Handbook of HERAKLIT

modeling – composing – abstracting – verifying

Peter Fettke[1,2][0000−0002−0624−4431] and Wolfgang Reisig[3][0000−0002−7026−2810]

[1] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
peter.fettke@dfki.de
[2] Saarland University, Saarbrücken, Germany
[3] Humboldt-Universität zu Berlin, Germany
reisig@informatik.hu-berlin.de

Overview

This handbook presents the formal foundations of HERAKLIT, along with extensive text that motivates the concepts of HERAKLIT. A consistent example explains the formal concepts, and shows the intended way of using them.

This handbook is not a textbook, but emphasizes the embedding of HERAKLIT in the scientific context. For example, the description of static and dynamic structures is motivated as a dynamic extension of predicate logic.

Alongside this handbook are the extensive HERAKLIT case studies [2–7]. They demonstrate that a user does not necessarily need to know the formal concepts of this handbook in depth. The intuitive notations and representations of HERAKLIT or the text HERAKLIT – *in a nutshell* [8] are sufficient for understanding.

Foreword

Modeling of computer-integrated systems: current state

The manufacturing of a complex product or system is always preceded by a planning process in which the structure, operation, intended effects, et cetera, of the product are formulated. Here, central tools and aids are *models*. A given complex system can also be better understood and analyzed by modeling it. A model emphasizes some aspects of the system, often with graphical, standardized representations. Commonly known types include models in urban planning or architecture, but electrical circuits, chemical structures, et cetera are also formally modeled. A good model is not only more descriptive and easier to understand than a colloquial representation; it can also be used in many ways: one can identify functionality and performance bottlenecks, estimate costs of a system, prove correctness against a specification, optimize parameters, and much more.

cite as: FETTKE, P.; REISIG, W.: *Handbook of* HERAKLIT. 2021. – HERAKLIT working paper, v1.1, September 10, 2021, http://www.heraklit.org

Compared to other engineering disciplines, models are not well used in informatics and business informatics. Models are not considered too helpful. An example is diagrams of the *Unified Modeling Language* (UML) for software systems. The various diagram types illustrate some aspects of a software system, but provide little support for analysis questions. Other example are the *Business Process Modeling Notation* (BPMN) or *event-driven process chain* (EPC) diagrams popular in business informatics. Such diagrams are limited to identifying abstract activities and representing flow of control. They support abstraction and composition, but do not help much to deal with concrete or abstract data and objects, to describe behavior, or to prove the correctness of behavioral models against a specification. The *Architecture of Integrated Information Systems* (ARIS) does allow for an integrated description of data, functions, and processes. However, powerful modularization concepts are lacking, and ARIS models are only partially formalized. As a theoretically sound method, a Petri net models distributed control flow, and higher-level nets also model data. But structuring and abstracting principles for modeling complex systems are lacking so far.

For small systems, the above-mentioned modeling techniques are more or less helpful. But where modeling is especially needed, for really large systems, there is a lock of comprehensive concepts. Such systems are modeled in parts at best, often with completely different modeling techniques that only fit together with difficulty. Currently available modeling methods are difficult to combine; there are no comprehensive model views into which completely different submodels can then be integrated.

What is needed

Modern and future digital systems, cyber-physical systems, digital information-based infrastructures, the *Internet of people, things and services*, et cetera, tend to be increasingly complex. Furthermore, they converge to dense networks of components of different types. Consequently, there are multiple challenges to overcome in the development of computer-integrated systems. What is needed are modeling techniques that are truly useful, especially for large systems. A number of aspects are particularly important for this:

- *Composition and hierarchical refinement of local components:* What does it mean conceptually that computer-integrated systems are "large"? It means, first of all, that some concepts adequate for modeling "small " systems are no longer useful. This is most obviously true for global states and steps. And it means that a large system is generally not monolithic or uniform in design, but is composed in some way of smaller systems. *Composition* and *refinement* are thus fundamental concepts for modeling large systems.
- *Models from the user's perspective:* There are several ways to conceptualize informatics in general and modeling in informatics in particular. One approach often used emerges from theoretical informatics, starting with alphabets, words, and formal languages, and their transformation. Likewise popular is the approach from the technical side, with digital circuits and

their abstraction, to software, databases, et cetera. Particularly useful for understanding large systems is the approach from the point of view of users, operators, and the purposes for which large systems are operated.
– *Systematic transfer of informal, intuitive ideas about a given or intended system into a formal model*: Informatics provides structured concepts for this purpose.
– *Parametrized, schematic modeling:* Often, it is not only a single system that is to be modeled, but a set of similarly structured and behaviorally similar models.
– *Integrated, unified, peer-to-peer handling of digitized and human-based processes:* An example is automated digital human resource management, together with the instruction of an employee about how to handle dangerous goods, and his confirmation by signature.

How Heraklit fulfills the requirements

Heraklit supports modeling of systems that are composed of subsystems and form hierarchies in which concrete products, machines, people, documents, data stores, et cetera are composed, transformed, computed, processed, transported, created, or deleted. Heraklit models can be abstract, detailed, or schematic; data, objects, and steps describing dynamic behavior can be formulated in detail, but also abstractly.

Concrete and abstract data structures, as well as behavior, are modeled in an integrated and interrelated way. Heraklit offers coordinated means of expression for this, which support the modeler in the formulation of their ideas. The modeler is not restricted; they have complete freedom in terms of content in choosing the degree of abstraction and the type of composition of modules; Heraklit, however, suggests conceptually unifying representations. Heraklit integrates new concepts with proven, deeply motivated concepts of modeling software, business processes and other systems. In addition, Heraklit provides a concept for abstracting concrete systems and thus allows one to symbolically model a large class of systems that are similar in structure and behavior.

With its few but expressive and integrated concepts, Heraklit models are:

– more manageable for the developer,
– easy to understand for the user,
– less error-prone and easier to verify,
– simpler to change, and
– faster to produce and less expensive than other methods, especially for very large systems.

From a technical point of view:

– Heraklit supports hierarchical partitioning of a large system into modules;
– one can use Heraklit to compose modules into large systems in a technically simple, but flexible and expressive way;

- Heraklit describes discrete steps in large systems only locally in single modules on freely chosen levels of detail;
- Heraklit represents all kinds of modules with the same concepts, whether the modules are intended for implementation or not;
- Heraklit represents data at freely chosen abstraction levels and considers data dependencies in the control flow;
- Heraklit can abstract from concrete data to capture in a schema behaviorally similar instantiations;
- with Heraklit it is possible to generate models that are scalable, systematically changeable, and extensible; and
- Heraklit supports proving that a model has desired properties.

Heraklit uses proven mathematically-based and intuitively easy-to-understand concepts (abstract data types and Petri nets) that have also been used for decades to specify systems; Heraklit recombines them and adds the composition calculus.

As an example throughout this handbook, when presenting the central Heraklit concepts, the business process of settling a claim with a car insurance company is modeled.

Part I: the basic Heraklit constructs

In the design of integrated, digital systems, Heraklit distinguishes three basic aspects ("dimensions"): the *architectural*, the *static*, and the *dynamic* dimension. Initially, they independently address specific issues and methods of designing and analyzing systems. Then, in the framework of a Heraklit *module*, they form an integrated concept.

In this first part of the handbook, we present the three Heraklit dimensions and motivate the basic notions associated with them.

1 Architecture: modules

First, we explain the basic idea of the module concept of Heraklit. Then we consider interfaces of modules in their general form as labeled and ordered finite sets. This is followed by the central notion of a module, again in its most general form: A module is a graph with interfaces. Interfaces have interesting properties, whose description will fill a full section; they then play a decisive role in the composition of modules.

1.1 The basic idea

We begin with the obvious observation that a "large" system S is generally composed of interrelated subsystems $S_1, \ldots, S_n$. One would denote the system S in the form

$$S = S_1 \bullet \cdots \bullet S_n, \tag{1}$$

where "$\bullet$" is a composition operator for subsystems. Typical examples of notation (1) are models for supply chains, machines in a production line of a manufacturing plant, assembly stations, et cetera, arranged in a sequence.

Heraklit refers to such systems and subsystems as *modules*. However, Heraklit uses the notation (1) not only for obvious sequences of subsystems, but also in cases where such a sequence is not relevant.

We begin the example of the business process of a car insurance company by modeling four individual modules A, B, F, and V in figure 1a. Without going into too much detail here, the chosen inscriptions show, sufficiently precisely, the function of each module. Figure 1b composes the modules A and V, written as $A \bullet V$. All four modules are composed in figure 1c as $A \bullet V \bullet B \bullet F$.

Besides composing (sub)systems, *refining* systems is a key concept for structuring large systems. Figure 2a refines the module V into a sequence $C \bullet D \bullet E$; figure 2b embeds this refinement into figure 1c, resulting in the composition $A \bullet B \bullet C \bullet D \bullet E \bullet F$. Conversely, one can also abstract a given system. As an example, figure 2c abstracts the entire accident into a single module.

In the linear notation of composed modules of the form $S_1 \bullet \cdots \bullet S_n$, each module S_i has a *left* and a *right* neighbor (but S_1 has no left neighbor and S_n has no right neighbor). Heraklit uses this property and forms *two* interfaces for each module M, the *left* and *right* interface of M, written ${}^*M$ and M^*. The composition $M \bullet N$ only uses M^* and ${}^*N$.

As in figures 1 and 2, a module is represented graphically as a rectangle, with two main components:

1. its left and right *interface*: each of the two interfaces consists of a set of ordered *gates*; each gate has a *label*, or inscription; and
2. its *interior*: this can be quite arbitrary in design; three variants are particularly common. The interior:
   - consists only of the name of the module as in figures 1a and 2c, or
   - in turn consists of modules, possibly composed as in figures 1b and 2a, or
   - describes dynamic behavior (details are given in chapter 3).

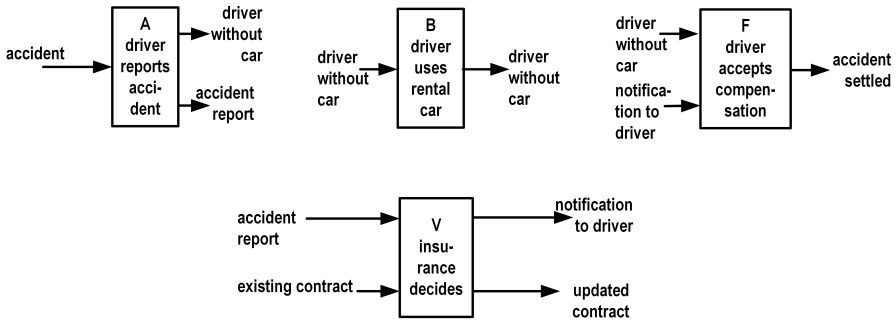
1.2 Interfaces

A module has two interfaces; each is a labeled and ordered set of nodes:

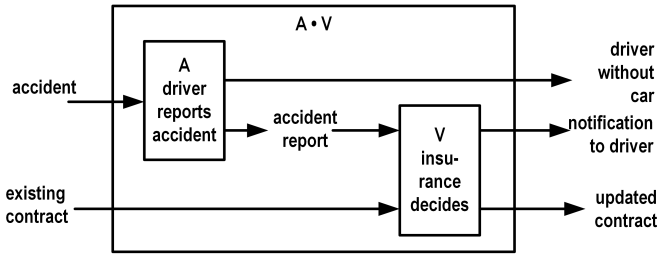
Definition 1 (labeling). *Let M and Λ be sets. A* labeling of M over Λ *assigns a label from Λ to each element of M.*

Often Λ consists of symbols. A labeling may well assign the same label from Λ to different elements of M.

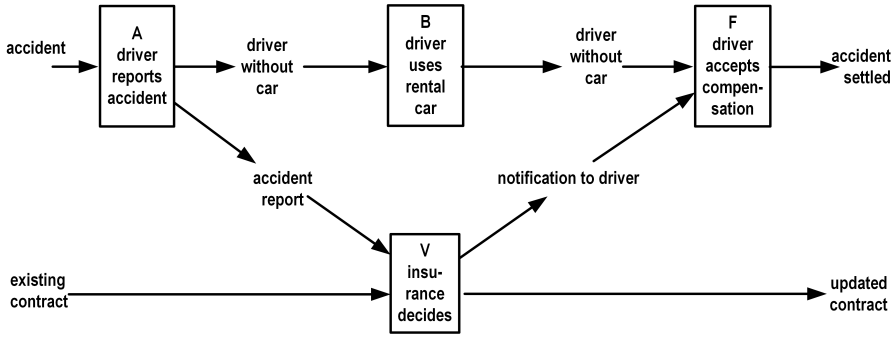
We often use *ordered* sets. Order is often defined as a total, reflexive relation. In contrast, we mainly use partial, irreflexive (strict) orders and therefore define:



(a) four individual modules



(b) the composition $A \bullet V$ of the two modules A and V from figure 1a

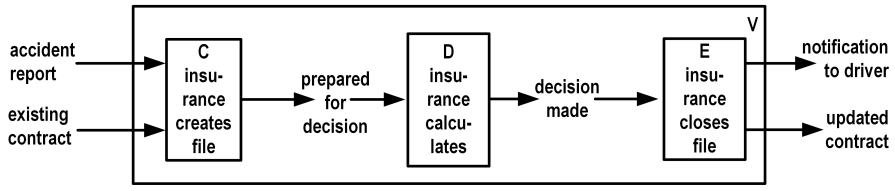


(c) the composition $A \bullet V \bullet B \bullet F$

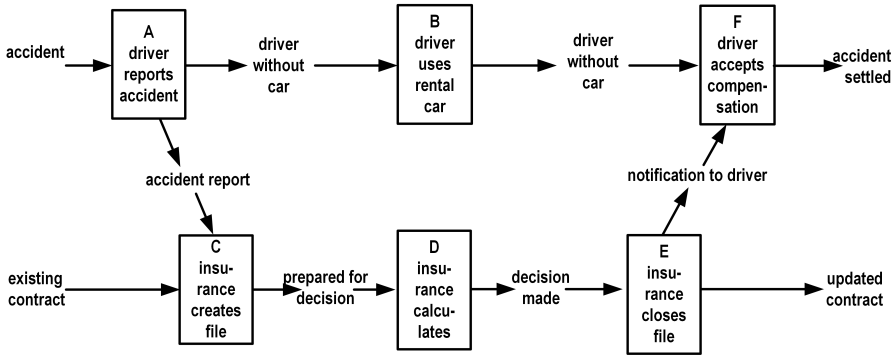
Fig. 1: Composition of modules

Definition 2 (order). *Let M be a set.*

1. *A relation $< \subseteq M \times M$ is an* order on M *if $<$ is transitive and irreflexive, that is, if for all $a, b, c \in M$ it holds that:*
   (a) if $a < b$ and $b < c$, then also $a < c$;



(a) the composition $C \bullet D \bullet E$



(b) the composition $A \bullet B \bullet C \bullet D \bullet E \bullet F$

(c) the entire accident model abstracted as one module

Fig. 2: zooming in and out

(b) not $a < a$.

2. *An order* $<$ *is* total *if in addition, for all* $a, b \in M$*, it holds that: either* $a < b$ *or* $b < a$ *or* $a = b$.

With this we can now define:

Definition 3 (interface). *Let* Λ *be a set of symbols. An* interface over Λ *is a finite, labeled over* Λ*, and totally ordered set.*

The order of differently labeled elements of an interface is irrelevant; *equivalent* interfaces differ only in this aspect.

Definition 4 (equivalence of interfaces). *Let* R *and* S *be interfaces.* R *and* S *are* equivalent*, written* $R \sim S$*, if:*

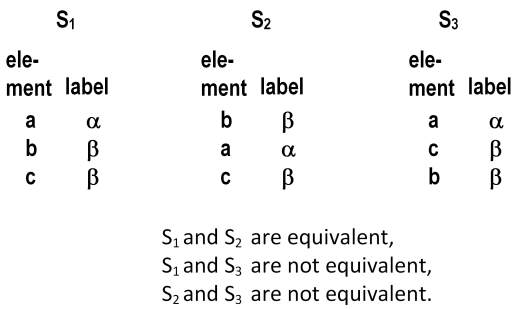


Fig. 3: equivalent and non-equivalent interfaces

- *R and S contain the same labeled elements;*
- *elements with the same label are ordered equally in R and S.*

Figure 3 depicts three interfaces over the same set $\{a, b, c\}$. The elements are ordered top-down. The labels are given explicitly. Two of the interfaces are equivalent.

When multiple elements of an interface A have the same label, the order on A becomes relevant. When composing two interfaces, elements labeled identically are fused together if in addition they also have the same *index*. The index is the number of equally labeled smaller elements:

Definition 5 (index of an element of an interface). *Let R be an interface. Let l be a label, and let $r \in R$ be l-labeled in R. Let R contain n l-labeled elements preceding r in the order of R. Then $n + 1$ is the* index of r in R.

If an element occurs in different interfaces, it may have a different index in each interface. An example for this case are the elements f, g and h in the sets S and T in figure 4.

Lemma 1 (indices are compatible with equivalence). *Let R and R' be two interfaces. R and R' are equivalent if and only if the index of every element in R coincides with its index in R'.*

The assertion follows directly from definitions 4 and 5.

1.3 Modules

In the formal definition of modules, graphs provide the general framework; in fact, many concrete modules have the structure of graphs. We define graphs as usual:

Definition 6 (graph). *Let V be a set and let $E \subseteq V \times V$ be a relation. Then $G = (V, E)$ is a* directed graph. *V and E are the* vertices *and the* edges *of G, written V_G and E_G, respectively.*

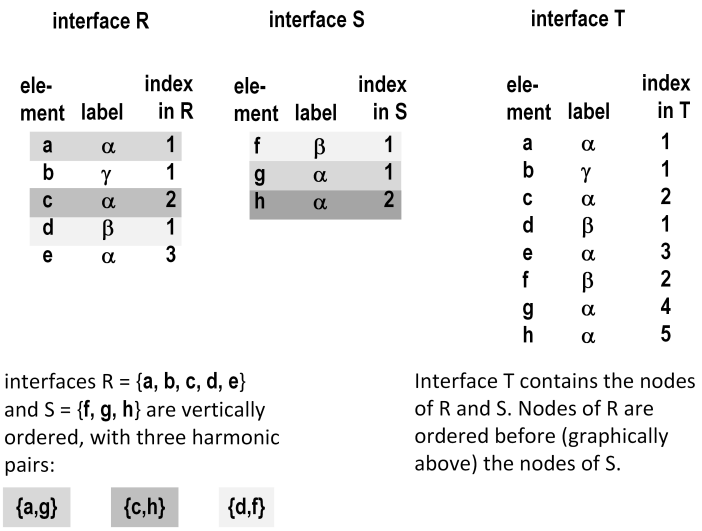


Fig. 4: harmonic pairs

As usual, we depict the nodes and edges of a graph as points and arrows between points.

We consider *undirected* graphs as a special case of directed ones, and as usual we call a graph *acyclic* if no arrow sequence forms a circle:

Definition 7 (undirected acyclic graph). *Let $G = (V, E)$ be a graph.*

1. *G is* undirected *if for every edge (a, b) in E, (b, a) is also an edge in E.*
2. *G is* acyclic *if the transitive closure E^+ of E is irreflexive (and thus an order).*

In the graphical representation of an undirected graph, the arrowheads are omitted. A finite acyclic graph is represented either horizontally with the smallest elements on the left, or vertically with the smallest elements on top.

We now have all the prerequisites to define a module as follows:

Definition 8 (module). *Let $G = (V, E)$ be a graph and let ${}^*G, G^* \subseteq V$ be two interfaces. Then G together with ${}^*G$ and G^* is a* module.

Notation 1 (components of modules). *Let G be a module.*

1. *${}^*G$ and G^* are the* left *and* right *interfaces of G, respectively; their elements are called* gates.
2. *$V_G \setminus ({}^*G \cup G^*)$ is the* interior *of G, written I_G.*
3. *"G" is the* name *of the module.*

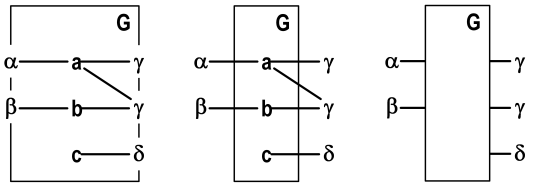


Fig. 5: variants of the representation of interfaces

We depict a module G as usual for graphs: A node is represented as a point, and an edge as an edge between points. These graphical elements are embedded into a rectangle. For individual gates of a module often only the label is interesting, not the node itself. Then only the label is depicted.

Nodes of the left and right interfaces ${}^*G$ and G^* are placed on the left and right surface of the rectangle of G, respectively. The order on ${}^*G$ and on G^*, respectively, grows from top to bottom in graphical representations.

It is not always convenient to depict the interface elements on the surface of the module rectangle. Especially in graphical representations of the composition of modules, it is occasionally convenient to detach the elements from the surface and draw them outside the rectangle, along with the attached edges. Figure 5 shows these variants for a technical example.

Graphs provide a very general framework for representing the interior of modules. This includes flowcharts, BPMN diagrams, several UML diagrams and much more. To describe dynamic behavior, Heraklit uses Petri nets with their graphical representation. A program can also be the interior of a module; this program can send *remote procedure calls* to its interfaces and receive corresponding responses. It is also possible that inside a module there are just text documents or other informal objects. In this case, the graph inside the module degenerates to a single labeled node. Examples will be given in the second part of the handbook.

The interfaces ${}^*G$ and G^* of a module G are naturally disjoint in many applications. The second part of the handbook will present examples of useful modules with non-disjoint left and right interfaces.

1.4 Properties of pairs of interfaces

The concept of *interface* is very elementary; yet pairs of interfaces have interesting properties that constitute the composition of modules.

Elements of two labeled and ordered sets are *harmonic partners*, forming a *harmonic pair*, if their labels and their indices coincide:

Definition 9 (harmonic partners and harmonic pair). *Let R and S be interfaces. Two elements $r \in R$ and $s \in S$ are* harmonic partners of R and S *if r and s have the same label and the index of r in R coincides with the index of s in S. In this case, the set $\{r, s\}$ is a* harmonic pair *of R and S.*

Clearly, each $r \in R$ and each $s \in S$ has at most *one* harmonic partner in S and R, respectively. Figure 4 shows a technical example.

Lemma 2 (harmonic partners are compatible with equivalence). *Let R, R', S and S' be interfaces. Let $R \sim R'$ and $S \sim S'$. Then $\{r, s\}$ is a harmonic pair of R and S if and only if $\{r, s\}$ is a harmonic pair of R' and S'.*

The assertion follows directly from definitions 4 and 9.

Notation 2 (label and index of a harmonic pair). *Let $\{r, s\}$ be a harmonic pair of R and S. Let l be the label and n the index of r and s. Then l and n are denoted as the* label and the index of $\{r, s\}$.

Lemma 3 (harmonic pairs). *Let R and S be interfaces; let $\{r, s\}$ be a harmonic pair of R and S, and let q be an element of R or S with the label of $\{r, s\}$, but without a harmonic partner. Then the index of $\{r, s\}$ is smaller than the index of q.*

Proof. Let l be the label and n the index of $\{r, s\}$. For $n = 1$, the proposition is trivial according to definition 9. So, let $1 <= m < n$. Then there exists an element $r_m \in R$ and an element $s_m \in S$ with label l and index m (by definition 5). Then $\{r_m, s_m\}$ is a harmonic pair of R and S (by definition 9). Then m is not the index of q (by assumption on q). □

1.5 The composition of modules

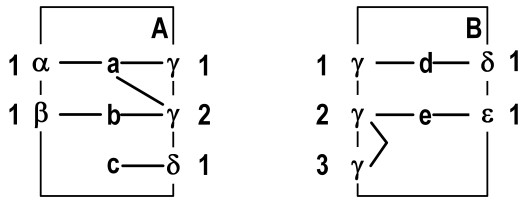
The composition $A \bullet B$ of two modules A and B is again a module. In $A \bullet B$, each element of A^* is either merged with an identically labeled element of ${}^*B$ and placed inside $A \bullet B$, or it is attached to B^*. Correspondingly, each element from ${}^*B$ is either merged with an element from A^*, or attached to ${}^*A$. Figure 1b shows an example.

Technically formulated, the interior of $A \bullet B$ consists of the interior of A, the interior of B, and the harmonic pairs of A^* and ${}^*B$. The left interface ${}^*(A \bullet B)$ of $A \bullet B$ consists of ${}^*A$ extended by those gates of ${}^*B$ which have no harmonic partners in A^*. The right interface $(A \bullet B)^*$ of $A \bullet B$ consists of B^*, extended by those gates of A^* which have no harmonic partners in ${}^*B$. The edges of $A \bullet B$ are the edges of A together with the edges of B. Here, a node of A or B "carries its edges to $A \bullet B$" if it has a harmonic partner in A^* or ${}^*B$, or if it is part of the new interface ${}^*(A \bullet B)$ or $(A \bullet B)^*$. Figure 6 shows a technical example.

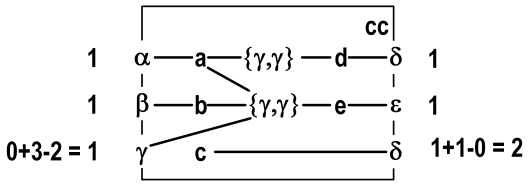
Formally, the composition of modules is now defined as follows:

Definition 10 ($A \bullet B$). *Let A and B be modules. For each node x of A or of B, let $x' = \{x, y\}$ if $\{x, y\}$ is a harmonic pair of A^* and ${}^*B$; let $x' = x$ if no such harmonic pair exists. Then the module $A \bullet B$ is defined as follows:*

1. *The nodes of $A \bullet B$:*
   Let x be node of A or of B. Then $x' \in A \bullet B$.
2. *The left interface ${}^*(A \bullet B)$ of $A \bullet B$:*



(a) Two modules A and B. The number at each gate of an interface indicates it's index in the interface.



(b) The sum at each gate is the result of the computation of $p + n - m$ as in Definition 10, 2b and 3b.

Fig. 6: technical example of the composition of modules

   (a) *Let $x \in {}^*A$ with label l and index n in ${}^*A$. Then $x' \in {}^*(A \bullet B)$ with label l and index n.*
   (b) *Let $x \in {}^*B$ without a harmonic partner in A^*, let l be the label and n the index of x in ${}^*B$, let m be the maximal index of l-labeled harmonic pairs of A^* and ${}^*B$, and let p be the maximal index of l-labeled elements of ${}^*A$. Then $x \in {}^*(A \bullet B)$ with label l and index $p + n - m$.*

3. *The right interface $(A \bullet B)^*$ of $A \bullet B$:*
   (a) *Let $x \in B^*$ with label l and index n in B^*. Then $x' \in (A \bullet B)^*$ with label l and index n.*
   (b) *Let $x \in A^*$ without a harmonic partner in ${}^*B$, let l be the label and n the index of x in A^*, let m be the maximal index of l-labeled harmonic pairs of A^* and ${}^*B$, and let p be the maximal index of l-labeled elements of B^*. Then $x \in (A \bullet B)^*$ with index $p + n - m$.*

4. *The edges of $A \bullet B$:*
   Let (x, y) be an edge of A or of B. Then (x', y') is an edge of $A \bullet B$.

Intuitively formulated, each node a of A and b of B is a node of $A \bullet B$. However, if a and b are harmonic partners of the interfaces A^* and ${}^*B$, they are "glued" in the set $\{a, b\}$. This set replaces both a and b in $A \bullet B$.

Each node a of the left interface ${}^*A$ of A is a node of the left interface ${}^*(A \bullet B)$ of $A \bullet B$. However, if a coincidently belongs to A^* and has a harmonic partner b

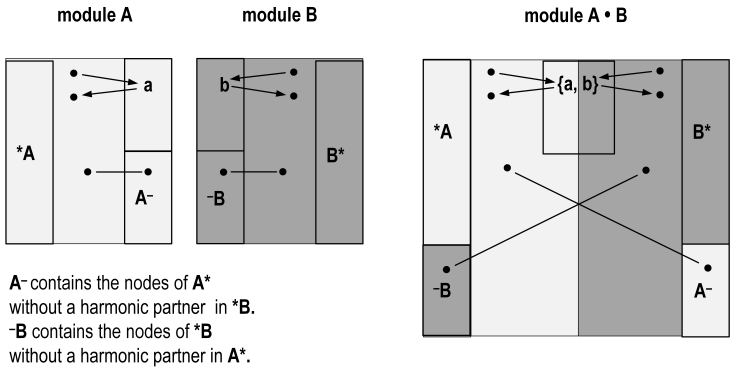


Fig. 7: sketch for composition

in ${}^*B$, then a is replaced in ${}^*(A \bullet B)$ by $\{a, b\}$. In both cases, a inherits its label and its index from ${}^*A$.

Furthermore, each node b of ${}^*B$ without a harmonic partner in A^*, belongs to ${}^*(A \bullet B)$, inheriting its label from ${}^*B$. To compute the index of b in ${}^*(A \bullet B)$, we start with the index n of b in ${}^*B$, and reduce this value by the number m of identically labeled nodes in ${}^*B$ with harmonic partners. Lemma 3 guarantees that the value of m is smaller than the value of n. In ${}^*(A \bullet B)$, the node b should be ordered behind (in graphical representations, below) the identically labeled nodes from ${}^*A$. Therefore, the index is increased by the maximal index p of such nodes from ${}^*A$.

The interface $(A \bullet B)^*$ is likewise constructed.

The edges of $A \bullet B$ are the edges of A and the edges of B. If a node x of A or B that has been replaced by a set $\{a, b\}$ in $A \bullet B$, then an edge (x, z) or (z, x) of A or B is in $A \bullet B$ replaced by $(\{x, y\}, z)$ or $(z, \{x, y\})$..

In addition to the example from figure 6, figure 7 outlines the principle of composition.

The notion of equivalence transfers from interfaces to modules:

Definition 11 (equivalence of modules). *Let A and A' be modules. A and A' are* equivalent *(written $A \sim A'$) if ${}^*A \sim {}^*A'$ and $A^* \sim A'^*$.*

Composition of modules is compatible with equivalence:

Lemma 4 (composition is compatible with equivalence). *Let A, A', B and B' be modules. Let $A \sim A'$ and $B \sim B'$. Then $A \bullet B \sim A' \bullet B'$ holds.*

The assertion follows immediately from definitions 10 and 11 and lemmas 1 to 3.

In section 1.1 we discussed that a "large" system S is generally composed of subsystems $S_1, \ldots, S_n$ in the form $S = S_1 \bullet \cdots \bullet S_n$. This bracket-free representation is justified only because the composition operator is *associative*, thus, the following theorem holds:

Theorem 1 (composition is associative). *Let A, B, and C be modules. Then it holds that:*

$$(A \bullet B) \bullet C = A \bullet (B \bullet C).$$

A proof of this theorem can be found in [9].

As already discussed, different elements of an interface may carry the same label. In particular, such interfaces may arise in the *composition* of modules. As an example, consider a machine that produces a product from given material. Figure 8a outlines a production step N of this machine. Figure 8b shows the packaging W of a product. Figure 8c shows how two production steps in a row produce two products from two material supplies. $N \bullet N$ has two material items in the left interface ${}^*(N \bullet N)$ and two products in the right interface $(N \bullet N)^*$. Now, the composition $N \bullet N \bullet W$ in figure 8d is interesting: The top product, which is the last product produced, is packaged. Intuitively, this becomes obvious if we first form $N \bullet W$ and then add N from the left to form $N \bullet (N \bullet W)$. Figure 8e shows the packaging of both products.

2 Statics: structures and signatures

2.1 Sets and operations

In a large system, many kinds of *objects* are interwoven, including data, items, algorithms, people, actions of people, steps of organizations, products, parts, services, customers, rights, obligations, pieces of money, assets, debts, et cetera. A description of such a system includes both *tangible objects* as well as *intangible objects*. To design such a system or even to understand parts of it, one has to understand aspects of these objects as far as they are important for the system. One of the aspects is the composition of such an object from other objects. Another aspect concerns what can happen to an object, namely to what extent it is involved in changes of (local) states, or subject to changes.

In the conflict between expressiveness, precision, and manageability, we propose representations based on a notion of *sets* as *Cantor* described it already in 1895 [1, p. 481]:

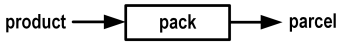
> *Unter einer ‚Menge' verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die ‚Elemente' von M genannt werden) zu einem Ganzen.* (*By a 'set' we understand any aggregate M of particular, well-differentiated objects m of our intuition or thought (which are called the 'elements' of M) into a whole*; translation by the authors.)

We do not use constructions such as sets that contain themselves as elements and thereby generate contradictions, and we avoid sets with unclear boundaries.

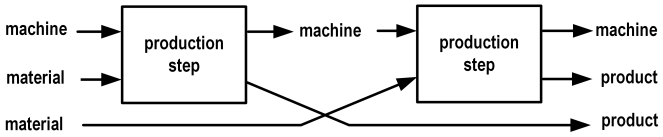
Objects of a system can often be pragmatically classified; for example:

- data that can created, deleted, differently represented, copied, updated, or composed;

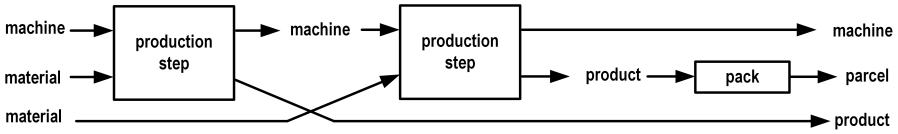
(a) production step N



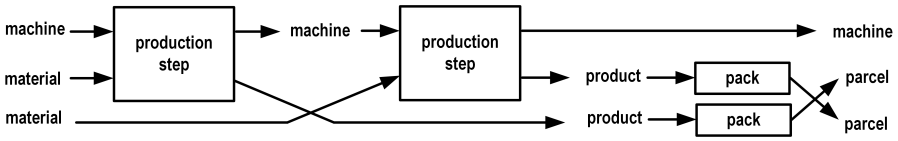
(b) behavior W: packing a product



(c) two successive production steps: $N \bullet N$



(d) composition $N \bullet N \bullet W$: the last produced product is packed



(e) composition $N \bullet N \bullet W \bullet W$: both products are packed

Fig. 8: production steps

– objects that contain deadlines, such as invoices with payment terms, or validity of identification documents;
– individual objects, such as vehicles with license plates;
– multiple objects, such as apples in retail stores;
– abstract objects, such as bank account balances, e-books, software, services, orders, or contracts;
– composed objects that consist of different components.

In its basic form, Heraklit treats all such objects equally neutrally. Special sets of objects have their own particular properties. Most of the concepts used by Heraklit to deal with such objects and sets are derived from predicate logic, and are well known in informatics.

Sets are the basis for *operations*. For example, an operation assigns a customer number to each customer or a price to each good. Such an operation can also be n-ary: for example it can generate a sales contract for a good and a customer.

To model an existing or a planned system, a *symbol* is agreed upon for each involved set, for some special elements of these sets, and for each operation. These symbols are put together in a *signature*; they can be composed into *terms*. An example is the term $f(a, g(b))$, with symbols a and b for special elements, and f for a binary and g for a unary operation. Thus the system can be represented symbolically by a *scheme*.

In contrast to chapter 1 with the new composition calculus, the concepts of this chapter are all well-known from the literature on general algebra and especially on predicate logic, but also on specification languages in informatics, such as ASM, B, RAISE/RSL, VDM, Z, et cetera, and to some extent on relational databases. We consistently develop the concepts from the everyday aspects of the applications; only in a few cases do we deviate from usual representations of predicate logic or common specification languages.

All concepts presented are explained using the example of an insurance company, as already done in chapter 1.

2.2 Structures

In the systematic development of mathematics and logic, *structures* were established in the 20th century: a structure (also: *Tarski structure*) consists of some sets, some specially distinguished elements of these sets (*"constants"*), and some *operations* on these sets. The *arity* of an operation f describes from which sets the arguments $a_1, \ldots, a_{n-1}$ for f and the result $a_n = f(a_1, \ldots, a_{n-1})$ are taken.

In HERAKLIT, the sets and operations of a structure can be chosen quite freely. They are not, as is usual in informatics, restricted to sets of symbol chains characterized by automata, grammars, or similar concepts. HERAKLIT uses the general concept of sets and functions to model, in particular, liveworld or imagined objects, operations, and facts, and to include them in formal arguments.

We begin with the very elementary concepts of sets and functions, and their combination in structures:

Definition 12 (domain, arity of a function). *Let $A_1, \ldots, A_n$ be sets and let $f : A_1 \times \cdots \times A_{n-1} \to A_n$ be a function. The sets $A_1, \ldots, A_n$ are the* domains *of f; the tuple $(A_1, \ldots, A_n)$ is the* arity *of f.*

The arity of a function is often defined as the number of arguments. This is sufficient in the homogeneous case, when there is only one domain. Later, however, we will need different domains.

Notation 3 (constant). *As usual in mathematics, the case $n = 1$ denotes a "0-arity" function, and thus a single element in the set A_1, and is called a* constant of type A_1.

structure Σ_1: *accident*

sets
drivers: *driver's license holders*
names: *the drivers' names in quotes*
cars: *all cars with valid license plates*
license-plates: *the valid license plates*
damages: *short texts*
insurance-types: {*compreh., liability*}
damage-sums: { € n | 0 < n < 100.000}

constants
Bob, Alice: drivers
'Bob', 'Alice': names
A-01, A-02: cars
'A-01', 'A-02': license-plates
at-tree, in-ditch: damages
compreh., liability: insurance-types
€ 0, € 500, € 1000: damage-sums

functions
f: cars ⟶ insurance-types
f(A-01) = compreh.
f(A-02) = liability
...

h: damages × insurance-types
⟶ damage-sums
h(at-tree, compreh.) = € 1000
h(in-ditch, compreh.) = € 500
h(at-tree, liability) = € 0
h(in-ditch, liability) = € 0
...

'_': drivers ⟶ names
'(Alice)' = 'Alice'
'(Bob)' = 'Bob'
...

'_': cars ⟶ license-plates
'(A-01)' = 'A-01'
'(A-02)' = 'A-02'
...

Fig. 9: the structure *accident*

Definition 13 (structure). *Let $M_1, \ldots, M_m$ be sets, and let $f_1, \ldots, f_k$ be functions with domains in $\{M_1, \ldots, M_m\}$. Then*

$$S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$$

is a structure. *The sets M_i are the* basic sets*, and the functions f_j are the* basic functions *of S.*

Thus, using the above notation, a structure can contain finitely many constants.

In the running example of car insurance, we use the structure *accident* in figure 9. The names of the basic sets, constants, and functions of this structure are partially self-explanatory by its application: The function f denotes for each car its insurance type, and the function h denotes the decision of the insurance company for a given report and contract. How these functions are defined in detail is irrelevant here; however, we assume, in the running example,

$$f(\textit{A-01}) = \textit{compreh.} \text{ and } h(\textit{at-tree}, \textit{compreh.}) = 1000. \tag{2}$$

The two functions

$$'\ldots' : \textit{driver} \to \textit{name} \text{ and } '\ldots' : \textit{cars} \to \textit{numbers} \tag{3}$$

assign a digital representation (a *digital twin*) to each driver x and each car y. As a convention, we write $'z'$ for the digital twin of a real or imaginary object z.

Structures provide the ability to apply functions in a nested fashion; for example, with the above assumptions (2):

$$h(\textit{at-tree}, f(\textit{A-01})) = 1000. \tag{4}$$

In general, an infinite number of elements of a basic set can be accessed in this way. The simplest example is the structure

$$S = (\mathbb{N}; 0, \textit{suc}), \tag{5}$$

with the set $\mathbb{N}$ of natural numbers as the only domain, the number 0 as a constant, and the successor function *suc* assigning the number $i + 1$ to each number i. Thus any number n can be addressed by applying to the constant 0 the *suc* operation n times.

The systematic way from informal descriptions of real or imaginary objects, operations and properties to formal descriptions begins with structures: the modeler is in no way restricted in the choice of basic sets and functions; they can refer to *Cantor's* liberal notion of sets described above. Here, the modeler can incorporate his interests, objects, and purposes into the model.

The basic sets of a structure can themselves be infinite. The restriction to *finitely many* basic sets or functions can easily be overcome: For example, infinitely many sets $M_1, M_2, \ldots$, are gained by the natural numbers $\mathbb{N}$ as another basic set, and the set $(M_1 \cup M_2 \cup \ldots) \times \mathbb{N}$ as another single basic set.

2.3 Signatures

Since a structure consists of basic sets, constants and functions of a real or an imagined world, the question of a symbolic, textual, machine-processable *representation* of such sets, constants and functions, and the nested application of functions, arises. This is accomplished by a *signature* of a structure: for each basic set, each constant, and each operation of the structure, a corresponding signature contains its own symbol. Each function symbol also comes with its arity. The symbols of the signature can be chosen freely; they only have to be pairwise different. Often a symbol of a signature is composed of other symbols to outline its intended use. Of course, after all, the elements of a structure S are themselves already named in some way; in the structure *accident*, for example, *driver, Bob* and f. It is technically simple to use the symbol $\underline{M_i}$ for each set M_i of a structure S in the signature of S, and for each operation $f : A_1 \times \cdots \times A_{n-1} \to A_n$ the composed symbol $\underline{f : A_1 \times \cdots \times A_{n-1} \to A_n}$. Thus, by underlining the name of an object, a symbol is created. These symbols are compiled analogously to the structure and form a *signature* for the structure:

Definition 14 (symbols for sets and functions).

1. *Let A be a set. Then let $\underline{A}$ be a symbol, denoted as a* sort symbol for A.

signature for the structure *accident*

sort symbols	constant symbols	function symbols
drivers	Bob, Alice: drivers	f: cars ⟶ insurance-types
names	'Bob', 'Alice': names	h: damages × insurance-types
cars	A-01, A-02: cars	⟶ damage-sums
license-plates	'A-01', 'A-02': license-plates	' ': drivers ⟶ names
damages	at-tree, in-ditch: damages	' ': cars ⟶ license-plates
insurance-types	compreh., liability: insurance-types	
damage-sums	€ 0, € 500, € 1000: damage-sums	

Fig. 10: signature of the structure *accident*

2. *Let $A_1, \ldots, A_n$ be sets and let $f : A_1 \times \cdots \times A_{n-1} \to A_n$ be a function. Then the symbol sequence*
$$\underline{f : A_1 \times \cdots \times A_{n-1} \to A_n}$$
is an operation symbol for *f.*
3. *An operation symbol $\underline{f : A_1 \times \cdots \times A_{n-1} \to A_n}$ is usually just written as $\underline{f}$.*
4. *The sequence $\underline{A_1}, \ldots, \underline{A_n}$ of sort symbols of $A_1, \ldots, A_n$ is the* arity *of $\underline{f}$.*
5. *For the case $n = 1$, the operation symbol $\underline{f : \to A_1}$ is a* constant symbol *of type A_1.*

Definition 15 (signature for a structure).
Let $S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$ be a structure. Then
$$\Sigma = (\underline{M_1}, \ldots, \underline{M_m}; \underline{f_1}, \ldots, \underline{f_k})$$
is a signature for S. *The structure S is a* Σ-structure.

Figure 12 shows the signature Σ_1 of structure *accident*. A signature for structure S in (5) is:
$$\Sigma_2 = (\underline{\mathbb{N}}; \underline{c : \mathbb{N}}, \underline{s : \mathbb{N} \to \mathbb{N}}). \tag{6}$$

Observation. A signature is a sequence of pairwise different symbols. As already explained, the symbols can be freely chosen. The choice of symbols in the above definition, and the abbreviated notation, is merely practically motivated. Thus, the different signatures of a structure differ only in the choice of the symbols. On the other hand, completely different structures can have the same signature. This will be used later to characterize classes of similarly formed structures.

2.4 Basic terms of a signature

The nested application of functions of a structure S can be symbolically reproduced with the symbols of a signature of S. For this purpose, *terms* are formed according to the arity of the symbols:

Definition 16 (basic term of a signature). *Let*

$$S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$$

be a structure, let Σ be a signature for S, and let $A \in \{M_1, \ldots, M_m\}$. The set T_Σ of basic terms of Σ *of type $\underline{A}$ is defined inductively:*

1. *Every constant symbol of Σ of type $\underline{A}$ is a* basic term *of Σ of type $\underline{A}$.*
2. *Let $\underline{f}$ be a function symbol of Σ with arity $\underline{A_1}, \ldots, \underline{A_n}$, $n > 1$. For $i = 1, \ldots, n-1$ let t_i be a fundamental term of type $\underline{A_i}$. Then $\underline{f}(t_1, \ldots, t_{n-1})$ is a* basic term *of type $\underline{A_n}$.*

As an example, the basic term $\underline{\mathbf{h}(\mathbf{at\text{-}tree}, \mathbf{f}(\mathbf{b}))}$ of the signature *accident* has the type $\underline{\text{damage-sums}}$. The basic term $\underline{\mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{c})))}$ of the signature Σ_2 has the type $\underline{\mathbb{N}}$.

For a basic set A of a structure S, each basic term of type A of a signature of S denotes an element of the set A. This element is inductively defined along the structure of the basic terms:

Definition 17 (meaning of a basic term). *Let*

$$S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$$

be a structure, let $\Sigma = (\underline{M_1}, \ldots, \underline{M_m}; \underline{f_1}, \ldots, \underline{f_k})$ be a signature for S and let $t \in T_\Sigma$ be a basic term. The meaning of t in S, *written t_S, is defined inductively:*

1. *For a constant symbol $t = \underline{f}$, t_S is defined as f.*
2. *For $t = \underline{f}(t_1, \ldots, t_n)$, t_S is defined as $f(t_{1S}, \ldots, t_{nS})$.*

In the structure *accident* from figure 9, the basic term

$$\underline{\mathbf{h}(\mathbf{at\text{-}tree}, \mathbf{f}(\mathbf{A\text{-}01}))} \tag{7}$$

describes the amount of € 1000 that the insurance company pays the driver *Bob* for the accident, based on the definitions in (2) and the calculation in (4).

2.5 Terms with variables

With the basic terms of the signature *accident* only a few objects of the structure *accident* can be represented: two drivers, cars, damages and insurance types, and three damage amounts.

In the structure *accident*, every holder of a driver's license is a potential driver, all registered cars are recorded, et cetera. To introduce each of these drivers and cars as a constant of the structure is unrealistic. However, if a customer is not a constant, there is no basic term to denote this driver. We solve this problem, as is common in mathematics, by using *variables* for drivers and cars: Wherever there is a constant symbol in a basic term, there can now also be a variable instead. This leads to the following definition:

Definition 18 (term with variables). *Let $S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$ be a structure, let $\Sigma = (\underline{M_1}, \ldots, \underline{M_m}; \underline{f_1}, \ldots, \underline{f_k})$ be a signature for S, and let $A \in \{M_1, \ldots, M_m\}$.*

1. *Let X be a set of symbols. For each $x \in X$, let the* type of x *be one of the basic sets M_i. Then X is a* set of variables for S.
2. *Let X be a set of variables for S. The set $T_\Sigma(X)$ of* terms over Σ and X of type $\underline{A}$ *is defined inductively:*
   - *Each constant symbol of type $\underline{A}$ is a* term in $T_\Sigma(X)$ of type $\underline{A}$.
   - *Each variable $x \in X$ of type $\underline{A}$ is a* term in $T_\Sigma(X)$ of type $\underline{A}$.
   - *Let f be a function symbol with arity $\underline{A_1}, \ldots, \underline{A_n}$, $n > 1$. For $i = 1, \ldots, n-1$, let t_i be a* term in $T_\Sigma(X)$ of type $\underline{A_i}$. *Then $\underline{f}(t_1, \ldots, t_{n-1})$ is a* term in $T_\Sigma(X)$ of type $\underline{A_n}$.

As an example of the signature *accident*, let

$$X = \{x, y\} \tag{8}$$

be a set of variables. Let x be of type *cars* and y be of type *insurance-types*. Then

$$\mathbf{h}(\textbf{at-tree}, \mathbf{f}(\mathbf{x})) \text{ and } \mathbf{h}(\textbf{at-tree}, \mathbf{y}) \tag{9}$$

are two terms in $T_{\Sigma 1}(X)$ of type *damage-sums*.

A term is given a meaning in a structure S by *assigning* each of its variables an element of the corresponding basic set of S and then *calculating* the term:

Definition 19 (assignment of a variable set). *Let*

$$S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$$

be a structure. Let X be a set of variables for S. Let $\beta : X \to M_1 \cup \cdots \cup M_m$, where for each variable x of type M_i the following holds: $\beta(x) \in M_i$. Then β is an assignment of X in S.

As an example, consider the structure *accident* from figure 9 and the set $X = \{x, y\}$ from (8). Then

$$\beta : \{x, y\} \to \text{cars} \cup \text{insurance-types with } \beta(x) = \text{A-01 and } \beta(y) = \text{compreh.} \tag{10}$$

is an assignment of X in the structure *accident*.

A term with variables gets a meaning in a structure S under an assignment of its variables:

Definition 20 (meaning of a term). *Let S be a structure, let Σ be a signature for S, let X be a set of variables for Σ, let β be an assignment of X in S, and let t be a term in $T_\Sigma(X)$. The* meaning of t in S under the assignment β, *written $t_{(S,\beta)}$, is defined inductively:*

1. *For a constant symbol* $t = \underline{f}$, $t_{(S,\beta)}$ *is defined as* f.
2. *For a variable* $t = x$, $t_{(S,\beta)}$ *is defined as* $\beta(x)$.
3. *For* $t = f(t_1, \ldots, t_{n-1})$, $t_{(S,\beta)}$ *is inductively defined as* $f(t_{1_{(S,\beta)}}, \ldots, t_{n-1_{(S,\beta)}})$.

As an example, for the two terms from (9) and β from (10) it holds that:

$$\mathbf{h}(\mathbf{at\text{-}tree}, \mathbf{f}(\mathbf{x}))_{(\mathbf{accident},\beta)} = \mathbf{h}(\mathbf{at\text{-}tree}, \mathbf{y})_{(\mathbf{accident},\beta)} = \mathbf{1000}. \quad (11)$$

Variables in terms and the concept of assignments can be used to label an infinite number of elements.

2.6 Propositions, predicates, and relational structures

Objects and functions form a structure; using the terms and variables of a matching signature, one can represent objects and functions of that structure. In addition, one would like to formulate *properties* and *relationships between objects*, either as assertions about a given structure, or as a requirement for a structure to be constructed. Such an assertion or claim can adequately be formulated as a *proposition* in the sense of propositional logic.

Typical examples of such propositions in the context of the example of a car insurance company are:

Driver Bob has an accident with the car number A-01. (12)

Bob has no car. (13)

There is an accident report about Bob's accident with the car with number A-01. (14)

Such a proposition can be arbitrarily complex in terms; as an elementary formal construct, a proposition has a truth value, and is either *true* or *false*. In an evolving dynamical system, however, this value is not necessarily constant forever, but may swap to the other truth value.

The domains of a HERAKLIT model can be arbitrarily large, in particular infinitely large, and the question arises how to formulate propositions about "many" objects. To do this, we start from the observation that propositions are often similarly structured: they consist of a *predicate* and differ only in the objects to which the predicate *applies*. For example, the proposition in (12) consists of the predicate *accident* and the object tuple *(Bob, A-01)*. Here, we can easily think of object tuples consisting of other drivers and other cars for the predicate *accident*.

We require concise formulations for such properties, especially in the context of graphical representations, and write the object tuples in parentheses after the predicate. Thus, for the propositions (12) to (14) with the predicates *driver-without-car* and *accident-report*, we formulate the representation:

accident(Bob, A-01), (15)

$$\textit{driver-without-car(Bob)}, \tag{16}$$

$$\textit{accident-report}('\textit{Bob}', '\textit{A-01}', \textit{at-tree}) \tag{17}$$

We use predicates whose object tuples consist of elements of domains of a structure. This allows a predicate to be embedded in a structure: a predicate is technically a relation over the domains of the structure. To integrate predicates, we extend the concept of a structure to include relations, and form *relational structures*:

Definition 21 (relational structure). *Let $S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$ be a structure; let $A_1, \ldots, A_q \in \{M_1, \ldots, M_m\}$.*

1. *Let $R \subseteq A_1 \times \cdots \times A_q$. Then R is a* relation *of S. The tuple $(A_1, \ldots, A_q)$ is the* arity *of R.*
2. *Let $R_1, \ldots, R_p$ be relations of S. Then*

   $$S' = (M_1, \ldots, M_m; f_1, \ldots, f_k; R_1, \ldots, R_p)$$

   is a relational structure.
3. *Let R be as in 1., and let $a \in A_1 \times \cdots \times A_q$. Then a is a* tuple of R *and $R(a)$ is a* proposition about S'.
4. *A proposition $R(a)$ holds in S' if $a \in R$.*

In 1. with $q = 1$ the relation R is *unary*, that is, a subset of A_1.

Figure 11 outlines a set of predicates of the structure *accident*. These predicates complement the structure *accident* to form a relational structure. It is convenient to denote this relational structure by *accident* as well.

predicates

accident ⊆ drivers × cars
accident-report ⊆ names × license-plates × damages
driver-without-car ⊆ drivers
exisiting-contract ⊆ license-plates × insurance-types
decision-prepared ⊆ names × license-plates × damages × insurance-types
driver-with-rental-car ⊆ drivers
driver-before-decision ⊆ drivers
file ⊆ names × license-plates × damages × insurance-types
notification-to-driver ⊆ names × damage-sums
updated-contract ⊆ license-plates × insurance-types
accident-settled ⊆ names × damage-sums

The tuples of the single predicates are not represented.

Fig. 11: predicates for structure *accident*

Equations (15) to (17) depict examples of propositions about the relational structure *accident*. Whether they apply or not is left open here, since the predicates in figure 11 are not specified in detail.

sort symbols
drivers
names
cars
license-plates
damages
insurance-types
damage-sums

constant symbols
Bob, Alice: drivers
'Bob', 'Alice': names
A-01, A-02: cars
'A-01', 'A-02': license-plates
at-tree, in-ditch: damages
compreh., liability: insurance-types
€ 0, € 500, € 1000: damage-sums

function symbols
f: cars ⟶ insurance-types
h: damages × insurance-types
⟶ damage-sums
' ': drivers ⟶ names
' ': cars ⟶ license-plates

relation symbols
accident with arity drivers × cars
accident-report with arity names × license-plates × damages
driver-without-car with arity drivers
exisiting-contract with arity license-plates × insurance-types
decision-prepared with arity names × license-plates × damages × insurance-types
driver-with-rental-car with arity drivers
driver-before-decision with arity drivers
file with arity names × license-plates × damages × insurance-types
notification-to-driver with arity names × damage-sums
updated-contract with arity license-plates × insurance-types
accident-settled with arity names × damage-sums

variables
x: drivers
y: cars
z: damages

Fig. 12: signature of relational structure *accident* and variable set X

2.7 Signatures for relational structures

In analogy to signatures of structures, signatures of relational structures are defined:

Definition 22 (signature for a relational structure).
Let $S = (M_1, \ldots, M_m; f_1, \ldots, f_k, R_1, \ldots, R_p)$ be a relational structure.

1. *For the relation R_i of arity $(A_1, \ldots, A_q)$, $\underline{R_i}$ is the* relational symbol *of R_i of arity $(\underline{A_1}, \ldots, \underline{A_q})$.*
2. *$\Sigma = (\underline{M_1}, \ldots, \underline{M_m}; \underline{f_1}, \ldots, \underline{f_k}; \underline{R_1}, \ldots, \underline{R_p})$ is a* signature for *S.*

Notation 4. *1. Variables and terms for the structure $(M_1, \ldots, M_m; f_1, \ldots, f_k)$ are also variables and terms for S.*
2. For a relational symbol R with arity $(\underline{A_1}, \ldots, \underline{A_q})$ and terms t_i with arity $\underline{A_i}$, the term tuple $(t_1, \ldots t_q)$ has the arity of *R.*
3. A relational structure with signature Σ is a Σ-relational structure.

Figure 12 shows a signature for the relational structure *accident*.
Relational symbols are the basis for logical expressions:

Definition 23 (logical expression). *Let Σ be a signature for relational structures, let X be a set of variables for Σ, and let R be a relational symbol with the arity $(\underline{A_1}, \ldots, \underline{A_q})$. For $i = 1, \ldots, q$, let t_i be a term over Σ and X of arity $\underline{A_i}$. Then $\underline{R}(t_1, \ldots, t_q)$ is a* logical expression *over Σ and X.*

Typical logical expressions over the signature from figure 12 are: $\underline{accident}$ ($\underline{\text{Bob}}$, $\underline{\text{A-01}}$) and $\underline{contract}$ ($\underline{'y'}$, $\underline{f(y)}$).

A logical expression holds or does not hold in a relational structure together with an assignment of the variables:

Definition 24 (validity of a logical expression). *Let Σ, X and $\underline{R}(t_1, \ldots, t_q)$ be as in definition 23 above. Let S be a Σ-structure and let β be an assignment of X in S. The logical expression $\underline{R}(t_1, \ldots, t_q)$* is valid under β in S if $(t_{1(S,\beta)}, \ldots, t_{n(S,\beta)}) \in R$.

Predicate logic composes such logical expressions with propositional operators "and", "not", et cetera.

2.8 Conventions and notations

Sections 2.1 to 2.7 describe the mathematical foundations of dealing with data and objects of all kinds, and their relation to symbolic representations. To make their practical application more manageable, we use a number of conventions and notations:

1. To write down a structure $S = (M_1, \ldots, M_m; f_1, \ldots, f_k)$, its basic sets, constants, and functions must somehow be symbolically denoted; we use $\underline{M_1}, \ldots, \underline{M_m}$ and $\underline{f_1}, \ldots, \underline{f_k}$. As a writing convention, we form a signature with these symbols, whose only interpretation of interest to us is the structure S. With this signature we can form terms with variables, and interpret them in S, as described above. Occasionally, we omit the underlining.
2. In applications, structures, especially relational structures, can become very large. The linear representation

$$S = (M_1, \ldots, M_m; f_1, \ldots, f_k, R_1, \ldots, R_p)$$

   is then replaced by groups of sorts, constants, functions, and predicates. In such a representation, the variables used are also listed.
3. We use Cartesian products of domains in structures as derived domains and as relations, and correspondingly as sorts of domains and relations in signatures, without explicitly introducing them as domains or sorts.
4. In this section, we have assumed an arbitrary given structure S and constructed a signature Σ as symbolic representations for S. Occasionally, one starts the other way around from a signature Σ, and considers several structures with Σ as a signature. Such a structure S is an *interpretation of* Σ.

3 Dynamics

This section is about *dynamic behavior* of systems. We start with *local states* and *local steps*. Such a *step* describes how in a subsystem some local states are left and others are reached. For example: Immediately after a traffic accident, a driver is

in the local state *accident* with his car. In a first step he leaves the local state *accident*, creates an accident report, and reaches the local state *driver-without-car*. His report reaches the insurance company, where it causes the local state *accident-report*. In a subsequent step, an accident report and the corresponding current insurance policy are compiled and a decision on payment is prepared. Thus, several steps occurring one after the other can form a sequence of any length. However, steps are not always in sequence; they may occur *independently of each other*. For example: Independently of the preparation and the decision of the insurance company about a payment, the driver without a car rents a rental car. With such steps, partly sequential and partly independent of each other, a *run* gradually evolves, until finally the concrete insurance case is settled. Concrete objects are involved in this process: a particular car, a driver, a report, a contract, et cetera.

HERAKLIT describes dynamics on three levels:

- A single HERAKLIT *run* (for short: run) is a single behavior of a (possibly composed) module. Based on the framework of structures from chapter 2, a single run involves individual objects and predicates of a relational structure.
- A HERAKLIT *system* (for short: system) characterizes a (generally infinitely large) set of (generally unboundedly growing) runs. Objects and operations are captured in *a* structure. To deal with large, in particular infinite, sets of objects, a suitable signature is used to form terms with variables. With cyclically recurring patterns, sets of unboundedly growing runs can be characterized this way.
- A HERAKLIT *schema* (for short: schema) describes a set of systems. Technically, a schema is based on a signature (as in section 2.3), so it is a purely symbolic representation. Any interpretation of this signature yields a system.

All three levels use the same principles to represent behavior. In the following section, we discuss these principles and how to deal with them formally. We then turn to the three levels of modeling dynamic behavior, as described above.

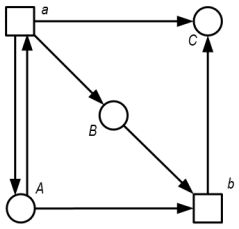
3.1 Nets

We begin with the observation that in describing dynamic behavior, a conceptual distinction can be made between state-like (passive) and step-like (active) elements. Such elements are structurally, causally, or factually connected. Thereby, a state-like element is directly connected only to step-like elements, and, vice versa, a step-like element is connected only to state-like elements. This leads to the basic definition of a *net*:

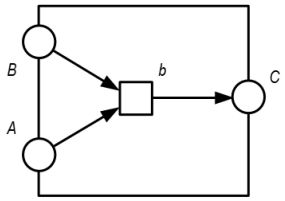
Definition 25 (net). *Let P and T be disjoint sets. Let $F \subseteq (P \times T) \cup (T \times P)$. Then $N = (P, T, F)$ is a* net.

A number of notations and conventions support the diverse use of nets:

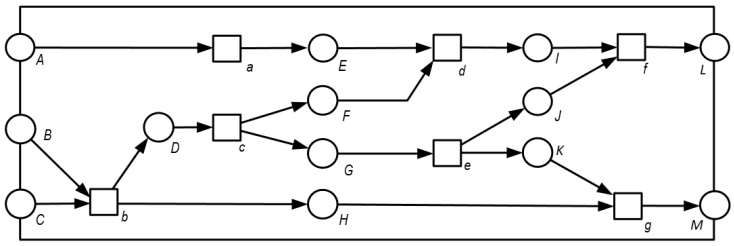
Notation 5 (components of nets). *Let $N = (P, T, F)$ be a net.*



(a) net



(b) net-atom as module



(c) causal net as module

Fig. 13: graphical representation of nets

1. *The set P contains the* places *of N; a place is used to represent a state-like, data-storing element of N, graphically circular- or elliptic-shaped.*
2. *The set T contains the* transitions *of N; with a transition an active, state-changing element is represented, graphically square- or rectangular-shaped.*
3. *The set F contains the edges of the* flow relation *of N; with an edge, a (directed) spatial or factual relation between a place and a transition is represented, graphically arrow-shaped.*
4. *$(P \cup T, F)$ is obviously a directed graph; thus all definitions and properties of graphs apply to nets as well.*

Definition 26 (basic net concepts). *Let $N = (P, T, F)$ be a net.*

1. *For $x \in P \cup T$, $\{y \mid yFx\}$ is the* pre-set *of x, written ${}^\bullet x$, and $\{y \mid xFy\}$ is the* post-set *of x, written $x^\bullet$.*
2. *$x \in P \cup T$ is* backward *(or respectively,* forward*)* branched *if ${}^\bullet x$ (or respectively, $x^\bullet$) contains more than one element.*
3. *For a net (P', T', F'), a bijective mapping $f : P \cup T \to P' \cup T'$ is a* net isomorphism *if for every pair $(x, y) \in (P \times T) \cup (T \times P)$ it holds that $(x, y) \in F$ if and only if $(f(x), f(y)) \in F'$.*

Figure 13a shows an example of the graphical representation of a net.

To describe system behavior we use special net structures, in particular net-atoms with only one transition:

Definition 27 (net-atom, atomic module).

1. *A place-labeled net of the form* $N = (P, \{t\}, F)$ *is a* net-atom.
2. *A net-atom can be interpreted as a module, with* ${}^*N = {}^\bullet t$ *and* $N^* = t^\bullet$.
3. *Let* N *be a net. Each transition* t *of* N *defines the net-atom* $({}^\bullet t \cup t^\bullet, \{t\}, ({}^\bullet t \times \{t\}) \cup (\{t\} \times t^\bullet)$*, written as* atom(t).

Figure 13b shows a net-atom as a module. Acyclic nets with unbranched places are also important:

Definition 28 (causal net, causal net as module). *Let* $N = (P, T, F)$ *be a place-labeled net with at least one transition.*

1. N *is a* causal net *if* N *is acyclic and no place of* N *is forward or backward branched.*
2. *Conceived as a module,* ${}^*N$ *contains all places* $p \in P$ *with* ${}^\bullet p = \emptyset$*, and* N^* *contains all places* $p \in P$ *with* $p^\bullet = \emptyset$.

Figure 13c shows an example of a causal net. This naming follows its later use to represent causal relations of steps.

Each net-atom is obviously a causal net, and each causal net is composed of net-atoms:

Lemma 5 (causal net is composed of atoms). *Let*

$$K = (P, T, F)$$

be a finite causal net. The transitions of K *can be ordered in the form* $(t_1, \ldots, t_n)$ *such that* $K = atom(t_1) \bullet \cdots \bullet atom(t_n)$.

Proof. Induction over n: case $n = 1$: A net-atom is obviously itself a causal net. Assumption: the assertion holds for $n-1$. There is a transition t_1 of K such that ${}^\bullet p = \emptyset$ for every $p \in {}^\bullet t_1$. Let F_1 be the set of arrows of $atom(t_1)$. Now let K' be defined as $K' = (P \setminus {}^\bullet t_1, T \setminus \{t_1\}, F \setminus F_1)$. By construction, K' is a causal net with $n - 1$ transitions. By the induction assumption, the transitions of K' can be arranged in the form $(t_2, \ldots, t_n)$ such that $K' = atom(t_2) \bullet \cdots \bullet atom(t_n)$. Then $K = atom(t_1) \bullet \cdots \bullet atom(t_n)$. □

3.2 Local states

Dynamical systems are often described in natural and technical sciences, but also in economics and social science, as an evolution in the flow of time; thus formally as a (continuous) function over the (time) axis of real numbers.

Since the advent of computer-integrated systems, behavior has also been represented in *discrete steps*; formally, then, with automata models or in the general form as transition systems. A step is a transition from one global state to another global state; and a single behavior, a *run*, is a sequence of such steps. Undisputed in this context is the use of *global* states. In the composition of two

systems A and B, all combinations (a, b) of states a of A and of states b of B are used as global states of the composed system.

However, Heraklit takes a different approach. For intuitive motivation, let us consider a business consisting of several (sub)systems, for example, departments for production, human resource management, customer service, balance sheet accounting, et cetera. The departments are – as usual – tailored to work relatively autonomously with relatively little flow of data and objects. The behavior of each individual department is represented as a transition system with states that are global within the department. Now, how should the behavior of the entire company be modeled? If, as described above, one composes the local states of all departments into a global state of the entire system, for example, the processing of a workpiece in the production department updates the global state of the entire company. Intuitively, this would be highly artificial and would formally lead to a massive flood of global states that would contribute little to the intuitive understanding and formal analysis of the system.

Heraklit models discrete steps but avoids global states. This is based on the observation that a global state is mostly composed of smaller, *local* states, and that a step mostly does not update an entire global state, but only a few smaller local states.

Heraklit uses the smallest concept conceivable for this purpose to formulate local states: a local state is a *proposition* in the sense of formal logic (detailed in section 2.3), with which a place of a net is labeled. In the context of the flow of events, we take this proposition to be *true* exactly when the corresponding local state has been reached.

Definition 29 (local state over S). *Let S be a relational structure, let N be a net, let p be a place of N, and let q be a proposition over S. Then p with label q is a* local state of N over S.

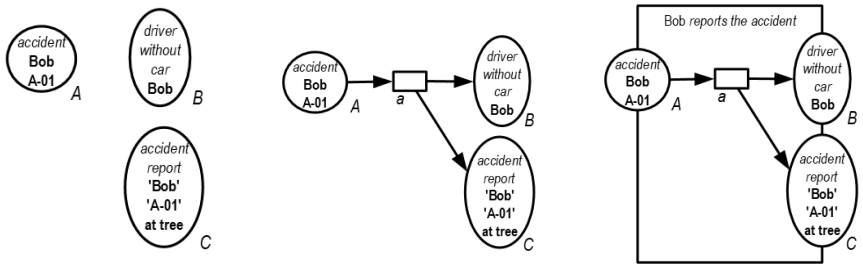
Figure 14a shows three local states: places A, B, and C with propositions of the relational structure *accident*, written according to the conventions from (15) to (17). For space reasons, brackets and commas have been omitted.

3.3 Heraklit runs

In this section we deal with dynamic behavior, that is, *changes* of local states. Dynamics arise by reaching a local state z: then the proposition z is *true*. When z is left, z switches to *false*. Reaching and leaving local states is organized by *steps*: We design dynamics such that multiple local states can be left or reached at the same time, this means *in one step*.

We technically describe a step as a labeled net-atom, N: each place of N is a local state. A local state is left when an arrow leads from the corresponding place to the transition of N. A local state is reached when an arrow leads from the transition to the corresponding place.

Figure 14b shows such a step, with the local states from figure 14a. Each arrow connects to a local state that is left (arrowhead at the rectangle) or reached



(a) three local steps (b) one step *Bob-reports-accident* (c) module *Bob-reports-accident*

Fig. 14: prepare accident report

(arrowhead at the ellipse). This representation follows the convention from section 3.1, according to which each rather passive, state-like element is represented as an ellipse and thus round-shaped:

Definition 30 (step over S). *Let S be a relational structure and let N be a net-atom. Let each place of N be a local state over S. Then N is a* step over S.

Intuitively formulated, a step describes dynamic behavior: After the local states in the pre-set ${}^\bullet t$ have all been reached, they are all left coincidently, and the local states in the post-set $t^\bullet$ are reached.

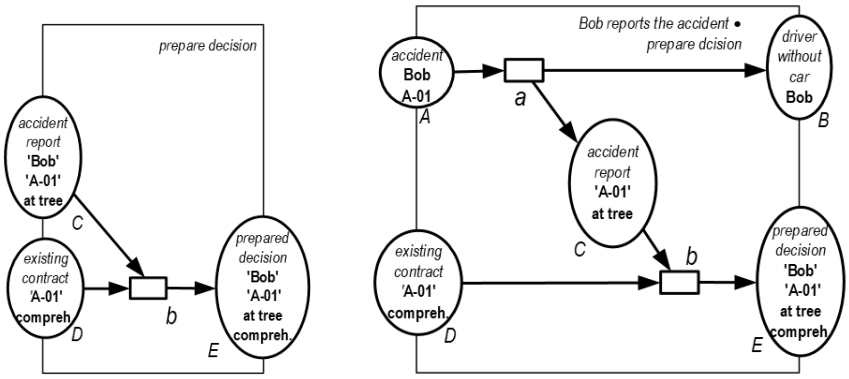
Figure 14b shows a step over the set Q of propositions of the relational structure *accident*. Figure 14c shows the corresponding module.

Figure 15a shows the preparation of the decision of the insurance company: An accident report and the corresponding insurance contract are compiled in one module.

Steps can be *composed*; a *run* is created. The composition of *Bob-reports-accident* and *prepare-decision* in figure 15b describes two steps in sequence: the step *Bob-reports-accident* yields the local state *accident-report*(**'Bob', 'A-01', at-tree**), which in turn is one of the two prerequisites for the step *prepare-decision*. This achieves two local states, *driver-without-car*(**Bob**), and *decision-prepare*(**'Bob', 'A-01', at-tree, compreh.**). These two local states are now prerequisites for further steps, b and f in figure 16. Both steps are independent of each other. The step c is interesting: It occurs after b, and is – just like b – independent of f. Finally, step d synchronizes the two sequences $a - e - f$ and $a - b - c$.

This example illustrates a general principle of modeling single runs, here a single insurance case: a run is an ordered set of steps; however, this order is not necessarily total: two steps can also be unordered (such as b and e, c and e, b and f, and c and f).

Formally, such a behavior, a *run*, is a labeled causal net composed of steps over a set S of propositions: each place is a local state.



(a) module *prepare decision* (b) module *Bob-reports-accident • prepare-decision*

Fig. 15: step *prepare-decision*, composition with *prepare-accident-report*

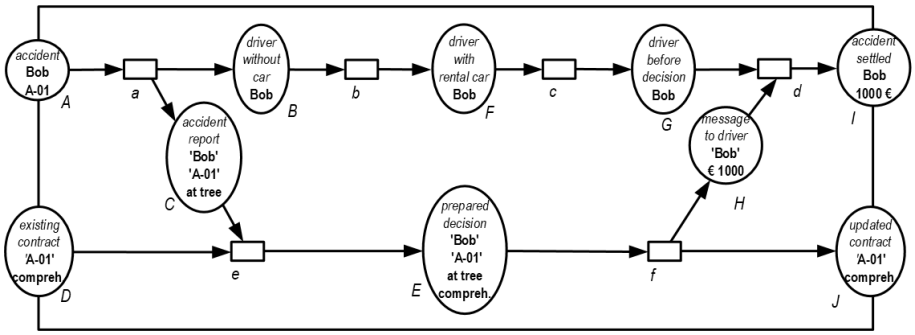


Fig. 16: Bob's accident

Definition 31 (run over S). *Let S be a relational structure and let K be a causal net. Let each place p of K be a local state over S. Then K is a* run over *S.*

Figure 16 shows a run over the set of propositions of the relational structure *accident*.

3.4 Heraklit systems

By analogy with figure 16, figure 17 shows another insurance claim, structurally identical to Bob's accident, but with different data. With more drivers, cars, accident reports, and insurance contracts, we want to be able to use a more compact representation rather than having to enumerate all possible insurance cases individually. In addition, we want a *purely symbolic* representation, which technically exclusively uses elements of signatures, as explained in section 2.3.

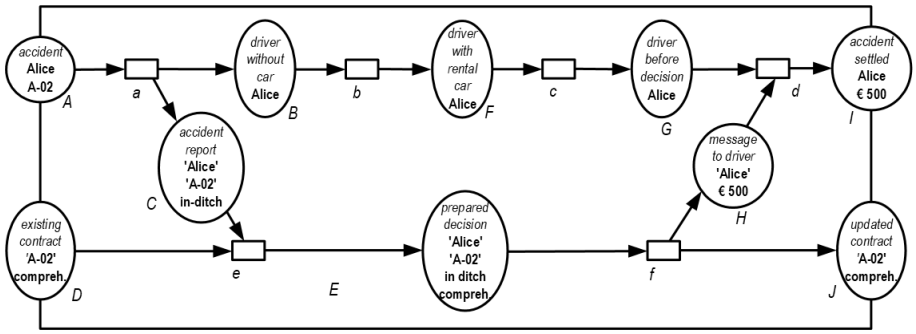


Fig. 17: Alice's accident

For this purpose, we propose a representation as a HERAKLIT *system*. Such a system describes, in general, infinitely many and infinitely long runs. A HERAKLIT system employs logical expressions of the form *predicate(tuple)*, as discussed in section 2.3, and a schematic representation of steps.

Remember that, to model a system, section 2.3 proposes to construct a structure from the basic sets and operations of the system. With a matching signature Σ and a set X of matching variables, the set $T_\Sigma(X)$ of terms provides a symbolic description of the objects and operations of the system. To capture dynamic behavior, we now extend this construction to relational structures and corresponding signatures.

Logical expressions are formed with the relational symbols of such a signature. Typical logical expressions over the signature from figure 12 are: *accident*(**Bob, A-01**) and *accident-report*$('\mathbf{x}', '\mathbf{y}', \mathbf{z})$.

We now symbolically conceive a step N as an *event*, with a net-atom E, isomorphic to N. Each transition of E is the relational symbol of the corresponding transition of N, and each edge of E is labeled with the tuple of the corresponding transition of N. As an example, figure 18a shows the step from figure 16, and the corresponding event. Figure 18b shows the corresponding event to the first step from figure 17. The two events are structurally identical, differing only with respect to the individual driver, car, and accident report. With variables for drivers, cars, and accident reports, the two events can now be depicted in a single representation. Figure 18c shows this representation. Together with the structure $Q =$ accident from figure 9, the variable assignment:

$$\beta(x) = \text{Bob}, \beta(y) = \text{A-01 and } \beta(z) = \text{at-tree} \tag{18}$$

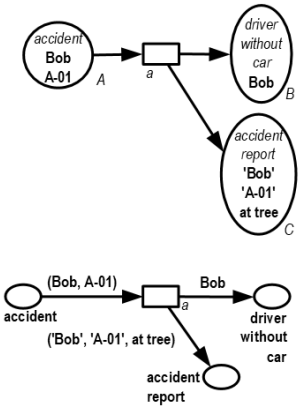
produces the event in figure 18a.

Similarly, the variable assignment:

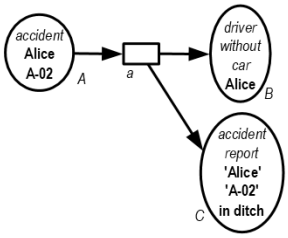
$$\beta'(x) = \text{Alice}, \beta'(y) = \text{A-02 and } \beta'(z) = \text{in-ditch} \tag{19}$$

produces the event in figure 18b.

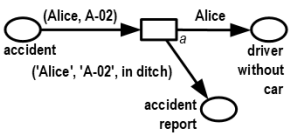
The following definition describes the general concept for events:

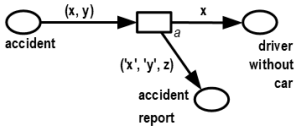


(a) step and event *Bob-reports-accident*



(b) step and event *Alice-reports-accident*





(c) schema *driver x reports accident z with car y*

Fig. 18: events

Definition 32 (event of a signature). *Let Σ be a signature for relational structures and let X be a set of variables for Σ. Let $E = (P, T, F)$ be a net-atom. Let each place $p \in P$ be labeled with a relational symbol p' of Σ. Let each arrow $f = (p, t)$ or $f = (t, p)$ be labeled with a tuple τ of Σ-terms with variables from X and the arity of p'. Then E is an* event over Σ and X.

Definition 33 (step of an event). *Let S be a relational structure, let Σ be a signature for S, let X be a set of variables for Σ, and let E be an event over S with Σ and X.*

*Let β be an assignment of X in S. Let N be a net-atom with an isomorphism $\sigma : E \to N$ such that for each place $p \in P$ and each arrow $f = (p, t)$ or $f = (t, p)$ of E with term label τ, it holds that the place $\sigma(p)$ of N is a local state with proposition $(p, \tau_{(S,\beta)})$. Then N is an S-*step of E.

Thus, by definition 29, a S-step is a step over the propositions of the relational structure S. Each assignment of variables yields a specific step. Given a large number of drivers, insured cars, insurance types, damage types, et cetera, all conceivable damage cases can be represented by assignments of the three variables in figure 18c.

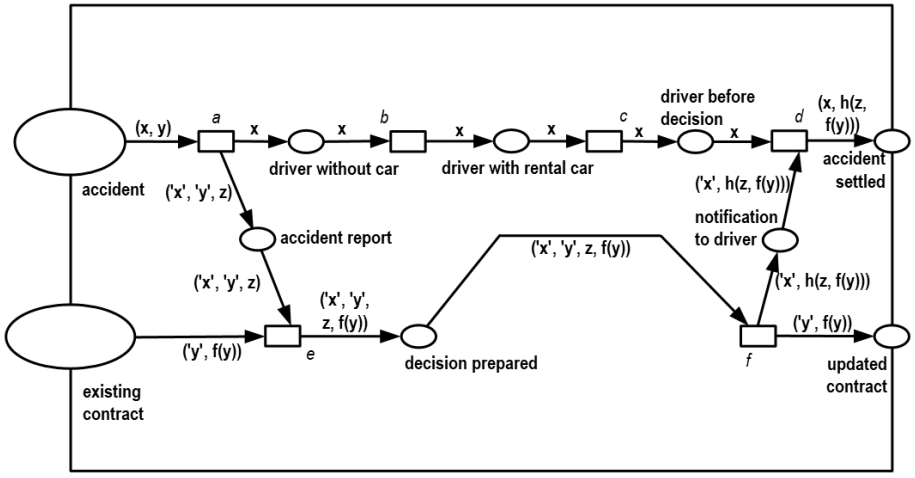


Fig. 19: system net

Thus, all the prerequisites for defining the fundamental notion of a HERAKLIT *system* are available: a HERAKLIT system over a relational structure S is a net structure consisting of events of S:

Definition 34 (HERAKLIT system). *Let S be a relational structure, let Σ be a signature for S, and let X be a set of variables for Σ.*

Let N be a net structure such that each atom of N is an event over Σ and X. Then N is a HERAKLIT system over S with Σ and X.

Definition 35 (run of a HERAKLIT system). *Let S be a relational structure, let Σ be a signature for S, let X be a set of variables for Σ, and let N be a* HERAKLIT *system over S. Let K be a run over S such that each net-atom of K is an S-step of N. Then K is a* run of N with Σ and X.

Observation. Each place p of a system net over Σ and X is a relation of Σ as a predicate.

The runs of figures 16 and 17 are runs of the HERAKLIT system in figure 19. Figure 20 extends this system by cycles describing that the driver rents a car several times if necessary (transition i), or that the insurance company performs additional consultations if necessary (transitions g and h). Figures 21 and 22 show additional runs of the system. In total, this system has an infinite number of runs.

3.5 Initial marking

According to the definition of runs in section 3.4, a run of a HERAKLIT system can begin and end with completely arbitrary steps of the system. In the system of

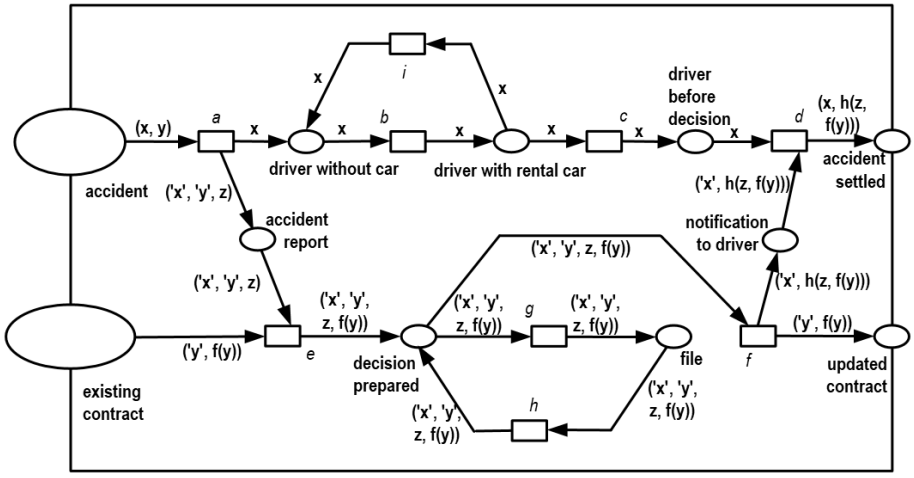


Fig. 20: system net with cycles

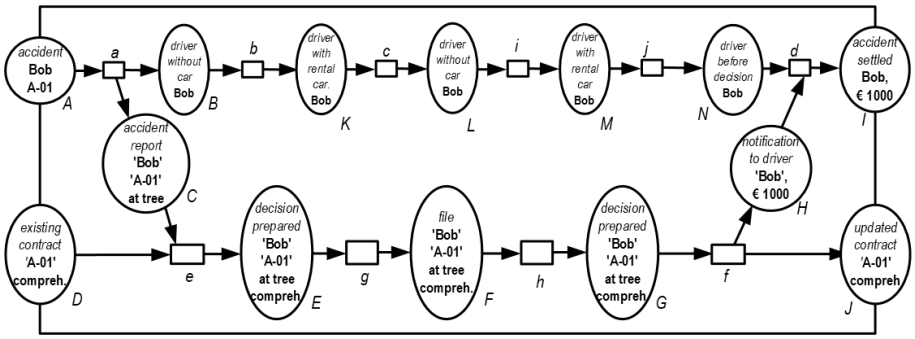


Fig. 21: Bob's accident with consultation

figure 19, however, one might wish to characterize only runs that begin with the writing of an accident report (transition a); perhaps one would like to represent only the two runs of *Bob's accident* and *Alice's accident*.

One can characterize such a restriction to one or a specific set of runs by specifying the left interface of the "intended" runs, using an *initial marking*, as in figure 23.

Definition 36 (Heraklit system with initial marking M_0). *Let S be a relational structure and let N be a* Heraklit *system over S. For each place p, let $M_0(p)$ be a set of elements of type p in S. Then N is a* Heraklit *system over S with initial marking M_0.*

Definition 37 (initially marked run of a Heraklit system with initial marking). *Let S be a relational structure and let N be a* Heraklit *system*

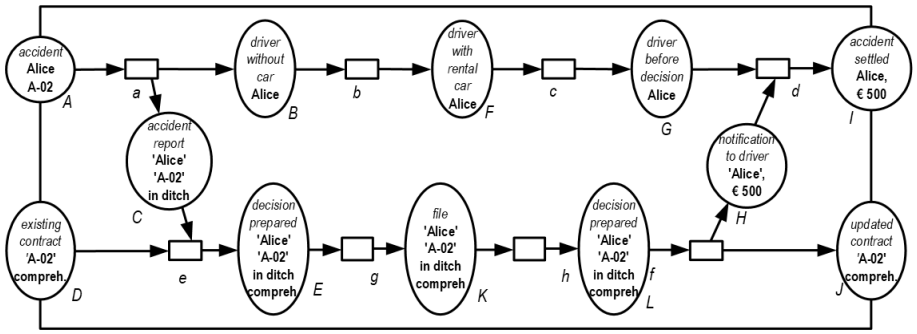


Fig. 22: Alice's accident with consultation

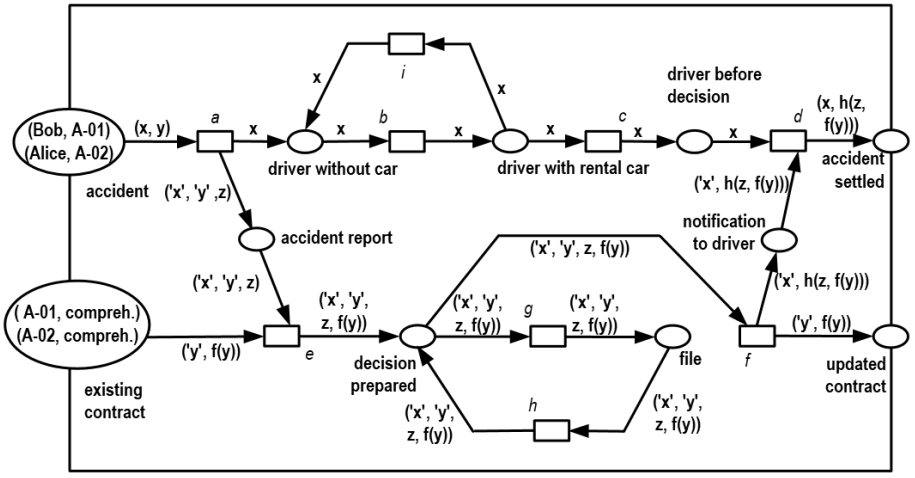


Fig. 23: system net with initial marking

over S with initial marking M_0. Let K be a run of N where the local states of ${}^*K$ are all distinct, and for each local state $(p, (a_1, \ldots, a_n))$ of ${}^*K$ it holds that $(a_1, \ldots, a_n) \in M_0(p)$. Then K is an initially marked run of N.

For example, *Bob's accident* and *Alice's accident* of figures 16, 17, 21, and 22 are two initially marked runs of the system net in figure 23.

Thus, for a system scheme N with initial marking over a signature Σ, any interpretation with a Σ-structure S specifies for runs K of S a maximal set of local states in ${}^*K$. Often one would like to leave the initial marking for a place p open and specify it independently. For example, in figure 23, given the structure *accident*, one would like to freely choose a set of tuples $(\mathbf{driver}, \mathbf{car})$. Thus, the obvious idea is to add in the signature Σ_1 (figure 10) of the structure *accident* a symbol A of type $\underline{driver} \times \underline{cars}$ and to use it in the system net in figure 23 as

the initial marking of the place *accident*. Given an instantiation of the system net, A can then be instantiated by any set of tuples of the form (driver, car).

However, this idea falls short: already by type, the inscription (x, y) of the arrow starting at *accident* expects a tuple of driver and car, but A is a set. From a logical point of view, the predicate *accident* does not apply to a set of pairs of this form, but to each individual element of this set, so it is not $accident(\{a_1, a_2, \dots\})$, but rather

$$accident(a_1) \wedge accident(a_1) \wedge \dots \tag{20}$$

We denote this as

$$accident(elm(A)). \tag{21}$$

Figure 24 shows the system net of insurance cases with the use of the *elm* notation in the initial marking: the system net can be instantiated with arbitrary drivers, cars, and insurance types.

3.6 HERAKLIT schemes

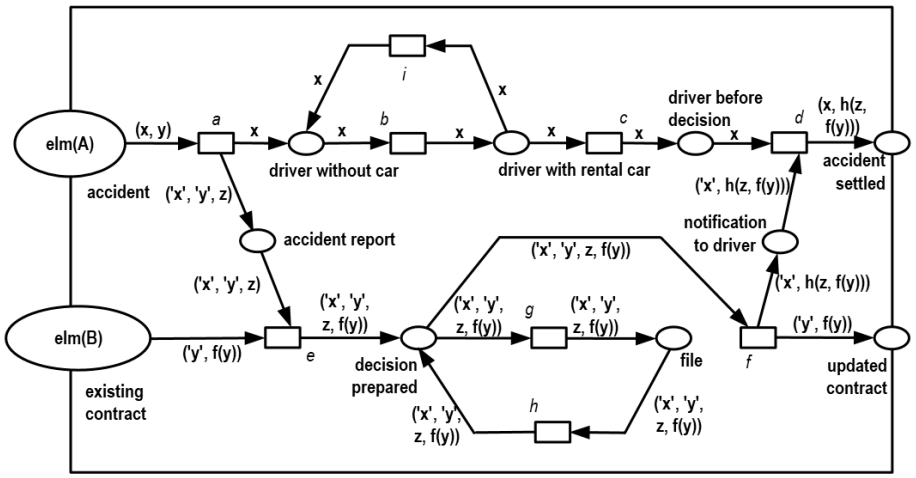
A HERAKLIT scheme describes behavior of elements of a relational structure S using the operations and relations of S. An example is the car insurance from chapter 2, with a fixed set of insurance types, possible damages, et cetera. One can now imagine other, special insurance companies for trucking companies, cab companies, or the like, which manage other data and whose functions are defined differently, but whose behavior is otherwise similar to the given behavior. We want to capture such variants of the insurance company in a *scheme* for insurance companies. Technically, this is accomplished very simply on the basis of the observation that, for example, the HERAKLIT systems of figures 19 and 20 use only elements of the *signature* of the relational structure *accident*, but not elements, functions, or relations of the relational structure *accident*.

Thus, the signature Σ of the systems of figures 19 and 20 is now interpreted not only with the relational structure *accident* from figure 10 and the relations from figure 11, but with arbitrarily chosen structures of this signature. This idea leads to the definition of HERAKLIT *schemes*. Here we compose events of a signature analogously to definition 34, but this time without reference to a special Σ-structure:

Definition 38 (HERAKLIT scheme). *Let Σ be a signature and let X be a set of variables for Σ. Let N be a net structure such that each atom of N is an event of Σ and X. Then N is a* HERAKLIT scheme over Σ.

This definition assumes only a net structure N, a signature Σ and variables. With an interpretation S of Σ, as in remark 4 at the end of section 2.8, Σ is a signature for S, and forms a HERAKLIT system according to definition 34.

Definition 39 (instantiation of a HERAKLIT scheme). *Let S be a relational structure, let Σ be a signature for S, and let N be a* HERAKLIT *scheme over Σ. The* HERAKLIT *scheme over S with Σ and X is the* instantiation of N with S.



sort symbols
drivers
names
cars
license-plates
damages
insurance-types
damage-sums

constant symbols
Bob, Alice: drivers
'Bob', 'Alice': names
A-01, A-02: cars
'A-01', 'A-02': license-plates
at-tree, in-ditch: damages
compreh., liability: insurance-types
€ 0, € 500, € 1000: damage sums
A: subset of drivers × cars
B: subset of license-plates × insurance-types

function symbols
f: cars ⟶ insurance-types
h: damages × insurance-types
⟶ damage-sums
' ': drivers ⟶ names
' ': cars ⟶ license-plates

variables
x: drivers
y: cars
z: damages

relation symbols:
accident with arity drivers × cars
accident-report with arity names × license-plates × damages
driver-without-car with arity drivers
exisiting-contract with arity license-plates × insurance-types
decision-prepared with arity names × license-plates × damages × insurance-types
driver-with-rental-car with arity drivers
driver-before-decision with arity drivers
file with arity names × license-plates × damages × insurance-types
notification-to-driver with arity names × damage-sums
updated-contract with arity license-plates × insurance-types
accident-settled with arity names × damage-sums

Fig. 24: two new constant symbols and the *elm* notation

Conceptual foundations of Heraklit

Heraklit has not been designed as a completely new modeling infrastructure, but is based in large part on well-established conceptual foundations of tra-

ditional areas of mathematics and informatics. We outline these relationships below; in doing so, we follow the structure of this handbook, namely, the three dimensions *architecture*, *statics*, and *dynamics*. Further details and references will be given in the second part of the handbook.

Architecture dimension

Modules with their *left* and *right interfaces* and their *composition* are the central concepts of the *composition calculus*: a finite set Σ of labels characterizes the set of all modules with Σ-labeled gates. Modules and their composition form a *monoid*; that is, modules behave first of all like words over an alphabet Σ. Moreover, the composition is compatible with the *refinement* of modules. There are many proposals for modules and their composition in the literature, but none has comparably elegant properties.

The composition calculus is new; without it, no convincing Heraklit modules for large systems could be formed.

Phrased as one tagline:

$$\text{Heraklit architecture} = \text{modules} + \text{composition} + \text{refinement}.$$

Statics dimension

Any mathematical construction can be conceptualized very abstractly as a *structure*, formed by sets and operations on these sets. Predicate logic is founded on this basic idea, and thereon concepts of informatics are based; Heraklit continues this approach.

Predicate logic extends abstract structures by *predicates* (technically, relations) to *relational structures* to capture properties of special structures. To this end, logical expressions are extended by quantifiers ("for all ..." and "there exists ...") and propositional operators ("and", "or", "not").

Phrased as one tagline:

$$\text{predicate logic} =$$
$$\text{relational structures} + \text{signatures} + \text{quantors} + \text{propositional logic}.$$

Heraklit adopts concepts of predicate logic, in particular relational structures, and different interpretations of one signature.

Specification languages (for example, ASM, B, RAISE/RSL, VDM, Z) describe relational structures as discussed in chapter 2. Relations are initially used in the development process to characterize properties of the the real or an imagined world; later in the design process, they are occasionally used for verification. This use of relational structures often generates very large specifications, comprising hundreds of pages or more.

Heraklit follows the tradition of specification languages with the use of "large" relational structures.

Phrased as one tagline:

$$\text{Heraklit statics} = \text{signatures} + \text{relational structures} + \text{multiple interpretations}.$$

Dynamics dimension

Specification languages provide only a limited ability to formulate behavior. In contrast, HERAKLIT provides expressive concepts for formulating behavior, modeling static and dynamic aspects as tightly interwoven and on an equal footing. Relations form local state components; several such local state components can be updated simultaneously. From the point of view of applications, logic is thus dynamized: in logic, a proposition is and remains either true or false. In HERAKLIT, a step can change this truth value.

Phrased as one tagline:

$$\text{HERAKLIT dynamics} = \text{steps} + \text{runs} + \text{locality}.$$

HERAKLIT now integrates all three dimensions. Again, phrased as one tagline on the technical level:

$$\text{HERAKLIT} = \text{modules} + \text{relational structures} + \text{local steps},$$

and on the level of calculi:

$$\text{HERAKLIT} = \text{composition calculus} + \text{logic} + \text{Petri nets}.$$

Conclusion

The first part of this handbook of HERAKLIT presents the conceptual and systematic foundations of the HERAKLIT modeling infrastructure, and emphasizes the scientific foundation of HERAKLIT in logic and general algebra.

The second part of the handbook delves into some important best practice aspects for using HERAKLIT in system development. These include specific structures and modules, for example digital twins, adapters, and design patterns, as well as ways to construct domain-specific classes of HERAKLIT modules. This part of the handbook will also include techniques for formulating and proving application-dependent properties of modules.

References

1. Cantor, G.: Beiträge zur Begründung der transfiniten Mengenlehre. Mathematische Annalen **46**, 481–512 (1895)
2. Fettke, P., Reisig, W.: HERAKLIT case study: 8-second hell (2020), HERAKLIT working paper, v1, December 12, 2020, `http://www.heraklit.org`
3. Fettke, P., Reisig, W.: HERAKLIT case study: adder (2020), HERAKLIT working paper, v1, December 5, 2020, `http://www.heraklit.org`
4. Fettke, P., Reisig, W.: HERAKLIT case study: parallel adder (2020), HERAKLIT working paper, v1, December 5, November 2020, `http://www.heraklit.org`
5. Fettke, P., Reisig, W.: HERAKLIT case study: retailer (2020), HERAKLIT working paper, v1, December 21, 2020, `http://www.heraklit.org`

6. Fettke, P., Reisig, W.: HERAKLIT case study: service system (2020), HERAKLIT working paper, v1, November 20, 2020, http://www.heraklit.org
7. Fettke, P., Reisig, W.: Modelling service-oriented systems and cloud services with HERAKLIT. In: Zirpins, C., Paraskakis, I., Andrikopoulos, V., Kratzke, N., Pahl, C., El Ioini, N., Andreou, A.S., Feuerlicht, G., Lamersdorf, W., Ortiz, G., Van den Heuvel, W.J., Soldani, J., Villari, M., Casale, G., Plebani, P. (eds.) Advances in Service-Oriented and Cloud Computing. pp. 77–89. Springer International Publishing, Cham (2021)
8. Fettke, P., Reisig, W.: HERAKLIT – in a nutshell (2021), HERAKLIT working paper, v1, August 2, 2021, http://www.heraklit.org
9. Reisig, W.: Associative composition of components with double-sided interfaces. Acta Informatica **56**(3), 229–253 (2019)