

OBJECT-ORIENTED DATABASES
WORKFLOWS: OBJECT MODELING,
INTEROPERABILITY, REUSE

(ODB Course)

Leonid Kalinichenko

Institute of Informatics Problems
Russian Academy of Science

E-mail: leonidk@synth.ipi.ac.ru

WORKFLOW CONCEPTS AND WFMS ARCHITECTURES

- Basic workflow concepts
- WFMS standards

COMPONENT-BASED INFORMATION SYSTEMS DEVELOPMENT

- Semantic interoperation reasoning framework
- Type specification manipulation
- Process of semantically interoperable information systems design

A UNIFORM SCRIPT-BASED MULTIACTIVITY FRAMEWORK

- Script types as canonical specification model
- Homogenizing specification of various kinds of dynamic behaviour
- Capturing of structural aspects of multiactivity modeling
- Mapping of pre-existing workflow specifications into the canonical model

WORKFLOW COMPONENT-BASED DESIGN

- Basic notions of process algebras, bisimulation
- Script processes and refinement of scripts
- Script type reducts and conformances
- Process of script design with reuse

OVERVIEW

- Semantic interoperation challenge
- Component-based information systems development
- Related researches and technologies

SEMANTIC INTEROPERATION CHALLENGE

Middleware architectures and accompanied technologies (like OMG CORBA) open a way to **component-based** information systems (IS) development.

But the gap between the existing **Object Analysis and Design** (OAD) methods (top-down) and the demand of the middleware architectures for the IS development intended for a **composition of interoperating components** remains to be large.

The problem in component-based development is that components do not have sufficiently **clean semantic specifications** to rely on **for their reuse**. OMG work for Object Analysis and Design that resulted in UML did not even try to close the gap: **component-based design is beyond the scope** of this work.

On the other hand, OMG work for Meta Object Facility, Business Object Facility and Object Analysis and Design clearly reflects the **diversity of various object models** and **lack of sound foundations**.

But probably the **largest obstacle** for the interoperability of components consists in the **application semantics** of components technically interrelated through the middleware. Reconciliation of their application concept base that is an obvious prerequisite for their interoperation constitutes a problem.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(4)

SYNTHESIS PROJECT APPROACH: OBJECT MODEL and APPLICATION SEMANTICS

Objective: component-based information systems development using semantic interoperability as the kernel idea.

Basic belief: information resources (components) can **meaningfully interoperate** only in the context of a specific application.

We focus on the issues of **semantics of the specifications** we get on different phases of the information system development

Object model semantics

The "canonical" semi-formal object model is used for **uniform representations** of various object, process and data model specifications **in one paradigm**.

Exact meaning of the canonical model is provided by **mapping** of the model into the **formal notation**. Inherent properties: **two-dimensional uniformity of specifications; consistency check of specifications; refinement of specifications** relying on pre-existing components; **equivalent specification of components** in the canonical model.

Specifications of components of **existing software or legacy systems** are extracted and **transformed** into a collection of homogeneous and **model-equivalent specifications** for further reuse at the design phase.

Application semantics

Application semantics of specifications is treated in frame of the **ontological approach**. Ontological definitions provide a **conceptual framework** for talking about application domain. Ontological specification is considered as a well organised collection of **concept definitions**. Concepts are composite descriptions of individuals or types. Semi-formal and formal (model-based) specifications for concepts are provided.

For each of the component specification suspected to be **relevant** to the application the **reconciliation** of its ontological context with the application domain ontological context should be made. The notions of **loose and tight ontological relevance** of specifications are provided.

SYNTHESIS PROJECT APPROACH: PROCESS OF IS DEVELOPMENT

A process of the **component-based IS development** arranged around the middleware concept is elaborated as a **CASE-like activity**.

Phases: (i) **requirement planning and analysis phases** of the conventional system development process are **augmented with ontological specifications and complete specifications of type invariants and operations** defined in the canonical model; (ii) the **design phase** is completely reconsidered: this is the **design with reuse** of the pre-existing components **homogeneously specified** in the canonical model.

Relevant components selection:

a **search** of the component **constituents ontologically relevant** to the the respective **application constituents**

selection among probably relevant constituents those that really may be used for the **concretization** of an application domain type (class).

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(6)

SYNTHESIS PROJECT APPROACH: REUSABLE FRAGMENTS AND COMPOSITIONS

For **reuse** a model of **composite object integrating data and behaviour** from various sources is applied.

Type specifications and their **reducts** are chosen as the **basic units** of specification manipulation.

Algebra of type specifications is introduced. The operations of the algebra are used to produce the specifications of the respective compositions of their operands.

Reducts of the component type specifications are used as **minimal fragments** potentially **reusable** for the respective **reducts of the analysis model types**.

Taking a type reduct is a basic operation of manipulation and transformation of type specifications.

The identification of the fact of **type reducts reusability** is the main concern of the design. Reusable type specifications should be **conformant** to an analysis model type.

The **heuristic procedure** for the **most common reduct** construction for a pair of ontologically relevant type specifications is the basic one for reuse. **Common reducts** are used in type algebra expressions and object calculus formulae to define new types that should be constructed at the design phase. The **mediating definitions** above the reusable reducts of the **component types** are involved. **Correctness** of the results of design can be **verified** using **formal facilities** of the canonical model.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(7)

SYNTHESIS PROJECT APPROACH: WORKFLOWS

Nonconcurrent (algorithmic) type components and **concurrent** (e.g., workflow) specifications are distinguished

Workflow specification – a megaprogram defined above various components of IS (including workflows themselves). Workflow specification is a complex construct **highly integrated** with specifications of other types. Designing a workflow we should consider **semantics of workflow entities** and entities related to them in an integrated way.

Canonical model for specification of a workflow-like **dynamic behaviour**, uniform definitions of industrially supported workflow models, for specification of multiactivities, declarative interresource constraints and multiresource applications. An orthogonal set of the canonical workflow model facilities includes:

- **high level Petri net** and script-based specifications making possible definition of **concurrent execution, data and control flow** between the execution components (such as activities or heterogeneous information resources). **Long-running application processes** with complex patterns of temporal, sequential, causal and other interactivity dependencies are expressible.
- function (**object calculus-based**) specification facilities defines **activities and assertions** over heterogeneous information resources;
- **type specification facilities** providing a relevant information resource model.

Identification of common reducts for workflow-like type definitions is the basic issue for **component-based workflow design**.

Reusable workflow reducts are identified using behavioural abstractions of scripts in terms of **process algebras** and the notion of **bisimulation**.

Complete justification of workflow **reducts reusability** is based on the notion of their **refinement relation**.

RELATED RESEARCH AND TECHNOLOGIES

Component-based E-commerce technology as a recent trend towards resolving the e-commerce challenge at both system and application levels.

Is based on Object-Oriented Technologies and Component-Based Programming, technologies of development of Containers, Wrappers, Mediators, Frameworks and respective standards.

Document-Centric Interoperability issue (in contrast to the API-based) is highly related: considering specifications we deal with document-centric interoperability.

Computational Business Process Components are considered specifically.

The vision for reusable process components is to enable the design, integration, and enactment of virtual enterprises, eventually leading to a community of virtual enterprises and virtual markets.

These components can model, support and execute business process activities within an organization, and can also interconnect and coordinate business processes across organizations, such as to support inter-organizational workflow.

As a framework, it is possible to compose appropriate business process components that provide computer-based support for common business processes such as purchasing, accounts payable, accounts, receivable, and other corporate financial operations.

OMG TRADERS **Trader** is a third party object that enables clients to find **suitable servers** in a distributed system:

- accepts **service advertisemant** from **exporters** of services;
- accepts **service requests** from **importers of services** when importers require knowledge about appropriate service providers;
- searches its **service offer database** to match the **importer's request**.

Associated with each service type is an **interface signature type**.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(9)

RELATED RESEARCH AND TECHNOLOGIES (II)

Workflow Management Coalition (WfMC): establishment of **standards for interoperability and connectivity between Wf products.**

Workflow Interoperability interface (4) provides for the ability of **two or more workflow engines to interoperate** in order to coordinate and execute **workflow process instances** across those engines.

Interface (4) defines the **IDL specifications** intended as an abstract representation of the **operations** required to effect **interoperability** between two (or more) workflow engines.

We distinguish the **nested sub-process WfMC model** assuming that a **process instance enacted on a workflow engine** causes the creation and enactment of a **sub-process instance on a second engine** before carrying on with its own enactment.

OMG Workflow Facility is intended to define **interfaces and their semantics** required to manipulate and execute **interoperable workflow objects** and their metadata. Among others, the following capabilities of the Workflow Facility are emphasized:

1. **integration** defined as a utilization of workflow- unaware applications and/or objects within the workflow context.
2. **interoperability** among different Workflow Facility implementations.
3. **support of reuse of workflow schemas.**

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(10)

SEMANTIC INTEROPERATION REASONING FRAMEWORK

- Transition from conventional object analysis and design to component-based development
- The overall SIR framework
- General structure of the method for development of semantically interoperable information systems

DEMAND FOR COMPONENT-BASED IS DEVELOPMENT

Problems:

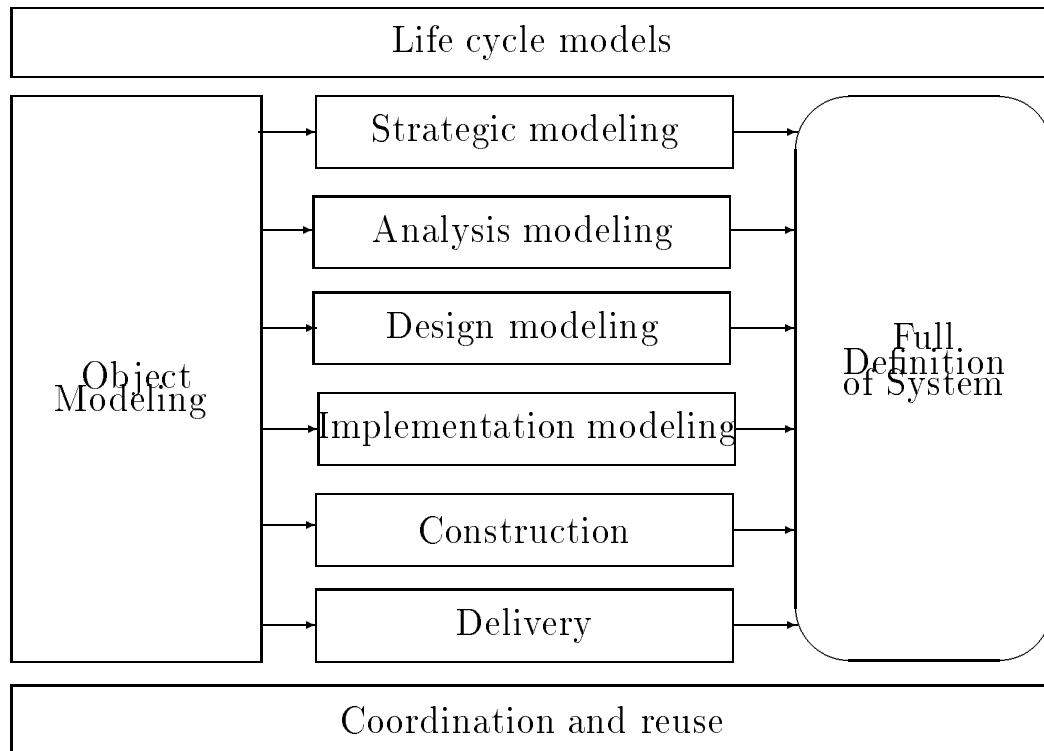
- IS development is **labour-intensive** process
- IS projects are **behind the schedule** and **over budget**
- Completed results **do not deliver** what has been expected
- IS are **inflexible** and difficult to maintain
- IS evolve from autonomous to **integrated** constructs
- Informational and instrumental **heterogeneity** of IS **components**
- Distribution and **autonomy** of IS constituents
- IS **re-engineering** is a continuous process of refining and changing of requirements
- IS **oppose changes** and have a trend of quite fast becoming the burden for their organizations (**legacy systems**)
- **Migration** (incremental) to new requirements of domains and technologies

New IS development culture is needed to make a process of development a **capital-intensive** one in which we invest more in the development of (object-oriented) **components** and automated process based on **semantic knowledge** to guide the assembly of those components into the desired target systems.

Component (information resource) is a replaceable unit of development which encapsulates design decisions and which can be composed with other components as part of a larger units.

Component-based IS development is now driven by open distributed computing/telecommunications technologies

DEVELOPMENT IN OBJECTS: CONVENTIONAL FRAMEWORK



Life cycle models: determine how to package the development into the activities of the framework.

Waterfall model: splitting of development into independent steps carried out in sequence.

Rapid prototyping: getting more early feedback for the user

Spiral model: re-evaluation of objectives on each particular cycle

Strategic Modeling: intentions of an enterprise, context for future developments, a systems strategy, organizational, financial and technical constraints.

OBJECT-ORIENTED DATABASES

GENERAL SCHEMA OF SIR

Substantial gap: technical ability to interoperate and reasoning that components possess **required capabilities** and **ability of semantic inter-operation** in a proper application context

A notion of **semantic interoperation reasoning** (SIR) as a collection of basic models and methods leading to **component-based** IS development

SIR: looking for resources applicable to the problem, choosing among them of the most appropriate and coherent with the problem domain context, composing of the chosen resources into a **megaprogram**

Definition. By **megaprogramming** we mean a technology of programming in large components (megamodules) that is based on knowledge of functional capabilities of services provided by heterogeneous information resources.

SIR should provide necessary modeling, methodological and architectural capabilities component-based information systems design based on SIR.

SIR should complement industrial core interoperation technology.

Definition. Context is a collection of information provided for correct interpretation of extensional and intensional sentences that constitute a specification of an information resource. A context is determined by a problem domain.

Definition. Ontological specification is a set of definitions of **context-specific knowledge representation** primitives (concepts) for describing of context vocabulary in a form that is both human and machine readable.

Ontological specifications play the role of **coupling interfaces** among components providing the basis for them to interoperate.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(36)

GENERAL SCHEMA OF SIR (II)

Definition. Semantic interoperability means interoperability of components intended for support of a specific application.

Components can **meaningfully interoperate** only in a context of specific application: application contexts of the information resources involved should be **coherent** and their **composition should be consistent** within the context of the intended application.

SIR) is a specific culture of the information systems development in HIRE characterized by:

- reaching of **coherence of ontological contexts** of resources and application;
- searching for resources and their subcomponents that could serve as **concretizations** of the application;
- creation of **compositions of resources** serving as a consistent, coherent concretization of application;
- **justifying** that constructed specification is truly a **concretization** of the application.

SIR PROPOSITIONS

Prerequisites of semantic interoperability

Specificational propositions

- **Complete specifications** of IR and of the application requirements

Completeness for reasoning for: **applicability** of a resource, **decomposability** of an application specification into fragments concretizable by available resources, **composability** of resource fragments into a consistent, coherent entity reusable for a given problem (or its fragment).

- **Homogeneous ("canonical")** equivalent specifications for resources

Specific methods should provide for equivalence of the canonical specifications.

- **Uniform set of specification facilities should be used for different phases of development**

Complexity of specifications (ontological, behavioural, assertional) significantly adds to the complexity of transformations if different representational models and languages would be associated with different levels.

- **Sound foundations** to support **provable** requirement concretization and coherent resource composition reasoning

Resource states, functions, assertions or transactions may serve as a concretization of the specification of the application states, functions, assertions or activities.

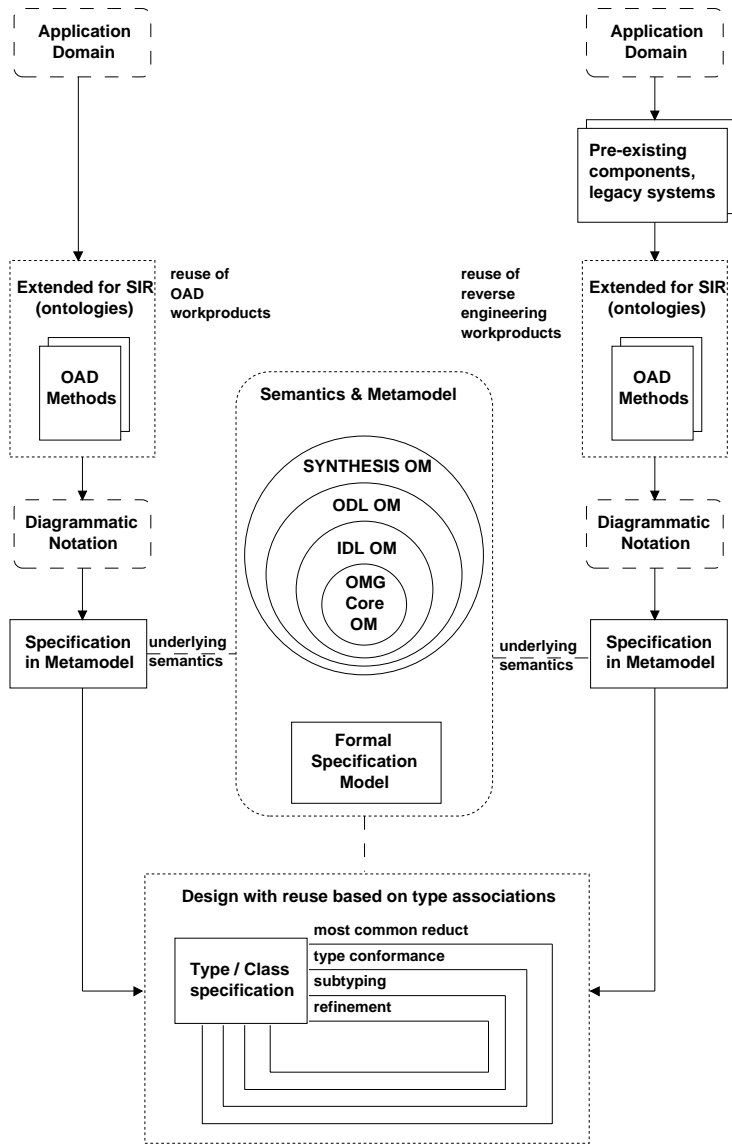
Model-based specifications represent **statics and dynamics** of an information system.

The notion of execution of a model-based specification consists of the **proof of the initial consistency** of the model and the **preservation of the invariants** by the operations. Provable way of development of programs from specifications through stepwise refinement. "Off-the-shelf" components.

SIRF MODELS

1. **Ontological model:** of application contexts definition; a combination of **verbal specification** and **canonical object model** for concept definitions.
2. **Requirement planning / Analysis models:** RP for specification of the problem domain and of IS requirements. AM leads to specification of IS in accordance with the requirements. **Canonical OM** and its **mapping to formal** notation.
3. **Design model: component-based** design in HIRE. Canonical OM contains features for **type compositions, refinements and verification, reconciliation of ontologies.**
4. **Implementation model:** basic notions are **servers, adapter types, implementation types.** Decisions: distribution of implementations over the set of servers; choosing component adapters to support a design type, arranging implementations into **groups** for a single design type, **hosting** of implementations by different machines.
5. **Information resource specification model.** SIR-complete specification of the pre-existing components using ontological, design and implementation modeling.

SIR FRAMEWORK



FORMAL METHODS

Formal methods are mathematically based techniques for describing system properties; have an underlying theoretical model against which a description can be verified.

Model-based specifications use typed set theory and sets as formally defined mathematical entities. VDM, Z, RSL, AMN are well known model-oriented [sequential] languages.

Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS) and Petri Nets are different model-oriented **concurrent** methods.

Larch is an **algebraic** sequential language.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(41)

COMPONENT-BASED DESIGN

Design: component-based process of concretization of a specification obtained on an analysis phase by an interoperable composition of pre-existing information resources from HIRE.

The **concretization** of the analysis specification and its constituents **follows the steps** (techniques):

1. **Ontological Contexts Integration:** establishing the resource/application ontological concepts relationship.
2. Searching for **ontologically relevant** components: an associative correlation based on verbal description of the concepts; interconcept associations as a subtyping relationships.

Metaphoric thinking: We rely on similarities between ontological concept definitions to infer metaphoric structure of the concept relevant entities (types).

3. **Preliminary identification of reusable fragments** Reducts (projections of object types) are considered as patterns of reuse and composition.

Software service reuse. The functional **reducts** of ontologically relevant resource types are discovered justifying that they are the refinements of reducts of the specification types.

Data service reuse: instances of the relevant data reducts can be reused as fragments of instances of the specification class. Relying on unique keys, the data fragments from different sources can be joined.

4. **Concretization type (view)** construction: composition type definition above the reducts of the resource types involved. Views constructed above the identified relevant mergeable data reducts are specified.
5. **Verification** of the concretizations For the verification necessary **proof obligations** are automatically generated and proved.

TYPE SPECIFICATION MANIPULATION: COMMON TYPE REDUCTS, SPECIFICATION COMPOSITION

- Concept of refinement
- Type reducts and type algebra
- Common reducts as reusable fragments

TYPE SPECIFICATION DECOMPOSITION: REDUCTS

Type reduct A signature reduct R_T of a type T is defined as a subsignature Σ'_T of type signature Σ_T that includes a carrier V_T , a set of symbols of operations $O'_T \subseteq O_T$, a set of symbols of invariants $I'_T \subseteq I_T$.

More generally: a type reduct R_T is a **subspecification** (with a signature Σ'_T) of a specification of a type T . R_T becomes a supertype of T .

Taking a reduct: the occurrences of the discarded attributes of the original type into the remaining operations and invariants of the reduct should be **existentially quantified and properly ranged**.

Decomposing a type specification, we can get its **different reducts** on the basis of **various type specification subsignatures**.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(44)

TYPE REFINEMENT

Type U is a **refinement** of type T iff

- there exists a one-to-one correspondence Ops between O_T and O_U ;
- there exists an abstraction function Abs that maps each state of U into the respective state of T ;
- $\forall x \in V_T, y \in V_U (Abs(x, y) \Rightarrow I_T \wedge I_U)$
- for every operation o in T there exists an operation $Ops(o) = o'$ in U such that o' is a refinement of o . To establish an operation refinement it is required that operation precondition $pre(o)$ should **imply** precondition $pre(o')$ and operation postcondition $post(o')$ should **imply** postcondition $post(o)$.

For type U that is a subtype of T we require that the type reduct R_U to the operation set including all operations $o \in O_U$ such that $Ops(o) \in O_T$ should be a **refinement** of T .

COMMON REDUCTS

A **common reduct** for types T_1, T_2 is such reduct R_{T_1} of T_1 that there exists a reduct R_{T_2} of T_2 such that R_{T_2} is a **refinement** of R_{T_1} . Further we refer to R_{T_2} as to a **conjugate** of the common reduct.

Hints for **inclusion of operations** of T_1 to its common reduct with T_2 : similarity of their signatures in T_2 modulo variable renaming, parameters ordering and parameter type redefinition (contravariant for the argument types and covariant for the result types type differences for T_2 are acceptable).

A specification of an operation of T_1 to be included into the resulting reduct is chosen among such pre-selected pairs of operations of operand types if the operations in a pair are in a **refinement order** (for the common reduct (resulting supertype) more abstract operation should be chosen).

If the pre-selected operations are not in a proper **refinement order** then they are considered to be **different operations** and will not be included into the common reduct.

A **most common reduct** $R_{MC}(T_1, T_2)$ for types T_1, T_2 is a reduct R_{T_1} of T_1 such that there exists a reduct R_{T_2} of T_2 that refines R_{T_1} and there can be no other reduct $R_{T_1}^i$ such that $R_{MC}(T_1, T_2)$ is a reduct of $R_{T_1}^i$, $R_{T_1}^i$ is not equal to $R_{MC}(T_1, T_2)$ and there exists a reduct $R_{T_2}^i$ of T_2 that refines $R_{T_1}^i$.

A [**common**] **type reduct** is the basic notion for the **component-based design**: it constitutes a basis for determining **reusable fragments**.

TYPE SPECIFICATION COMPOSITION: MEET

Type meet operation. An operation $T_1 \sqcap T_2$ produces a type T as an **'intersection'** of specifications of the operand types. Generally the result T of the meet operation is formed as the **merge of two most common reducts** of types T_1 and $T_2 : R_{MC}(T_1, T_2)$ and $R_{MC}(T_2, T_1)$. The merge of two reducts includes union of sets of their operation specifications. If in the union we get a pair of operations that are in a refinement order then only one of them, the more abstract one is included into the merge. Invariants created in the resulting type are formed by **disjuncting of invariants** taken from the most common reducts specifications.

If T_2 (T_1) is a **subtype** of T_1 (T_2) then T_1 (T_2) is a result of the meet operation.

Type T is placed in the type hierarchy as a **direct supertype** of the arguments types and a direct subtype of all common direct supertypes of the argument types.

If T_1 and T_2 are **object types** then T is an **object type**.

If one of the argument types of meet is an **object** type and another is a **non-object** type then the result is a **non-object type**.

If T_1 and T_2 are non-object types then T — the result of meet — is a non-object type.

Meet operation produces a type T that contains common information contained in types T_1 and T_2 .

TYPE SPECIFICATION COMPOSITION: JOIN

Type join operation. An operation $T_1 \sqcup T_2$ produces a type T as is a **join** of specifications of the operand types. Generally T includes a **merge of specifications** of T_1 and T_2 . Common elements of specifications of T_1 and T_2 are included into the resulting type only once. The **common elements** are determined by the **merge of the conjugates of two most common reducts** of types T_1 and $T_2 : R_{MC}(T_1, T_2)$ and $R_{MC}(T_2, T_1)$. The merge of two reduct conjugates includes union of sets of their operation specifications. If in the union we get a pair of operations that are **in a refinement order** then only one of them, the **more refined one** (belonging to the **conjugate of the most common reduct**) is included into the merge. Invariants created in the resulting type are formed by **conjuncting of invariants** taken from the original typss. The resulting invariant should be consistent.

If $T_2 (T_1)$ is a **subtype** of $T_1 (T_2)$ then $T_2 (T_1)$ is a result of a join operation.

A type T is placed in the type hierarchy as a **direct subtype** of the join operand types and a **direct supertype** of all the common direct subtypes of the argument types.

In case when both types T_1 and T_2 are **object types** or at least one of the argument types is an object type then T – the result of **join** – is an **object type**. In all such cases T should become a **strong or weak behavioral** subtype of the argument types.

In case when both types T_1 and T_2 are **non-object types** then the result of *join* T is a **non-object type**.

PROPERTIES OF TYPE COMPOSITIONS

P.O. set of types \mathcal{T} with a subtyping relation and \top and \perp types is a **lattice**: for all $S, U \in \mathcal{T}$ there exists least upper bound (l.u.b.) and greatest lower bound (g.l.b.).

Meet $T = S \sqcap U$ produces T as the **g.l.b.** for types S, U in the $(\mathcal{T}, \sqsubseteq)$ lattice.

Join $T = S \sqcup U$ produces T as the **l.u.b.** for types S, U in the $(\mathcal{T}, \sqsubseteq)$ lattice.

$\langle \mathcal{T}; \sqcap, \sqcup, * \rangle$ is an algebra with three binary operations and properties:

1. **Commutativity:** $S \sqcap U = U \sqcap S$ and $S \sqcup U = U \sqcup S$
2. **Associativity:** $S \sqcap (U \sqcap S) = (S \sqcap U) \sqcap S$ and $S \sqcup (U \sqcup T) = (S \sqcup U) \sqcup T$
3. **Idempotence:** $S \sqcap S = S$ and $S \sqcup S = S$
4. **Absorption:** $S \sqcap (S \sqcup U) = S$ and $S \sqcup (S \sqcap U) = S$

Two another rules relate subtyping with operations meet and join:

$$S \sqcap U = U \equiv S \sqsubseteq U$$

$$S \sqcup U = S \equiv S \sqsubseteq U$$

A product operation is neither commutative nor associative. Using of parenthesis influences the result of a product in the following way:

$(T_1 * T_2) * T_3$ two injecting f.

$T_1 * (T_2 * T_3)$ two injecting f.

$T_1 * T_2 * T_3$ three injecting f.

FORMAL METHODS

Formal methods are mathematically based techniques for describing system properties; have an underlying theoretical model against which a description can be verified.

Model-based specifications use typed set theory and sets as formally defined mathematical entities. VDM, Z, RSL, AMN are well known model-oriented [sequential] languages.

Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS) and Petri Nets are different model-oriented **concurrent** methods.

Larch is an **algebraic** sequential language.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(50)

MODEL-BASED SPECIFICATIONS

Static aspects: include the **states** a system can occupy and the **invariant relationships** (constraints) that should be preserved as the system moves from state to state.

Dynamic aspects: include possible operations and changes of state that happen.

Specification of an operation (function) consists of a definition of properties and relationships that state transitions caused by the operation should satisfy. For that predicates relating values of state variables before and after operation (expressing **mixed pre- and post-conditions**) are defined.

The notion of execution of a model-based specification consists of the **proof** of the initial **consistency** of the model and the **preservation of the invariants** by the operations.

The provable way of development of programs from specifications by proper **concretization** of abstract data types and operations of the specification by concrete data types and programs satisfying strict concretization conditions.

Starting with VDM, model-theoretic methods evolved to Z and Object-Z Notations, Raise and J.-R. Abrial's Abstract Machines technology that had reached the level of "industrial strength" methods supported by the proprietary dedicated program packages.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(51)

SPECIFICATIONS IN B

An exact **axiomatic definition (specification)** of the modeled entity properties in AMN may be analysed formally.

AMN program is used for proving **correctness** of definitions and for **symbolic transformation** of initial program (composition, refinement, etc.)

An abstract machine definition contains: the **machine states** and **operations** allowing to get and change such states.

States are defined by **variables** determining the state components and **invariants**.

Operations describe properties and relationships that must be satisfied during a **change of a state** within the **limits of the invariants**.

To express **logical assertions** relating values of state variables **before and after** operation executing, AMN uses **calculus of substitutions**.

Properties of operations are expressed in terms of **predicate transformers** which bind with **post-condition** of Op its **weakest precondition**.

Generalized substitutions (which are operators of such calculus) may be considered as **abstract machine commands**.

To prove that an operation preserves invariants, every invariant is considered as a post-condition to which the operation (i.e., a predicate transformer) is applied. This application results in forming of a new predicate which must be proved too.

REFINEMENT OF ABSTRACT MACHINES

Step-wise restriction of the abstract specification to make them finally implementable is a **refinement**.

Algorithmic refinement consists in removing of non-determinism by being more and more precise about the way our operations are to be eventually made concrete through sequencing and loop. At the same time we should relax pre-conditions.

Data refinement consists in removing completely all variables whose types are too complicated to be implemented as such and in replacing them by simpler variables whose types correspond to those found in programming notations. Data refinement also includes some algorithmic refinement at the same time.

Abstraction relation Assume two substitutions S and T working within two different machines (two distinct variable spaces are represented by two variables x and y , $x \in s$ and $y \in t$ are respective invariants of these machines). A binary relation v from s to t such that $ran(v)$ is equal to t is an **abstraction** relation.

Algorithmic refinement just appears to be a special case of data refinement with an **identity** abstraction relation. It is important that algorithmic and data refinements are **monotonic** on all generalized substitution constructs introduced.

A machine N is said to **refine** a machine M if a user can use N instead of M without noticing it.

Both machines must have the same operation signatures.

Sufficient condition for N to refine M is the requirement that each operation of N **data refines** the corresponding operation of M (by a certain abstraction relation).

REFINEMENT CONSTRUCT

refinement is a construct that resembles a machine. A refinement can refine either a machine or another refinement.

The **invariant clause** of refinement is just the abstraction relation. The operations of the **refinement** only involve the variables of the **refinement**, not of the construct being refined. **Pure algorithmic refinement:** variables of the **refinement** and of the construct being refined are the same.

machine

AM Identifier

variables

x

invariant

P

initialization

S

operations

$z \leftarrow \text{OpName} \cong \text{pre } Q \text{ then } T \text{ end};$

end

Refinement of the abstract machine

machine

Identifier

refines

AM Identifier

variables

y

invariant

R

initialization

U

operations

$z \leftarrow \text{OpName} \cong \text{pre } L \text{ then } V \text{ end};$

end

PROOF OBLIGATIONS FOR REFINEMENT

The **proof obligations template** which concern the relationship between a machine and its refinement via the **abstraction mapping** R on the variables of the two machines:

$$\begin{aligned} & \exists(x, y) \cdot (P \wedge R) \\ & [U] \neg [S] \neg R \\ & \forall(x, y) \cdot (P \wedge R \wedge Q \Rightarrow L \wedge [V'] \neg [T] \neg (R \wedge z = z')) \end{aligned}$$

R is a predicate containing both a typing invariant and other properties of the local state y and the abstraction mapping which relates y and x . V' stands for substitution V within which the variable z has been replaced by z' .

The proof obligations meaning:

1. The existence of a model for a combined abstract and refined state which satisfies the abstraction relation
2. The correct refinement of initialization under the assumption of the constraints and properties of both machines.
3. The correct refinement of operations.

TYPE COMPOSITION EXAMPLE: TYPE DECLARATIONS

```
{Project;
  in: type;
  area: string;
  grade: real;
};

{Rproject;
  in: type;
  supertype: Project;
  coordinator: Organization;
  leader: Professor;
  priority_theme: string;
  cooperate_with: {set_of: Rproject};

  candidate_proj: {in: function;
    params: { j/Rproject, -c/Rproject};
    {{this.area = j.area & this.priority_theme = j.priority_theme &
      c' = j}}};

  area_constr: {in: predicate, invariant;
    {{ all p/Rproject (p.area = 'comp-sci' => p.grade = 5 &
      (p.priority_theme = 'open systems' |
      p.priority_theme = 'interoperability'))}}};

  leader_constr: {in: predicate, invariant;
    {{ all p/Rproject (p.leader.degree = 'PhD') }}}
}
```

TYPE COMPOSITION EXAMPLE: TYPE DECLARATIONS

```
{Iproject;
  in: type;
  supertype: Project;
  coordinator: Company;
  cooperate_with: {set_of: Project};
  sponsor: Company;

  candidate_proj: {in: function;
    params: {j/Project, -c/Project};
    {{this.area = j.area & c' = j}}};

  area_constr: {in: predicate, invariant;
    {{ all p/Iproject (p.area = 'comp-sci' => p.grade >= 3 )}}}

}
```

MOST COMMON REDUCT FOR Rproject and Iproject TYPES

```
{RIrmc;
  in: type;
  supertype: Project;
  coordinator: Organization;

  candidate_proj: {in: function;
    params: {j/Rproject, -c/Project};
    {{this.area = j.area &
      ex p/Rproject (this = p/RIrmc &
        p.priority_theme = j.priority_theme) &
        c' = j}}};

  area_constr: {in: predicate, invariant;
    {{ all r/RIrmc (r.area = 'comp-sci' => r.grade = 5 &
      ex p/Rproject (r = p/RIrmc &
        (p.priority_theme = 'open systems' |
          p.priority_theme = 'interoperability'))))}}};

  leader_constr: {in: predicate, invariant;
    {{ all r/RIrmc
      ex p/Rproject (r = p/RIrmc &
        p.leader.degree = 'PhD')}}}
}
```

MOST COMMON REDUCT FOR Iproject and Rproject TYPES

```
{IRrnc;  
  in: type;  
  supertype: Project;  
  cooperate_with: {set_of: Project};  
  
  area_constr: {in: predicate, invariant;  
    {{ all r/IRrnc (r.area = 'comp-sci' => r.grade >= 3  
  }  
}
```

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(59)

CONJUGATE FOR Rproject, Iproject COMMON REDUCT

```
{RIRmc_C;  
  in: type;  
  supertype: Project;  
  coordinator: Company;  
  
  candidate_proj: {in: function;  
    params: {j/Project, -c/Project};  
    {{this.area = j.area & c' = j}}};  
  
  area_constr: {in: predicate, invariant;  
    {{ all r/RIRmc_C (r.area = 'comp-sci' => r.grade >= 3 )}}}
```

CONJUGATE FOR Iproject, Rproject COMMON REDUCT

```
{IRRmc_C;  
  in: type;  
  supertype: Project;  
  cooperate_with: {set_of: Rproject};  
  
  area_constr: {in: predicate, invariant;  
    {{ all r/IRRmc_C (r.area = 'comp-sci' => r.grade >= 5 &  
      ex p/Rproject (r = p/IRRmc_C &  
        (p.priority_theme = 'open systems' |  
          p.priority_theme = 'interoperability'))}}});  
  
  leader_constr: {in: predicate, invariant;  
    {{ all r/IRRmc_C  
      ex p/Rproject (r = p/IRRmc_C &  
        p.leader.degree = 'PhD')}}}}
```

MEET Rproject, Iproject

```
{RImeet;
  in: type;
  supertype: Project;
  coordinator: Organization;
  cooperate_with: {set_of: Project};

  candidate_proj: {in: function;
    params: {j/Rproject, -c/Project};
    {{this.area = j.area &
      ex p/Rproject (this = p &
        p.priority_theme = j.priority_theme) &
        c' = j}}};

  area_constr: {in: predicate, invariant;
    {{ all r/RImeet (r.area = 'comp-sci' => r.grade = 5 &
      ex p/Rproject (r = p/RImeet &
        (p.priority_theme = 'open systems' |
          p.priority_theme = 'interoperability')) |
      all r/RImeet (r.area = 'comp-sci' => r.grade >= 3 &
        ex p/Iproject (r = p/RImeet))}}};

  leader_constr: {in: predicate, invariant;
    {{ all r/RImeet
      ex p/Rproject (r = p/RImeet &
        p.leader.degree = 'PhD') }}}
}
```

JOIN Rproject, Iproject

```
{RIjoin;
  in: type;
  supertype: Rproject, Iproject;
  coordinator: Company;
  cooperate_with: {set_of: Rproject};
  leader: Professor;
  priority_theme: string;
  sponsor: Company;

  candidate_proj: {in: function;
    params: {j/Project, -c/Project};
    {{this.area = j.area & c' = j}}};

  area_constr: {in: predicate, invariant;
    {{ all p/RIjoin (p.area = 'comp-sci' => p.grade = 5 &
      (p.priority_theme = 'open systems' |
      p.priority_theme = 'interoperability') &
      (p.area = 'comp-sci' => p.grade >= 3))}}};

  leader_constr: {in: predicate, invariant;
    {{ all p/RIjoin (p.leader.degree = 'PhD') }}}
}}
```

PROCESS OF SEMANTICALLY INTEROPERABLE INFORMATION SYSTEMS DESIGN

- Basic design principles
- Common signature reduces identification
- Most common reduces construction
- Concretizing types construction

BASIC STEPS OF THE DESIGN PROCESS

1. Establishing **ontological conformances** for each constituent of the analysis model

For each constituent Con_s of the analysis model we get a collection $Scon_r$ of constituents of the **resource specifications of the same kind** that are ontologically relevant to Con_s . **Loose or/and tight ontological relevance** is established.

2. **Common reducts** identification and construction

For each pair of type specifications t_s and t_r (each t_r should belong to ontological conformance of t_s) we try to **construct their common reduct**. We start with identification of their **common signature reduct** taking into account only signatures of types. In case of success, most common reducts are attempted. This makes possible to identify the maximal reduct of t_s and the respected concretization reduct of t_r .

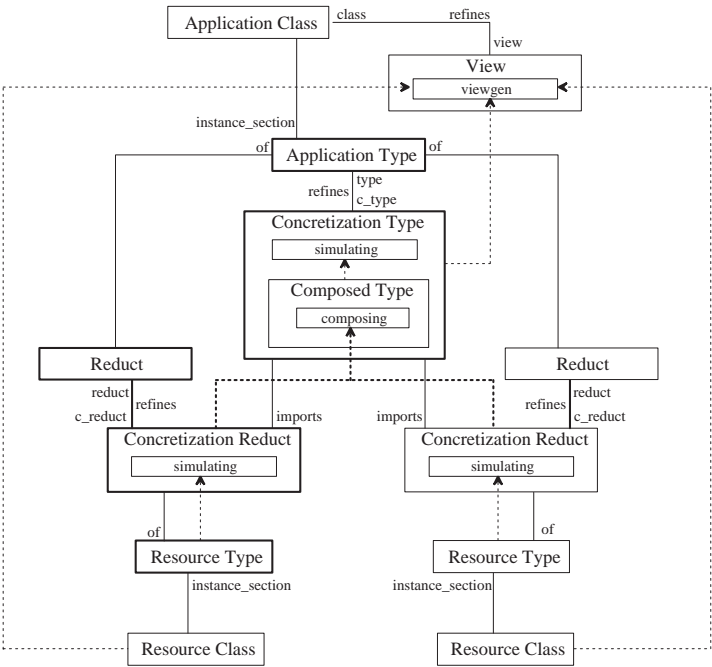
3. Construction of concretization **type compositions and views**

Using concretization reducts for t_s , we create a **composition of types** that serves as a concretization of t_s . Using concretizing types and relying on knowledge of real world extensions of resource classes, we construct views serving as concretizing classes for classes of the analysis model.

4. **Justification** of concretization by formal proofs

Finally, we can justify the concretizations constructed so far by **formal proofs**.

SPECIFICATION HIERARCHY FOR COMPONENT-BASED DESIGN



A UNIFORM SCRIPT-BASED MULTIACTIVITY FRAMEWORK

- Script types as a canonical specification model
- Homogenizing specification of various kinds of dynamic behaviour
- Capturing of structural aspects of multiactivity modeling
- Mapping of pre-existing workflow specifications into the canonical model

OBJECTIVES OF THE HIRE DYNAMIC BEHAVIOR SPECIFICATION

- **Application specification of requirements** for information systems design:

Description of information systems with complex patterns of dynamic behavior and constraints. The behavior may be defined by means of **concurrent actions** (interactions) of various application domain objects (such as take-off or landing of a plane).

- **Pre-existing information resource specification:**

Behavioral abstraction of heterogeneous information resources

- **Dynamic behavior design/implementation**

Is intended ultimately for the **reuse of pre-existing behavior**

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(67)

ASPECTS OF THE INTEROPERABLE SYSTEM DYNAMICS

- **Multiactivity modeling**

Long-running application multiactivities with complex patterns of temporal, sequential, causal and other interdependencies

(Dayal, Meersman)

- **Active execution modeling**

Class of execution models supporting events, conditions, timing requirements leading to the execution of predefined activities.

Classical scheme of ECA form of rules: event detectors, composite events, coupling of triggered condition (action) of a rule with a transaction, rule management (Ceri, Dayal, Widom)

- **Declarative Interresource Constraints Specification**

Maintaining consistency among interdependent data stored in HIRE that may be allowed to be inconsistent within certain limits.

Compensating activities correcting constraint violation.

Consistency req. support includes automatic generation (synthesis) of a set of transactions that manage interdependent data.

Polytransaction (Rusinkiewicz, Sheth); Carnot constraints

- **Multi-resource megaprogramming specification and execution**

1. definition of **tasks for autonomous software packages involved**
2. definition of **data and control flow between such tasks** (query, constraint, multitransaction);
3. **common interface and protocol** for communication of various autonomous software packages supporting information resources.

DOL, L.0, Rosette; Rusinkiewicz, Sheth, Tomlinson

THE ORTHOGONAL SET OF THE LANGUAGE FEATURES SUPPORTING THE HIRE DYNAMICS

We provide a **uniform model and language supporting executable specification of information resource interoperation** in a heterogeneous multiresource environment **in every aspects of the HIRE dynamics**:

- application multiactivity specification,
- declarative multiresource constraint definition,
- multi-resource interoperation (mega)programming.

Orthogonal set of HIRE dynamics specification facilities that are object-oriented, logic-based and concurrent includes:

- **A high level Petri net based script-oriented specification of concurrent and asynchronous behaviour of HIRE** as a discrete event dynamic system providing for definition of data and control flow between the execution components (such as subactivities or software systems).
- **Function/Predicate declaration facilities introduced to define activities/assertions over the heterogeneous information resources.** (An essential feature is powerful typed first order language expressing predicates and functions over the HIRE components serving for conditions and activities in the Petri net).
- **Composite Events (internal/external) specification, event expressions and algebra, activities/constraints enforcement and coupling rule declaration.**
- **And finally, the object-orientation of the approach allows specifying multiactivity class hierarchies, multiactivity specialization and classification.**

SYNTHESIS Language: FRAMES

The declaration of any **entity** (including the entities of the language itself, such as **types, classes, functions, assertions**) is given in the SYNTHESIS language by means of a **frame**.

Generally frame may be considered as a **structured symbolic model** of some entity or of some concept used to represent such individuals.

Syntactically a frame is always represented in brackets { and }. The **slot** names and their **values** are separated by a colon. The values of a slot are separated by commas. Atomic value, frame, collection of formulae of object calculus, set of values may be used as slot values. Different slots in a frame are separated by semicolons.

Each frame may be declared to belong to some **class**. After such declaration the frame becomes an object (an instance of the class (classes) mentioned). Such class membership is given by a slot in :< class name list > .

SYNTHESIS Language: FUNCTION SPECIFICATIONS

Syntactically each functional attribute of a type is defined by declaration of a function:

< function declaration > ::= { < function identifier >;
in : function; [params : { < formal parameter list > };]
[< specification >] }
< formal parameter identifier > ::= < parameter sort symbol > < typed variable >
< parameter sort symbol > ::= - | + | < empty >
< typed variable > ::= < variable > [/ < type qualifier >]
< type qualifier > ::= < type name > | < class name > | < attribute name >

The meaning of a parameter sort symbol:

+ input parameter; - output parameter; < empty > input & output parameter.

OBJECT-ORIENTED DATABASES

EVENT / CONSTRAINT DECLARATION

Every information resource assertion is considered to be an instance of an **assertion class**.

```
< assertion > ::= { < frame name >;
    metaframe coupling : {enum; immediate, deferred,
        decoupled, causal_decoupled}end;
    [enforcement : < event expression >; ]
    [activity : < function designator >; ]
    metaslot coupling : {enum; immediate, deferred,
        decoupled, causal_decoupled}end;
    < assertion predicate >
    }

< event expression > ::= < event term > | < event sequence > |
    < event choice > | < event iteration > | < event interleaving > |
    < parallel event composition > | < nondeterministic event choice >
```

ASSERTION EXAMPLE. Assertion predicate: projects in which an employee is participating should be developed by a company to which the employee belongs.

```
{empl_assert;
    metaframe coupling : deferred end;
    enforcement : on_update_companies(engaged_in) + every(month);
    activity : restore_consistency;
    metaslot coupling : immediate end;
    all e/employees(works_on(e) <= belongs_to(e).engaged_in) }
```

```
{companies;
in: class;
instance_section:
{name: {string;};
engaged_in: {set_of: project}}}
{employee;
in: class ;
superclass: person;
instance_section:
{ belongs_to: company;}}
```

SCRIPT TYPES

Script is used in SYNTHESIS to capture the behavior as a pattern of actions that should satisfy temporal, sequential causal requirements.

Script combines (in an object-oriented style) 1) **high level Petri net model** providing for concurrency, 2) **typed first order language facilities** used for predicate (**assertions**) and for function (**activities**) specification, 3) **event algebra**, 4) **activity coupling** declarations.

Each instance of a script type corresponds to a particular activation of a multiactivity defined by a script type. As an object, such instance is characterized by an object identifier and a collection of script attributes.

```
< script type specification > ::=
    { < type identifier >; in : script;
      [ params : { < formal parameter list >; } ; ]
      [ supertype : < supertype name list >; ]
      instance_section : { [ < attribute specification list >; ]
        [ initial : < initial marking >; ]
        states : < state section >;
        [ gates : < state section >; ]
        [ transitions : < transition section >; ] } }
```

A script models the **dynamic system behaviour** in terms of **states and state transitions**. A net is represented by a **bipartite directed graph** arcs of which connect nodes taken from two sets: a set of states and a set of transitions.

Script dynamics are modelled by **typed tokens** that are produced in the initial states of a script on its creation, in the output states of transitions or are coming to the external states of a script from the outside. In one script instance a number of tokens of different types can coexist. The states are places where collections of tokens can be accumulated to wait for appropriate conditions to enter transitions.

SCRIPT TYPES (II)

< transition specification > ::=

{< transition name >; }

from :< list of input state names >; [bind_from :< binding list >;]

to :< list of output state names >; [bind_to :< binding list >;]

A transition specification may include a list of **transition conditions** that should be simultaneously satisfied in order that a transition could be fired and a description of **activity** (function declaration that may call another activities) that should be taken on such firing:

[conditions :< assertion list >;]

activity :< function declaration > }

Types of input (output) tokens of a transition should coincide or be the subtypes of the types of the corresponding input (output) parameters of the transition activity.

A **state (place)** description of a script can include a definition of **assertions** that should be satisfied in order that a token which is placed into the state could be **activated**.

< state specification > ::=

{< state name >;

[token : {< identifier >:< type >; }]

[assertions : {< assertion list >}]}

The external input states may be defined by the functional attributes of a script. **The external output states** are declared in the state section as **gates**. In a **to** list of a transition references to external states of another scripts (through **gates**) are allowed.

SCRIPT TYPES (III)

On definition of a transition the rules for **binding** its input and output may be given.

< element of binding list > ::=
 {< state identifier > [, < input/output parameter name >
 [, < factor >][, < condition >]}
< factor > ::= < variable > | < integer >

Binding rules provide for establishing of the correspondence of tokens ingoing from (outgoing to) the defined states to input (output) parameters of the transition activity. Declare also **logical conditions** that tokens ingoing from (outgoing to) the defined states should satisfy and (or) declare **a number of tokens** that should be **consumed (produced)** on the input (output) of the transition. On default this number is equal to one.

A collection of active tokens taken from all of the input states (selected in the given order in quantity established for each input state) should be considered together to **fire** the transition. If the transition fires, all tokens of this collection are removed from the input states. If several transitions are conflicting for consuming of the same tokens for the transitions to be fired, nondeterministic choice of the transition is made.

Hierarchical scripts in which a transition may be substituted by the whole (child) script can be defined. The parameters of the **child script** should include input and output **states** of the transition that should be substituted. In the action of the transition to be substituted instead of the function declaration the child script can be referred as follows.

< child script > ::= {< script identifier >;
 in : script; [params : {< formal parameter list >}]; }

Type **state** may be used for typing of formal parameters of a transition and of a child script.

AN EXAMPLE OF HIERARCHICAL SCRIPT DEFINITION

<pre> {trscript; in: script; states: {s1; ...}; {s2; ...}; transitions: </pre>	<pre> {Q; from: ss1; to: ss3, sp2; activity: {subscript2; in: script; params: {ss1, ss3, sp2 }} }, </pre>
<pre> {A; from: s2; to: s1; activity: {subscript1; in:script; params: {s2, s1 } } }, </pre>	<pre> {R; from: ss3, sp1; to: ss2; activity: ... } }; </pre>
<pre> {B; from: s1; to: s2; activity: ... }; }; </pre>	<pre> {subscript2; in: script; params: {ps1/state, ps2/state, ps3/state }; states: {sss1; ...}; transitions: </pre>
<pre> {subscript1; in: script; params: {sp1/state, sp2/state}; states: {ss1; ...}, {ss2; ...}, {ss3; ...}; transitions: </pre>	<pre> {U; from: ps1; to: sss1, ps2; activity: ... }, </pre>
<pre> {P; from: ss2; to: ss1; activity: ...}, </pre>	<pre> {V; from: sss1; to: ps3; activity: ... }} </pre>

SCRIPT DEFINITION

Script type may be abstracted as a variant of coloured Petri nets:

- $(P, T, A; N)$ is a **net structure** (directed bipartite graph) with finite set of places P , finite set of transitions T , finite sets of arcs A and a node function N from A into $P \times T \cup T \times P$;
- $C : P \rightarrow \text{Type}$ is a **type function** mapping places into Synthesis types; Each token of this place must be of the given type or of its subtypes.
- $Ap : P \rightarrow \text{Oce}^*$ is an **activation predicate function** mapping places into object calculus expressions defining formulae that should be satisfied for a token of the place to become active;
- $G : T \rightarrow \text{Oce}$ is a **guard function** mapping each transition into an object calculus expression defining a predicate that should be satisfied for a transition to fire;
- $F : T \rightarrow \text{Func}$ is an **activity function** mapping each transition into a function definition determining a function to be applied on firing of a transaction; binding of input and output parameters of the function is given by an arc expression function;
- $E : A \rightarrow (\text{ParxOce} \times \text{Z})^*$ is an **arc expression function** mapping each arc into an object calculus expression defining a subset of transition input (output) sets of tokens that might be bind to some input (output) parameter of a transition function. This subset may be forwarded from (into) the state on transition firing. Additionally an integer may be added stating a number of tokens that might be consumed (produced) on transition firing in this particular binding;
- $H : T \rightarrow \Sigma$ is a **labelling function** mapping transitions into an alphabet that includes also an empty sequence;
- $I : P \rightarrow (\text{VxZ})^*$ is an **initialization function** mapping each place into an expressions giving values of the token type defined for the place and a number of those values that should be created initially in the places.

SCRIPT TYPE = INTERACTIVE TYPE

Give possibility to impose:

- **chronological order** on operation calls on a type instance interface
- **concurrent dependencies** on type operations
- **choice of the type behaviour** influenced by its environment
- **environment constraints** on the type behaviour, etc.

Classical "effective computability" was **purely algorithmic**. Interactive computations cannot inherently be completely specified by the first order predicate logics. Turing machines are not enough.

Each **script type operation** can be defined by a **script** expressing its historical and concurrent interrelationships with other operations

We treat a **workflow specification as a script type** with operations represented by **subscripts** including places and transitions embedded by the ingoing and outgoing places of the operations.

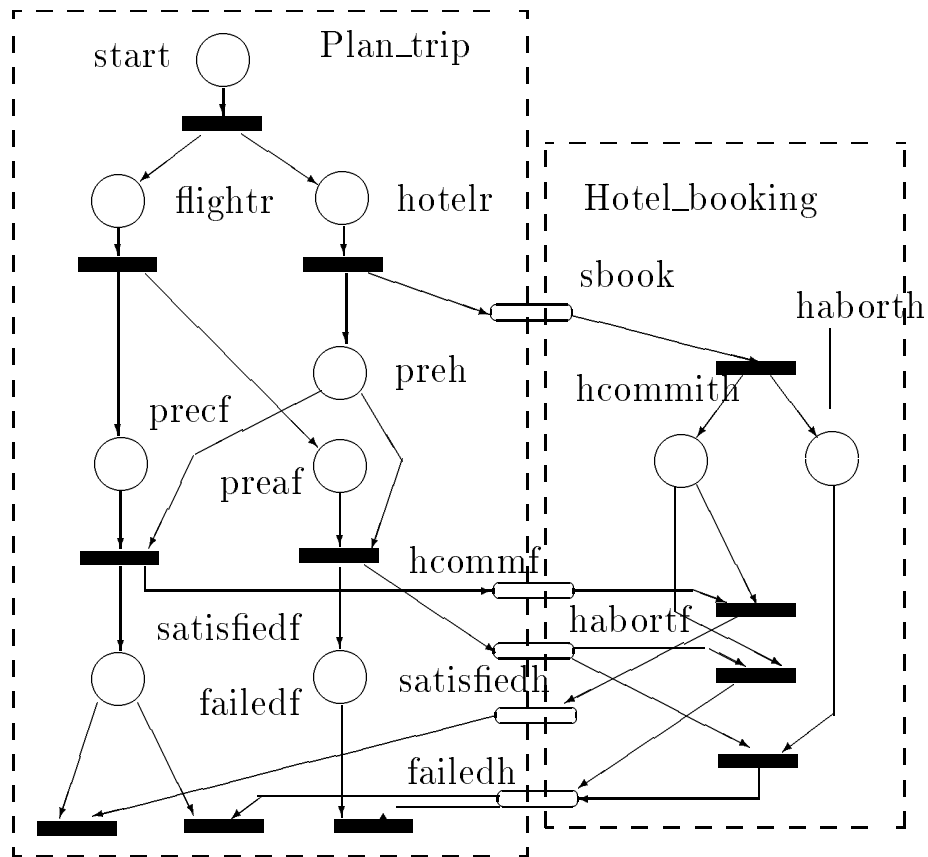
All considerations and specification algebra including decomposition and composition introduced first for conventional types are applicable also to script (interactive) types.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(77)



MULTITRANSACTION SPECIFICATION EXAMPLE

Application of the **script-based model to the definition of the semantics of the subtransaction execution dependencies** in a multitransaction model. **Termination dependency** will be shown here assuming the existence of the prepare-to-commit state in subtransactions.

Planning a trip includes two subgoals: to find a seat on a flight and reserving a hotel. A hotel should be reserved only if a flight was rented. A script **plan_trip** is declared as a subtype of a nested transaction type. An instance of this type is a **top transaction making a planning**. Two **concurrent subtransactions** are started.

PLAN TRIP (TOP TRANSACTION SCRIPT) I

```
{plan_trip; in: script;
supertype: nested_transaction ;
initial: start;
for_whom: person;
line: airline;
destination: city;
date: time;
which_hotel: hotel;
period: time;
satisfiedh: {in: function; params: r/reservation }
failedh: {in: function };
  states:
    {pref; assertion: {{flight ≠ null }}},
    {preaf; assertion: {{flight = null }}},
    {satisfiedh; },
    {failedh; };
    {satisfiedf; assertion: {{flight ≠ null }}},
    {failedf; assertion: {{flight = null }}};
  gates: {hcommitf; }, {habortf; }, {sbook; };
  transitions:
    {from: start; to: flightr, hotelr; activity: },
    {from: flightr; to: pref, preaf;
      activity: {flres; in: function; params: -flight/flight,-seat/seat,
        -scr/self;
{comment; Types flight, seat, etc. are assumed to be declared};
  {{spawn(reserve(for_whom,line,destination,date,flight,seat));
  assocval(scr,this) }}}},
{comment; assocval is a built-in function associating an object value (here
this) with an object identifier(scr); },
```

PLAN TRIP (TOP TRANSACTION SCRIPT) II

```
{from: hotelr; to: preh;
  activity: {book; in: function; params: -obj/hotel_booking;
    {{ new(obj, hotel_booking);
{comment; New instance of the hotel_booking script is created.};
    sbook(obj, for_whom, which_hotel, period) }}}},
{comment; New instance of the hotel_booking script is started parallelly
through the gate.}
  {from: precf, preh; to: satisfiedf;
    activity: {hcomm; in: function;
      params: flight/flight, seat/seat, +scr/plan_trip, +obj/hotel_booking;
      {{ hcommitf(obj, scr) }}}},
{comment; To inform the hotel_booking script that it may be committed.},
  {from: preaf, preh; to: failedf;
    activity: {habor; in: function; params: +scr/plan_trip, +obj/hotel_booking;
      {{ habortf(obj, scr) }}}},
{comment; To inform the hotel_booking script that it should be aborted.},
  {from: satisfiedf, satisfiedh; to: ;
    activity: {csatisf; in: function; params: +flight/flight, +seat/seat,
      +r/reservation;
      {{ satisfaction_reported; remove(this) }}}},
{comment; remove leads to removing of the referred object},
  {from: satisfiedf, failedh; to: ;
    activity: {freserv; in: function; params: +flight/flight, +seat/seat;
      {{ flight_reservation_reported; remove(this) }}}},
  {from: failedf, failedh; to: ;
    activity: {{ failure_reported; remove(this) }}}
}
```

HOTEL BOOKING SUBTRANSACTION SCRIPT

```
{hotel_booking; in: script;
supertype: nested_transaction ;
sbook: {in: function; params: +for_whom/person,+which_hotel/hotel,+period/time };
hcommitf: {in: function; params: o/plan_trip };
habortf: {in: function; params: o/plan_trip };
  states:
    {sbook; },
    {hcommitf; },
    {habortf; },
    {hcommith; assertion: {{r ≠ null}}},
    {haborth; assertion: {{r = null}}};
  gates:
    {satisfiedh; },
    {failedh; };
  transitions:
    {from: sbook; to: hcommith, haborth;
      activity: {book; in: function;
        params: +for_whom/person, +which_hotel/hotel, +period/time, -r/reservation;
        {{which_hotel.actual_vacancy > 0 ->
          r.where='which_hotel & r.for_whom'=person &
          r.for_period'=period & added(r,reservation) &
          which_hotel.actual_vacancy'=which_hotel.actual_vacancy-1) }}}},
    {comment; Here hotel, person, reservation denote predefined types. Their attribute
names are self-explanatory.},
    {from: hcommitf, hcommith; to: ;
      activity: {hcom; in: function; params: +o/plan_trip, +r/reservation ;
        {{ satisfiedh(o, r); commit(this) }}}},
    {from: habortf, hcommith; to: ;
      activity: {habor; in: function; params: +o/plan_trip;
        {{ failedh(o); abort(this) }}}},
    {from: habortf, haborth; to: ;
      activity: {habor; in: function; params: +o/plan_trip;
        {{ failedh(o); abort(this) }}}
  }
}
```

DECLARATIVE INFORMATION RESOURCE CONSTRAINT SPECIFICATION

Polytransaction model. The constraints are separated from the application programs to facilitate the maintenance of data consistency requirements and flexibility of their implementation. To specify **the interdatabase dependency the Data Dependency Descriptors** D^3 include:

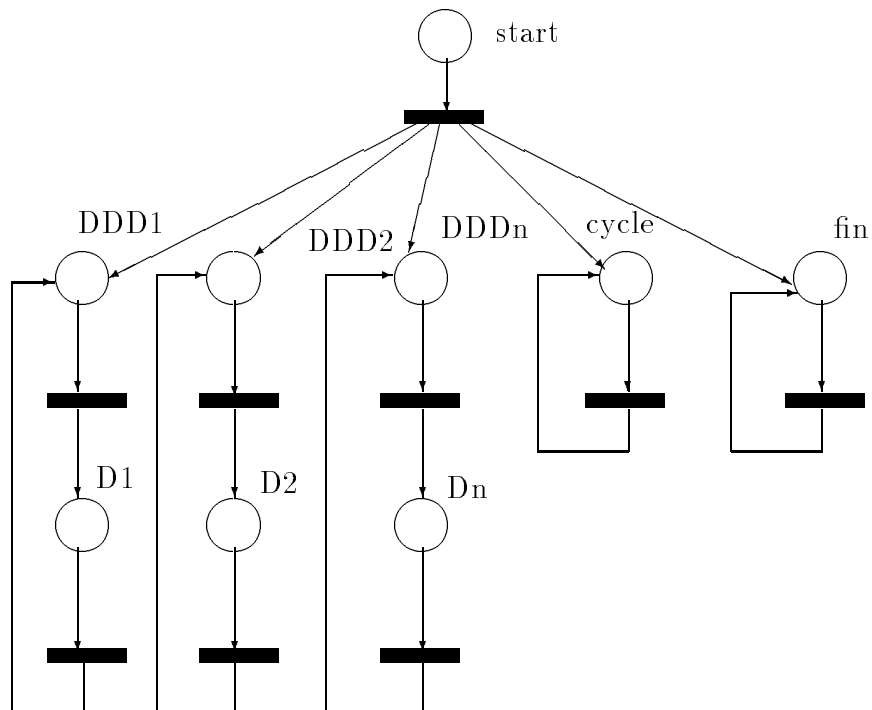
- P is a **data dependency predicate**, specifying a relationship between the data objects;
- C is a **mutual consistency predicate** stating consistency requirements and defining when P must be satisfied;
- A is a collection of **consistency restoration procedures** needed to restore consistency and to ensure that P is satisfied.

Interdatabase Dependency Schema (IDS) is a set of all D^3 to be enforced in a multidatabase system. **A polytransaction is a "transitive closure" of a transaction T submitted to an interdependent data management system.** Several execution modes of child transactions are proposed: **coupled/decoupled, vital/non-vital.**

Transformation of the polytransaction specification into a SCRIPT:

- Data dependency descriptor D^3 \rightarrow state of a script DDD_i and a transition leading from this input state.
- Data dependency predicate P of D^3 \rightarrow an assertion associated with the corresponding state DDD_i .
- Consistency predicate C of D^3 \rightarrow a condition included into a transition associated with the corresponding DDD_i as an input state.
- Restoration procedures of the D^3 \rightarrow activity function of a transition associated with the corresponding DDD_i as an input state.

To demonstrate usage of the script concept for declarative constraint specification the **deferred** execution model for the child transactions was chosen.



POLYTRANSACTION SPECIFICATION EXAMPLE

{specific_multiactivity_name; in: script; [params : ...;]

supertype: **deferred_transaction** ;

states:

 {DDD1; assertions: { {<formula>}},

{comment; Here <formula> denotes a formula of SYNTHESES expressing the data dependency predicate P of the D^3 },

 {{ **current wave is empty** }}}

 {DDD2; assertions: { {<formula>}},

 {{ **current wave is empty** }}}

 ...

 {DDDn; assertions: { {<formula>}},

 {{ **current wave is empty** }}}

POLYTRANSACTION SPECIFICATION EXAMPLE (II)

transitions:

```
{from: start; to: DDD1, DDD2, ..., DDDn, cycle, fin;
  activity: ACTIVITY_FUNCTION (<params>)},
{comment; ACTIVITY_FUNCTION represents a top transaction function.
},
  {from: DDD1; to: D1;
    condition: {{mutual_consistency_cond_1}}};
{comment; mutual_consistency_cond is represented by a SYNTHESIS formula that may include temporal and data state consistency terms. },
  activity: RESTORATION_FUNCT_1(<params>)},
{comment; General form of a RESTORATION_FUNCTION:
restoration_function: {in:function; params: ... ;
  {{if <condition> then deferred_spawn(ci, p, ti).
if <condition> then deferred_spawn(cj, p, tj).
...
if <condition> then deferred_spawn(ck, p, tk).}}}}
For the restoration the script is considered to be a creator of the child transactions  $t_i, t_j, \dots, t_k$  in types  $c_i, c_j, \dots$  or  $c_k$  for their proper parent  $\mathbf{p}$ . In our case  $\mathbf{p}$  is the script itself.},
  {from: DDD2; to: D2;
    condition: {{mutual_consistency_cond_2}}};
    activity: RESTORATION_FUNCT_2(<params>)},
    ...
  {from: DDDn ; to: Dn;
    condition: {{mutual_consistency_cond_n}}};
    activity: RESTORATION_FUNCT_n(<params>)},
```

POLYTRANSACTION SPECIFICATION EXAMPLE (III)

{comment; Transitions from states D1,D2, ..., Dn defined below take place when deferred subtransaction t_i, t_j, \dots, t_k generated on transition from DDDm to Dm had been executed. To detect this situation it is sufficient to check that the set of generated subtransactions is not a subset of both current_wave and next_wave sets.},

{from: D1; to: DDD1;
 condition: $\{\{ \neg((\{t_i, t_j, \dots, t_k\} \subseteq \text{current_wave}) \vee \{t_i, t_j, \dots, t_k\} \subseteq \text{next_wave}))\}\}$ };
 activity: },

{from: D2; to: DDD2;
 condition: $\{\{ \neg((\{t_i, t_j, \dots, t_k\} \subseteq \text{current_wave}) \vee \{t_i, t_j, \dots, t_k\} \subseteq \text{next_wave}))\}\}$ };
 activity: },

...

{from: Dn; to: DDDn;
 condition: $\{\{ \neg((\{t_i, t_j, \dots, t_k\} \subseteq \text{current_wave}) \vee \{t_i, t_j, \dots, t_k\} \subseteq \text{next_wave}))\}\}$ };
 activity: },

{from: cycle; to: cycle;
 condition: $\{\{ \text{current wave is empty and next wave is not empty, no transition is possible but this one } \}$ };
 activity: cycle(this) },

{from: fin; to: ;
 condition: $\{\{ \text{current wave is empty and next wave is empty, no transition is possible but this one } \}\}$ };
 activity: commit(this) }}

MULTI-RESOURCE PROGRAMMING EXAMPLE

Support of the heterogeneous computing environment consisting of autonomous software systems using the script-based model.

Example. Forming class **organization** over the distributed heterogeneous classes **firm**, **company**, **enterprise** interpreted by different SQL-like databases (oracle1, oracle2, db2 correspondingly).

{organization;	{firm;	{company;	{enterprise;
in:class;	in:class;	in:class;	in:class;
name: string;	fname: string;	cname: string;	ename: string;
industry: string;	ind: string};	trade: string;	turnover: real;
hq: string}		city: string}	hq: string;
			profile: string }

The SYNTHESIS rule forming the class **organization** looks as follows:

```
+organization(o):- {o [name,industry,hq] | industry='Hotel'  
∨ industry='Banking' & (firm(name/fname, industry/ind)  
& company(name/cname, industry/trade, hq/city)) ∨  
turnover > 10E7 & enterprise(name/ename, industry/profile, hq) }
```

According to this rule, an **object instance of the organization class** is created as an **intersection of firm and company classes** belonging to the Hotel or Banking industries and **union of enterprise class instances** with a turnover greater than 10^7 .

The rule is compiled into the following script:

```
{org_eval;in: script;  
tr1_status:{integer} metaslot init: 0 end;  
tr2_status:{integer} metaslot init: 0 end;  
tr3_status:{integer} metaslot init: 0 end;  
tr4_status:{integer} metaslot init: 0 end;  
tr5_status:{integer} metaslot init: 0 end;  
initial: start;
```

MULTI-RESOURCE PROGRAMMING EXAMPLE (II)

states:

```
{st6;assertion: { {¬ (tr1_status=0 | tr2_status = 0 |  
tr3_status=0 | tr4_status = 0 | tr5_status=0) } } };
```

transitions:

```
{tr0; from: start; to: st6, stin1, stin2, stin3;  
activity: },
```

```
{tr1; from: stin1; to: st1;  
activity: {firm_restr; in: function; params: [-rf];  
  { { open('oracle1', site1, tr1_status);  
    task('oracle1', tr1_status, [ ], [rf],  
    {create view rf(name, industry)  
      as select fname as name, ind as industry  
        from firm  
          where ind='Hotel' or ind='Banking' } } );  
    close('oracle1',tr1_status) } } } },
```

```
{tr2; from: stin2; to: st2;  
activity: {company_restr; in: function; params: [-rc];  
  { { open('oracle2', site2, tr2_status);  
    task('oracle2', tr2_status, [ ], [rc],  
    {create view rc(name, industry, hq)  
      as select cname as name, trade as industry, city as hq  
        from company  
          where trade='Hotel' or trade='Banking' } } );  
    close('oracle2', tr2_status) } } } },
```

MULTI-RESOURCE PROGRAMMING EXAMPLE (III)

```
{tr3; from: stin3; to: st3;
  activity: {enterprise_restr; in: function; params: [-re];
    {{ open('db2', site3, tr3_status);
      task('db2', tr3_status, [ ], [re],
        {create view re(name, turnover, hq)
          as select name, turnover, hq
            from enterprise
              where turnover > 10E7});
      close('db2', tr3_status) }}},
{tr4; from: st1, st2; to: st4;
  condition: {{ tr1_status=0 & tr_2status=0 }};
  activity: {rfc_form; in:function; params: [+rf,+rc,-rfc];
    {{task('synthesis', tr4_status, [rf,rc], [rfc],
      {intersect(rf, rc, rfc)}})}}},
{tr5; from: st3, st4; to: st7;
  condition: {{ tr3_status=0 & tr_4status=0 }};
  activity: {organization_form; in:function; params: [+re,+rfc,
    -organization];
    {{task('synthesis', tr5_status, [re,rfc], [organization],
      {union(rfc, re, rfce);
        formclass rfce, organization)}}}}},
{tr6; from st6; to: fin;
  activity:{{report failure}}};
{tr7; from st7; to: fin;
  condition: {{ tr5_status = 0 }};
  activity: {{report success}}}}
```

Local commands written in the language of the software system are submitted by the **task** function of the format:

```
task('service_name', status, '['list of service input variables']', '['list of service output variables']', {commands of the service in its native lang })
```

STRUCTURAL ASPECTS OF EXTENDED MULTIACTIVITY MODELING

- **Generalization of models** A cooperative transaction hierarchy is a structured set of cooperative transactions. The internal nodes are *transaction groups* and the leaves are cooperative transactions. *Operation machines* are user definable synchronization mechanisms for specifying patterns and conflicts reflecting cooperative transaction structural dependencies. An operation machine is a finite state automata.

Brown University, M.Nodine, S.Zdonik

- **Declarative intercomponent dependency specification**

First-order logic based formalism declaratively specifying component transaction and related dependencies that should be preserved. Structural dependencies, operational effects, visibility rules are specified.

ACTA Framework, K.Ramamritham, P.Chrysantis

- **Dependency finite states automata** Structural dependencies are modelled as a dependency finite state automaton whose paths represent the computations that satisfy the dependency. Temporal propositional logic is used to specify the dependencies. The scheduler receives events corresponding to a possible task execution. It queries the applicable dependency automata to execute the event.

MCC, Carnot, P.Attie, A.Sheth, M.Rusinkiewicz

- **Transaction Specification and Management Environment (TSME)** is proposed as a facility to support the definition and construction of specific extended transaction models corresponding to application requirements. The basic idea is to separate transaction specification from implementation. TSME specification language must provide modeling primitives similar to those in the ACTA framework. ECA rules are used for enforcing of the structural dependencies.

GTE Labs, D.Georgakopoulos, F.Manola, M.Hornick

SEQUENTIAL SAGA STRUCTURAL DEPENDENCIES

SAGA: is a set of relatively independent (component) transactions T_1, T_2, \dots, T_n that execute in a predefined order (e.g., sequential). Each component transaction ($T_i (0 \leq i < n)$) is associated with a compensating transaction CT_i . T_i and CT_i behave like ACID transactions (they can commit without waiting for other component transactions or the saga to commit). SAGA commits if all its component transactions commit in the prescribed order. On abort compensating transactions are executed in the reverse order of commitment of the component transactions

Structural dependency axioms for sequential SAGAs:

- **Constituent transaction sequentiality:**

$$\text{post}(\text{Begin}_S) \rightarrow T_i \text{ BCD } T_{i-1} \wedge CT_j \text{ WCD } CT_{j+1} \wedge CT_{n-1} \text{ BAD } S \\ 1 < i \leq n, 1 \leq j < n - 1$$

- **The relationship between a saga and the components**

$$\text{post}(\text{Begin}_{T_i}) \rightarrow S \text{ AD } T_i \wedge T_i \text{ WD } S \wedge CT_i \text{ BCD } T_i; \quad 1 \leq i < n$$

- **Compensation enforcement**

$$\text{post}(\text{Commit}_{T_i}) \rightarrow CT_i \text{ CMD } S \wedge CT_i \text{ BAD } S; \quad 1 \leq i < n$$

- **Commit of the whole SAGA:** $\text{post}(\text{Begin}_{T_n}) \rightarrow S \text{ SCD } T_n$

AD = abort dependency: if T_i aborts, S aborts

BAD = begin-on-abort-dependency: CT_{n-1} cannot begin execution until S aborts

BCD = begin-on-commit-dependency: CT_i cannot begin execution until T_i commits

CMD = force-commit-on-abort dependency: if S aborts, CT_i commit

SCD = strong-commit-dependency: if T_n commits then S commits

WCD = weak-begin-on-commit dependency: if CT_{j+1} commits CT_j can begin executing after that commit

WD = weak-abort-dependency: if S aborts and T_i have not yet committed then T_i aborts

CHUNK MULTIACTIVITY MODEL

Multiactivity is treated as a **structure that is imposed on a collection of constituent "activities" or tasks**. Interfaces of activities are assumed to be defined in some interface definition language (e.g., IDL).

Chunk is a piece of a multiactivity between "significant events".

Multiactivity is abstracted as a **sequence of chunks** with significant events in between.

Compensating chunk (cochunk) is a chunk providing of compensating activity for the respective chunk that might have committed before committing of the whole multiactivity or of the parent chunk.

Chunkpack is a sequence of chunks separated by call/return of the components. Each component activity is represented as a chunk or a chunkpack.

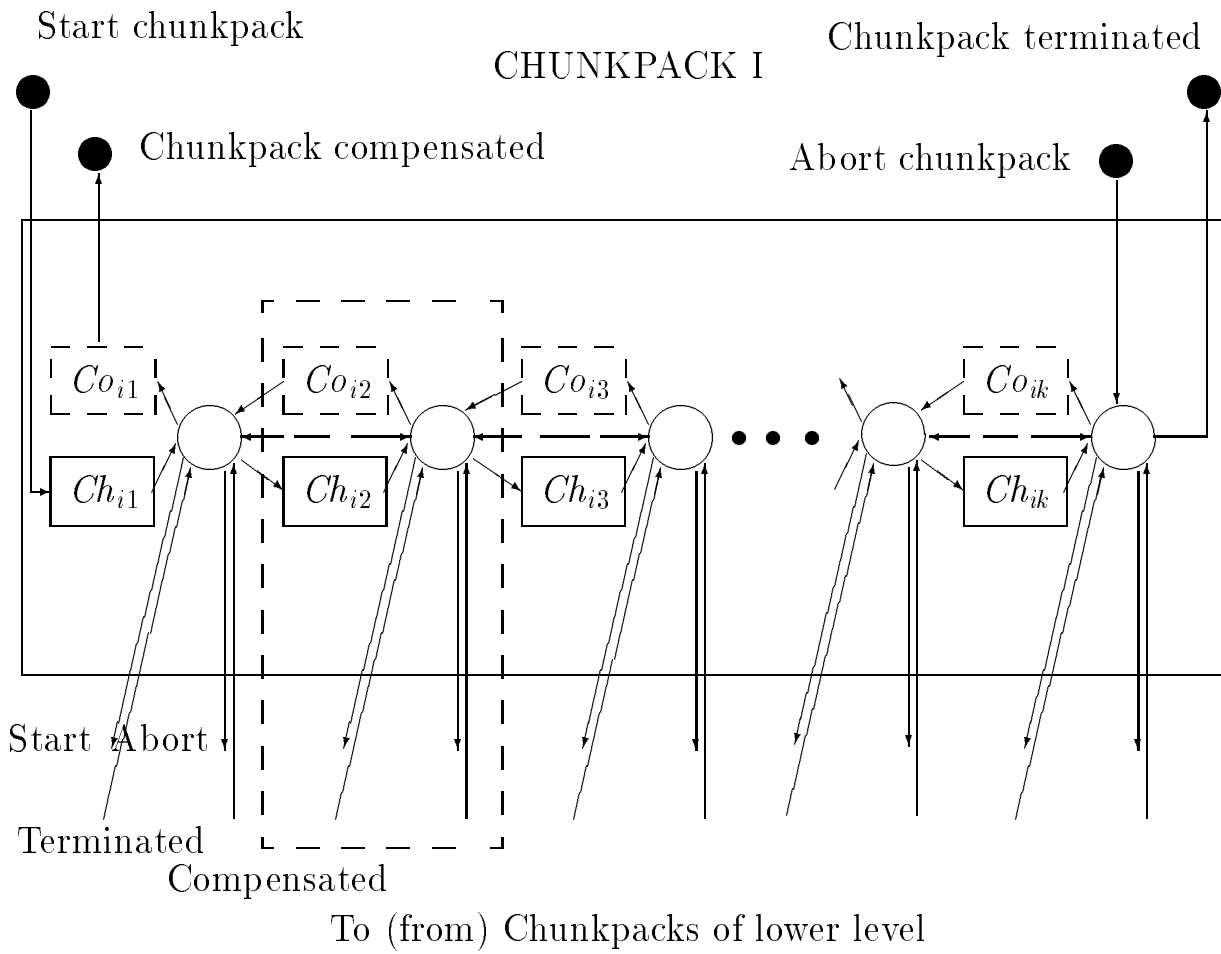
General structure of the chunkpack with the interface:

1. start(ChP_i, params) to call execution of the pack
2. abort(ChP_i, params) that should cause rollback/compensate

```
chunkpackChPi(params)
    chunk : Chij[(params)]; cochunk : Coij[(params)];
        [start(ChPm[(params))]; ][abort(ChPm[(params))]; ]
    chunk : Chik[(params)]; cochunk : Coik[(params)];
        [start(ChPn[(params))]; ][abort(ChPn[(params))]; ]
    chunk : Chil[(params)]; cochunk : Coil[(params)];
        [start(ChPo[(params))]; ][abort(ChPo[(params))]; ]
    ...
end
```

Chunk junctions (Intercomponent Control Modules) accumulate facilities for proper interpretation of significant events preserving **interactivity dependencies** defined for a **specific multiactivity model**

Using these simple abstractions, a **multilayered structure** can be specified where on each layer **several chunkpacks** may be located. A chunkpack is convenient to represent by a hierarchical script.



CHUNKPACK STRUCTURE

JUNCTION SCRIPT

The **junction script** for each constituent of the skeleton should be individually generated as a **child script** from the generic junction script once designed for each specific multiactivity (multitransaction) model.

The generation consists in providing of parameters for the generic script to set parameters for concrete chunk and cochunk and to call a child chunkpack and its compensation actions (if such child is required).

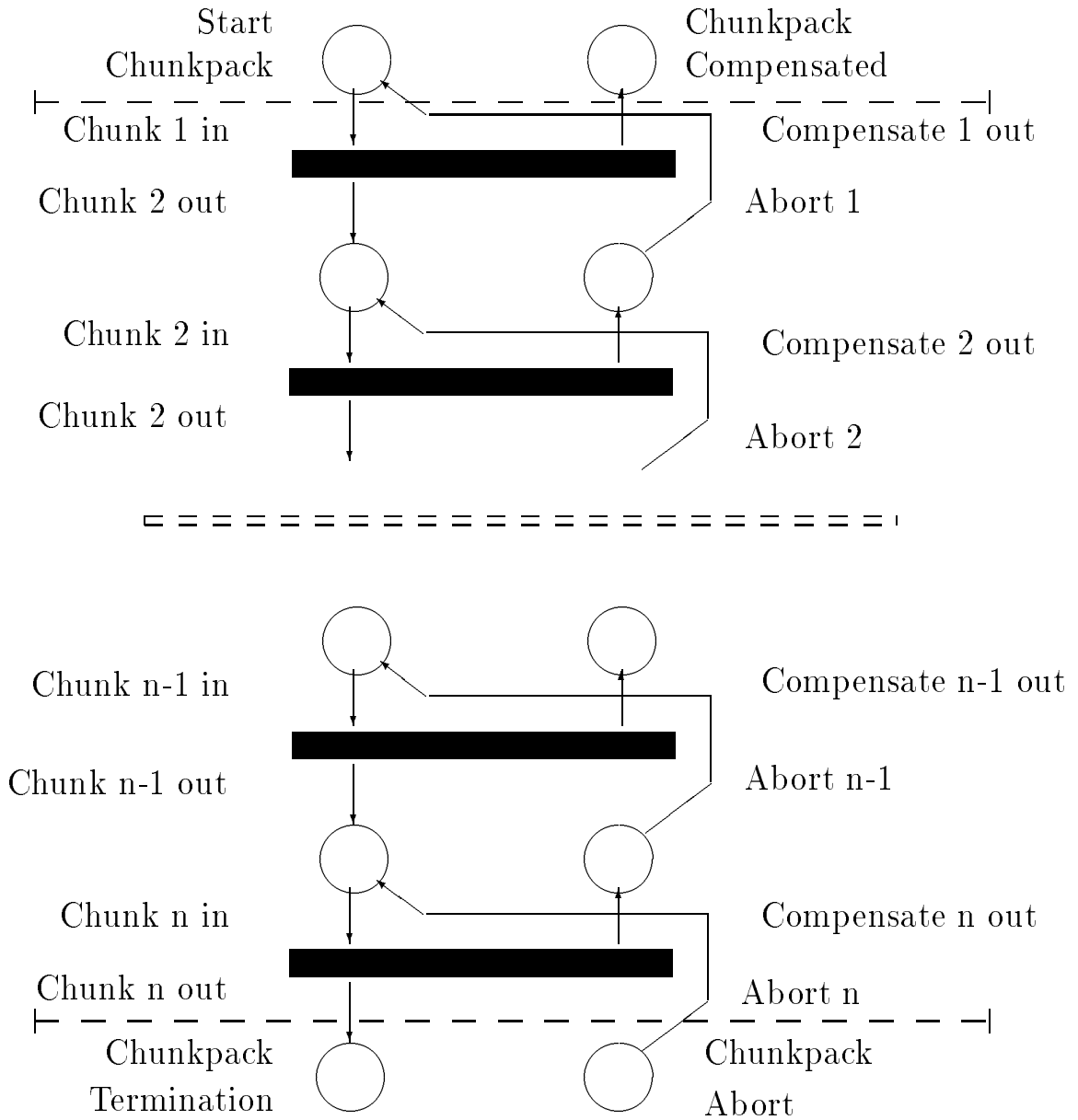
Therefore, a generic junction script (**Generic Intercomponent Control Module** or GICM) should abstract control features suitable for a **particular multiactivity model** that could be reused in the structure of a particular multiactivity represented by its skeletons.

In GICM we can capture **multiactivity dependencies** such as commit-dependency, abort-dependency, weak-abort-dependency, termination-dependency, exclusion-dependency, compensation-dependency, begin-dependency, serial-dependency, etc.

These dependencies may include also **execution order, executional conflicts, causal dependencies, mutual exclusion, termination dependency, exclusive dependency**, etc. Such dependencies are expressible in the script-based facilities.

The **correctness properties** of different multiactivity models are also expressible (e.g., 1) an activity A can occur only after another activity B ; 2) an activity A can occur only if certain condition is satisfied (a necessary condition for the activity); 3) reaching of a condition requires the occurrence of an activity A (a sufficient condition for the activity)).

CHUNKPACK SKELETON



DEFINITION OF GENERIC SAGA JUNCTION SCRIPT

The script reflects dependencies between significant events (begin, commit, abort, rollback, compensate) of a subtransaction and between subtransactions for a sequential SAGA model.

```
{saga_junction;  
in: script;  
  params: {out_par_type:type};  
{comment; the param out_par_type is used to generate a specific junction  
script };  
instance_section:  
  params: {subtr_obj: transaction; post_subtr/state,  
  abort/state, subtr_out/state,  
  precomp/state, comp_out/state };  
{comment; the states and subtransaction object should be passed from the  
chunkpack containing this junction};  
  initial: {fixed, 1, none/tnone};  
  states: {post_subtr; token: {tstr: {ttype; in:type;  
  term: {enum; enum_list:{OK:,NOK:} },  
  par: [ out_par_type ] }},  
  {abort;}, {subtr_out;}, {precomp;}, {comp_out;},  
  {exok;}, {fixed; }, {prerollb;}, {postcomm; };
```

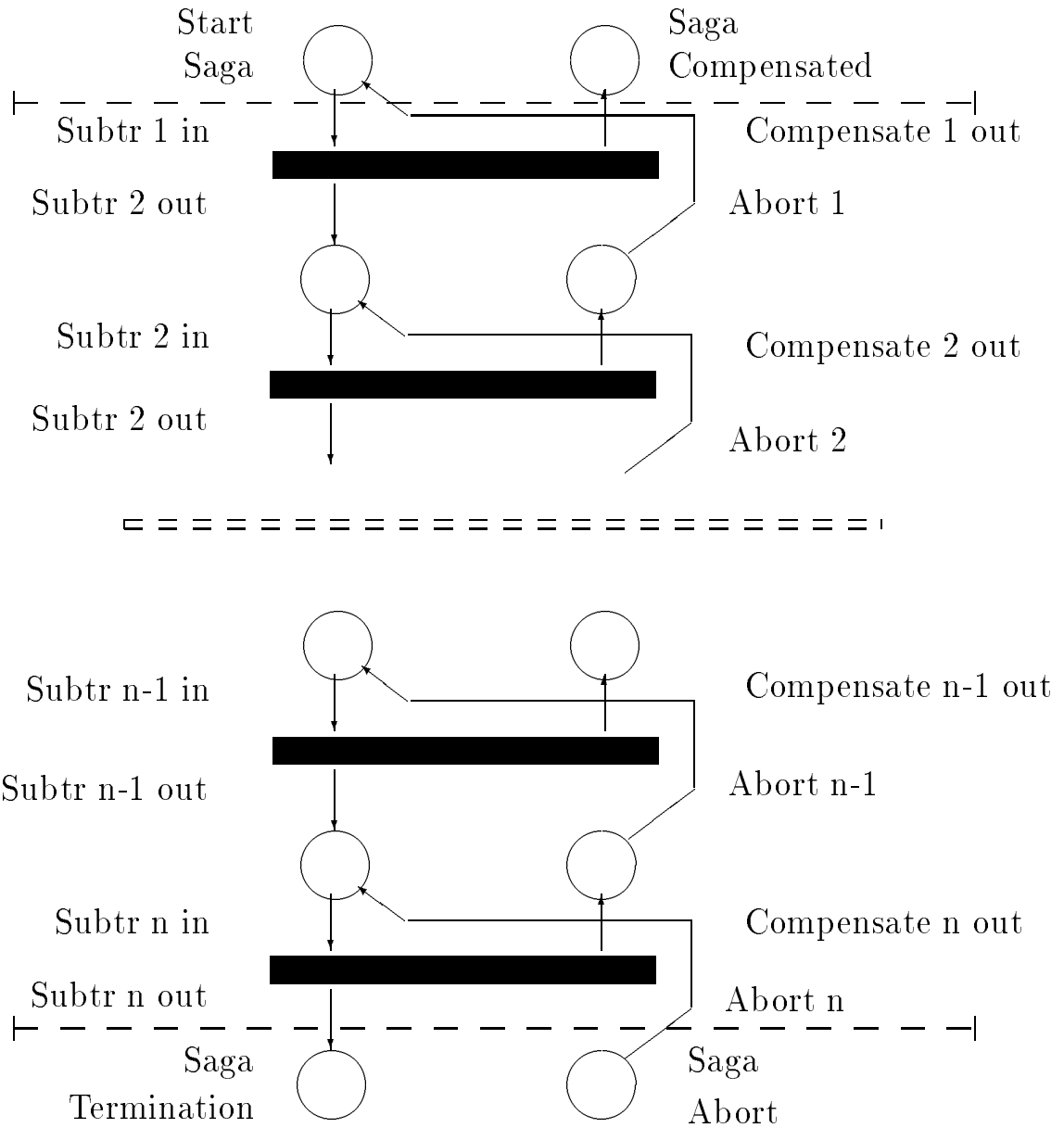
OBJECT-ORIENTED DATABASES

IPI RAS

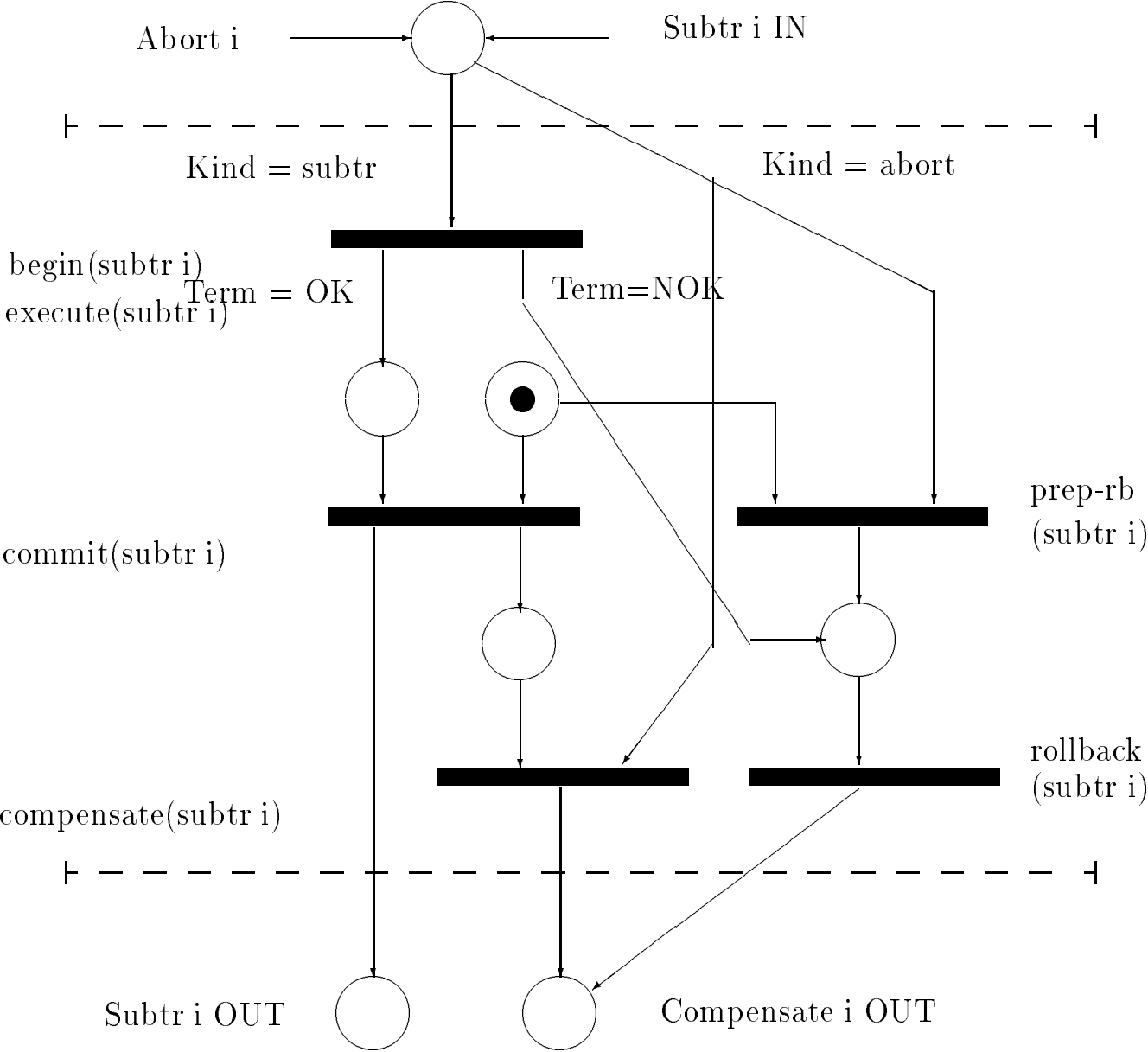
Leonid Kalinichenko

(95)

SAGA SKELETON SCRIPT



GENERIC CHILD SCRIPT FOR SAGA SKELETON



DEFINITION OF GENERIC SAGA JUNCTION SCRIPT (II)

transitions:

```
{postact;
  from: post_subtr;
  to: exok, prerollb; bind_to: {exok; term='OK' },
  {prerollb; term='NOK' };
  action: {dummy; in: function;
  params: +i/par, -o/par;
  {{ o'=i }}}},
{comm;
  from: exok, fixed;
  to: subtr_out, postcomm;
  action: {commitment; in: function;
  params: +i/par, -o/par;
  {{commit(subtr_obj); o'=i }}}},
{precompens;
  from: postcomm, abort; to: precomp;
  action: {{}}};
{comment; it is assumed that compensation function has no params; any
params could be delivered; in that case abort of subtr or of a saga should
provide parameters for ANY of subtr}; },
  {preroll;
    from: post_subtr, fixed, abort; to: prerollb;
    action: {{}} },
  {rollb;
    from: prerollb ; to compens_out: ;
    action: {{rollback(subtr_obj)}}}
};
```

ROLE OF THE CHUNK MODEL

Chunk model provides for **decomposition** of the multiactivity **concretization** problem into subproblems for which the conventional **refinement** methodology could become applicable. Such decomposition is based on the following principles:

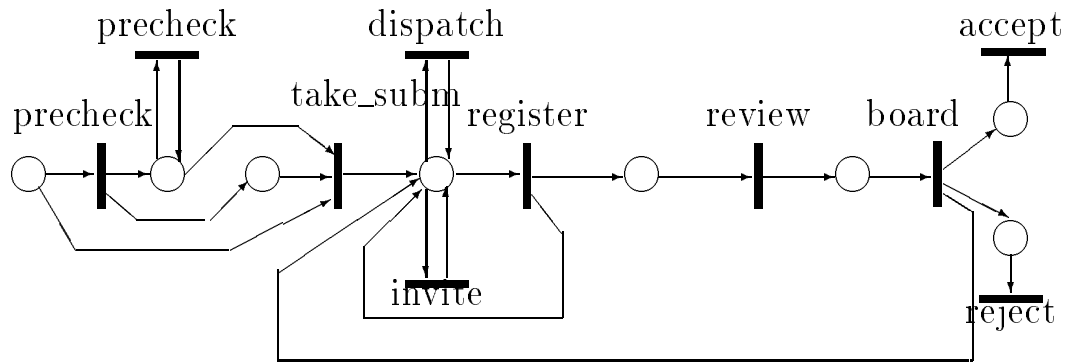
- **isolation** of the pure **functional** kind behaviour and the **concurrent** kind;
- **structural localization of concurrency control** in GICM (**junctions**);
- **multiactivity specification abstraction** of particular concurrency model. The specification can be made free of unnecessary concurrency details (imposed by efficiency, reliability and implementation constraints).

Application of these principles leads to the following **general framework** suitable for the multiactivity concretization:

1. treatment of chunks as **pure functional** behaviour **refining** them using conventional refinement technique;
2. **aggregation of chunks** linked by junctions having no specific concurrency requirements;
3. **imposing of a refinement order** on various generic junction specifications;
4. **treating separately** stepwise concretization of junctions with a specific **concurrency requirement**.

WORKFLOW COMPONENT-BASED DESIGN

- Basic notions of process algebras, bisimulation
- Script processes and refinement of scripts
- Script type reducts and conformances
- Process of script design with reuse



AGENCY WORKFLOW SPECIFICATION

```

{agency_workflow;
in: script;
instance_section:
{
states:
{new_prop; token: {npt: Proposal}},
{next_check; token: {nct: Proposal}},
{checked; token: {cht: Proposal}},
{assign_experts; token: {aet: Review}},
{pre_rev; token: {prt: Review}},
{post_rev; token: {art: Review}},
{for_accept; token: {fat: Review}},
{for_reject: {frt: Review}};

```

{comment; We assume that the mutable types **Proposal** and **Review** are defined elsewhere. The type **Review** is a subtype of the **Proposal** type. Note, please, that tokens used by the script obtain the **Proposal** or the **Review** type. }

AGENCY WORKFLOW SPECIFICATION (II)

transitions:

```
{  
    from: new_prop;  
    bind_from:  
        {new_prop, Proposal.leader.works_for <> 'Academy' };  
    to: next_check, checked;  
    bind_to:  
        {next_check, Proposal.checked = F},  
        {checked, Proposal.checked = T };  
    activity: {precheck; in: script;  
        params: {sb/Proposal }}  
},
```

{comment; The **precheck** activity assures that a submitted proposal contains all formally required information. This activity assumes a correction iteration with the authors (if required). **Proposal.checked** reflects whether a preliminary check (**precheck**) for a proposal had been completed. If after the iteration the check is positive, **Proposal.checked** is set to T otherwise to F. Proposals coming from the Institutes of the Academy of Sciences do not need prechecks.

```
}  
{  
    from: next_check;  
    bind_from:  
        {next_check, Proposal.checked = F };  
    to: next_check;  
    activity: {precheck; in: script;  
        params: {sb/Proposal }}  
},
```

{comment; Here a "loop" of prechecks is planned to make a proposal formally clean. }

AGENCY WORKFLOW SPECIFICATION (III)

```
{
  from: new_prop, next_check, checked;
  bind_from:
    {new_prop, Proposal.leader.works_for = 'Academy' },
    {next_check, Proposal.checked = T };
  to: assign_experts;
  activity: {take_subm; in: script;
    params: {+sb/Proposal, -rev/Review }}
},
{ comment; The Review type as a subtype of the Proposal type adds to the
Proposal type an attribute defined as an array of expertises nominated to dif-
ferent experts. Preparation of expertises includes nomination of experts (the
process of nomination is reflected by marking in the expertise data the cur-
rent experts' reachability (Review.Expertise[i].reach) and their agreement to
take responsibility for an expertise (Review.Expertise[i].taken). The take_subm
activity forms an instance of the Review type. On creation of the instance,
values of the mentioned attributes are set to Null. Review.expnumb contains
actual number of experts nominated for reviewing of a proposal. On creation
of a Review instance, this value is set to 0). }
{
  from: assign_experts;
  bind_from:
    {assign_expert, Review.expnumb = 0 or
    Review.Expertise[Review.expnumb].reach = F or
    (Review.Expertise[Review.expnumb].reach = T &
    Review.Expertise[Review.expnumb].taken = F)
    };
  to: assign_experts;
  activity: {dispatch; in: script;
    params:{rev/Review}}
},
```

AGENCY WORKFLOW SPECIFICATION (IV)

{comment; The dispatch activity tries to nominate next expert candidate and checks whether he/she is reachable. Returns Review.Expertise[Review.expnumb].reach = F if the expert is not reachable. If an expert is a "staff" expert he need not be invited and can be nominated by dispatch. In this case Review.Expertise[Review.expnumb].taken gets T value. }

{

```
from: assign_experts;
bind_from: {assign_expert, Review.expnumb <> 0 &
  (Review.Expertise[Review.expnumb].reach = T &
  Review.Expertise[Review.expnumb].taken = Null) };
to: assign_experts;
activity: {invite; in: script; params:{rev/Review}}
```

},

{comment; The invite activity communicates with the next expert candidate and checks whether he/she agrees to provide an expertise for this proposal. The activity finally sets Review.Expertise[Review.expnumb].taken = F or T depending on such decision. }

{

```
from: assign_experts;
bind_from: {assign_expert, Review.expnumb <> 0 &
  (Review.Expertise[Review.expnumb].reach = T &
  Review.Expertise[Review.expnumb].taken = T) };
to: assign_experts, pre_rev;
bind_to: {assign_expert, Review.expnumb < K },
  {pre_rev, Review.expnumb = K };
activity: {register; in: function; params: {rev/Review}}
```

},

{comment; The register activity finalizes registering of the next expert. }

AGENCY WORKFLOW SPECIFICATION (V)

```
{
    from: pre_rev;
    to: post_rev;
    activity: {review; in: script; params:{rev/Review}}
},
{comment; The review activity provides reviewing of a proposal by the nominated experts. }
{
    from: post_rev;
    to: for_accept, for_reject, assign_experts;
    bind_to:
        {assign_experts, return = 'additional_expertize' },
        {for_accept, return = 'accept' },
        {for_reject, return = 'reject' };
    activity: {board; in: script; params:{rev/Review, -return/string }}
}
{comment; The Agency board makes a decision concerning acceptance or rejection of a proposal, or in complicated cases, assigns a new reviewing process. }
{
    from: for_accept; to: :
    activity: {accept; in: function; params:{rev/Review }}
},
{
    from: for_reject; to: :
    activity: {reject; in: function; params:{rev/Review }}
}
{comment; The accept and reject activities provide the required notification of the authors. }
}};
```

FLOWMARK FEATURES

FlowMark is a robust workflow management system intended to automate production workflows.

Scenario def: A **process** in FlowMark is presented as a weighted, coloured, directed graph of activities where both control and data flows occur as different kinds of edges. Activities represent a piece of work that the assigned actor or program can complete.

Two kinds of **connectors** linking activities:

1. A **control connector** which can have a **transition condition** associated to it allowing the control flow through it only when the condition evaluates to true. By defining multiple connectors starting from one activity (and having transition conditions evaluated to true) it is possible to define **parallel** control flows.
2. A **data connector** specifies the data flow between the data containers of the activities. There can also be **multiple data connectors** leaving from or incoming to one activity. The data are written into and read from the data containers by the programs associated to the activities via APIs.

Scenario management It is possible to set start conditions and exit conditions for the activities.

The start condition enables to **synchronise** the re-joining control flows since it is possible to choose the start condition to evaluate to true either if **at least one** of the incoming connectors evaluates to true or only if **all** incoming connectors evaluate to true.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(106)

MAPPING OF FLOWMARK MODEL TO THE CANONICAL WORKFLOW MODEL

1. Each **activity** A is mapped into a **transition** T_A with the activity specified corresponding to the program (manual) activity. If this is a **process activity**, a **nested script** will be created.
2. To a **connector** linking an activity A and activity B two arcs and a state P of a script correspond.

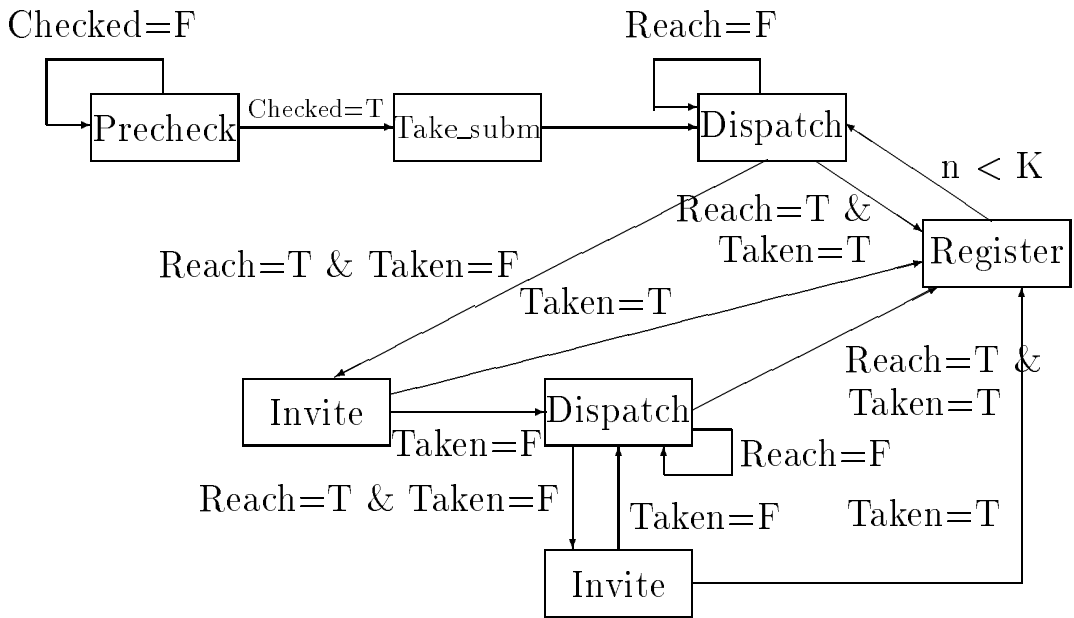
A **type of a token** corresponds to a **type of data** in the output **container** of A associated to this connector. We assume that to a **control connector** such data type also corresponds containing data required for routing. If there is a **transition condition** associated to a connector then it is mapped into a condition of an element of the **output binding list** of T_A related to P .

3. An activity A **start condition** is mapped into a **guard** of T_A (an assertion included into a **conditions**: list of T_A or into into conditions of elements of the input binding list of T_A or into both).

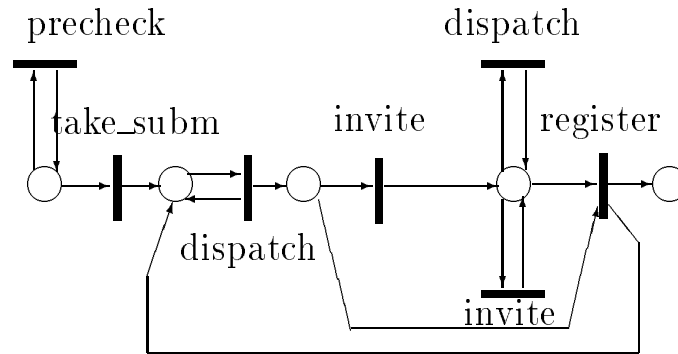
An activity A **exit condition** is mapped into assertions included into **all states** listed in the **to**: list of output state names of T_A .

4. To start the process, transitions corresponding to **starting activities** of a workflow should be identified and tokens should be placed into the **input states** of these transitions. This will establish an initial marking of the script.

FLOWMARK PROCESS DIAGRAM



CANONICAL FLOWMARK PROCESS REPRESENTATION



STAFFWARE FEATURES

Staffware is intended for automating administrative and production tasks with well-defined rules.

Scenario def: a procedure including steps (activities) with attached addressee (actors) and forms ('data types')

Icons represent different kinds of steps and control structures (conditional routing icon, wait icon, document icon)

Three kinds of steps in Staffware:

1. A (normal) step is the most common step type to which always a form is attached.
2. An automatic step executes an external program without user interaction.
3. Events are activities that can be used for controlling an already running case, altering the data related to it.

Each step consists of four possible parts: (i) an **addressee**, (ii) a **form** that the addressee receives, (iii) **activity** that takes place after the release of the step, (iv) a **deadline** (optional) of the step completion.

Routing features: sequentiality, forks, joins, loops and conditional routing. Routing from one step to another is specified by **activities** defined in the steps.

Scenario management: a case is started with the first step from the procedure definition; the form belonging to the step is sent to the addressee or automatic step is executed; after releasing the step the control flow goes to the step that was defined as an activity of the released step.

MAPPING OF STAFFWARE MODEL TO THE CANONICAL WORKFLOW MODEL

1. Each **step** S is mapped into a **transition** T_S with the activity specified corresponding to the program (manual) activity of the step. Places P are generated appropriately in between.

For a **conditional activity** the condition is mapped into a condition of an element of the **output binding list** of T_S related to P .

A **type of a token** in P corresponds to a **form** that should be attached to the next step.

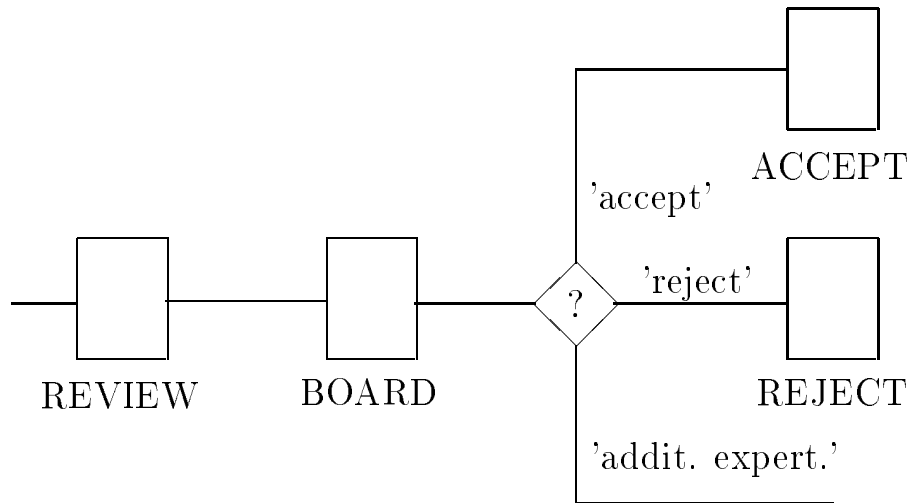
2. An activity A **start condition** is mapped into a **guard** of T_A that may be interpreted differently by the script facilities

An activity A **exit condition** is mapped into **assertions** included into all states listed in the **to:** list of output state names of T_A .

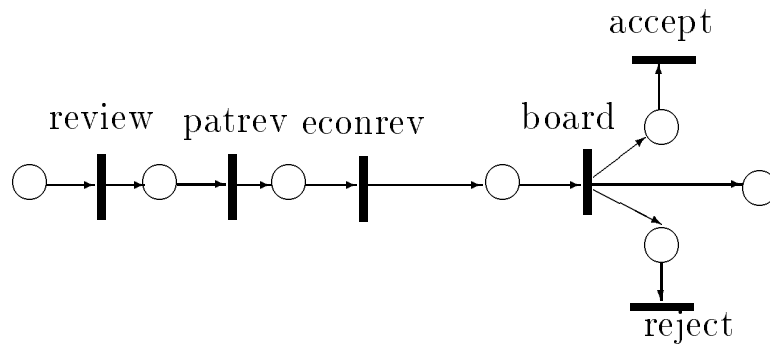
3. In case of a **deadline** constraint additional transition and step is generated with conditions leading to firing of the transition in case of the deadline violation.

4. To start the procedure, a token should be placed into the **input state** of the first step of the procedure. This will establish an **initial marking** of the script.

STAFFWARE PROCEDURE DIAGRAM



CANONICAL STAFFWARE PROCEDURE REPRESENTATION



THEORETICAL MODELS OF CONCURRENCY

Basic dichotomies:

- **Intensionality vs. extensionality** (operational vs. denotational, what systems do vs. what outside observer sees)
Intensional: state/transition systems, labeled transition systems, process algebras (ACP, CCS, CSP), Petri nets
Extensional: trace, sequence of atomic actions; acceptance trees, failure sets, synchronization trees
- **Interleaving vs. true concurrency** (interleaving models reduce concurrency to nondeterminism choosing between actions)
ACP, CCS and CSP employ **interleaving**, as do the trace-based and synchronization-tree extensional models
Petri nets represent **concurrency**, as do specific traces and event structures.
- **Branching-time vs. linear time**
Traces are **linear-time** models while synchronization trees and event structures are **branching-time** theories.

Approaches

- **Temporal logic** (linear and branch time), dynamic logics
Two systems are equivalent iff they satisfy the same formulas, proof systems rely on proving implications between temporal formulas.
- **System-based approach:** “high-level” systems as specifications for lower-level ones
Behavioral equivalences and pre-orders approach intends to show that the implementation provides “at least” the behavior dictated by the specification. Algebraic proof systems using equational reasoning.

Strategic directions (Concurrency Working Group): correctness, a unifying semantic framework of concurrency including its presentation for potential users, design and verification methodologies

PROCESS ALGEBRAS: BASIC NOTIONS

To make a workflow specification refinement with reuse of pre-existing behaviours more tractable we get an abstraction of a script by a PA definitions. PA is an algebraic theory for the description of process behaviour. PA is given by a signature and a set of axioms.

Transition system can be defined as a tuple $\langle S, L, T, s \rangle$ where

S is a **set of states**, L is a **set of labels**,

$T \subseteq (S \times L \times S)$ is a **transition relation**, $s \in S$ is the **initial state**.

The **signature** Σ of PA specifies **constants** and **function symbols**. Terms over signature Σ are defined as:

a variable $v \in V$ is a **term**,

if $c \in \Sigma$ is a **constant symbol**, then c is a term,

if $f \in \Sigma$ is an n -ary **function symbol** and t_1, \dots, t_n are terms then $f(t_1, \dots, t_n)$ is a term.

The set of all terms over $\Sigma : \Upsilon(\Sigma, V)$, the set of all ground terms : $\Upsilon(\Sigma)$.

Constants of PA: a finite set A of atomic activities a, b, \dots and specific constants (e.g., $\varepsilon =$ **empty activity** (or **skip**) and \surd ('tick') = **successful termination of a process**.) The atomic activities are taken as labels in the definition of a transition system.

PA axiom system: Ω . $\Psi(\Omega)$ is a class of processes evolving over Ω and containing A as elementary processes.

For $a \in A$ a **transition** $p \xrightarrow{a} p'$ expresses that by executing a the process p evolves into p' . p' represents the **remainder** of p to be executed.

$p \xrightarrow{a} \surd$ expresses that the process p can **terminate** after executing a .

T can be defined by rules that should be **admissible** in frame of Ω .

Functional terms in Σ are constructed using operators of a particular algebra. Processes are specified using algebraic expressions over Σ and V .

BISIMULATION

To reason of a possibility of reusing one process specification for another, they are compared on the basis of **bisimulation**.

Definition Two process terms are **bisimilar** if they have the same branching structure.

More formally, two processes $p, q \in \Psi(\Omega)$ are **bisimilar** (this fact is denoted by $p \underline{\leftrightarrow} q$) if there exists a **binary, symmetric** relation R between processes such that:

1. $R(p, q)$
2. if $R(p', q')$ and $p' \xrightarrow{a} p''$ then there is a transition $q' \xrightarrow{a} q''$ such that $R(p'', q'')$
3. if $R(p', q')$ and $p' \xrightarrow{a} \surd$, then $q' \xrightarrow{a} \surd$

Bisimulation equivalence is a congruence relation with respect to all operators of a process algebra defined. Axiomatization of an algebra should be **sound with respect to bisimulation** equivalence. It means that equality of process terms in an axiom ($p = q$) implies $p \underline{\leftrightarrow} q$. In well-defined algebras Ω is a **complete axiomatization** with respect to bisimulation equivalence, i.e., if $p \underline{\leftrightarrow} q$ then $p = q$.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(114)

BASIC PROCESS ALGEBRA WITH ITERATION

BPA^* (J.Bergstra, I.Bethke, J. Baeten) binary operators: $+$ and \cdot are called the alternative and sequential composition.

The **alternative composition** of the processes p and q is the process that either executes process p or process q but not both.

The **sequential composition** of the processes p and q is the process that first executes process p and upon completion thereof starts with the execution of process q .

For the **iteration** the binary Kleene operator x^*y is used that is understood as $y + x(x^*y)$.

In expressions \cdot will often be omitted, so pq denotes the same as $p \cdot q$. As binding conventions, $*$ binds stronger than \cdot , which in turn binds stronger than $+$.

BPA COMPLETE AXIOMATIZATION

We shall use the terms equality defined by the axioms to deduce terms bisimulation and therefore to deduce that we can substitute a term (actually, a script) instead of another term or a script.

(A1)	$x + y$	=	$y + x$
(A2)	$x + (y + z)$	=	$(x + y) + z$
(A3)	$x + x$	=	x
(A4)	$(x + y)z$	=	$xz + yz$
(A5)	$(xy)z$	=	$x(yz)$
(A6)	$x \cdot (x^*) + y$	=	x^*y
(A7)	$x^*(y \cdot z)$	=	$(x^*y) \cdot z$
(A8)	$x^*(y \cdot ((x + y)^*) + z)$	=	$(x + y)^*z$
(A9)	$x \cdot \varepsilon$	=	x
(A10)	$\varepsilon \cdot x$	=	x

SCRIPT PROCESSES

To simplify we restrict the consideration with the **deterministic processes** such that the behaviour of a process on its first $n + 1$ steps is determined by the behaviour of the process on its first n steps only. Another restriction comes out of a class of script behaviour patterns representable by the terms of BPA^* .

In BPA^* a script type will be abstracted as a pair $\langle \alpha, e \rangle$ where:

- α is the set of the atomic activities which are relevant for a script type;
- e is a process algebra expression composed of algebraic operators and atomic activities to abstract the script type behaviour (denoting all legal script firing sequences).

For a set of atomic activities of a script type (constants of a process algebra) we shall use a set of names of activities defined in transitions of a script assuming that such names are unique. We say that this set (α) forms an **alphabet** of a script type.

A script language Λ is formed by the set of all firing sequences (denoted by sequences of a script alphabet tokens). We abstract Λ as a class of processes $\Psi(\Omega)$ evolving over an axiom system of BPA^* .

Selectors for a script type $N = \langle \alpha, e \rangle$ include:

- $S_a(N) = \alpha$ (a set of **activity types**);
- $S_e(N) = e$ (a **process algebra expression** over α and Ω).

Projection of a process algebra expression e to a script alphabet B is defined as $e \setminus B$ that is equal to e with replacing of all activity types in e not belonging to alphabet B by ε (an empty activity).

REFINEMENT OF SCRIPTS

Script Q **refines** script N iff:

1. $S_a(N) \subseteq S_a(Q)$;
2. $S_e(N) = S_e(Q) \setminus S_a(N)$ that is **process algebra abstractions** of N and of a reduction of Q to N are **bisimulation equivalent**;
3. Each transition t_Q whose activity is included into the projection $S_e(Q) \setminus S_a(N)$ should be in **one-to-one** correspondence with a transition t_N in N . For each pair (t_Q, t_N) input places of t_Q should be in one-to-one correspondence with the input places of t_N , to each output place of t_N an output place of t_Q should correspond. For corresponding places (p_N, p_Q) a **type** of a token $C(p_Q)$ should be a **subtype** of $C(p_N)$ and $Ap(p_Q)$ should logically imply $Ap(p_N)$. $F(t_Q)$ should be a **refinement** of $F(t_N)$ and $G(t_N)$ should logically imply $G(t_Q)$ for (t_N, t_Q) pair.

All places of Q corresponding to input and output places of N should be declared as respected ports in Q (incoming places or outgoing places (gates) respectively). For each pair of input places a token type of Q place should be a supertype of a token type of the N place. For each pair of output places a token type of Q place should be a subtype of a token type of the N place. For each pair of places an initialization of a place in Q should be a refinement of initialization of a place in N .

4. For corresponding pairs of transitions their ingoing and outgoing correlated arcs (a_N, a_Q) (their relationship is established by the correlation of input and output places of the transition pair) should satisfy the condition that $E(a_Q)$ should be a **refinement** of $E(a_N)$. It means that $E(a_Q).Par$ type should be a subtype of $E(a_N).Par$, $E(a_Q).Oce$ should logically imply $E(a_N).Oce$ and $E(a_N).Z = E(a_Q).Z$.
5. Initial marking M_Q of Q should refine an initial marking M_N of N so that for the corresponding pairs of places (p_N, p_Q) , $M_Q(p_Q)$ should be a refinement of $M_N(p_N)$.

SCRIPT TYPE REDUCTS

Definition Script type reduct A reduct R_V of a script type V is defined as a subspecification of V such that V is a script subtype of R_V .

Decomposing a script specification, we can get its different reducts on the basis of various script specification subsignatures.

Definition A common script reduct for script types V_1, V_2 is such script reduct R_{V_1} of V_1 that there exists a script reduct R_{V_2} of V_2 such that R_{V_2} is a refinement of R_{V_1} . Further we refer to R_{V_2} as to a *conjugate* of the common script reduct.

Definition A most common script reduct $R_{MC}(V_1, V_2)$ for script types V_1, V_2 is a script reduct R_{V_1} of V_1 such that there exists a script reduct R_{V_2} of V_2 that refines R_{V_1} and there can be no other script reduct $R_{V_1}^i$ such that $R_{MC}(V_1, V_2)$ is a script reduct of $R_{V_1}^i$, $R_{V_1}^i$ is not equal to $R_{MC}(V_1, V_2)$ and there exists a script reduct $R_{V_2}^i$ of V_2 that refines $R_{V_1}^i$.

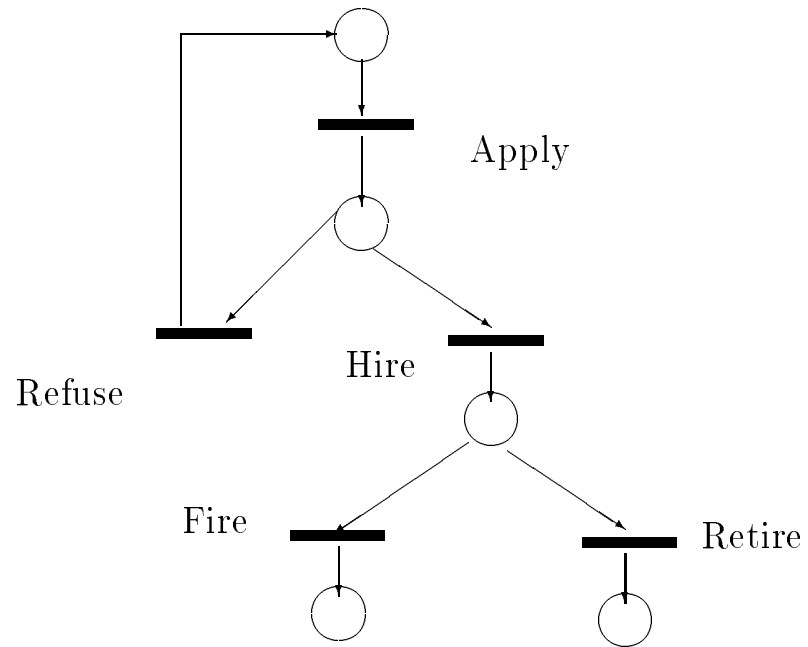
OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(118)

EXAMPLE: JOB APPLICATION SCRIPTED TYPE

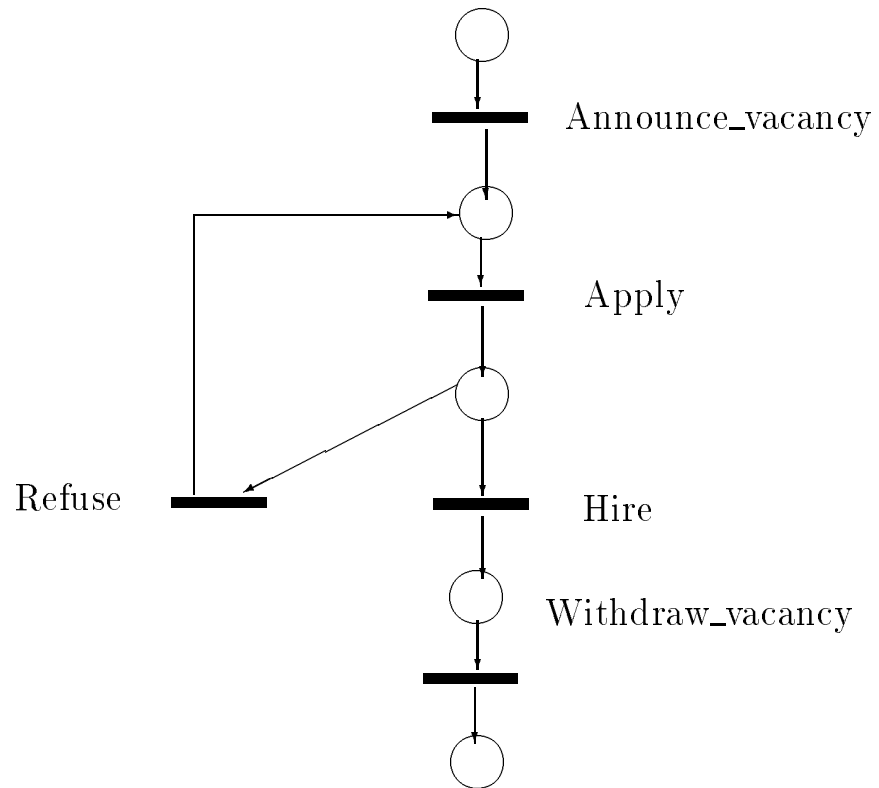


$$Apply.(1 + (Refuse.Apply)^*).$$

$$Hire.(Fire + Retire)$$

A process algebra expressions shown on the pictures abstract operations orderings.

EXAMPLE: JOB VACANCY SCRIPTED TYPE

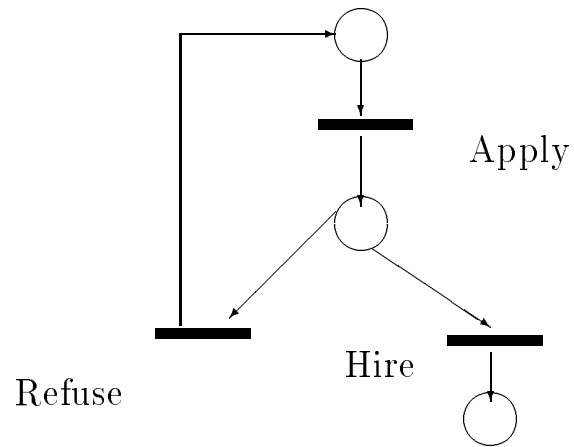


Announce_vacancy.Apply.

$(1 + (Refuse.Apply)^*)$.

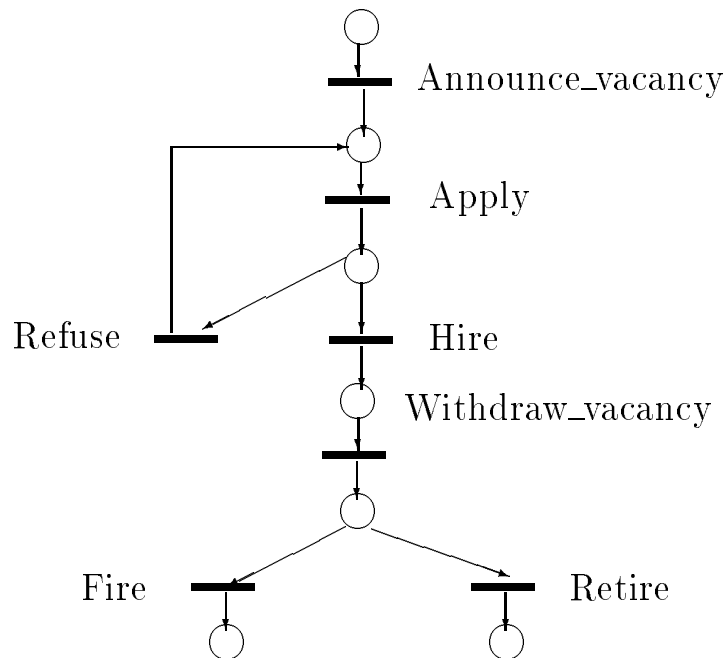
Hire.Withdraw_vacancy

EXAMPLE: MEET OF JOB APPLICATION AND VACANCY TYPES



$$Apply.(1 + (Refuse.Apply)^*).Hire$$

EXAMPLE: JOIN OF JOB APPLICATION AND VACANCY TYPES



Announce_vacancy.Apply.
(1 + (Refuse.Apply)).*
Hire.Withdraw_vacancy.
(Fire + Retire)

SCRIPT CONFORMANCES

Definition A **conformance association** between script types V_1, V_2 holds (we say " V_2 conforms to V_1 ") iff there exists a common script reduct of V_1 and V_2 .

For a collection of script types T we define a *conformance of a script type V* as a sub-collection of types $S = \{t_1, t_2, \dots, t_n\}$ in T such that any script type t_i in t conforms to S .

The conformance of V characterizes a reuse perspective for a script type V .

Ontological conformance of a script type V is a sub-collection of script types $S = \{t_1, t_2, \dots, t_n\}$ in T such that any type t_i in S is [loosely or tight] ontologically relevant to V .

Ontological conformances of constituents of script type specifications (such as places, transitions, gates etc.).

Thus we rely on a range of conformances starting with a loose ontological conformance and ending with a conformance based on the most common reduct of script types.

GENERAL PROCEDURE OF SCRIPT DESIGN WITH REUSE

In a script design phase we distinguish between the following basic steps:

1. Establishing ontological conformances (loose and tight) for each constituent of the analysis model
2. Common reducts identification and construction
For each pair of script type specifications t_s and t_r (each t_r should belong to ontological conformance of t_s) we try to construct their most common reduct (starting from their **common signature reduct**).
3. Construction of hierarchical compositions of scripts

SCRIPT TYPES MOST COMMON REDUCT IDENTIFICATION

For each **pair of ontologically conformant script types** t_s, t_r we look for a **maximal collection** A of pairs of constituents (at_s^i, at_r^j) that are also ontologically relevant and satisfy preliminary constraints such that at_r^j could be reused as at_s^i .

General approach to form A for the pair of ontologically relevant types t_s and t_r is the following:

1. All ontologically relevant **pairs of places** (at_s^i, at_r^j) of the script types belong to A if type of at_r^j is a supertype of type of at_s^i . We should check also that an activation predicate of at_r^i should logically imply an activation predicate of at_s^j .
2. For all ontologically relevant **pairs of transitions** their activity declarations (at_s^i, at_r^j) belong to A if signatures of functions at_s^i and at_r^j satisfy the requirements established for a pair of ontologically relevant pair of functional attribute of types to be included into a common reduct of a type.

After that we should check that at_r^j is a **refinement** of at_s^i and that the guarding function of a pre-existing script transition should **imply** the guarding function of an application script transition.

In this process establishing mediating functions reconciliating possible type and value conflicts is assumed.

REACHING OF REFINEMENT

Construct a **candidate common script type reduct** N and its conjugate Q out of the constituents in A .

Take their **PA expressions** $S_e(N)$ and $S_e(Q) \setminus S_a(N)$ and apply **equivalent transformations** to $S_e(N)$ based on the **PA axiom system** trying to reduce $S_e(N)$ to $S_e(Q) \setminus S_a(N)$. We repeat this procedure for different subscripts N and Q until we get satisfying subexpressions or fail.

Thus we are looking for potentially reusable patterns of pre-existing behaviours to get bisimilar processes that identify finally the common reducts. Repeat the process for all script types t_r conformant to t_s .

Among successful discoveries of most common reducts, we choose those **dis-joint common reducts** that gives the best cover of t_s .

Each common reduct in t_s is transformed to a **transition** that is to be **hierarchically substituted** by conjugate of a common reduct (a pre-existing script reduct).

Implementation of such strategy is in a good accordance with the **nested sub-process model** identified by WfMC.

The **uncovered remainder** of t_s identifies those part of the script type that is to be implemented from scratch.

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(125)

EXAMPLE OF BISIMULATION EQUIVALENT TRANSFORMATIONS IDENTIFYING COMMON SCRIPT REDUCT

For the **Agency workflow specification** a PA expression for the **workflow reduct** responsible for **submissions collection**:

$$\text{Agency_coll_nom} = (\text{precheck} \cdot \text{precheck}^* \text{take_subm} + \text{take_subm}) \cdot ((\text{dispatch} + \text{invite})^* \text{register})$$

PA expression for the **workflow reduct** responsible for **submissions review and acceptance**:

$$\text{Agency_rev_acc} = \text{review} \cdot \text{board}(\text{accept} + \text{reject} + \text{add_expertise})$$

For the pre-existing FlowMark workflow specification we get:

$$\text{IndLab_FM} = \text{precheck}^* \text{take_subm} \cdot (\text{dispatch}^* (\text{invite}((\text{dispatch} + \text{invite})^* \text{register}) + \text{register}))$$

For the **pre-existing Staffware workflow specification** we get:

$$\text{IndLab_SW} = \text{review} \cdot \text{patrev} \cdot \text{econrev} \cdot \text{board}(\text{accept} + \text{reject} + \text{add_expertise})$$

Projecting the last expression onto the alphabet of the Agency workflow:

$$\text{IndLab_SW}' = \text{review} \cdot \varepsilon \cdot \varepsilon \cdot \text{board}(\text{accept} + \text{reject} + \text{add_expertise})$$

Taking into account that according to the axiom A9:

$$\text{review} \cdot \varepsilon \cdot \varepsilon = \text{review}$$

we see that $\text{IndLab_SW}'$ is bisimulation equivalent to Agency_rev_acc .

Now, we apply equivalent transformations to Agency_coll_nom to reduce it to IndLab_FM :

According to the axiom A6 of the algebra axiom system, we get:

$$(\text{precheck} \cdot \text{precheck}^* \text{take_subm} + \text{take_subm}) = \text{precheck}^* \text{take_subm}$$

$$\text{According to the axiom A8, we get: } (\text{dispatch} + \text{invite})^* \text{register} = \text{dispatch}^* (\text{invite}((\text{dispatch} + \text{invite})^* \text{register}) + \text{register})$$

Applying these transformations, we obtain equivalent expressions thus justifying that the processes abstracted from the Agency script reduct and from the FlowMark to script mapping are bisimilar.

CONCLUDING REMARKS

For **component-ware** technologies, the components should have **complete semantic** specifications for their reuse. Reusable components are treated **semantically interoperable** in context of a specific application.

Uniform descriptions of heterogeneous **components** are provided in frame of **canonical object model** having formal semantics. Concept of refinement of specifications is inherent for the model.

Application semantics of specifications are considered in frame of the **ontological** approach. The same canonical model is used for the ontological modeling.

Component-based design with reuse is based on the **ontological contexts integration, search** of the ontologically relevant component **constituents**, identification of **reusable fragments**, construction of **compositions** of components concretizing specifications of requirements.

The concept of **common type reduct** constitutes a basis for determining **reusable fragments**.

The **script-based workflow** specification framework is defined.

The **workflow refinement** is based on systematic analysis of **functional and concurrent** workflow specification properties merging conventional well grounded **specification refinement technique** with detecting of **process bisimulation equivalence**. For the latter admissible patterns of activities of a script are modelled as **process algebra expressions** implied by a script model.

Complete **axiomatization** of process algebras provides for **equivalent transformation** of such expressions and their partial ordering to reason that a behaviour given by a pre-existing workflow fragment can be considered a **refinement** of the required multiactivity behaviour.

Further research is required to develop **more uniform modeling facilities** for adequate justification of reusable workflow fragments selection and composition.

RELATED PUBLICATIONS

1. L.A. Kalinichenko, A Declarative Framework for Capturing Dynamic Behavior in Heterogeneous Interoperable Information Resource Environment, *Proceedings of the RIDE -IMS'93*, Vienna, April 1993.
2. L.A. Kalinichenko, "Emerging semantic-based interoperable information system technology", *Computers as our better partners*, Proceedings of the International IISF/ACM Symposium, Tokyo, World Scientific, 1994.
3. L.A. Kalinichenko, "Homogeneous localization of structural interactivity dependencies in megaprograms using scripts", *Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS'94)*, Moscow, May 1994.
4. L.A. Kalinichenko, *SYNTHESIS: a language for description, design and programming of interoperable information resource environment*, Institute for Problems of Informatics of the Russian Academy of Sciences, September 1995
5. L.A. Kalinichenko, "Structural and Behavioral Abstractions of the Multiactivities Intended for their Concretizations by the Pre-existing Behaviors", *Second International East-West Database Workshop*, Klagenfurt, Springer, 1995.
6. Kalinichenko L.A. Workflow Reuse and Semantic Interoperation Issues. In *Advances in workflow management systems and interoperability*. A.Dogac, L.Kalinichenko, M.T. Ozsü, A.Sheth (Eds.). NATO Advanced Study Institute, Istanbul, August 1997.
7. Kalinichenko L.A. Component-based Development Infrastructure: A Systematic Approach, *OMG-DARPA-MCC Workshop on "Compositional Software Architecture"*, Monterey CA, January 6 - 8, 1998.
8. Kalinichenko L.A. Composition of type specifications exhibiting the interactive behaviour. In *Proceedings of EDBT'98 Workshop on Workflow Management Systems*, March 1998, Valencia
9. Briukhov D., Kalinichenko L. Component-based information systems development tool supporting the SYNTHESIS design method. *Proceedings of the East European Symposium on "Advances in Databases and Information Systems"*, September 1998, Poland, Springer, LNCS N 1475, 1998

OBJECT-ORIENTED DATABASES

IPI RAS

Leonid Kalinichenko

(128)