Systems engineering : a formal approach. Part II. Frameworks

Eindhoven University of Technology

Department of Mathematics and Computing Science

Systems Engineering: a Formal Approach

Part II : Frameworks

by

K.M. van Hee

93/10

Computing Science Note 93/10
Eindhoven, April 1993

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

Information Systems Engineering: a Formal Approach

by

K.M. Van Hee

March 30, 1993

This report is part of a preliminary version of a book that will be published.

Contents

Chapter 7

Introduction

In this part we define the formalisms we use to model and analyze systems. The concepts are already introduced in part I. As said before, a *formalism* consists of a mathematical framework and a language. The mathematical *framework* is represented by a *tuple* of mathematical entities, called *attributes*, and a set of *requirements* to be fulfilled by the attributes.

formalism
framework

As an illustration we define the framework of an automaton. An *automaton* is a tuple

$$(S, I, O, T, B)$$

where

- S, I and O are sets, called state space, input set and output set respectively,
- $T \in S \times I \nrightarrow S \times O$ is called the transition function,
- $B \subset S$ is called the set of initial states.

If we refer to an automaton, we give it a name, for instance A, and we may refer to an *attribute* of A by using the name of the automaton as a subscript for the attribute name. For instance S_A denotes the state space of the automaton A. If we are considering only one automaton then we drop the dependency of the name. We refer for instance to the "state space S of the automaton" if there is only one (maybe arbitrary) automaton in the context.

Note that a framework is in fact a set of functions with a common domain. Each function of a framework is called a *model*. The elements of the common domain are (traditionally) listed in a tuple. In the example above the framework is called "automaton" and it is formally defined by:

model

$$\begin{aligned}\{A \quad | \quad & A \textit{ is a function } \wedge \operatorname{dom}(A) = \{S, I, O, T, B\} \wedge \\ & A(S),\ A(I),\ A(O) \textit{ are sets } \wedge \\ & A(T) \in A(S) \times A(I) \nrightarrow A(S) \times A(O) \wedge \\ & A(B) \subset A(S)\}\end{aligned}$$

However, we will use the shorter notation as given above: note that $S_A = A(S)$. In other situations we often write f_x instead of $f(x)$, but it will always be clear from the context what the function and what the argument is.

We use a similar technique for specifying models in the specification language (cf. part V). There we use so-called *schemas* to specify models. The difference between a schema and a framework is, that in a schema each attribute has a *type*, i.e. a set to which the (function) value of the attribute belongs. So if we would have used a schema to specify the automaton, we should have specified to which sets $A(S)$, $A(I)$ and $A(O)$ belong. So schemas are "typed" and frameworks not. (Note that the term "tuple" is used in the specification language in a slightly different sense).

transition systems framework

object framework

actor framework

The first framework we present is called the *transition systems framework*. It formalizes the concept of discrete dynamic systems and their properties. The second framework is called the *object framework*. It formalizes the concepts of *simplexes* and *complexes*. These are *static* concepts used for modeling state spaces (or data bases). The third framework is called the *actor framework*, because it formalizes the concepts of actors and networks of actors. It generalizes the framework of Petri nets.

The three frameworks fit together, since the object framework is used to define the objects in the actor framework and the actor framework is used to model discrete dynamic systems.

Chapter 8

Transition systems framework

We start with some notations and preliminary definitions. Let $\mathcal{N}$ be the set

$\mathcal{N}$

$$\{x \subset \mathbb{N} \mid \forall i \in x, j \in \mathbb{N} : j < i \Rightarrow j \in x\}$$

So $\mathcal{N} = \{\emptyset, \{0\}, \{0,1\}, \{0,1,2\}, \ldots\}$. Also $\mathbb{N} \in \mathcal{N}$.

A *sequence* p is a function such that $\mathrm{dom}(p) \in \mathcal{N}$. The sequence is called infinite if $\mathrm{dom}(p) = \mathbb{N}$, otherwise it is finite with *length* $|p|$. The empty sequence with domain $\emptyset$ is denoted ϵ, other sequences with optional $\langle\ \rangle$ brackets. So

sequence

ϵ

$$\langle a, b, c\rangle = \{(0,a), (1,b), (2,c)\}.$$

Note that in fact $\epsilon = \emptyset$. For $i \in \mathrm{dom}(p)$ we write p_i instead of $p(i)$. If p is a sequence of length $n > 0$ and a is an arbitrary element, we define

$(\ ;\ )$

$$(p\,; a) = \langle p_0, \ldots, p_{n-1}, a\rangle.$$

For the sequence of length 0 we define $(\epsilon\,; a) = \langle a\rangle$.

Let A be a set. An A-sequence is a sequence with range A. A^n is the set of all A-sequences of length n, A^* the set of all finite A-sequences and A^∞ the set of all infinite A-sequences. Further we define A^+ as $A^* \cup A^\infty$.

A^n, A^*, A^∞, A^+

The *prefixes* of a sequence p are the sequences $p^i = \langle p_0, \ldots, p_{i-1}\rangle$ with $i \in \mathrm{dom}(p)$. By convention $p^0 = \epsilon$ and for finite p we have $p^{|p|} = p$. Note that $(p^i; p_i) = p^{i+1}$.

prefix p^i

A set $D \subset A^+$ is called *prefix-closed* if and only if

prefix-closed

$$\forall p \in D, i \in \mathrm{dom}(p) : p^i \in D$$

It is called *suffix-closed* if and only if

suffix-closed

$$\forall p \in A^\infty : (\forall i \in \mathrm{dom}(p) : p^i \in D) \Rightarrow p \in D$$

It is called *closed* if it is both prefix- and suffix-closed. A prefix-closed set D contains all prefixes of its elements; if a suffix-closed set D contains all prefixes of an infinite p, it contains p itself. Note that the empty set is closed, A^* is prefix-closed and A^∞ is suffix-closed.

We shall define now what we call *transition systems*. Transition systems consist of possible *events*, from some set E. Elements of E^+ are called *traces*. The important part of a transition system is a function $\mathcal{L}$, the *transition law*, telling what events come next when a certain finite trace has occurred.

transition system

Definition 8.1 A transition system is a pair $(E, \mathcal{L})$, where

event set E

- E is a set, called the *event set*,

transition law $\mathcal{L}$

- $\mathcal{L} \in E^* \to \mathbb{P}(E)$, called *transition law*.

trace
autonomous trace

A *trace* of $(E, \mathcal{L})$ is an element of E^+. An *autonomous trace* of $(E, \mathcal{L})$ is a sequence $p \in E^+$, such that

$$\forall i \in \text{dom}(p) : p_i \in \mathcal{L}(p^i)$$

maximal aut. trace

An autonomous trace p is called *maximal* if and only if it is infinite, or finite and such that $\mathcal{L}(p) = \emptyset$. The set of all autonomous traces of $(E, \mathcal{L})$ is called its *autonomous behavior*. The set of all maximal autonomous traces is called its *maximal autonomous behavior*.

autonomous behavior
maximal aut. behavior

□

For a trace p the set $\mathcal{L}(p)$ is the set of *possible extensions* of p. Note that ϵ is an autonomous trace for any transition system. It is easy to prove that the autonomous behavior of a transition system is closed. (The proof is an exercise.) The next theorem shows that any non-empty closed set of sequences can be described as the autonomous behavior of some transition system. This result is important because it proves that the autonomous behavior of a transition system can be obtained by restricting the set of all possible traces by so-called *dynamic constraints*.

Theorem 8.1 Let E be a set and let $D \subset E^+$ be non-empty and closed. Then there is a transition system $(E, \mathcal{L})$ with D as autonomous behavior.

Proof. Let $p \in E^*$. Then we set

$$\mathcal{L}(p) = \{e \mid (p\,; e) \in D\}.$$

We shall prove that $(E, \mathcal{L})$ has D as autonomous behavior. First let $q \in D$. We shall prove that q is an autonomous trace. Take $i \in \text{dom}(q)$. We have to prove that $q_i \in \mathcal{L}(q^i)$. By the definition of $\mathcal{L}$,

$$q_i \in \mathcal{L}(q^i) \iff (q^i; q_i) \in D.$$

Because $(q^i; q_i) = q^{i+1}$ we have

$$q_i \in \mathcal{L}(q^i) \iff q^{i+1} \in D.$$

The last assertion is true by the prefix-closedness of D. So q is indeed an autonomous trace of $(e, \mathcal{L})$.
Conversely, let q be an autonomous trace of $(E, \mathcal{L})$. We have to show that $q \in D$. Since D is non-empty and prefix-closed we have $\epsilon \in D$. Moreover $q^0 = \epsilon$, so

$$q^0 \in D.$$

If $q^i \in D$ for some $i \in \text{dom}(q)$, then $q_i \in \mathcal{L}(q^i)$, because q is an autonomous trace. Hence, by the definition of $\mathcal{L}$, we obtain $(q^i; q_i) \in D$, i.e. $q^{i+1} \in D$. This proves

$$\forall i \in \text{dom}(q) : q^i \in D \Rightarrow q^{i+1} \in D.$$

By induction, either q (if finite) or all its prefixes (if infinite) are in D. By the suffix-closedness of D we conclude that $q \in D$.
□

As a corollary, every prefix-closed subset of E^* is the intersection of E^* with the autonomous behavior of some transition system.

The maximal autonomous behavior of a transition system can be obtained by removing all sequences from the autonomous behavior that are prefixes of some autonomous trace. Conversely, the autonomous behavior is derived from the maximal autonomous behavior by adding all prefixes. A maximal autonomous trace has "maximal length", which explains the term "maximal". We shall now define some properties of transition systems.

Definition 8.2 An autonomous trace p of a transition system $(E, \mathcal{L})$ is said to *deadlock* if and only if $\mathcal{L}(p) = \emptyset$. A transition system $(E, \mathcal{L})$ is called *deterministic* if and only if its maximal autonomous behavior consists of a single sequence.
□

deadlock

deterministic transition system

Often we deal with transition systems where only the last event of a finite trace determines the set of next events. Such transition systems are called *memoryless* and their transition law can be characterized by a binary relation over E, called the *transition law relation*. We denote by $\ell(p)$ the last event of a non-empty trace $p \in E^*$, i.e.

$\ell(p)$

$$\ell(p) = p_{|p|-1}.$$

Definition 8.3 Let $(E, \mathcal{L})$ be a transition system. If

$$\forall p, q \in E^* : (p \neq \epsilon \wedge q \neq \epsilon \wedge \ell(p) = \ell(q)) \Rightarrow \mathcal{L}(p) = \mathcal{L}(q)$$

then $(E, \mathcal{L})$ is called *memoryless*. For such a memoryless transition system the binary relation $T\ell \subset E \times E$ satisfying

memoryless transition system

$$T\ell = \{(e_1, e_2) \mid \exists p \in E^* : \ell(p) = e_1 \ \wedge \ e_2 \in \mathcal{L}(p)\}$$

is called the *transition law relation*.
□

transition law relation $T\ell$

Theorem 8.2 Let $(E, \mathcal{L})$ be a memoryless transition system with transition relation $T\ell$. Then

$$\forall p \in E^* : \mathcal{L}(p) = \{e \in E \mid (\ell(p), e) \in T\ell\}.$$

Proof. If $e \in \mathcal{L}(p)$, then clearly $(\ell(p), e) \in T\ell$. Suppose that, conversely, $(\ell(p), e) \in T\ell$. Then there is a $q \in E^*$ such that $\ell(p) = \ell(q)$ and $e \in \mathcal{L}(q)$. Since $\mathcal{L}$ is memoryless, $\mathcal{L}(q) = \mathcal{L}(p)$, so $e \in \mathcal{L}(p)$.
□

time domain T
state space
σ, τ

Now we shall further elaborate our event set. Events have a *time* and *state* component; we assume that allowed event times are a subset of our standard time domain T, the non-negative real numbers. From now on, event sets have the shape $St \times T$, where St is a set called a *state space*. Given an event e in some E, $\sigma(e)$ will denote its state component and $\tau(e)$ its time component, so $e = (\sigma(e), \tau(e))$. We fix the start time of transition systems by setting for any transition law $\mathcal{L}$

$$\forall e \in \mathcal{L}(\epsilon) : \tau(e) = 0.$$

We introduce two behavioral properties of transition systems.

monotonous transition system

Definition 8.4 A transition system is *monotonous* if and only if

$$\forall p \in E^* : \forall e \in \mathcal{L}(p) : \tau(e) \geq \tau(\ell(p)).$$

eager autonomous trace

An autonomous trace p of a transition system is called *eager autonomous traceeager* if and only if

$$\forall i \in \text{dom}(p) : \forall e \in \mathcal{L}(p^i) : \tau(p_i) \leq \tau(e).$$

eager autonomous behavior

The set of all eager autonomous traces is called the *eager autonomous behavior*.
□

From now on we will only consider monotonous transition systems. The eager autonomous behavior of $(E, \mathcal{L})$ is the autonomous behavior of $(E, \mathcal{L}')$, where $\mathcal{L}'$ is derived from $\mathcal{L}$ by deleting all elements from $\mathcal{L}(p)$ for which the time component is not minimal. Strange enough, a maximal autonomous trace of $(E, \mathcal{L}')$ does not have to be an eager maximal autonomous trace of $(E, \mathcal{L})$, because it is possible to construct an $\mathcal{L}$ such that $\mathcal{L}(p)$ is an infinite set without minimal element for some $p \in \text{dom}(\mathcal{L})$. Then $\mathcal{L}'(p)$ is empty, so p has no eager continuation. However, if $\mathcal{L}(p)$ is finite and non-empty, then p has an eager continuation.

strongly memoryless transition system

Definition 8.5 A transition system is called *strongly memoryless* if and only if

$$\forall p, q \in E^* : \sigma(\ell(p)) = \sigma(\ell(q)) \Rightarrow \mathcal{L}(p) = \mathcal{L}(q).$$

□

Definition 8.6 Let $(E, \mathcal{L})$ be a transition system. A *livelock* of $(E, \mathcal{L})$ is an infinite autonomous trace p such that **livelock**

$$\exists t \in T : \forall i \in \operatorname{dom}(p) : \tau(p_i) \leq t.$$

□

Our idea of a livelock is a trace that makes infinitely many transitions in finite time.

We now define the *path* of a trace: the function giving the state for each point in time. A time point t is mapped to the state resulting from the last event that occurred before or at t.

Definition 8.7 Let p be a non-empty trace of a transition system $(St \times T, \mathcal{L})$. The *path* of p is the function $W_p \in T \to St$ satisfying $W_p(t) = \sigma(p_i)$, where i is defined by **path W_p**

$$i \in \operatorname{dom}(p) \ \wedge \ \tau(p_i) \leq t \ \wedge \ \forall j \in \operatorname{dom}(p) : j > i \Rightarrow \tau(p_j) > t.$$

□

Thus if a trace contains more events with the same time $\tau(p_i)$ the last one determines W_p. Note that a trace has no path if it livelocks, since it has no last event for every $t \in T$; we could define its path on a subset of T.

We conclude by defining some *similarity relations* on the set of transition systems. They are used later to compare transition systems. For instance, if the events of two transition systems have different names but their autonomous behavior is the same after renaming the events, we would like to call them "similar". In fact they are *isomorphic*, which is the strongest form of similarity.

Definition 8.8 Let A and B be transition systems. Further let $X \subset E_A \times E_B$ be a given binary relation.

- $p \in E_A^*$ and $q \in E_B^*$ are called X*-similar* (notation $\sim_X$) if and only if **X-similar $\sim_X$**

$$\operatorname{dom}(p) = \operatorname{dom}(q) \ \wedge \ \forall i \in \operatorname{dom}(p) : (p_i, q_i) \in X.$$

- A is called *similar* to B with respect to X if and only if **similar**

$$\begin{aligned}&\forall p \in E_A^*, q \in E_B^* : p \sim_X q \Rightarrow \\ &\qquad \forall x \in \mathcal{L}_A(p) : \exists y \in \mathcal{L}_B(q) : (x, y) \in X\end{aligned}$$

- A and B are called *bisimilar* with respect to X if and only if A is similar to B with respect to X and B is similar to A with respect to **bisimilar**

$$X^{-1} = \{(y, x) \mid (x, y) \in X\}.$$

□

The following theorem establishes an obvious relationship between the autonomous behaviors of two similar transition systems.

Theorem 8.3 Let A and B be two transition systems such that A is similar to B with respect to $X \subset E_A \times E_B$. Further let P_A and P_B be the autonomous behaviors of A and B respectively. Then:

$$\forall p \in P_A : \exists q \in P_B : p \sim_X q.$$

Proof. Let $p \in P_A$. Clearly $\epsilon \sim_X \epsilon$ and $p_0 \in \mathcal{L}_A(\epsilon)$. So $\exists q_0 \in \mathcal{L}_B(\epsilon)$: $(p_0, q_0) \in X$. We apply induction on the length of an autonomous trace. Assume the assertion holds for autonomous traces of length n in P_A. Let $p = (p^n; p_n) \in P_A$. By the induction hypothesis we have the existence of an autonomous trace $q \in P_B$ with length n such that $p^n \sim_X q$. Since $p_n \in \mathcal{L}(p^n)$ we have, by the similarity, the existence of $q_n \in \mathcal{L}(q)$ such that $(p_n, q_n) \in X$. Hence $(p^n; p_n) \sim_X (q; q_n)$ and $(q; q_n) \in P_B$.
□

The next theorem establishes two important properties of similarity.

Theorem 8.4 Similarity is *reflexive* and *transitive*.

Proof. Clearly a transition system $(E, \mathcal{L})$ is similar to itself with respect to the relation $\{(x, x) \mid x \in E\}$. This is the reflexivity. To prove the transitivity, let A, B and C be transition systems and let A be similar to B with respect to X and let B be similar to C with respect to Y. Further let $Z = Y * X$ be defined by

$$Z = \{(x, z) \mid \exists y \in E_B : (x, y) \in X \ \wedge \ (y, z) \in Y\}.$$

Finally let $p \in E_A^*$ and $r \in E_C^*$ such that $p \sim_Z r$. Then there is a $q \in E_B^*$ with $p \sim_X q$ and $q \sim_Y r$. From the two similarities we derive

$$\forall x \in \mathcal{L}_A(p) : \exists z \in \mathcal{L}_C(r) : \exists y \in \mathcal{L}_B(q) : (x, y) \in X \ \wedge \ (y, z) \in Y.$$

Hence

$$\forall x \in \mathcal{L}_A(p) : \exists z \in \mathcal{L}_C(r) : (x, z) \in Z.$$

So A is similar to C with respect to Z.
□

Note that, under the assumptions of the theorem, there is for each autonomous trace p of A an autonomous trace q of B and an autonomous trace r of C such that $p \sim_X q$ and $p \sim_Z r$. Moreover there is an autonomous trace $\tilde{r}$ of C such that $q \sim_Y \tilde{r}$. However, we may not conclude that $q \sim_Y r$. If we know that there is only one r such that $p \sim_Z r$, then this problem is solved. This is the case if the relations over the event sets are *graphs* of functions, i.e. if there is a function f such that $X = \{(x, y) \mid x \in E_A \ \wedge \ y = f(x)\}$ and a function g with $Y = \{(y, z) \mid y \in E_B \ \wedge \ z = g(y)\}$.

graph of function

Symmetry is lacking in the similarity relation and therefore it is not an equivalence relation. Bisimilarity is symmetrical, so that relation should be an equivalence relation.

Theorem 8.5 Bisimilarity is an *equivalence* relation.

Proof. The reflexivity is trivial. The symmetry follows from the fact that $(X^{-1})^{-1} = X$. The transitivity follows from the fact that $X^{-1} * Y^{-1} = (Y * X)^{-1}$.
□

An important case is the situation where $X \subset E_A \times E_B$ is the graph of a bijective function f. Then bisimilarity of A and B means that A and B are *isomorph* and f induces a bijective function between the autonomous behaviors of the two transition systems.

isomorph

According to the definitions so far, similarity of transition systems establishes relations between autonomous traces of the same length only. However, if one transition system needs several transitions to *simulate* one transition of another transition system, then these transition systems are not (bi)similar according to our definitions. In order to allow transition systems to be called "similar" in such cases we introduce the notion of an extended transition law. Let $(E, \mathcal{L})$ be a transition system. The *extended transition law* $\tilde{\mathcal{L}}$ is defined by:

$$\tilde{\mathcal{L}}(p) = \mathcal{L}(p) \cup \{\ell(p)\}.$$

So we added "dummy" events, i.e. repetitions of events. We say transition system A is *weakly (bi)similar* to B if the (bi)similarity holds with respect to the extended transition laws. In particular $(E, \mathcal{L})$ and $(E, \tilde{\mathcal{L}})$ are bisimilar with respect to the identity relation $(\{(x, x) \mid x \in E\})$.

A nice application of the similarity is that there is for each non-memoryless transition system a memoryless transition system that is similar to it.

Theorem 8.6 Let an arbitrary transition system $(E, \mathcal{L})$ be given. Then there is a memoryless transition system $(E', \mathcal{L}')$, that is similar to $(E, \mathcal{L})$ with respect to C, where

- $E' = E^*$,
- C satisfies: $(e', e) \in C \Leftrightarrow \ell(e') = e$.

Proof. The proof is an exercise.
□

Chapter 9

Object framework

In this chapter we introduce a framework to model complexes. Complexes will be used to define tokens and therefore to define the state space of a system. We start with the definition of a *class model*, which contains all information to define the structure of complex classes. A class model is an *abstract syntax* for class diagrams. (In the database literature the term *database model* is also used instead of class model.) Then we define the concept of an *instance model*, which contains the information to define simplexes and complexes. Afterwards we introduce *constraints*. Constraints are used to define properties of complexes. Often the systems engineer only wants to consider a subset of a complex class instead of the entire class. The transition law of the system should guarantee that in all reachable states the complexes in the tokens satisfy the constraints. So the constraints are the *invariants* for the transition law of the system. It is a *proof obligation* for the systems engineer to show that the constraints are indeed invariant. There are several kinds of constraints that occur frequently in models such as *relationship*, *inheritance* and *tree* constraints. All constraints have a graphical notation in the class diagram. There are many other constraints possible (and often necessary) in models, they can be expressed with predicate calculus (cf. part III). A complete *object model* consists of

- a class model,
- an instance model,
- a set of constraints.

Definition 9.1 A *class model* is a 7-tuple **class model**

$$(CN,\ SN,\ RN,\ DM,\ RG,\ CB,\ CR)$$

where

- CN, SN and RN are mutually disjoint sets of names of complex classes, simplex classes and relationship classes respectively. There is one element in CN called the *universal* complex class. ***CN, SN, RN*** **universal complex class**

DM, *RG*

- The functions
$$DM \in RN \rightarrow SN$$
and
$$RG \in RN \rightarrow SN$$
give a domain simplex class (DM) and a range simplex class (RG) to every relationship.

CB, *CR*

- The functions
$$CB \in CN \rightarrow \mathbb{P}(SN)$$
and
$$CR \in CN \rightarrow \mathbb{P}(RN)$$
determine the body simplex classes (CB) and the relationship classes (CR) contained in a complex class, such that $\forall n \in CN$: $\forall r \in CR(n)$:
$$DM(r) \in CB(n) \wedge RG(r) \in CB(n).$$
Further $CB(universal) = SN$ and $CR(universal) = RN$.

□

Note that the bodies of complex classes may overlap. All complex classes are *subclasses* of the universal complex class. A class model is (partly) defined in a class diagram as shown in part I. A class diagram may display constraints as well. Note that complex classes may share simplex classes and relationships.

For the production/consumption example with class model X, the class diagram of figure 5.12 and the table of figure 5.13 show the following (we have written CN instead of CN_X, etc.):

$$\begin{array}{rcl}
CN &=& \{Consumer, Machine, Order, PendingOrder, Operation\} \\
SN &=& \{consumer, order, amount, machine, operation, duration, speed\} \\
RN &=& \{p,\ q,\ r,\ s,\ t,\ u\} \\
DM &=& \{(p, order), (q, order), (r, operation), (s, operation), ...\} \\
RG &=& \{(p, consumer), (q, amount), (r, order), (s, machine), ...\} \\
CB &=& \{(Consumer, \{consumer\}), (Machine, \{machine, speed\}), ...\} \\
CR &=& \{(Comsumer, \emptyset), (Machine, \{u\}), (Order, \{q\}), ...\}.
\end{array}$$

We shall now define the instance model of an object model.

instance model

Definition 9.2 Let a class model be given. An *instance model* is a 2-tuple (*sim*, *com*) where

sim

- *sim* is a set-valued function with dom(*sim*) = SN and for $n \in SN$ $sim(n)$ is the set of representations of all possible simplexes in the "world";

- *com* is a set-valued function that assigns to all $n \in CN$ the set of representations of possible complexes, where

com

$$\begin{aligned} com(n) = \{c \mid\ & c\ \textit{is a function} \wedge \mathrm{dom}(c) = CB(n) \cup CR(n) \wedge \\ & \forall m \in CB(n) : c(m) \subset sim(m) \wedge c(m)\ \textit{is finite} \wedge \\ & \forall r \in CR(r) : c(r) \subset c(DM(r)) \times c(RG(r))\} \end{aligned}$$

□

Note that if all pairs of complex classes differ at least in one simplex class or one relationship class, then we can determine the class of a given complex, because the domain of a complex contains the essential information.

The function *sim* specifies which values are used to denote "atomic" entities of a certain kind in the world. The values have to come from a *value universe U* that will be defined in part V. For instance *sim*(*chair*) is the set of representations of all possible chairs in the world. Although two different simplex classes may have the same set of representations, we can always distinguish them because we assume the class of a simplex is always known. Given these representations, we can define complex classes by means of the function *com* that determines which "molecular" entities in the world belong to a certain complex class. A complex is defined as a function that assigns a finite set of simplexes to each name of a simplex class and a finite set of simplex pairs to each name of a relationship class. As we have seen in part I, we can consider a complex as a graph with simplexes as nodes and relationships as edges, labeled with the names of the relationships; this is just another way of representing the function. The relationships may only connect simplexes that belong to the complex. The elements of simplex classes and complex classes are called *instances* of these classes.

instance

In the production/consumption example of figure 5.13 a possible instance (i.e. a complex) $c \in com(Operation)$ is

$$\begin{aligned} c \ = \ & \{(operation, \{operation1\}), \\ & (machine, \{machine3\}), \\ & (order, \{order123\}), \\ & (amount, \{10\}), \\ & (speed, \{35\}), \\ & (q, \{(order123, 10)\}), \\ & (r, \{(operation1, order123)\}), \\ & (s, \{(operation1, machine3)\}), \\ & (u, \{(machine3, 35)\}) \\ & \}. \end{aligned}$$

In this example the second element of each pair is always a singleton. (This is because all relations are functional and total and because there is a root simplex class due to a tree constraint.)

In specifications it is sometimes cumbersome to use the (function) representation of a complex class as defined above. Depending on the role of the complexes in processors the systems engineer can use a different representation. If he does so, he needs to define for each complex class n a bijective *representation function*

representation function

$$RF_n \in U \rightarrow com(n).$$

The next step in the definition of an object model is the definition of constraints. First we introduce, for notational convenience, some auxiliary functions and a set called the *object universe*, which contains all possible complexes in the universal complex class.

Definition 9.3 Let a class model and an instance model be given.

object universe OU

- $OU = \bigcup_{n \in CN} com(n)$ is the *object universe*.

$R_{r,c}$

- $R_{r,c}$ is, for each complex $c \in OU$ and relationship r, a function with $\text{dom}(R_{r,c}) = c(DM(r))$, that assigns to a simplex $x \in c(DM(r))$ the set of simplexes that have a relationship of class r with it:

$$R_{r,c}(x) = \{y \mid y \in c(RG(r)) \ \wedge \ (x,y) \in c(r)\}.$$

$D_{r,c}$

- $D_{r,c}$ is a similar function with $\text{dom}(D_{r,c}) = c(RG(r))$, but it concerns the domain of a relationship. For a simplex $y \in c(RG(r))$:

$$D_{r,c}(y) = \{x \mid x \in c(DM(r)) \ \wedge \ (x,y) \in c(r)\}.$$

□

constraint

In general, a constraint is a Boolean function (i.e. a function with range {*true*, *false*}) over a complex class. A Boolean function is expressed by a predicate with a free variable that denotes a complex.

Constraints will be expressed in the specification language (cf. part V). Here we will give an example of a constraint. Regard the class model displayed in figure 4.16. There we required that the person to which the student refers by means of the relationships a and b and the person he refers to by c and e are the same. This can be expressed by following predicate with u as an arbitrary universal complex:

$$\forall x \in u(student) : \ R_{b,u}(R_{a,u}(x)) = R_{e,u}(R_{c,u}(x)).$$

Here we silently extended the domain of R in order to apply the function to sets of complexes in the obvious way. Furthermore, we did not express that $R_{b,u}(R_{a,u}(x))$ and $R_{e,u}(R_{c,u}(x))$ should be singletons. There are more efficient notations possible, but in essence every constraint can be expressed like this one. A more efficient notation is used in part I, where the dependency on the complex u is deleted, where $u(n)$ is replaced by n (for a simplex class n) and where the functions $R_{n,u}$ and $D_{n,u}$ are replaced by n and n^{-1} respectively. With these conventions the formula above will read in the specification language:

$$\forall x : \mathit{student} \bullet b(a(x)) = e(c(x)).$$

There are several constraints that occur frequently in practice and therefore they have a representation in the class diagram, as shown in part I. We call them the *standard constraint*; they consist of relationship, inheritance and tree constraints. The *relationship constraint* consist of cardinality, key and exclusion constraints.

standard constraint
relationship constraint

Definition 9.4 Let a class model and an instance model be given. The function

$$FC \in RN \to \mathbb{P}(\{\mathit{total}, \mathit{functional}, \mathit{injective}, \mathit{surjective}\})$$

denotes the *cardinality* constraints. The cardinality constraints imply a set of requirements for a relationship r and each complex c with $r \in \mathrm{dom}(c) \cap RN$:

cardinality constraint *FC*

- if *total* $\in FC(r)$ then:

totality

$$\forall x \in c(DM(r)) : R_{r,c}(x) \neq \emptyset$$

- if *functional* $\in FC(r)$ then:

functionality

$$\forall x \in c(DM(r)) : \#R_{r,c}(x) \leq 1$$

- if *injective* $\in FC(r)$ then:

injectivity

$$\forall y \in c(RG(r)) : \#D_{r,c}(y) \leq 1$$

- if *surjective* $\in FC(r)$ then:

surjectivity

$$\forall y \in c(RG(r)) : D_{r,c}(y) \neq \emptyset$$

□

Definition 9.5 Let a class model and an instance model be given. The functions

$$\begin{aligned} DK &\in SN \to \mathbb{P}(\mathbb{P}(RN)) \\ RK &\in SN \to \mathbb{P}(\mathbb{P}(RN)) \\ DX &\in SN \to \mathbb{P}(\mathbb{P}(RN)) \\ RX &\in SN \to \mathbb{P}(\mathbb{P}(RN)) \end{aligned}$$

respectively denote the *domain key* constraints, the *range key* constraints, the *domain exclusion* constraints and the *range exclusion* constraints. They should satisfy

domain key constraint *DK*
range key constraint *RK*
domain exclusion constraint *DX*
range exclusion constraint *RX*

$$\forall n \in SN : \forall r \in \bigcup DK(n) : DM(r) = n$$
$$\forall n \in SN : \forall r \in \bigcup RK(n) : RG(r) = n$$
$$\forall n \in SN : \forall r \in \bigcup DX(n) : DM(r) = n$$
$$\forall n \in SN : \forall r \in \bigcup RX(n) : RG(r) = n.$$

A domain key constraint is an element of $DK(s)$ and analogously for the other constraints. For an arbitrary complex c these constraints imply respectively: $\forall n \in SN : \forall C \in DK(n) : \forall x, y \in c(DM(r)) :$

$$(\forall r \in C : R_{r,c}(x) = R_{r,c}(y) \neq \emptyset) \Rightarrow x = y.$$

$\forall n \in SN : \forall C \in RK(n) : \forall x, y \in c(RG(r)) :$

$$(\forall r \in C : D_{r,c}(x) = D_{r,c}(y) \neq \emptyset) \Rightarrow x = y.$$

$\forall n \in SN : \forall C \in DX(n) :$

$$\forall r_1, r_2 \in C : \forall x \in c(DM(r_1)) : R_{r_1,c}(x) = \emptyset \ \vee \ R_{r_2,c}(x) = \emptyset.$$

$\forall n \in SN : \forall C \in RX(n) :$

$$\forall r_1, r_2 \in C : \forall x \in c(RG(r_1)) : D_{r_1,c}(x) = \emptyset \ \vee \ D_{r_2,c}(x) = \emptyset.$$

□

Note that totality and surjectivity are each other's counterparts and also functionality and injectivity, in the sense that the roles of R and D are exchanged. Further note that totality and injectivity together imply a domain key constraint and similarly surjectivity and functionality imply a range key constraint.

For the production/consumption example the class diagram of figure 5.12 and the table of figure 5.13 show the constraints as displayed in figure 9.1.

The key constraints can be used to find representations for simplexes: we may choose as representation a combination of the representations of the simplexes involved in a key constraint. If for instance the simplex class *operation* is the domain simplex class of two relationships s and r with range simplex classes *machine* and *order* respectively and if s and r form a domain key constraint, then we can use the pairing of the representations of *machine* and *order* as the representation for *operation*. Note that we cannot combine domain and range key constraints into one "key constraint": if there is a relationship that connects a simplex class with itself, it is ambiguous if the domain or the range of this relationship should be used.

The next kind of constraints we consider are the *inheritance* constraints. First we introduce the notion of a relationship path.

relationship path

Definition 9.6 Let a class model and an instance model be given. A sequence of relationship class names $\langle r_1, \ldots, r_k \rangle$ is called a *relationship path* if and only if all elements are different and

$$\forall i \in \{1, \ldots, k-1\} : RG(r_i) = DM(r_{i+1}).$$

□

FC	$\{(p, \{functional, total\}),$ $(q, \{functional, total\}),$ $(r, \{functional, total\}),$ $(s, \{functional, total\}),$ $(t, \{functional, total\}),$ $(u, \{functional, total\})$ $\}$
DK	$\{(comsumer, \emptyset),$ $(order, \emptyset),$ $(amount, \emptyset),$ $(machine, \emptyset),$ $(operation, \{\{r, s\}\}),$ $(duration, \emptyset),$ $(speed, \emptyset)$ $\}$
RK DX RX	$\{(consumer, \emptyset),$ $(order, \emptyset),$ $(amount, \emptyset),$ $(machine, \emptyset),$ $(operation, \emptyset),$ $(duration, \emptyset),$ $(speed, \emptyset)$ $\}$
IC	$\emptyset$
TC	$\{(Consumer, consumer),$ $(Machine, machine),$ $(Order, order),$ $(PendingOrder, order),$ $(Operation, operation)$ $\}$
PC	$\emptyset$

Figure 9.1: Constraints for the production/consumption system.

inheritance relationship
inheritance constraint *IC*

Definition 9.7 Let a class model and an instance model be given. A relationship is called an *inheritance relationship* if it is total, functional and injective. An *inheritance constraint* IC is a set of inheritance relationships $IC \subset RN$ such that:

- the graph with nodes in SN and edges $\{(DM(r), RG(r)) \mid r \in IC\}$ is a directed acyclic graph,
- for all complexes c and all relationship paths $\langle r_1, \ldots, r_k \rangle$ and $\langle p_1, \ldots, p_l \rangle$ in $\mathrm{dom}(c) \cap IC$, with

  $$DM(r_1) = DM(p_1) \wedge RG(r_k) = RG(p_l)$$

  the following predicate should hold: $\forall x \in c(DM(r_1))$:

  $$R_{r_k,c}(R_{r_{k-1},c}, \ldots, R_{r_1,c}(x)) = R_{p_l,c}(R_{p_{l-1},c}, \ldots, R_{p_1,c}(x)).$$

□

An inheritance constraint IC is a set of total, functional and injective relationships with the property that they form a directed acyclic graph. If we follow two different paths formed of inheritance relationships, going from one simplex to another, then it should hold for each complex, that if we start in the first simplex and we follow the paths, then both paths will end in the second simplex.

An inheritance structure induces a partial order on the simplex classes. Sometimes it is useful to combine the inheritance constraint with the exclusion constraint. For instance, in the example of figure 4.16 we might want to exclude the states in which a school person is a student and a teacher at the same time, therefore we could add the set $\{c, d\}$ to $RX(schoolperson)$. Note that an inheritance constraint may contain several different class hierarchies. Inheritance can be used to obtain efficient representations in a database. We will discuss this topic in chapter 13.

The last kind of constraint we consider are the *tree constraint*. A tree constraint specifies that the complexes in a class have a tree-like structure, i.e. there is one simplex, called the *root simplex*, from which all the other simplexes can be reached by an *undirected* path of relationships. Furthermore the complex may contain only one simplex of the root simplex class. For a complex c we define

cont(*c*)

$$cont(c) = \bigcup_{k \in \mathrm{dom}(c) \cap SN} c(k),$$

the set of all simplexes enclosed in the complex c.

tree constraints *TC*

Definition 9.8 A *tree constraint* is an element of the function

$$TC \in CN \twoheadrightarrow SN$$

where $\forall n \in \mathrm{dom}(TC)$:

- $TC(n) \in CB(n)$ is called the *root simplex class*, root simplex class
- $\forall c \in com(n) : \#c(TC(n)) = 1$; this element is called the *root simplex*, root simplex
- $\forall c \in com(n) : \forall x \in cont(c)$:
  there is a sequence of simplexes $\langle z_1, \ldots, z_k \rangle$ in $cont(c)$ such that:
  - $z_1 \in c(TC(n)) \wedge z_k = x$
  - $\forall i \in \{1, \ldots, k-1\} : \exists r \in \mathrm{dom}(c) \cap RN$:

$$(z_i, z_{i+1}) \in c(r) \vee (z_{i+1}, z_i) \in c(r)$$

□

Note that z_1 is the root simplex.

Having defined the class model, instance model and constraints, we are now ready to define an object model.

Definition 9.9 An *object model* is a 4-tuple $(CM,\ IM,\ SC,\ PC)$, where object model

- CM is a class model, CM
- IM is an instance model, IM
- SC is a tuple of standard constraints, i.e. SC

$$SC = (FC,\ DK,\ RK,\ DX,\ RX,\ IC,\ TC)$$

  where $(FC,\ DK,\ RK,\ DX,\ RX)$ denote the relationship constraints, IC is an inheritance constraint and TC denotes the tree constraints,
- PC is a Boolean function, called the *free constraint*, such that: $\mathrm{dom}(PC) = CN$ and free constraint PC

$$\forall n \in CN : PC(n) \in com(n) \to \{true, false\}$$

□

Instead of the set of all complexes of a class n, $com(n)$, we are often interested in the subset of complexes that satisfy the standard and free constraints. In the following chapter an object model is used to define the state space. Note that there may be constraints on states, that cannot be expressed as constraints on objects. For instance, the constraint that "no two tokens have a common simplex of a certain class", is such a *global constraint*. global constraint

Chapter 10

Actor framework

In the preceeding chapter we introduced the object framework as a layer on top of the (transition) systems framework to facilitate the modeling of a state space. Here we introduce the *actor framework* to make the modeling of transition relations more easy. The actor framework may be regarded as a next layer, since it uses concepts from the object framework. However, the coupling between the two frameworks is rather loose, which implies that the systems engineer is free to start with object modeling or with actor modeling as he likes.

First we introduce the concept of an actor. We distinguish *flat nets* and *hierarchical actors*. Inside a flat net all actors are processors, so there are no actors that represent a (sub)network. A hierarchical actor is a network that contains also non-elementary actors and such a network can be transformed into a flat net. We first define a *flat net model*, which is in fact a formal definition of the actor networks (without hierarchy) as displayed in chapter 5. Secondly we define a hierarchical actor structure. Again this is a formal definition of the actor networks from chapter 5, now also including the non-elementary actors. Then we define the *actor model*, which encloses an object model and a flat net model. Subsequently we define how an actor model determines a transition relation. Finally we give some properties of actor models.

Note that a hierarchical net model is an *abstract syntax* for the diagrams of actor networks we draw and that a flat net model is a special case of a hierarchical net model. The semantics is defined by the actor model, in particular the transition system defined by the actor model.

Definition 10.1 A *flat net model* is a 6-tuple (L, P, C, I, O, M) where **flat net model**

- L (locations) is a finite set of *places*, L
- P is a finite set of *processors*, P
- C is a finite set of *connector* names, C
- $I \in P \rightarrow \mathbb{P}(C)$ assigns to a processor a set of input connectors, I
- $O \in P \rightarrow \mathbb{P}(C)$ assigns to a processor a set of output connectors, O

M • $M \in P \to (C \nrightarrow L)$ (match) assigns the connectors of each processor to places,

such that:

- L, P and C are mutually disjoint,
- $\forall p \in P : \mathrm{dom}(M_p) \subset I(p) \cup O(p)$,
- $\forall p \in P$: $I(p) \cap O(p) = \emptyset$, i.e. no connector name is input and output for the same processor,
- $\forall p \in P$: $I(p) \neq \emptyset$, i.e. each processor has at least one input connector,
- $\bigcup_{p \in P}(I(p) \cup O(p)) = C$, i.e. there are no "dangling" connectors.

□

It is easy to see how a flat net model can be represented graphically (cf. figure 5.5). Note that the same connector name may occur at different processors and that processors do not need to have output places. Further note that there may be processors with unconnected connectors. Such actors are called *open actors* while the others are called *closed*. Only closed (flat) actors will have a state space and a transition relation associated to them. Open actors are considered to be *components* out of which one can make closed actors. A single processor, without places, is an example of an open actor. A processor and a place may be connected by more than one input or output connector. Note that channels and stores are just special places and therefore we do not consider them here.

open actor
closed actor

We will now define the hierarchical net model; it is just a generalization of the flat net model.

hierarchical net model

Definition 10.2 A *hierarchical net model* is a 10-tuple

$$(L, P, A, C, I, O, top, HA, HL, M)$$

such that:

- L — L (locations) is a finite set of places,
- P — P is a finite set of processors,
- A — A is a finite set of actors, $P \subset A$,
- C — C is a finite set of connector names,
- I — $I \in A \to \mathbb{P}(C)$ assigns to each actor a set of input connectors,
- O — $O \in A \to \mathbb{P}(C)$ assigns to each actor a set of output connectors,
- top — $top \in A$ is called the top level actor,
- HA — $HA \in A \backslash \{top\} \to A$ assigns every processor or actor to an (enclosing) actor, except for top,

- $HL \in L \rightarrow A$ assigns every place to an actor, *HL*
- $M \in A \backslash \{top\} \rightarrow (C \mapsto L \cup C)$ (match) assigns connectors of actors to places or connectors, *M*

such that

- L, A and C are mutually disjoint sets,
- $\forall a \in A : I(a) \cap O(a) = \emptyset$, i.e. no connector name is input and output for the same actor,
- $\forall p \in P : I(p) \neq \emptyset$, i.e. all processors must have at least one input connector,
- $\forall a \in A \backslash \{top\} : dom(M_a) = I(a) \cup O(a)$, i.e. all connectors of an actor (except for *top*) are connected,
- $C = \bigcup_{a \in A}(I(a) \cup O(a))$, i.e. there are no dangling connectors,
- $\forall a \in A \backslash \{top\} : \exists k \in I\!N : HA^k(a) = top$, i.e. all actors are directly or indirectly mapped to the top level actor,
- $\forall \ell \in L : \exists k \in I\!N : HA^k(HL(\ell)) = top$, i.e. all places are directly or indirectly mapped to the top level actor,
- $\forall a \in A \backslash P :$

  $$\begin{array}{l} \forall c \in I(a) : \exists b \in A : \exists d \in I(b) : M_b(d) = c \;\wedge \\ \forall c \in O(a) : \exists b \in A : \exists d \in O(b) : M_b(d) = c, \end{array}$$

  so the connectors of a "high level" actor are internally connected to an actor: input to input and output to output,
- $\forall a \in A \backslash \{top\} : \forall c \in I(a) \cup O(a) :$

  $$\begin{array}{l} M_a(c) \in L \Rightarrow HL(M_a(c)) = HA(a) \;\wedge \\ M_a(c) \in C \wedge c \in I(a) \Rightarrow M_a(c) \in I(HA(a)) \;\wedge \\ M_a(c) \in C \wedge c \in O(a) \Rightarrow M_a(c) \in O(HA(a)), \end{array}$$

  which means that if a connector is connected to a place then this place belongs to the same higher-level actor as the actor itself and if a connector of an actor is connected to another connector, then this last one is a connector of the higher-level actor and of the same kind.

□

Note that the second last requirement of the above definition does not imply that a connector of a non-elementary actor a is internally connected to an actor *enclosed* in a: for this we also need the last requirement. The top level actor (called *top*) is the only actor that may have unconnected connectors. An actor that has unconnected connectors is

open actor **closed actor** **stores**

called *open*, otherwise it is *closed*. Further note that we did not introduce *stores* yet. A store is just a special place in the sense that it is always connected to one input and one output connector for each processor to which it is connected. We consider it to be syntactical "sugar".

In figure 10.1 we display a hierarchical net model graphically. In figure 10.2 the same net model is presented in table format. It is easy to verify that all requirements are satisfied.

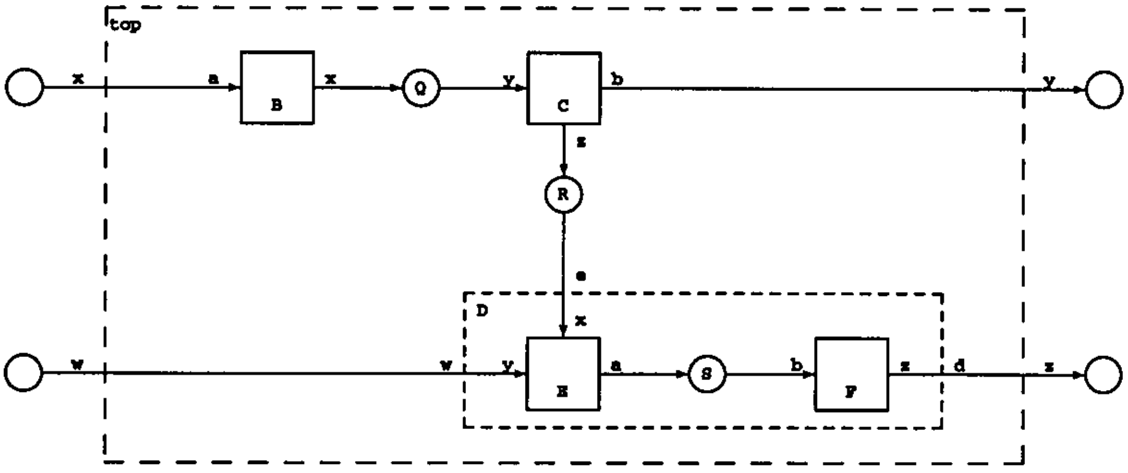

Figure 10.1: A hierarchical net model.

A	$I(a)$	$O(a)$	$HA(a)$
top	w, x	y, z	-
B	a	x	*top*
C	y	b, z	*top*
D	e, w	d	*top*
E	x, y	a	D
F	b	z	D

L	$HL(\ell)$
Q	*top*
R	*top*
S	D

A	C	$M_a(c)$
B	a	x
	x	Q
	y	Q
C	b	y
	z	R
	e	R
D	w	w
	d	z
	x	e
E	y	w
	a	S
F	b	S
	z	d

Figure 10.2: A hierarchical net model, table format.

Each hierarchical net model determines precisely one flat net model. In fact we define the transition system associated with a closed hierarchical actor to be the one that is associated with (closed) flat net model. We formulate the transformation from a closed hierarchical net model to a flat net model as a theorem.

Theorem 10.1 Let $(L, P, A, C, I, O, top, HA, HL, M)$ be a closed hierarchical net model. Let the function $g \in A\backslash\{top\} \to (C \mapsto L)$ be defined by $\forall a \in A : \forall c \in I(a) \cup O(a)$:

$$M_a(c) \in L \Rightarrow g_a(c) = M_a(c) \ \wedge$$
$$M_a(c) \in C \Rightarrow g_a(c) = g_{HA(a)}(M_a(c)),$$

where $\mathrm{dom}(g_a) = I(a) \cup O(a)$. Then g is defined correctly and

$$(L, P, \tilde{C}, \tilde{I}, \tilde{O}, \tilde{M})$$

forms a flat net model, where $\tilde{C} = \cup_{p \in P}(I(p) \cup O(p))$ and $\tilde{I} = I \restriction P$, $\tilde{O} = O \restriction P$, $\tilde{M} = g \restriction P$.

Proof. There are two properties to be proven: first that g is defined correctly by the recursive definition and secondly that the defined tuple is a correct flat net model. The proof is an exercise.
□

Next we will define the concept of an *actor model.* It encompasses a flat net model and an object model. The definition is given first and an elucidation afterwards.

Definition 10.3 An *actor model* is an 8-tuple **actor model**

$$(FN, OM, CT,\ CA,\ T,\ ID,\ F,\ R)$$

where

- FN is a closed flat net model (cf. definition 10.1), *FN*
- OM is an object model (cf. definition 9.9), *OM*
- $CA \in L \to CN$ is called the *class assignment* function, it determines for each place a complex class, *CA*
- $CT \in P \to (C \nrightarrow CN)$ assigns to each connector of a processor a complex class such that *CT*

$$\forall p \in P : \mathrm{dom}(CT_p) = I(p) \cup O(p)$$
$$\wedge$$
$$\forall c \in \mathrm{dom}(M_p) : CT_p(c) = CA(M_p(c)),$$

  which means that a connector and the place to which it is connected have the same complex class,
- T is a subset of the non-negative real numbers that contains 0, it is called the *time domain*, **time domain T**
- ID is a countable set of *identities*, *ID*
- $F \in ID \nrightarrow ID$ is called the *parent function* and it satisfies: **parent function F**

$$\forall i \in \mathrm{dom}(F) : \exists n \in \mathbb{N} :\ F^n(i) \in ID\backslash\mathrm{dom}(F),$$

- $R \in P \rightarrow I\!P(C \nrightarrow ID \times OU \times T)$, where R_p is called the *processor relation* of processor p and the elements of R_p are called *firing rules*. The processor relation should satisfy $\forall p \in P : R_p \neq \emptyset$ and $\forall p \in P : \forall r \in R_p :$ **processor** **firing rules**

  1. $\mathrm{dom}(r) \subset I(p) \cup O(p) \wedge \mathrm{dom}(r) \cap I(p) \neq \emptyset$
  2. $\forall a \in \mathrm{dom}(r) : \pi_2(r(a)) \in com(CT_p(a))$
  3. $\forall a \in I(p) \cap \mathrm{dom}(r) : \forall b \in O(p) \cap \mathrm{dom}(r) : \pi_3(r(a)) \leq \pi_3(r(b))$
  4. $\exists x \in rng(r \restriction I(p)) : \forall y \in rng(r \restriction O(p)) : F(\pi_1(y)) = \pi_1(x)$
  5. $\forall x, y \in \mathrm{dom}(r) \cap O(p) : x \neq y \Rightarrow \pi_1(r(x)) \neq \pi_1(r(y))$

□

The sets T and ID are used to give a complex a time stamp and an identity, respectively. The function CA assigns a name of a complex class to each place. Objects in a place should always belong to the complex class assigned to the place.

F The function F is used to "create" new identities out of old ones. This proceeds as follows; given an identity i a new identity j should satisfy $F(j) = i$. If F^{-1} denotes the inverse of F then $F^{-1}(i)$ is the set of new identities, created out of i. We will only use a finite subset of this set. The requirement on F implies that no identity is a descendant of itself. The set $ID \backslash \mathrm{dom}(F)$ is the set of *start identities*, they do not have ancestors. As an example of an identity set consider the set $I\!N^*$, the set of all sequences of natural numbers. The "children" created by an identity $i \in I\!N^*$, are all sequences $(i; j)$ where $j \in I\!N$ and ";" denotes concatenation. The function F applied to a non-empty sequence gives the sequence with the last element removed. It is also clear that there is an $n \in I\!N$ for every identity such that F^n applied to this identity gives the empty sequence. Here $\text{dom}(F) = I\!N^*\backslash\{\epsilon\}$. (Later we will derive some properties of F.)

R_p

The definition of R_p is quite complicated and requires some explanation. Informally, a firing rule in R_p contains connectors of p and "things" (tokens) produced or consumed for these connectors in a firing of p. Note that a firing rule contains "things" that are almost tokens, the only difference with tokens is, that the place of a token is replaced by a connector, and that the order of components is a bit different. This makes it possible to use the same processor relation in an actor several times, in combination with different places. A processor p *executes* or *fires*, according to one firing rule in R_p.

The first requirement of R_p states, that only tokens are consumed from, or produced for connectors of p and that there is at least one input token (which is important for the activation of the processor and for the identification of new tokens).

The second requirement states that consumed or produced tokens should have the right class.

The third requirement says that all produced tokens should have a time stamp larger or equal to the time stamps of all consumed ones (which is important for the monotonicity of time, as will be seen later. Note that we usually do not specify the time stamps of the new tokens, but only a *delay* that has to be added to the time of the transition.

The fourth and fifth requirement concern the identification of new tokens. Remember that we have approached the identification of tokens in a constructive way, by introducing a specific identification mechanism. All produced tokens get their identity from one consumed token: they get an identity i such that $F(i)$ equals the identity of the consumed token that is selected for identification. So the new tokens get different identities, that are descendants of the identity of the selected token.

Now we are ready to define the state space of an actor model.

state space St

Definition 10.4 Let an actor model be given. The *state space* St is defined by

$$St \subset ID \nrightarrow OU \times T \times L,$$

where

- OU is the object universe (cf. definition 9.3),
- $\forall s \in St : \forall i \in \text{dom}(s) : \pi_1(s(i)) \in com(CA(\pi_3(s(i))))$
- $\forall s \in St : s$ is finite,
- $\forall s \in St : \forall i, j \in \text{dom}(s) : i \neq j \Rightarrow \neg\exists n \in I\!N : i = F^n(j)$.

state
token

A *state* is an element of St; the elements of a state are called *tokens*.
□

So a state is a set of 4-tuples with a unique first component, denoting the identity, the complex, the availability time (time stamp) and the place of a token, respectively. The second requirement states that the tokens in a place should carry a complex that belongs to the class of the place. The last requirement says that in a state no identity is the ancestor of another one.

The next step is the introduction of the transition relation. The transition relation relates a state to a possible successor state. In the definition of a transition relation we use the concept of a *firing assignment*. A firing assignment assigns to a non-empty set of processors a firing rule, i.e. an element of R_p, for each processor in the set. So several processors may fire simultaneously, but no processor may fire simultaneously with itself. (In other Petri net formalisms this is sometimes allowed too.)

firing assignment f

Definition 10.5 A *firing assignment* f is a function that satisfies:

- $f \in P \nrightarrow (C \nrightarrow ID \times OU \times T)$,
- $\mathrm{dom}(f) \neq \emptyset$,
- $\forall p \in \mathrm{dom}(f) : f(p) \in R_p$.

□

So $f(p)$ is a firing rule of processor p. Each transition is caused by the firing of one firing assignment, that determines one or more firing rules.

Note that a token is of the form (i, o, t, ℓ) where $i \in ID$, $o \in OU$, $t \in T$, $\ell \in L$, while a firing rule is of the form (c, i, o, t), with $c \in C$. This is because in a processor relation the place is not known, but only the connector, to which the token corresponds. Further note that a state is a function of identities and a firing rule a function of connectors. Therefore the structure of a firing rule differs from the structure of a state.

FA
transition relation *Tr*

Definition 10.6 Let an actor model with state space St be given. Further let FA be the set of all firing assignments. The *transition relation* Tr, with $Tr \subset St \times St$, satisfies $\forall (s, s') \in Tr : \exists f \in FA$:

1. $In(f) \subset s$
2. $\forall\, p, p' \in \mathrm{dom}(f) : p \neq p' \Rightarrow in(p, f(p)) \cap in(p', f(p')) = \emptyset$
3. $time(s) = tim(f)$
4. $s' = (s \backslash In(f)) \cup Out(f)$

where $\forall p \in P : \forall r \in R_p$:

$$\begin{aligned}
in(p,r) &= \{(i,(x,t,\ell)) \mid \exists c \in I(p) \cap \mathrm{dom}(r) : r(c) = (i,x,t) \wedge M_p(c) = \ell\} \\
In(f) &= \bigcup \{in(p, f(p)) \mid p \in \mathrm{dom}(f)\} \\
out(p,r) &= \{(i,(x,t,\ell)) \mid \exists c \in O(p) \cap \mathrm{dom}(r) : r(c) = (i,x,t) \wedge M_p(c) = \ell\} \\
Out(f) &= \bigcup \{out(p, f(p)) \mid p \in \mathrm{dom}(f)\} \\
tim(f) &= \max\{\pi_3(x) \mid x \in In(f)\} \\
time(s) &= \min\{tim(f) \mid f \in FA \wedge In(f) \subset s\}.
\end{aligned}$$

applicable firing assignment

A firing assignment f is called *applicable* for state s if it satisfies the

requirements 1, 2 and 3.
□

Each transition is the firing of a non-empty set of processors according to firing assignment f. For each processor p a firing rule in the processor relation is chosen ($f(p)$). The set of all tokens that are consumed by the firing is $In(f)$. This set of tokens should be available in the state s (requirement 1). Further no two processors may consume the same token (requirement 2). Requirement 3 states that the set of consumed tokens is the earliest possible set: $tim(f)$ determines the maximal time stamp of the tokens in $In(f)$, requirement 3 says that we only may use firing assignment f if the maximal time stamp is minimal, so there is no transition possible at an earlier time. This property makes the transition law *eager*. Note that the function *time* assigns to each state the time of the first possible transition. We call this the *transition time* of the state. The definition of $time(s)$ is subtle: it is the minimal firing time of a set of firing rules that consume only tokens from the given state, but we did not require that two different processors are not consuming the same tokens (as requirement 2 for the used firing rule). The reason is, that, even if two processors would consume a same token, we can delete one of them (from the domain of the firing assignment) without increasing the minimum time. Requirement 4 specifies how the new state is computed: first delete all consumed tokens, then add the newly produced tokens.

transition time

Although we have defined a state space St and a transition relation Tr for an actor model, we did not define a *transition law* (cf. definition 8.1). The transition law defines a transition system for an actor model. Here a set of *initial states* has to be given, because otherwise the transition law is undefined. Note that the definition of a transition relation is independent of initial states.

Definition 10.7 Let an actor model with state space St, transition relation Tr and set of initial states $S_0 \subset St$ be given. The *transition law* $\mathcal{L}$ satisfies

transition law

$$\mathcal{L} \in (St \times T)^* \to I\!P(St \times T)$$

such that

$$\mathcal{L}(\epsilon) = \{(s, 0) \mid s \in S_0\}$$

and $\forall p \in (St \times T)^* \backslash \{\epsilon\}$:

$$\mathcal{L}(p) = \{(s, t) \mid (\sigma(\ell(p)), s) \in Tr \ \wedge \ t = time(\sigma(\ell(p)))\}.$$

We say the pair (St, Tr) *induces* the transition system $(St \times T, \mathcal{L})$.
□

induced transition system

Here $\ell(p)$ denotes the last event of a non-empty trace p and $\sigma(e)$ and $\tau(e)$ denote the state and time conΦcomponentΦonent of an event e (as defined in chapter 8). Note that the *event set* (see definition 8.1) is $St \times T$. Instead of giving a set of initial states for an actor model we may also give a transition law; from one we can derive the other.

Theorem 10.2 Let an actor model with state space St and transition law $\mathcal{L}$ be given. The transition law is *strongly memoryless.*

Proof. Consider two finite traces p and q. Let $\ell(p) = \ell(q) = s$. Then we have

$$\mathcal{L}(p) = \mathcal{L}(q) = \{(s', t) \mid (s, s') \in Tr \wedge t = time(\sigma(s))\}.$$

Hence only the last reached state determines the event set. So, according to definition 8.5, $\mathcal{L}$ is strongly memoryless.
$\square$

Now we know that the transition law is (strongly) memoryless we can apply theorem 8.2, so the transition law $\mathcal{L}$ is generated by the transition law relation $T\ell$ defined in definition 8.3. The following relationship between Tr and $T\ell$ exists:

$$T\ell = \{((s,t),(s',t')) \mid (s,s') \in Tr \ \wedge \ t \leq time(s) = t'\}.$$

actor model properties

Now we have given the definition of the actor model, we will verify some general properties of the actor model and the autonomous behavior induced by it. For instance, it is not clear whether $s' = (s \backslash In(f)) \cup Out(f)$ is an element of the state space or not. (We will prove it is.) It obviously is a set of tokens, but it is not clear whether it satisfies all the requirements of a state space. Note that the definition of the transition law remains correct, because if s' is not an element of St, then the pair (s, s') does not belong to Tr. We also prove that $s \backslash In(f)$ and $Out(f)$ are disjoint, which means that the produced tokens are indeed new. Further we will show that the transition relation Tr determines a *monotonous* and *eager* transition law. Another property, that appeals to our intuition, is that if processors may fire simultaneously, the next state can also be reached by firing all the processors individually in some arbitrary order. This property is called *serializability.* We also give a sufficient condition for a system to prevent *livelock.* These properties can be considered as proofs that the framework is "sound", in the sense that it corresponds to our intuition.

In the definition of a state space (definition 10.4) we required the property that in a state no identity is the ancestor of another one. We will show (in lemma 10.1) that this property is an invariant of the mechanism to create identities: when a processor fires, one input token is chosen and all new tokens get an identity that is derived from this one, i.e. their identity is mapped by F to the identity of the chosen input token.

Lemma 10.1 Let the Boolean function q on $\mathbb{P}(ID)$ be defined by $\forall I \in \mathbb{P}(ID)$:

$$q(I) \ = \ \forall i,j \in I, \, n \in \mathbb{N} \backslash \{0\} : F^n(i) \neq j$$

(Function q means that no identity in I is the ancestor of another one.) If for some $J \subset ID$ holds that $q(J) = true$ then

$$\forall j \in J, \, i \in ID : F(i) = j \Rightarrow i \notin J$$

and

$$\forall j \in J : q((J \backslash \{j\}) \cup \{i \mid F(i) = j\}) = true.$$

(So q still holds if one replaces an identity j by its "children" i.)

Proof. Fix some $j \in J$. We start with the first assertion. Assume for some $i \in ID$ with $F(i) = j$ that $i \in J$. This violates the property $q(J) = true$ (with $n = 1$). So the first assertion holds. Next we consider the second assertion. Let, for some $j \in J$,

$$J' = (J \backslash \{j\}) \cup \{i \mid F(i) = j\}.$$

Assume that $x, y \in J' \wedge x \neq y \wedge \exists n \in I\!N \backslash \{0\} : F^n(x) = y$. We prove that this implies a contradiction:

- if $x, y \in J \backslash \{j\}$ then the contradiction follows from $q(J) = true$,
- if $F(x) = F(y) = j$ then we have $F^{n+1}(x) = F(y) = j$ and so $F^n(F(x)) = F(x)$ which is a contradiction because of the property of F (see definition 10.3),
- if $F(x) = j \wedge F(y) \neq j$ then $y \in J \backslash \{j\}$ and $y = F^n(x) = F^{n-1}(j)$ which is a contradiction because $y, j \in J$ and $q(J) = true$ (if $n = 1$ we have $y = F(x) = j$, which is a contradiction because $y \in J \backslash \{j\}$),
- if $F(x) \neq j \wedge F(y) = j$ then $F^{n+1}(x) = j$ and $x \in J \backslash \{j\}$ which is also a contradiction because $q(J) = true$.

So in all cases there is a contradiction, which proves the second assertion.
□

Theorem 10.3 Let an actor model with state space St be given. Let $s \in St$ and let f be an applicable firing assignment. Then we have:

- no two processors p and p' $(p \neq p')$ produce the same tokens, i.e.

$$out(p, f(p)) \cap out(p', f(p')) = \emptyset$$

- the produced tokens are different from the consumed tokens, i.e.

$$(s \backslash In(f)) \cap Out(f) = \emptyset$$

- consumption of $In(f)$ and production of $Out(f)$ gives a new state, i.e.

$$(s \backslash In(f)) \cup Out(f) \in St.$$

Proof. We will prove the theorem for an f with $\#(f) = 2$, i.e. a firing assignment in which two processors fire. The case where only one processor fires and the general case are easily derived from this case. Let $\text{dom}(f) = \{p_1, p_2\}$.

We will first prove: all newly produced tokens have a different identity, which implies the first assertion. Note that tokens produced by one processor have different identities, because of property 5 of the processor relation R_p (definition 10.3). Let the tokens, selected for creation of new identities in $rng(f(p_1))$ and $rng(f(p_2))$, have identities j_1 and j_2 respectively. Clearly $j_1 \neq j_2$ (because of requirement 2 of the transition relation, definition 10.6, and the fact that tokens in state s have different identities). Consider two arbitrary tokens in $out(p_1, f(p_1))$ and $out(p_2, f(p_2))$ with identities i_1 and i_2, respectively. Then $i_1 \neq i_2$ because $F(i_1) = j_1 \neq j_2 = F(i_2)$.

Next we consider the second assertion. Let J be the set of all identities in state s. It follows from the first assertion of lemma 10.1 that neither i_1 nor i_2 belongs to $J \backslash \{j_1, j_2\}$. This proves the second assertion.

To prove the last assertion note that s satisfies the constraints of the state space, which means that $q(J) = true$, where q is defined in lemma 10.1. According to that lemma the set

$$J_1 = (J \backslash \{j_1\}) \cup \{i \mid F(i) = j_1\}$$

satisfies $q(J_1) = true$. And similarly satisfies the set

$$J_2 = (J_1 \backslash \{j_2\}) \cup \{i \mid F(i) = j_2\}$$

also $q(J_2) = true$. However the set of all identities of tokens in the new state is a subset of J_2 (the proof of this is an exercise), so this set also has property q. This proves the last assertion.
□

The following assertion is an immediate consequence of this theorem:

$$\forall (s, s') \in Tr \Rightarrow s \neq s'.$$

monotonous The next theorem shows that the induced transition system is monotonous and that every autonomous trace is eager. In particular, if $(s, s') \in Tr$, then

$$time(s) \leq time(s').$$

Theorem 10.4 Let an actor model with state space St, transition relation Tr and transition law $\mathcal{L}$ be given. Then the induced transition system is *monotonous* and every autonomous trace is *eager*.

Proof. We first prove the monotonicity. (Recall definition 8.4.) Let $p = \langle (s_0, t_0), \ldots \rangle$ be an arbitrary autonomous trace. We will prove by induction that for $n \in I\!N$:

$$\tau(p_n) \leq time(\sigma(p_n)).$$

Note that $\tau(p_n) = t_n$ and that (by definition 10.7) $time(\sigma(p_n)) = time(s_n) = t_{n+1}$, so we will prove $t_n \leq t_{n+1}$.

We start with $n = 0$. Since all tokens in s_0 have non-negative time stamps, we have $time(s_0) \geq 0$. On the other hand $t_0 = 0$ by definition, so we have $t_0 \leq time(s_0) = t_1$.

Assume we have $t_n \le t_{n+1}$. Then we consider $t_{n+2} = time(s_{n+1})$. Note that, for some applicable firing assignment $\tilde{f}$ it holds that

$$s_{n+1} = (s_n \backslash In(\tilde{f})) \cup Out(\tilde{f})$$

where $In(\tilde{f}) \subset s_n$. It follows from the definition of the processor relation (definition 10.3, property 3) that

$$\forall p \in dom(\tilde{f}) : \forall x \in in(p, \tilde{f}(p)), y \in out(p, \tilde{f}(p)) : \pi_3(x) \le \pi_3(y).$$

Therefore:

$$\max\{\pi_3(x) \mid x \in in(p, \tilde{f}(p))\} \le \min\{\pi_3(y) \mid out(p, \tilde{f}(p))\}.$$

Note that

$$tim(\tilde{f}) = \max\{\max\{\pi_3(x) \mid x \in in(p, \tilde{f}(p))\} \mid p \in \mathrm{dom}(\tilde{f})\}$$

and that $tim(\tilde{f}) = time(s_n)$. Hence

$$\forall p \in \mathrm{dom}(\tilde{f}) : \max\{\pi_3(x) \mid x \in in(p, \tilde{f}(p))\} = time(s_n),$$

because otherwise we could delete a processor p from $\mathrm{dom}(\tilde{f})$ in order to obtain an $f \in FA$ with a smaller maximum. So we have

$$\forall p \in \mathrm{dom}(\tilde{f}) : \forall y \in out(p, \tilde{f}(p)) : \pi_3(y) \ge time(s_n)$$

and therefore

$$\forall x \in In(\tilde{f}), y \in Out(\tilde{f}) : \pi_3(x) \le \pi_3(y). \qquad (*)$$

We use this property to show that

$$tim(\tilde{f}) = \min\{tim(f) \mid f \in FA \wedge In(f) \subset s_n \cup Out(\tilde{f})\}.$$

Let $f^* \in FA$ be defined by

$$tim(f^*) = \min\{tim(f) \mid f \in FA \ \wedge \ In(f) \subset s_n \cup Out(\tilde{f})\}.$$

Then $tim(\tilde{f}) \ge tim(f^*)$, because the set over which the minimum is taken for f^*, is at least as large as the set for $\tilde{f}$. Assume

$$tim(\tilde{f}) > tim(f^*)$$

which is equivalent to

$$\max\{\pi_3(x) \mid x \in In(\tilde{f})\} > \max\{\pi_3(x) \mid x \in In(f^*)\}.$$

This is only possible if $In(f^*) \cap Out(\tilde{f}) \ne \emptyset$. However, then there is a $y \in Out(\tilde{f}) \cap In(f^*)$ such that:

$$\pi_3(y) \le tim(f^*) < tim(\tilde{f}) = \max\{\pi_3(x) \mid x \in In(\tilde{f})\}.$$

This contradicts $(*)$. So we have

$$\begin{gathered} time(s_n) = \min\{tim(f) \mid f \in FA \wedge In(f) \subset s_n\} \\ = \\ \min\{tim(f) \mid f \in FA \wedge In(f) \subset s_n \cup Out(\tilde{f})\} \\ \leq \\ \min\{tim(f) \mid f \in FA \wedge In(f) \subset (s_n \backslash In(\tilde{f})) \cup Out(\tilde{f})\} \\ = \\ time(s_{n+1}). \end{gathered}$$

The inequality is justified by the fact that the minimum is taken over a larger set on the left-hand side. So this proves for all traces p:

$$\tau(\ell(p)) \leq time(\sigma(\ell(p))).$$

To verify the monotonicity we have to prove that for all events e in the set

$$\mathcal{L}(p) = \{e \mid e = (s,t) \ \wedge \ (\sigma(\ell(p)), s) \in Tr \ \wedge \ t = time(\sigma(\ell(p)))\}$$

the inequality $\tau(e) \geq \tau(\ell(p))$ holds. This is the case because

$$\tau(e) = time(\sigma(\ell(p))) \geq \tau(\ell(p)).$$

To verify eagerness note that all events in $\mathcal{L}(p)$ have the same event time, which means that always one with minimal time stamp is chosen.
□

serializability The next theorem gives the *serializability* property.

Theorem 10.5 Let an actor model with state space St, transition relation Tr and set of firing rules FA be given. Let $f \in FA$ be applicable for state s and $(s, s'') \in Tr$ such that

$$s'' = (s \backslash In(f)) \cup Out(f)$$

If we divide f into two firing rules $g, h \in FA$ such that $f = g \cup h$ and $g \cap h = \emptyset$, then:

- g is an applicable firing rule for s,
- h is an applicable firing rule for $s' = (s \backslash In(g)) \cup Out(g)$,
- $time(s') = time(s)$,
- $s'' = (s' \backslash In(h)) \cup Out(h) = (s \backslash (In(g) \cup In(h))) \cup Out(g) \cup Out(h)$.

Proof. Note first that $In(g) \cap In(h) = \emptyset$, $Out(g) \cap Out(h) = \emptyset$ and that

$$In(f) = In(g) \cup In(h) \wedge Out(f) = Out(g) \cup Out(h). \qquad (*)$$

Further note that $time(s) = tim(f) = tim(g) = tim(h)$. Hence g (and also h) is applicable in s. Since $In(h) \subset s \backslash In(g)$ we have $In(h) \subset s'$, hence $time(s') \leq tim(h)$. By the former theorem we have $time(s) \leq time(s')$. Hence h is applicable for s' and $time(s') = time(s)$. The last assertion follows from $(*)$.
□

As a consequence of this theorem we may split every applicable firing assignment into a sequence of elementary firing assignments with domains that contain only one processor. The order in which these processors fire is irrelevant for the final result. This property is used in the next theorem, which states that an actor model in which only one processor may fire in each transition, is *similar* to the same actor model in which several processors may fire simultaneously.

Theorem 10.6 Let an actor model with state space St and a set of initial states be given. We consider two transition relations for this model: the "standard" transition relation Tr defined in definition 10.6 and the transition relation Tr' defined by:

$$Tr' = \{(s, s') \in Tr \mid \exists f \in FA : \#(f) = 1\}.$$

Let the transition systems induced by (St, Tr) and (St, Tr') be A and B respectively. Then B is similar to A with respect to the identity relation on $St \times T$.

Proof. It is obvious that B is similar to A , because each transition of B is also a transition of A.
□

Note that A is not similar to B.

The next theorem gives a sufficient condition to avoid livelock. Remember that a trace has livelock if it can make infinitely many transitions in a finite time interval. Hence no livelock means that the system makes *progress*. **livelock free**

Theorem 10.7 Let an actor model and a transition law be given such that, for some $\epsilon \in \mathbb{R}^+$, the processor relation satisfies $\forall p \in P : \forall r \in R_p$:

$$\max\{\pi_3(r(a)) \mid a \in I(p) \cap \mathrm{dom}(r)\} \leq \epsilon + \min\{\pi_3(r(b)) \mid b \in O(p) \cap \mathrm{dom}(r)\}$$

then the system is *livelock free*.

Proof. Let p be an infinite autonomous trace. Note that $\sigma(p_i)$ contains only finitely many tokens for all $i \in \mathbb{N}$, since all states are finite. Let, for $n \in \mathbb{N}, k_n$ be defined by:

$$k_n = \{i \in dom(p) \mid 0 \leq \tau(p_i) < n\epsilon\}$$

Hence k_n is the set of (indexes of) events happening before $n\epsilon$. First we show k_1 is finite. Note that for all applicable firing rules f in the initial state

$$\max\{\pi_3(x) \mid x \in In(f)\} \leq \epsilon + \min\{\pi_3(y) \mid y \in Out(f)\}$$

because of the requirement on R. Hence all tokens, produced in events k_1, have a time stamp greater or equal to ϵ and cannot be consumed in an event during $[0, \epsilon)$. So all events in k_1 are caused by tokens in $\sigma(p_0)$ and this is a finite set. Suppose we have proven that k_n is finite.

Hence the number of tokens produced before $n\epsilon$ is finite. In all events occurring in $[n\epsilon, (n+1)\epsilon)$ only tokens produced before $n\epsilon$ are consumed, since tokens produced after $n\epsilon$ are not available before $(n+1)\epsilon$. Hence the number of events in $[n\epsilon, (n+1)\epsilon)$ is finite and therefore k_{n+1} is finite. So we have proven by induction that k_n is finite for all $n \in I\!N$.
□

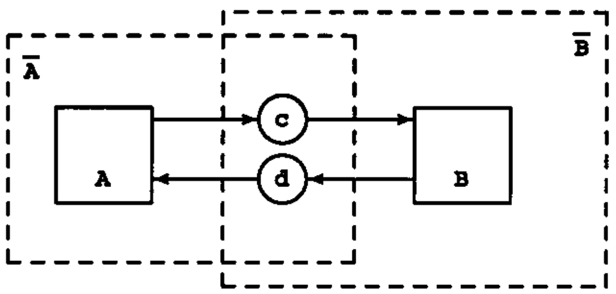

Figure 10.3: The composition of two open systems.

system composition

The next topic we consider in this chapter is the composition of two actor models. Composition of actor models is used to make complex systems out of more simple ones. As an example let us start with two open net models A and B. We add places and we connect the input and output connectors of A and B to these places. Then we have made the *closures* $\bar{A}$ and $\bar{B}$ of A and B respectively. The composition is denoted by $\bar{A} * \bar{B}$. In figure 10.3 we illustrate this. Here A and B both have two unconnected connectors. They are connected to two channels c and d. An event in which the contents of c or d are changed by a processor of B, is called an *external event* for $\bar{A}$. In order to define the composition of two actor models we have to define the composition of two object models and the composition of two flat net models first, because an actor model encompasses these entities.

external event

composition of object models

Definition 10.8 Let A and B be two object models. They are *composable* if and only if

- $\forall r \in RN_A \cap RN_B : DM_A(r) = DM_B(r) \ \wedge \ RG_A(r) = RG_B(r)$,
- $\forall n \in CN_A \cap CN_B : CB_A(n) = CB_B(n) \ \wedge \ CR_A(n) = CR_B(n)$,
- $\forall n \in SN_A \cap SN_B : sim_A(n) = sim_B(n)$.

Their *composition*, denoted by $A * B$, is defined as their component-wise union. (So if $C = A * B$ then $SN_C = SN_A \cup SN_B$, $sim_C = sim_A \cup sim_B$ etc.).
□

Lemma 10.2 Let A and B be two composable object models and let $C = A * B$. Then:

- C is a correct object model, i.e. it satisfies all requirements of definition 9.1 and 9.2,

- $\forall n \in CN_A \cap CN_B : com_A(n) = com_B(n)$.

Proof. The proof is an exercise. (Check the appropriate definitions.)
□

Definition 10.9 Let A and B be two flat net models. *composition of flat net models*They are *composable* if and only if

composition of flat net models

- $P_A \cap P_B = \emptyset$,
- $L_A \cup L_B$, $P_A \cup P_B$, $C_A \cup C_B$ are mutually disjoint.

Their *composition*, denoted by $A * B$, is their component-wise union. (So if $C = A * B$ then $L_C = L_A \cup L_B$ and $M_C = M_A \cup M_B$ etc.).
□

Lemma 10.3 Let A and B be two composable flat net models and let $C = A * B$. Then C is a correct flat net model.

Proof. The proof is trivial because all requirements concern P and $P_A \cap P_B = \emptyset$. So if the requirements hold for A and B they hold for C.
□

Definition 10.10 Let A and B be two actor models such that:

composition of actor models

- FN_A and FN_B are composable,
- OM_A and OM_B are composable,
- $\forall \ell \in L_A \cap L_B : CA_A(\ell) = CA_B(\ell)$,
- $T_A = T_B \ \wedge \ ID_A = ID_B \ \wedge \ F_A = F_B$.

Then A and B are *composable* and their composition, denoted by $A*B$, is defined by:

$$(FN_A * FN_B, OM_A * OM_B, CA_A \cup CA_B, CT_A \cup CT_B, T_A, ID_A, F_A, R_A \cup R_B).$$

□

Lemma 10.4 Let A and B be two composable actor models and let $C = A * B$. Then C is a correct actor model.

Proof. We only have to verify the requirements for CT_C and R_C. For CT_C the requirement follows from $P_A \cap P_B = \emptyset$ and the composability of OM_A and OM_B. In order to verify the requirement for R_C note that

$$OU_C = \bigcup_{n \in CN_C} com_C(n) = \bigcup_{n \in CN_A} com_C(n) \cup \bigcup_{n \in CN_B} com_C(n).$$

By the second assertion of lemma 10.2 we have

$$OU_C = OU_A \cup OU_B.$$

Since $ID_A = ID_B$ and $T_A = T_B$ we have

$$R_C \in P_C \to I\!P(C_C \nrightarrow ID_C \times OU_C \times T_C).$$

The rest of the requirements follow from the fact $P_A \cap P_B = \emptyset$.
□

The composition operators, all denoted by $*$, are associative and commutative. This is very important for the design of systems, because it gives us freedom in the way we want to decompose a complex system.

Theorem 10.8 The composition operators $*$ for object models, flat net models and actor models are *associative* and *commutative*.

Proof. The proof is an exercise.
□

Now we know how to compose actor models, we are interested in the relationship between the state spaces and transition relations of a composed system and its components. The next theorem gives an answer.

Theorem 10.9 Let A and B be composable actor models and let $C = A * B$. (The subscripts A, B and C are used to distinguish the model attributes.) If $(s, s') \in Tr_C$ then $\exists s_A, s'_A \in St_A, s_B, s'_B \in St_B$:

- $s_A \cap s_B = \emptyset \ \wedge \ s'_A \cap s'_B = \emptyset$,
- $s = s_A \cup s_B \ \wedge \ s' = s'_A \cup s'_B$,
- for $i \in \{A, B\}$: $s_i \neq s'_i \Rightarrow (s_i, s'_i) \in Tr_i \ \wedge \ time_i(s_i) = time_C(s)$.

Proof. (For notational convenience we drop subscript C sometimes.) Let $(s, s') \in Tr_C$ and let $f \in TA_C$ be applicable in s, such that

$$s' = (s \backslash In(f)) \cup Out(f).$$

Further let $f_i = f \restriction P_i$. The following assertions are easy to verify:

1. $f_A \cap f_B = \emptyset$ and $f = f_A \cup f_B$,
2. $In(f) = In(f_A) \cup In(f_B)$ and $Out(f) = Out(f_A) \cup Out(f_B)$,
3. $In(f_A) \cap In(f_B) = \emptyset$ and $Out(f_A) \cap Out(f_B) = \emptyset$,
4. $In_i(f_i) = In(f_i)$ and $Out_i(f_i) = Out(f_i)$.

So we have

$$s' = (s \backslash (In_A(f_A) \cup In_B(f_B))) \cup Out_A(f_A) \cup Out_B(f_B).$$

By assertion 3 and 4 and the fact that $In(f) \subset s$, we can find s_A and s_B such that

- $s = s_A \cup s_B$,
- $s_A \cap s_B = \emptyset$,
- $In_i(f_i) \subset s_i$.

Let $s_i' = (s_i \setminus In_i(f_i)) \cup Out_i(f_i)$. Then clearly $s' = s_A' \cup s_B'$ and $s_A' \cap s_B' = \emptyset$ which proves the first two assertions of the theorem. If $f_i \neq \emptyset$ then (using the same arguments as in theorem 10.3)

$$time_C(s) = tim_C(f) = tim_C(f_i) = tim_i(f_i) = time_i(s_i),$$

which proves that f_i is applicable for s_i and therefore $(s_i, s_i') \in Tr_i$. Finally note that $f_i \neq \emptyset \Leftrightarrow s_i \neq s_i'$, which completes the proof.
□

Note that the opposite of this theorem is not true: if each of the components of a system can have an event at a certain time, then it is not sure that they can do their event both (neither simultaneous, nor in some order).

Next we consider the *processor characteristics*. They are important for the modeling of actors, because they are often known in an early stage of design, i.e. before the processor relation is specified. They are also important for the analysis of actors, because some analysis methods are only applicable for actors where the processors have specific processor characteristics.

processor characteristics

Definition 10.11 Let an actor model be given. The *processor characteristics* are defined by:

- *Totality*, which means that a processor will be enabled if and only if there are enough input tokens, independent of their values. Formally a processor p is total if and only if for all functions g with

  totality

  $$dom(g) \subset I(p) \ \wedge \ \forall a \in dom(g) : g(a) \in CA(M_p(a))$$

  it holds that

  $$(\exists h \in R_p : dom(h \restriction I(p)) = dom(g \restriction I(p))) \\ \Rightarrow \ \exists f \in R_p : f \restriction I(p) = g \restriction I(p).$$

- *Input completeness*, which means that the processor consumes in each event via all input connectors. Formally a processor p is input complete if and only if

  input completeness

  $$\forall f \in R_p : dom(f) \supset I(p).$$

- *Output completeness*, which means that the processor produces for every output connector in each event. Formally a processor p is output complete if and only if

  output completeness

  $$\forall f \in R_p : dom(f) \supset O(p).$$

  We call a processor *complete* if it is both input and output complete.

functionality

- *Functionality*, which means that the produced tokens are functionally dependent of the consumed ones. Formally a processor p is functional if and only if:

$$\forall f, g \in R_p : f \restriction I(p) \subset g \restriction I(p) \Rightarrow f = g.$$

□

Note that a total processor does not need to be input complete, however if it is enabled for some subset of input connectors it is enabled for all possible values of input objects. Functionality does not imply input completeness either; however if a "functional processor" is enabled for some set of input objects it is not enabled for any subset. Further functionality implies that two firing rules with the same input have the same output.

As said in the introduction, the actor framework is a generalization of the Petri net framework. With the processor characteristics we can express precisely in what sense it is a generalization: a (classical) Petri net can be defined as an actor model in which all processors are complete and total. In fact in classical Petri nets the identities, values and time stamps of tokens do not play any role. In classical Petri nets one is only interested in the number of tokens in a place of a state, which is called the *marking* of the state. In fact we may use trivial choices for the processor relations and the complex classes: all complex classes are the same and contain only one complex and the processors give all produced tokens a delay equal to zero.

stores

We conclude this chapter with some remarks on *stores*. As said before, from a formal point of view they are just places with some special properties, and therefore we did not consider them before in this chapter. A store always contains exactly one token and will always be available which means that for all $\ell \in L$, such that ℓ is a store the following requirements hold:

$$\begin{gathered} \forall s \in St : \#\{t \in s \mid \pi_3(t) = \ell\} = 1 \\ \wedge \\ \forall t \in s : \pi_2(t) \leq time(s). \end{gathered}$$

These requirements can be met if a consumed token of a store is replaced by a produced token with a delay 0.

channel

If we assume that each processor has at least one input *channel*, i.e. a non-store input place, then we may generalize theorem 10.7 by requiring that the processor relation R satisfies $\forall p \in P : \forall r \in R_p$:

$$\begin{gathered} \max\{\pi_3(r(a)) \mid a \in I(q) \cap \mathrm{dom}(r) \wedge a \textit{ is a channel}\} \\ \leq \\ \epsilon + \max\{\pi_3(r(a)) \mid a \in O(q) \cap \mathrm{dom}(r) \wedge a \textit{ is a channel}\}. \end{gathered}$$

The proof is an exercise.

References and Further Reading

In the bibliography we have given already some references for the formal frameworks used in the book. Here we give some more specialized references.

The theory of *transition systems* stands alone, although similar ideas can be found in literature. For example transition systems are studied in detail in [Hesselink, 1988]. Also the concept of *similarity* is considered there. The theory of *traces* is studied in detail in [Mazurkiewicz, 1984] and [Snepscheut, 1985]. There is related literature on *process algebras* as we have seen: [Hoare, 1985; Milner, 1980; Baeten and Weijland, 1990]. In process algebra similarity relations are studied as well. The *timed* transition systems are related to process algebraic formalisms with time, such as [Reed and Roscoe, 1988]. Another formalism that supports the notion of time is *temporal logic*, in which assertions about the behavior of a system can be proved. See [Pnueli, 1977] or a more general treatise on time in logic [van Benthem, 1983]. The idea of a *transition law* and the concept *memoryless* are borrowed from the theory of Markov chains, see [Revuz, 1975].

The *object framework* is closely related to the entity relationship model ([Chen, 1976]). There are many extensions of this framework, for example [Parent and Spaccapietra, 1985]. A survey is provided in [Spaccapietra, 1987]. Another important original framework is presented in [Abrial, 1974], it has only binary relations as we have here. Binary relations can be considered as set-valued functions and therefore our framework is also closely related to the *functional data model* of [Shipman, 1981], see also [Buneman and Frankel, 1979]. (Note that we will use the term "functional data model" in a more restrictive sense later.) All these data models are often called *semantic data models*. However there is an object framework that is called so, see [Hammer and McLeod, 1981]. A survey of semantic data models is given in [Hull and King, 1987]. The concept of complexes was first introduced in [van Hee and Verkoulen, 1991; van Hee and Verkoulen, 1992]. There are many other frameworks that have the notion of complex objects, for example the nested relational model, see [Schek and Scholl, 1986] and GOOD, see [Gyssens *et al.*, 1990]

The *actor framework* is the classical Petri net model extended with time, object identities and complex values for the tokens. It is not borrowed from other authors. A predecessor of this framework is presented in [van Hee *et al.*, 1989a] and the first version of the actor framework can be found in [van Hee *et al.*, 1989b; van Hee *et al.*, 1991]. The framework has many similarities with others, for example with the *colored Petri nets* of [Jensen, 1992]. The differences are that in our framework tokens may have a value that belongs to an infinite type, that tokens have a time stamp and an identity, and that the transitions are defined by a processor relation instead of arc and transition inscriptions. The colored Petri net model is an improvement of the *predicate/transition nets* of [Genrich, 1987; Genrich and Lautenbach, 1979]. The first ideas of Petri nets with distinguishable tokens are found in [Schiffers and Wedde, 1978]. Other frameworks of Petri nets with time can be found in [Sifakis, 1977; Sifakis, 1980; Ajmone Marsan *et al.*, 1985].

There is a large amount of papers on Petri nets and their analysis. The books of [Reisig, 1985; Jensen, 1992; Peterson, 1981] offer more detailed references. In [Pless and Plünnecke, 1980] a bibliography is given of the literature till 80. There is a Petri news letter that gives up-to-date information of new articles; Petri Net Newsletter, Gesellschaft für Informatik Bonn (ISBN 0173-7473).

The concept of *serializability* plays also an important role in distributed databases, see [Ceri and Pelagatti, 1984].

Exercises

1. Prove that the autonomous behavior of transition system is a closed set.

2. Prove theorem 8.6.

3. Prove theorem 10.1.

4. Prove the assertion in theorem 10.3 that all identities of tokens in the new state are in J_2.

5. Prove lemma 10.2.

6. Prove theorem ??.

7. Generalize theorem 10.7 to the following cases:

   - tokens in stores do not have a delay,
   - not all processors satisfy the requirement of the theorem, but in each *cycle* (i.e. a cycle in the bipartite graph of a flat net model) there is at least one processor that gives its output tokens in the places of the cycle a positive delay.

8. Consider an arbitrary transition system and transform it into a memoryless transition system, such that the systems are bisimilar.

9. Give a formal description of the following actor model and list the behavior of this actor model (i.e. the set of all possible processes starting in the initial state). The actor model has one processor p, two channels, called a and b and one store s. The object classes of the three places are simple: there is only one complex class and the complexes can be represented by natural numbers. Channel a is an input channel and b an output channel. The processor relation of p is such that the value of the complex in store s is the sum of the values of the input of p modulo 4 and the output objects have a value that is 0 if the store value has become even in an event and 1 if it has become odd. In the initial state the store value is 0 and in channel a reside three objects with values 2, 3 and 0. All time stamps and all delays are 0.

10. A *Turing machine* is a finite state machine extended by an infinite tape from which it can read symbols and on which it can write symbols. A Turing machine is characterized by a 4-tuple (M, Σ, T, m_0), where

    - M is the finite state space, not containing h the so-called *halt state*,
    - m_0 is the initial state,
    - Σ is a set of symbols not including the characters L and R, which are used to direct the tape head to the left and right respectively,

- T is a transition function such that:

$$T \in M \times \Sigma \to (M \cup \{h\}) \times (\Sigma \cup \{L, R\})$$

If the machine is in state m, symbol a is read by the head and $T(m, a) = (n, b)$ then the new state will be n and:

- if $b \in \Sigma$ then the symbol on the tape becomes b,
- if the $b \in \{L, R\}$ the tape head will move to the left or right respectively.

The machine stops if the state h is reached. (For more information of Turing machines see for instance [Lewis and Papadimitriou, 1981]).
A generally accepted definition of a computable function is, that for each domain value the function value can be computed by a Turing machine. A formalism to express computations is called *Turing complete*, if every computable function can be expressed in the formalism.
Prove that the actor framework is Turing complete. (Hint: design an actor model for an arbitrary Turing machine.)

11. Consider an arbitrary actor model N. Show there is another actor model M with the same object model and the same set of places, with only one processor p, such that N and M are bisimilar with respect to the identity relation. (Hint: give p an input connector for every input connector occurring in N and connect it to the same place as in N, after some renaming to avoid name clashes. Further let the processor relation R_p be the (modified) union of the processor relations of the processors of N).

12. A hierarchical net model is defined by:

$$\begin{aligned} N.L &= \{p,\ q,\ r,\ s,\ t,\ v,\ w\} \\ N.P &= \{C,\ D,\ E,\ F\} \\ N.A &= \{top,\ A,\ B\} \cup N.P \\ N.C &= \{p?, p!, q?, q!, r?, r!, s?, s!, t?, t!, v?, v!, w?, w!, x?, x!, y?, y!\} \end{aligned}$$

Further the functions $N.I$ and $N.O$ are given by:

	I	O
A	p?	q!
B	q?	p!
C	x?, w?	v!
D	v?	y!,w!
E	s?, t?	x!, r!
F	r?, y?	s!, t!

The function $N.M$ maps every connector with a ? or a ! to the place with the same name (without ? or !) if it exists. Further:

$$\begin{aligned} N.M_C(x?) &= p! \\ N.M_D(y?) &= q? \\ N.M_E(x!) &= p? \\ N.M_F(y?) &= q! \end{aligned}$$

And the functions HA and HL are given by:
$\forall i \in \{A, B\} : HA(i) = top$
$\forall i \in \{C, D\} : HA(i) = A$
$\forall i \in \{E, F\} : HA(i) = B$
$\forall i \in \{p, q\} : HL(i) = top$
$\forall i \in \{v, w\} : HL(i) = A$
$\forall i \in \{s, r, t\} : HL(i) = B$
Draw a diagram, determine the errors in this definition and give a correction.

13. A *data dictionary* is a *data base* that stores the actor models (including object models) of a system. So in a data dictionary the processors are objects, and object classes and the relationship classes of the object models as well!

    - Make an object model for a data dictionary in which only flat net models can be represented.
    - Extend this model to enable it to represent also hierarchical net models.
    - Make an object model in which an arbitrary object model can be represented including, the graphical expressible constraints.
    - Integrate both object models to obtain a model for a "complete" data dictionary.

14. Consider an arbitrary actor model. Introduce a store called *time* and connect it to a processor called *clock*, that triggers itself via one channel called *step*, that is both input and output channel for *clock*. The delay of the token in *step* is one time unit. The value of the object in *time* is a natural number that is increased by one in every firing. In the initial state the value in *time* is zero.
    Prove that in each state of an arbitrary trace, store *time* indicates the "right" time (with an error of at most one time unit).

Index

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for $F\omega$, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.