pi \) be a run. Each \( (x,y) \in A_\pi \) defines the post-events of \( y \) as forbidden, i.e., \( \kappa(x,y) = \{ \ell(e) \mid (y,e) \in F_o \cup A_o, e \in E_o \} \). Oclet \( o \) is enabled in \( \pi \) iff there exists a refinement \( \text{pre}' \preceq \kappa \text{pre} \) s.t. (1) \( \text{pre}' \subseteq \pi \), (2) max \( \text{pre}' \subseteq \max \pi \), and (3) \( X_{\text{con}(o)} \cap X_\pi = \emptyset \).
A set $R$ of oclets can be used to specify the behaviors of a distributed system. A specification $\Omega$ consists of a set of oclets $\pi$ that can be decomposed into basic oclets. Each concrete event $e$ is enabled in $\pi$ at location $o$ iff $o^e$ is enabled in $\pi$. For technical reasons, $o$'s contribution is assumed to be disjoint from $\pi$ so that it can be appended to $\pi$.

**Definition 7 (Continue a run with an oclet).** Let $o$ be an oclet and let $\pi$ be a distributed run. If $o$ is enabled in $\pi$, then the composition of $o$ and $\pi$ is defined as $\pi \triangleright o := (\pi \cup o) = (B_\pi \cup B_o, E_\pi \cup E_o, F_\pi \cup F_o, \ell', A_\pi \cup A_o)$ with $\ell'(x) = \ell(x)$, for all $x \in X_\pi$, $\ell'(x) = \ell_o(x)$, for all $x \in X_o$.

A specification $\Omega$ describes a set $R(\Omega)$ of abstract runs: that is, the prefixes of all runs that can be constructed by repeatedly appending enabled oclets of $\Omega$ to the initial run. The concrete system behaviors specified by $\Omega$ are the concrete runs that refine $R(\Omega)$.

**Definition 8 (Semantics of a specification).** Let $\Omega = (O, \pi_0)$ be a specification. The abstract runs of $\Omega$ are the least set $R(\Omega)$ of runs s.t.

1. $Pre(\pi_0) \in R(\Omega)$, and
2. for all $\pi \in R(O), o \in O$, if $o$ is enabled in $\pi$ at $\alpha$ then $Pre(\pi \triangleright o^\alpha) \in R(O)$.

A set $R$ of concrete runs satisfies $\Omega$ iff for each $\pi \in R(\Omega)$, $[\pi] \cap R \neq \emptyset$.

### 3.5 Operational semantics

A system designer can use oclets to specify the behaviors of a distributed system. Harel et al. [20] suggested to turn a specification into an executable system model by providing operational semantics for scenarios.

Operational semantics describe system behavior as occurrences of single events. Each event has a local precondition, if the precondition holds, the event can occur by being appended to the run. These principles are naturally captured by basic oclets that contribute just a single event; we define operational semantics of oclets by decomposing complex oclets into basic oclets.

We call an event $e$ of a run $\pi$ concrete iff $e$ has no abstract dependencies, i.e., $\forall (x, y) \in A_\pi : x \neq e \neq y$. Oclet $o$ is basic iff its contribution consists of exactly one concrete event $e$ (with post-conditions). If $o$ is not basic, then it can be decomposed into basic oclets. Each concrete event $e$ of $o$'s contribution induces the basic oclet $o[e]$ that contributes $e$ and $e$'s post-set and has as precondition all transitive predecessors $e \downarrow^* \leq_\pre$ of $e$ in $o$.

**Definition 9 (Decomposition into basic oclets).** Let $o = (\pi, \pre)$ be a basic oclet. A concrete event $e \in E_{con(o)}$ induces the basic oclet $o[e] = (\pi', \pre')$ with $X' = e \downarrow^* \leq_\pre \cup e$, $F' = F|_{X' \times X'}$, $A' = A|_{X' \times X'}$, $\ell' = \ell|_{X'}$, and $\pre' \subseteq \pi'$ s.t. $X_{\pre'} = e \downarrow^* \leq_\pre \setminus \{e\}$. The basic oclets of $o$ are $\hat{o} = \{o[e] \mid e \in E_{con(o)}, e \text{ is concrete}\}$.
There may be specifications where a particular action $a$ has no corresponding concrete event $e$, $\ell(e) = a$, i.e., it is always adjacent to some abstract dependency. In this case, the specification provides no information on how to refine these abstract dependencies. We found it useful for concise specifications, that in this case, a non-concrete event $e$ of an oclet $o = (\pi, \text{prv})$ also induces the basic oclet $o[e]$ where the abstract dependencies between $e$ and some condition $b$ are turned into direct dependencies, i.e., replace in $o$ each abstract dependency $(b, e) \in A_o, b \in B_o$ by an arc $(b, e) \in F_o$ and each $(e, b) \in A_o, b \in B_o$ by an arc $(e, b) \in F_o$, and then compute $o[e]$ as in Def. 9.

In both cases of Def. 9 and with additional basic oclets, the operational semantics of an oclet specification follows from its basic oclets.

**Definition 10 (Operational semantics).** Let $\Omega = (O, \pi_0)$ be a specification, $\hat{\Omega} = (\bigcup_{o \in O} \hat{o}, \pi_0)$ is the basic specification induced by $\Omega$. It defines the operational semantics of $\Omega$ as $R(\hat{\Omega})$. $\Omega$ is operational iff $R(\hat{\Omega})$ satisfies $\Omega$.

We call $R(\hat{\Omega})$ the operational semantics of $\Omega$ because $\hat{\Omega}$ describes a set of single events. Each event is enabled when its local precondition holds; an enabled event can occur (by appending it to the run). Not every specification has operational semantics that satisfy the specification. A non-operational specification needs to be refined to become operational. Yet, we can characterize operational specifications.

**Theorem 1.** Let $\Omega = (O, \pi_0)$ be a specification s.t. $\pi_0$ is concrete and each event in the contribution of each oclet in $\hat{\Omega}$ is concrete. Then $\hat{\Omega}$ is operational.

**Proof.** This theorem has been proven for oclets without abstract dependencies in [12]: by induction on the prefixes of an oclet’s contribution, an oclet’s contribution can be reconstructed from its basic oclets. In particular, whenever oclet $o$ is enabled in $\pi$, also each basic oclet of $o$ is enabled in $\pi$ or in a continuation $\pi \triangleright o[e_1] \triangleright \ldots \triangleright o[e_k]$. This reasoning applies also when $o$ has abstract dependencies in its precondition (still, each basic oclet $o[e]$ gets enabled whenever $o$ is enabled).

Theorem 1 states a rather strict sufficient condition for operational specifications (abstract dependencies only in preconditions). Next, we present a more general sufficient condition: $\Omega$ is operational if all abstract dependencies of $\Omega$ can be refined by its basic oclets.

Refining abstract runs lifts to oclets: oclet $o_2 = (\pi_2, \text{prv}_2)$ refines oclet $o_1 = (\pi_1, \text{prv}_1)$, written $o_2 \preceq o_1$, iff $\pi_2 \subseteq \pi_1$ and $\text{prv}_2 = \text{prv}_1$, i.e., we may only refine contributions. An oclet’s refinement can be justified by another oclet.

**Definition 11 (Justified by oclet).** Let $o_1, o_2$ be oclets s.t. $o_2$ refines $o_1$. The refinement from $o_1$ to $o_2$ is justified by an oclet $o$ iff there exists run $\rho \subseteq \pi_{o_2}$ s.t. $X_{o_2} \setminus X_{o_1} \subseteq X_{\text{con}(o)}, F_{o_2} \setminus F_{o_1} \subseteq F_{\text{con}(o)}, A_{o_2} \setminus A_{o_1} \subseteq A_{\text{con}(o)}$.

Let $\Omega = (O, \pi_0)$ be an oclet specification. The refinement from $o_1$ to $o_2$ is justified by $\Omega$ iff there is a sequence of refinements from $o_1$ to $o_2$ s.t. each refinement is justified by a basic oclet $o \in O$ (or by an isomorphic copy $o^a$ of $o$).

**Theorem 2.** Let $\Omega = (O, \pi_0)$ be a specification. If each oclet $o_1 \in O$ can be refined to an oclet $o_2$ justified by $\Omega$ s.t. all events of $o_2$’s contribution are concrete, then $\hat{\Omega}$ is operational.
Lemma 1. Let $\Omega = (O, \pi_0)$ be a specification. Let $o$ be an oclet that can be refined into an oclet $o'$, justified by $\Omega$, s.t. each $e \in E_{\text{con}(o')}^\ast$ is concrete. Let $\pi$ be a run s.t. $o$ is enabled in $\pi$. Then there exists a sequence $a_1, \ldots, a_n \in \hat{O} \cup \hat{o}$ of basic oclets s.t. $(\pi \triangleright o_1 \triangleright \ldots \triangleright o_n) = (\pi \triangleright o') \leq (\pi \triangleright o)$.

Proof (Lem. 1). Proof by the number $n = |E_{\text{con}(o')}|$. For $n = 0$, the proposition holds trivially. For $n > 0$, let $e \in E_{\text{con}(o')}^\ast$ be a maximal (no other event of $o$ succeeds $e$).

Case 1: $e \in E_{\text{con}(o')}^\ast$ and $e$ is concrete. Obtain $o\searrow e, o'\searrow e$ by removing $e$ and $e^\ast$ from $o, o'. o'\searrow e \subseteq o\searrow e$ justified by $\Omega$ and $\rho := \pi \triangleright o_1 \triangleright \ldots \triangleright o_{n-1} = \pi \triangleright o'\searrow e \subseteq \pi \triangleright o\searrow e$ by inductive assumption. From $e$ being complete follows $o[e] \in \hat{o}$ and $o[e]$ enabled in $\rho$. Thus $(\rho \triangleright o[e]) = (\pi \triangleright o'\searrow e \triangleright o[e]) = (\pi \triangleright o') \leq (\pi \triangleright o)$.

Case 2: $e \in E_{\text{con}(o')}^\ast$ and $e$ is not complete, or for some $b \in e^\ast, b \in \max \pi_o$ (s.t. $e$ is added by refining $(x, b) \in A_o$; see Def. 3). By Def. 11, a basic oclet $\hat{o}$ of $\Omega$ with $\hat{e} \subseteq \pi'$ justifies $e \in E_{\hat{o}} \cap E_o$ and all $(x, e), (e, y) \in F_{\hat{o}} \setminus F_o \subseteq F_{\text{con}(\hat{o})}$. Thus, there ex. oclet $o''$ s.t. $o'' \subseteq o'$ where $o'' \leq o'$ is justified by $\hat{o}$ (and the rest by $\Omega$). All events of $o''$ (except $e$) are concrete. Obtain $o\searrow e, o'\searrow e, o''\searrow e$ by removing $e$ and $e^\ast$ from $o, o', o''$. By construction holds $o''\searrow e = o'\searrow e$ and max $\hat{e} \subseteq \max \pi''\searrow e$. Refinement $o''\searrow e \leq o\searrow e$ is justified by $\Omega$, thus $\rho := (\pi \triangleright o_1 \triangleright \ldots \triangleright o_{n-1}) = (\pi \triangleright o''\searrow e) \leq (\pi \triangleright o\searrow e)$ holds by inductive assumption. Further, $\hat{e} \subseteq \pi''\searrow e \leq \rho$ and max $\hat{e} \subseteq \max \pi''\searrow e \leq \max \rho$ holds. Thus, $\hat{o}$ is enabled in $\rho$ and $(\rho \triangleright \hat{o}) = (\pi \triangleright o'\searrow e \triangleright \hat{o}) = (\pi \triangleright o') \leq (\pi \triangleright o)$.

Proof (Thm. 2). By induction on the semantics of $\Omega = (O, \pi_0)$. Let $\hat{\Omega} = (\hat{O}, \pi_0)$. Base: By Def. 8, $\pi_0 \in R(\Omega)$ and $\pi_0 \in R(\hat{\Omega})$. Step: Show for $\pi \triangleright o \in R(\hat{\Omega})$ (o enabled in $\pi$) that there ex. $\rho \in R(\hat{\Omega})$ s.t. $\rho \leq \pi \triangleright o$. By assumption there ex. a refinement $o' \leq o$ justified by $\Omega$. As $\hat{o} \subseteq \hat{O}$, Lem. 1 implies that $\pi \triangleright o' \in R(\hat{\Omega})$ and $(\pi \triangleright o') \leq (\pi \triangleright o)$.

4 Adding Data: $\Sigma$-Oclets

In the current model of oclets, conditions and events can be labeled with specific data values. But the language of oclets does not allow to concisely describe manipulation of data values and data-dependent enabling of events. In this section, we extend oclets with notions of data. As in Sect. 3, we first informally describe our approach and afterwards present our formal model.

4.1 Specifying Data

We propose to incorporate data into oclets in the same way as data has been introduced in Place/Transition nets (P/T nets) by several classes of high level Petri Nets, e.g., in Coloured Petri nets [27]. Recall that in CPNs, a marking distributes concrete values (from one or several domains) on places; expressions on arcs describe which values are consumed or produced by an occurrence of a transition; a guard expression at the transition may restrict consumable and
... producible values further. The method-wise relevant property of CPNs is that each coloured net can be unfolded w.r.t. all possible interpretations of its expressions into an equivalent P/T net. Then, a (black) token on a P/T net place \((p,v)\) denotes value \(v\) on colored place \(p\); likewise each colored transition unfolds to several P/T net transitions defined by the consumed and produced concrete values. A special class of CPNs are Algebraic Petri nets [32] where expressions and values are defined by a \(\Sigma\)-algebra with a signature \(\Sigma\).

We adapt the idea of Algebraic Petri nets to oclets and introduce \(\Sigma\)-oclets. In a run, a place \(p\) carrying the value \(v\) is labeled \((p,v)\). In a \(\Sigma\)-oclet, each condition is labeled with a pair \((p,t)\) where \(p\) is a name (e.g., of a component) and \(t\) is a term (over function symbols and variables of \(\Sigma\)) describing possible values on \(p\). Events are labeled with names of actions as before; an event may carry an additional guard expression (defined over \(\Sigma\)). A system designer can use different terms in the pre- and post-sets of an event to describe how values change by an occurrence of an event.

Fig. 6 shows \(\Sigma\)-oclet \(\text{pump}\) of the gas station example (left) and an unfolding \(\text{pump}^\beta\) (right) by assigning each variable a concrete value.

Fig. 7. A distributed run with data.
We now introduce some basic concepts from the theory of abstract data types. For \( \Sigma \) will first introduce some basic notions (concerning the representation of data) dependencies of Sect. 3 we introduce data on the level of runs. Further, each event is annotated with the ideas of Algebraic Petri nets: each condition of an oclet is annotated with a \( \Sigma \)-term that specifies which values may hold at this condition (or in Petri net terms, which values may be on the place). Each function symbol as follows: for all \( c : s \in F, c \in T(\Sigma)_s \); for all \( s \in S, V_s \subseteq T(\Sigma)_s ; \) if \( f : s_1 \ldots s_n \rightarrow s \in F \) and for \( 1 \leq i \leq n, t_i \in T(\Sigma)_s ; \) then \( f(t_1, \ldots , t_n) \in T(\Sigma)_s \). Finally, \( T(\Sigma) = \bigcup_{s \in S} T(\Sigma)_s \). \( V(t) \) denotes all variables occurring in term \( t \).

A \( \Sigma \)-algebra \( A = (S_A, F_A) \) interprets sorts as domains \( s_A \in S_A \) if \( s \in S \), and function symbols as functions \( f_A : s_{1,A} \ldots s_{n,A} \rightarrow s_{0,A} \in F_A \) if \( f : s_1 \ldots s_n \rightarrow s_0 \in F \); if \( n = 0 \), then \( f_A \) is a constant. An assignment \( \beta : V \rightarrow \bigcup S_A \) (for \( A \)) maps each \( v \in V_s \) to a value \( \beta(v) \in s_A \). \( \beta \) lifts to an evaluation of \( \Sigma \)-terms \( \text{val}^\beta : T(\Sigma) \rightarrow \bigcup S_A \) as follows: for all \( c \in T(\Sigma) \), \( \text{val}^\beta(c) = c_A \); for all \( v \in V \), \( \text{val}^\beta(v) = \beta(v) \); for \( t = f(t_1, \ldots , t_n) \in T(\Sigma) \), \( \text{val}^\beta(t) = f_A(\text{val}^\beta(t_1), \ldots , \text{val}^\beta(t_n)) \).

We consider Boolean signatures containing sort \( \text{Bool} \) and symbols =, \( \neg \), \( \lor \), \( \land \), \( \Rightarrow \), \( \text{True} \), and \( \text{False} \), and their usual interpretation. For the rest of this section, let \( \Sigma \) be an arbitrary, but fixed Boolean signature.

4.3 Syntax and Semantics of \( \Sigma \)-oclets

Syntax. As sketched in Sect. 4.1, we introduce data into oclets by following the ideas of Algebraic Petri nets: each condition of an oclet is annotated with a \( \Sigma \)-term that specifies which values may hold at this condition (or in Petri net terms, which values may be on the place). Each such \( \Sigma \)-term that specifies which values may hold at this condition (or in Petri net terms, which values may be on the place) is enabled in run \( \pi \) of Fig. 7, i.e., oclet \( \text{pump} \) is enabled for \( \text{pump} \Rightarrow 17 \), etc. We can continue \( \pi \) by appending the contribution of \( \text{pump} \) which updates the internal record of the \( \text{pump} \). Now \( \text{pump} \) is enabled for the assignment \( \text{todo} \Rightarrow 2 \). Thus, \( \text{pump} \) can occur two more times, i.e., until the assignment \( \text{todo} \Rightarrow 0 \) violates the guard expression.

This notion of data now allows to express data-dependent behavior: event \( \text{pump} \) repeats as often as specified by \( \text{todo} \). Further, we are now able to distinguish different instances of a component. Unlike in run \( \pi_2 \) of Fig. 4, we can now distinguish the two pumps run in \( \pi \) of Fig. 7. This permits to activate exactly the pump that was chosen by the customer.

We continue with the formal description of \( \Sigma \)-oclets. Similar to Sect. 3, we will first introduce some basic notions (concerning the representation of data) and afterwards formally describe the syntax and semantics of \( \Sigma \)-oclets.
Definition 12 (Σ-run). A Σ-run π = (B, E, F, ℓ, A, τ) is an abstract partially ordered run (B, E, F, ℓ, A) together with a term-assignment τ : X_π → T(Σ) s.t. τ(ℓ) ∈ T(Σ)_Bott for all ℓ ∈ E. We write V(π) for the set of all variables occurring in terms in π.

Each condition b of a run π implicitly gets sort s by its term τ(b) ∈ T(Σ)_s. All notions on runs canonically lift to Σ runs by preserving terms, e.g., prefixes, refinement. Σ-oclets follow canonically; a specification additionally fixes an interpretation of Σ.

Definition 13 (Σ-oclet). A Σ-oclet o = (π, pre) consists of a Σ-run π and a prefix pre ⊆ π of π.

Fig. 6 (left) shows the Σ-oclet pump. A node’s term is written below its label; the particular example uses tuples to express complex data at conditions. In all Σ-oclets, we omit the trivial guard True. The initial run of a Σ-specification contains no variables (the run is unique) and no guards (the run is always well-defined).

Definition 14 (Σ-specification). A Σ-specification Ω = (O, π₀, Α) consists of a finite set O of Σ-oclets, a Σ-run π₀ where V(π₀) = ∅ and τ(ℓ) = True for all ℓ ∈ E_π₀, and a Σ-algebra Α.

Semantics. The semantics of Σ-oclets follows the principles of Petri nets. An oclet o is enabled in a run π (w.r.t. an assignment β of o’s variables) if the evaluation of o’s precondition (replace all terms by their evaluation) occurs at the end of π, and the guards at o’s events evaluate to true. If o is enabled, append the evaluation of o’s contribution to π. The formal definition is slightly different: we unfold each Σ-oclet into a set of classical oclets, one oclet per assignment; the labels of the unfolded oclet carry the concrete values. The semantics of a Σ-specification is then the semantics of its unfolded oclets.

Definition 15 (Low-level unfolding of Σ-oclets). Let π be a Σ-run and Α be an interpretation of Σ. An assignment β satisfies π iff for each ℓ ∈ E_π, its guard τ(ℓ) evaluates to true, i.e., valβ(τ(ℓ)) = true. In this case, π unfolds to the abstract run valβ(π) = (B_π, E_π, F_π, ℓ’, A_π) where for each b ∈ B_π, ℓ’(b) = (l(b), valβ(τ(b))), and for each ℓ ∈ E_π, ℓ’(ℓ) = ℓ(ℓ).

A Σ-oclet o = (π, pre) unfolds to oclet valβ(π) = (valβ(π), valβ(pre)) for any assignment β that satisfies π (in this case, we also say that β satisfies o). The semantics of o is the set of all low-level oclets valβ(o) := {valβ(o) | β satisfies π} into which o unfolds. A specification Ω = (O, π₀, Α) unfolds to the low-level specification val(O) = (∪ o∈O valβ(o), valβ(π₀)).

A single Σ-oclet can unfold to infinitely many oclets if the domains of Α allow it. An unfolded oclet for a concrete assignment is shown in Fig. 6 (right). The low-level unfolding of a Σ-specification Ω is finite if each domain of Α is finite.

Definition 16 (Semantics of Σ-specifications). Let Ω be a Σ-specification. Ω describes the behaviors R(Ω) := R(val(Ω)) described by its unfolding. A set R of concrete runs satisfies Ω iff R satisfies val(Ω).
4.4 Operational Semantics

In principle, \(\Sigma\)-oclets gain operational semantics by the operational semantics of their unfolding. However, unfolding a \(\Sigma\)-specification into an (infinite) set of oclets seems impractical to operationalize a \(\Sigma\)-specification. A more practical approach is to directly decompose a \(\Sigma\)-oclet \(o\) into its basic \(\Sigma\)-oclets \(\tilde{o}\) by lifting Def. 9 to preserve each node’s terms.

**Definition 17 (Operational semantics).** The operational semantics of \(\Sigma\)-specification \(\Omega = (O, \pi, A)\) are the runs \(R(\Omega)\) of its basic \(\Sigma\)-specification \(\tilde{\Omega} = (\tilde{O}, \pi, A)\), i.e., \(R(\Omega) = R(\text{val}(\tilde{\Omega}))\).

While \(\tilde{\Omega}\) yields semantics based on occurrences of single events, it may introduce spurious behavior (not specified by \(\tilde{\Omega}\)). This spurious behavior arises if a basic \(\Sigma\)-oclet \(o[e]\) of \(o\) unfolds to some basic low-level oclet that is not defined by the unfolding of entire \(o\). For instance, for an oclet \(o\) with two events \(e_1, \tau(e_1) = (x > 0)\) and \(e_2, \tau(e_2) = (x < 5)\), \(e_1 \leq e_2\), assignment \(\beta(x) = 17\) permits a basic oclet \(o^\beta[e_1]\) while \(o^\beta\) is undefined because of \(\tau(e_2)\). With \(o^\beta[e_1]\), the operational semantics would “get stuck” in the middle of \(o\) after \(e_1\).

In the following, we characterize a class of \(\Sigma\)-specifications where each basic \(\Sigma\)-oclet \(o[e]\) of \(o\) unfolds by those assignments that also unfold the entire oclet \(o\).

**Definition 18 (Data-consistent).** An oclet \(o\) is data-consistent (w.r.t. a \(\Sigma\)-algebra \(A\)) iff for each event \(e \in E_o\) and any assignment \(\beta : V(o[e]) \rightarrow S_A\) holds: \(\beta\) satisfies \(o[e]\) iff there ex. \(\beta^+ : V(o) \rightarrow S_A\) that satisfies \(o\) and \(\beta^+|_{V(o[e])} = \beta\). A \(\Sigma\)-specification \(\Omega\) is data-consistent iff each \(o \in O_\Omega\) is data-consistent w.r.t. \(A_\Omega\).

The basic \(\Sigma\)-oclets of a data-consistent specification \(\Omega\) yield exactly the operational semantics of \(\tilde{\Omega}\).

**Lemma 2.** An oclet \(o\) is data-consistent (w.r.t. algebra \(A\)) iff \(\{o[e]^{\beta} | o[e] \in o, o[e]^{\beta} \in \text{val}_A(o[e])\} = \{o^\beta[e] | o^\beta \in \text{val}_A(o), o^\beta[e] \in o^\beta\}\). For each \(\Sigma\)-specification \(\Omega\) holds: \(R(\text{val}(\tilde{\Omega})) \subseteq R(\text{val}(\tilde{\Omega})) = R(\tilde{\Omega})\); and \(R(\text{val}(\tilde{\Omega})) = R(\text{val}(\tilde{\Omega}))\) iff \(\Omega\) is data-consistent.

Lemma 2 holds immediately by Def. 18 and the respective definitions of oclet semantics. Next, we characterize data-consistent specifications in terms of their structure and guards. In a data-consistent specification, two distant nodes only carry variables that also occur in their joint predecessors, and guards of two events are not contradictory.

**Definition 19 (Variable-consistent).** Let \(o\) be a \(\Sigma\)-oclet. Two nodes \(x, y \in X_o\) are variable-consistent iff \(V(\tau(x)) \cap V(\tau(y)) \subseteq V(\tau(x_{\leq o} \cap y_{\leq o}))\). Oclet \(o\) is variable-consistent iff all nodes of \(o\)’s contribution are pair-wise variable-consistent.

**Definition 20 (Guard-consistent).** Let \(o\) be a \(\Sigma\)-oclet and let \(A\) be a \(\Sigma\)-algebra. An assignment \(\beta\) is feasible for an event \(e \in E_o\) iff \(\beta\) satisfies the guard of each pre-event of \(f\), i.e., \(\forall f \in E_o \cap e_{\leq o} \setminus \{e\} : \text{val}_A^\beta(\tau(f)) = \text{true}\). Two events
e, f ∈ E_o are guard-consistent iff \( f \not\leq_o e \) implies that for all assignments \( \beta \) that are feasible for \( e \) and \( f \), \( \text{val}_A^\beta(\tau(e) \Rightarrow \tau(f)) = \text{true} \). An oclet \( o \) is guard-consistent iff all events of \( o \)’s contribution are pair-wise guard-consistent.

**Theorem 3.** Let \( \Omega \) be a \( \Sigma \)-specification. If each oclet \( o \in O_\Omega \) is variable-consistent and guard-consistent, then \( \Omega \) is data-consistent.

**Proof (Sketch).** In a guard-consistent oclet \( o \), a guard \( \tau(e) \) is constrained by the guards of \( e \)’s predecessors (by a feasible binding, \( \tau(e) \) holds only true if the predecessors’ guards do). Further, if \( \tau(e) \) holds, then the guard \( \tau(f) \) of any other event \( f \) that \( e \) does not depend on also holds true. Thus, an assignment \( \beta \) that satisfies \( o[e] (\tau(e) \) and all predecessor guards) can be extended to an assignment \( \beta^+ \) to satisfy \( o \) (all guards of \( o \)). That the extension is indeed possible (and does not require contradicting assignments of variables), follows from \( o \) being variable-consistent. \( \square \)

Theorem 3 decomposes data-consistency into smaller properties that are easier to check; though the help of a theorem prover may be needed to prove the implications stated in Def. 20.

Lemma 2 obviously allows to implement operational semantics for data-consistent \( \Sigma \)-specifications. The crucial part is to check whether a basic oclet \( o[e] = (\pi, \text{pre}) \) is enabled in a run \( \pi \). First, find a sub-net \( \rho \subseteq \pi \) s.t. \( \rho \) refines \( \text{pre} \) when ignoring all terms of \( o \). Then assign each variable \( v \) a value s.t. each term \( \tau(b) \) of a condition evaluates to the value \( a, \ell(\pi(b) = (p, a) \) in \( \pi \) and \( \tau(e) \) evaluates to true. If no such assignment exists, then \( o \) is not enabled. The occurrence of \( o[e] \) is straightforward with the chosen assignment.

### 4.5 Complete Gas Pump Example

Fig. 8 shows the complete set \( O \) of \( \Sigma \)-oclets for the gas pump example including its initial run \( \text{init} \). The signature \( \Sigma \) for the \( \Sigma \)-specification \( \Omega = (O, \text{init}, A) \) is Boolean and also contains the theory of integers with its usual interpretation, and additionally defines constant symbols \( \text{Price}, 6, 17 \) of sort integer. Further, it has a sort containing the constant symbols \( \text{Free}, \text{Active} \) and \( \text{Running} \). The variables are \( V = V_{\text{Nat}} = \{ \text{change}, \text{cid}, \text{todo}, \text{pid}, \text{left} \} \). The initial run \( \text{init} \) specifies an environment (\( \text{Env} \)), the operator, and two inactive pumps with ids 6 and 17.

Oclets \( \text{prepay} \) (when the customer pays in advance) and \( \text{pay later} \) (when the customer pays at the end) describe the system behavior at the most abstract level. All other oclets but \( \text{env} \) justify a refinement of \( \text{main} \). The specification is data-consistent, and the run of Fig. 7 is a run of this specification.

Oclet \( \text{env} \) describes the arrival and leaving of a new customer; technically a new instance of \( \text{Customer} \) is created with id \( (\text{cid}) \), the amount the customer wants to refill \( (\text{todo}) \), the id of the pump he wants to use \( (\text{pid}) \) and his payment \( ($) \). Only customers who can pay their desired amount of gas may participate. A customer instance is destroyed after the customer received his change.

When the customer paid or ordered his amount, the operator activates the pump (see \( \text{prepay} \) and \( \text{pay later} \). The \( \text{pid} \) assures to activate the pump that the
customer wants to use. The amount that has to be pumped is passed to the pump.

Afterwards, the pump can be started by the customer (see get gas). Then, the pump starts pumping (see pump). The customer may stop at any time, at the latest when all of the amount requested by him was pumped. When the pump finished, the actual amount that was pumped is passed to the operator. A customer who did not pay in the beginning can pay now (see pay later). Then, the change is calculated (see prepay and pay later). Note that because $cid$, $pid$, todo and $ occur in prepay's precondition, it is not necessary that the operator continuously carries any of these values up to the occurrence of change. Thus, he may start serving a second customer before the first one gets his change back.

Between starting the pump and leaving the gas station, the customer can get one free snack. Oulet snack can occur only once as an occurrence of get free snack after start prevents oclet snack from being enabled.
Tool support. The approach presented here is implemented in the Eclipse-based tool Greta [14]. Greta provides a graphical editor for oclets and animated execution via a simulation engine which implements the operational semantics of oclets including abstract dependencies and $\Sigma$-oclets. Greta finds variable assignments and evaluates terms using the simulation engine of CPN Tools via Access/CPN [40]. Additionally, Greta allows to verify whether a Petri net implements a given specification, and to synthesize a minimal labeled Petri net that implements a specification—both techniques operate on McMillan-prefixes for oclets [13]. Greta is available at www.service-technology.org/greta.

5 On the Relation of Oclets to Petri Nets

This section discusses the relation between oclets and Petri nets. As above, oclets without abstraction and data representation are called “classical.”

Classical oclets vs. P/T nets. Classical oclets contain P/T-nets: for each P/T net $N$ exists a specification $\Omega$ s.t. their behaviors are identical: $R(\hat{\Omega}) = R(N)$ [12]. The converse does not hold. Even classical oclets can mimic a Turing machine by their preconditions [13]. However, if $\Omega$ is bounded (i.e., there exists a $k$ s.t. no run $\pi \in R(\hat{\Omega})$ of $\Omega$ has more than $k$ maximal conditions with the same label), then there is a labeled Petri net $N$ with the same behavior: $R(N) = R(\hat{\Omega})$. $N$ can be synthesized automatically from $\hat{\Omega}$ by first building a McMillan-prefix [11] for $\Omega$ that finitely represents $R(\hat{\Omega})$ and then folding that prefix to $N$ [13].

$\Sigma$-oclets vs. P/T nets. $\Sigma$-oclets extend classical oclets. Clearly, a $\Sigma$-specification $\Omega$ with an infinite domain has no equivalent finite P/T net. If $\Omega$ has only finite domains, it unfolds to a finite low-level specification $\text{val}(\Omega)$. If abstract dependencies in $\text{val}(\Omega)$ can be refined s.t. they only occur in pre-conditions, then semantics of classical oclets carries over (Thm. 2). Thus not every $\Sigma$-specification has a P/T net with the same behavior. Yet it seems plausible that every bounded $\Sigma$-specification with finite domains has a net with the same behavior: finitely many oclets will allow to continue markings of finite size only in finitely many ways. This suggests that the synthesis from oclets [13] can also be generalized to $\Sigma$-oclets. If abstract dependencies in $\Omega$ cannot be refined, the synthesis of a P/T net also has to find a refinement of the abstract dependencies.

$\Sigma$-oclets vs. Algebraic Petri nets. As synthesizing Algebraic Petri nets from $\Sigma$-oclets is out of the scope of this paper, we just sketch some basic observations here. First, every Algebraic net $N$ with term-inscribed arcs and term-inscribed transition guards has an equivalent $\Sigma$-specification: translate each transition and its pre- and post-places to a basic oclet by moving arc inscriptions to the respective pre- and post-condition. The reverse direction is more difficult. Term annotations are not an issue as both models are based on the same concepts. But enabledness of an event of a $\Sigma$-oclet depends on a “history” in the execution while enabledness of a transition depends on the current marking only. Hee et al. [22] have shown how to express history of tokens in a data-structure of an Algebraic Petri net, and how to use them in guards. Whether all data-dependencies expressible in
Data and Abstraction for Scenario-Based Modeling with Petri Nets

Σ-oclets can be expressed this way is an open question. Yet, token histories of [22] have drawbacks regarding analysis. Thus, synthesizing a net without an explicit and complete recording of token histories is an interesting, open problem as well. In any case, the signature Σ of a specification has to be extended to allow remembering behavior in the past.

6 Conclusion and Future Work

In this paper, we proposed a formal model for scenarios that combines the advantages of HMSCs (simple semantics by composition of scenarios) with the advantages of LSCs (intuitive notion of scenarios with local preconditions and a flexible style of specifying systems). Our model called oclets is based on Petri nets and their distributed runs. The basic semantic notions of composing LSC-style scenarios to distributed runs have been proposed earlier; in this paper we have shown how the basic model of oclets can be lifted to the expressive means of LSCs by introducing a notion of abstract dependencies and adopted concepts from Algebraic Petri nets to represent data. All of our extensions are constructed such that (1) they generalize and embed the “classical” oclet concept and (2) their semantics can be described in terms of classical oclets. Existing operational semantics for oclets canonically lift to our extended model. Oclets deviate from LSCs where their semantics turns problematic for the aim of deriving specified behavior by composing scenarios (see Sect. 2). Our approach is implemented in the tool Greta and was validated on a number of elaborate examples. Finally, composition, decomposition, abstraction, refinement, and unfolding suggest oclets to be an interesting model for a calculus of scenarios. The contribution of this model of scenarios becomes obvious when considering existing works that relate scenario-based techniques to Petri nets.

From Scenarios to Petri nets. To bridge the gap between scenario-based specifications and Petri nets, several approaches have been proposed. Methods to transform UML sequence diagrams to Coloured Petri nets (CPNs) are described in [5], [16], and [41]; the latter approach is used in [15] to model a variant of the gas station example used in this paper. Further, there exist approaches to provide Petri net semantics for MSCs [25, 29] and to synthesize a Petri net out of a MSC specification [36]. While these approaches are straight-forward, the scenario languages lack expressive power (MSCs, UML sequence diagrams) or tend to yield complex specifications in practice (HMSCs, see Sect. 2).

Approaches to transform a LSC specification to a CPN have been described in [1] and [28]. As two LSCs of a specification may be contradictory [20], both synthesis approaches need to generate the state space of the LSC and the synthesized CPN to check equivalency of both models. Also operational semantics of LSCs [20] requires model checking to find a correct play-out step. We have shown in this paper that Σ-oclets do not require model checking for operational semantics. The gives rise to the hope that components can be synthesized from Σ-oclets as correct by construction (as in the case of HMSCs).
To the best of our knowledge, no other Petri net-based model for scenarios features data and abstract causal dependencies. The notion of history-dependent behavior in Petri nets was introduced in [22]. Oclets particularly relate to Token History Petri nets where each token records its “traveling” through the net; LTL-past guards at transitions restrict enabledness to tokens with a particular history. The scenario-based approach of oclets provides a graphical syntax for a subclass of these guards. Particularly, Token History Petri nets might allow to synthesize components from an oclet specification. There are numerous refinement and abstraction techniques for Petri net system models, e.g., by modular refinement [39] or using rules [3]. Refinement of actions of a distributed run has been studied in [18]; we think these results can be lifted to refine abstract dependencies in oclets in a systematic way.

Future work. A next step for research work is to develop symbolic semantics for $\Sigma$-oclets allowing to concisely describe infinitely many behaviors. This should support solving the main challenge of scenarios: to synthesize high level Petri net components from a $\Sigma$-specification without unfolding into a concrete low-level model.

References

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2009):

09/01 Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf
Compositional Service Trees

09/02 P.J.J. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin
Queue merge: a Binary Operator for Modeling Queueing Behavior

09/03 Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemsse
Breadth-Bounded Model Checking

09/04 Muhammad Atif and MohammadReza Mousavi
Formal Specification and Analysis of Accelerated Heartbeat Protocols

09/05 Michael Franssen
Placeholder Calculus for First-Order logic

09/06 Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle
POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy

09/07 Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Piegl and Hannes Werthner
Pattern-based Analysis of Windows Workflow

09/08 Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Swift mode changes in memory constrained real-time systems

09/09 Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko
Behavioural analysis of an FC Linux Driver

09/10 Ugur Keskin
In-Vehicle Communication Networks: A Literature Survey

09/11 Bas Ploeger
Analysis of ACS using mCRL2

09/12 Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote
Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic

09/13 Luca Aceto, Anna Ingolfsson, MohammadReza Mousavi and Michel A. Reniers
A Rule Format for Unit Elements

09/14 Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst
Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance

09/15 MohammadReza Mousavi and Emil Sekerinski, Editors
Proceedings of Formal Methods 2009 Doctoral Symposium

09/16 Muhammad Atif
Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems

09/17 Jeroen Keiren and Tim A.C. Willemsse
Bisimulation Minimisations for Boolean Equation Systems

09/18 Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran
On-the-fly Auditing of Business Processes

10/01 Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi
Analytical Software Design: Introduction and Industrial Experience Report

10/02 F.E.J. Kruseman Aretz
Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study
On Rule Formats for Zero and Unit Elements

Towards Model-Based Testing of Electronic Funds Transfer Systems

Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited

Locally unique labeling of model elements for state-based model differences

Converting existing analysis to the EDP resource model

Reconstruction and verification of group membership protocols

A linear translation from LTL to the first-order modal $\mu$-calculus

Extending an Open-source Real-time Operating System with Hierarchical Scheduling

1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)

3rd International Software Language Engineering Conference

Discrimination Aware Decision Tree Learning

Specification Guidelines to avoid the State Space Explosion Problem

GEM: a Distributed Goal Evaluation Algorithm for Trust Management

Rule Formats for Distributivity

Decompositional Reasoning about the History of Parallel Processes

Robustness of Behavioral Equivalence on Open Terms

Desynchronisability of (partial) closed loop systems

Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!

Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification

Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols

Benefits of Applying Formal Methods to Industrial Control Software

Semantics, bisimulation and congruence results for a general stochastic process operator

Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence

Transforming SOS Specifications to Linear Processes

A Component Framework where Port Compatibility Implies Weak Termination

Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol
11/10 Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi and Tim A.C. Willemse
Synchronizing Asynchronous Conformance Testing

11/11 Jeroen J.A. Keiren and Michel A. Reniers
Type checking mCRL2

11/12 Muhammad Atif, Mohammad Reza Mousavi and Ammar Osaiweran
Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems

11/13 J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius
Experience report on developing the Front-end Client unit under the control of formal methods

11/14 J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius
Analyzing a Controller of a Power Distribution Unit Using Formal Methods

11/15 John Businge, Alexander Serebrenik and Mark van den Brand
Eclipse API Usage: The Good and The Bad

Investigating the Effects of Designing Control Software using Push and Poll Strategies

11/17 M.F. van Amstel, A. Serebrenik and M.G.J. van den Brand
Visualizing Traceability in Model Transformation Compositions

11/18 F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber
Dogfooding the Structural Operational Semantics of mCRL2

12/01 S. Cranen
Model checking the FlexRay startup phase

12/02 U. Khadim and P.J.L. Cuijpers
Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP

12/03 M.M.J.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher
Revised budget allocations for fixed-priority-scheduled periodic resources

12/04 Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever
Experience Report on Designing and Developing Control Components using Formal Methods

12/05 Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse
A cure for stuttering parity games

12/06 A.P. van der Meer
CIF MSOS type system

12/07 Dirk Fahland and Robert Prüfer
Data and Abstraction for Scenario-Based Modeling with Petri Nets