Extensible Canonical Process Model Synthesis Applying Formal Interpretation

L.A. Kalinichenko, S.A. Stupnikov, N.A. Zemtsov

Institute of Informatics Problems, Russian Academy of Science
{leonidk, ssa, nazem}@ipi.ac.ru

Abstract. The current period of IT development is characterized by an explosive growth of diverse information representation languages. Applying integration and composition of heterogeneous information components it is required to develop the canonical information model serving for adequate expression of semantics of various information models used in the environment encompassing required heterogeneous components. Basic principles of the canonical model synthesis include fixing of its kernel, constructing the kernel extensions for each specific information model of the environment so that this extension together with the kernel could be *refined* by this information model, and forming the canonical model as a union of all such extensions. Previously these principles have been successfully applied to the synthesis of structural and object canonical models. This paper[1] applies this technique to synthesis of the process canonical model. The method proposed is based on interpretation of process model semantics in logics, and specifically, in the Abstract Machine Notation that made possible to construct provable refinements of process specifications. This method has been applied to the environment of process models defined by workflow patterns classified by W.M.P. van der Aalst. Thus the canonical process model synthesized possesses a property of completeness with respect to broad class of process models used in various Workflow Management Systems as well as the languages used for process composition of Web services.

1 Introduction

The present period of IT development is characterized by the process of explosive growth of various information representation models. This development takes place in frame of specific distributed infrastructures (such as OMG architectures (in particular, the model driven architecture (MDA)), semantic Web and Web services architectures, digital library architectures as collective memories of information in various subject domains, architectures of the information grid), as well as in the standards of languages and data models (such as, for example, ODMG, SQL, UML, XML and RDF stacks of data models), process models and

[1] This research has been partially supported by the grant N 05-07-90413 of the Russian Foundation for Basic Research as well as by the Program of Basic Research of the Department of Information Technologies and Computing Systems of RAS

workflow models, semantic models (including ontological models and models of metadata), models of digital repositories of data and knowledge in particular scientific domains (e.g., virtual observatories in astronomy).

This process is accompanied by another trend — the accumulation of based on such models information components and services, the number of which grows exponentially. This growth causes the accelerating need for integration of components and services represented in heterogeneous models in various applications, as well as their reuse and composition implementing new information systems. The indicated trends are contradictory: the more variety of used models we meet in various components and services, the more complex become problems of their integration and composition. These trends are not new, but with time, the diversity of various models and their complexity grow together with the increasing need for integration and composition of components and tools represented in different models. Scale of these phenomena, determining possibilities of designing of distributed information systems in various domains, reusing, trading and compositions of components, reaching their semantic interoperability, integration of heterogeneous information sources, is sufficient motivation for research and development of adequate methods for manipulation of various information representation models.

The basis of these methods is constituted by the concept of the canonical information model serving as the common language, "Esperanto", for adequate expression of semantics of various information models, surrounding us. To prove that a definition in a language can be substituted with a definition in another one, formal specification facilities and commutative model mappings are provided.

Initially ideas of mapping structured data models and canonical model construction for them were developed. The basic definitions of equivalence of database states, database schemas and data models were introduced to preserve operations and information while constructing of mappings of various structured data models into the canonical one [11], [12]. According to this approach, each data model was defined by syntax and semantics of two languages – data definition language (DDL) and data manipulation language (DML). The main principle of mapping of an arbitrary source data model into the target one (the canonical model) constituted *the principle of commutative data model mapping* . According to it preserving of operations and information of a source data model while mapping it into the canonical one could be reached under the condition, that the diagram of DDL (schemas) mapping and diagram of DML (operators) mapping are commutative [12]. At that time in the process of data model mappings construction the denotational semantics was used as a formalism (metamodel), allowing to prove a commutativity of the diagrams mentioned [12]. Such a proof had to be carried out manually.

Later, for the object data models, the method of data model mapping and canonical models constructions was modified as follows. As a formalism (metamodel) of the method the Abstract Machine Notation (AMN) was used instead of the denotational semantics. It allowed to define the model-theoretic specifications in the first order logics and to prove the fact of specification *refinement*

[3], [4]. The theory of refinement provided for developing of fundamental definitions of relationships between data types, data schemas, data models so that instead of equivalence of respective specifications, it could be possible to reason on their refinement [14]. It is said that specification A *refines* specification D, if it is possible to use A instead of D so that the user of D does not notice this substitution. Existence of special tools for AMN (B-technology)provides for conducting proofs of commutativity of mappings semi-automatically: theorems required for the proof of refinement are generated by B automatically, and their proof is (generally) conducted with the human assistance.

The main principle of canonical model synthesis is that its *extensibility* is required for semantic integration and information interoperability in heterogeneous environment, including various models. A kernel of the canonical model is fixed. For each specific information model M of the environment an extension of the kernel is defined so that this extension together with the kernel *is refined* by M. Such refining transformation of models should be *provably correct*. The canonical model for the environment is synthesized as *the union of extensions*, constructed for models M of the environment.

Applying similar principles, this paper deals with process models, required for describing activities of various organizations for solving of their tasks. For example, virtual organization models are based on composition of processes of real organizations involved in sphere of activity of the virtual organization. Another example is trading of processes and composition of processes implementing a required process (this is one of well-known tasks in semantic Web or in mobile systems). The processes are implemented in workflows in various Workflow Management Systems (WfMS). For the process languages they apply various concepts and paradigms incompatible for various WfMS. Irrespective of the model used, workflow specification is a complex construction, integrated with specifications of other types (usually, object-oriented).

While mapping processes at synthesis of their canonical model, it is required to preserve the semantics of concurrency. The main problem of such synthesis is that there is no general theory of concurrency. The early research has shown [15], that process algebras do not possess sufficient expressive power to serve as a kernel of the canonical process model. At the same time combining of two requirements – completeness of the canonical process model ability of interpretation of various workflow models with an ability to prove of correctness of arbitrary process model interpretation in the canonical process model – remained hard-reachable for quite a long time period . The possibility of interpretation of concurrent events, typical for process models, in logics, and specifically, in the Abstract Machine Notation has been discovered recently [8], [19], [17]. Algorithms of process specifications mappings into AMN were constructed [7], [18]. This approach allows to construct provable refinements of process specifications, applying the B-technology. This achievement is the necessary prerequisite for commutative mapping of process models. Simultaneously it was succeeded to classify and describe the diversity of workflow models by means of *workflow patterns* [1]. Due to these two events, the possibility of choice of a canonical process

model kernel and construction of its extensions, refined by various workflow patterns, became possible. Thus, the way to the canonical process model synthesis has been opened, and such synthesis has been developed [16] in the context of the work reported here.

The text of the paper is organized as follows. In section 2 an approach to the synthesis of extensible canonical process model is described. In section 3 the technique of construction of refining extensions for process model is considered. In conclusion the results are summarized, and perspectives of application of the methods described are discussed.

2 Construction of Canonical Process Model

The analysis of large number of WfMS process models [1] resulted in 20 workflow patterns – process constructions being typical in practice.

The set of the patterns mentioned is complete; it has appeared to be enough for representation of process models of various WfMS. This result allows to select the workflow patterns as the source process models for synthesis of the canonical process model. Thus it is possible to combine process completeness of the selected set of constructions with a possibility to interpret with this set an arbitrary process, expressed in process models of various WfMS's.

2.1 Definition of the Kernel of Canonical Process Model

According to the principles of the canonical model synthesis, the canonical process model is developed as a kernel, including basic primitives of process specifications, and extensions of the kernel.

A subset of scripts of the SYNTHESIS language [13], [15] was selected as the canonical process model kernel. Its capabilities are close to the colored Petri nets [10]. The kernel has the following properties.

1. It is based on the well-known model of Petri nets.
2. It embeds Petri nets in the object environment. As a result the control flow and the data flow are combined by means of tokens – objects of certain types having unique identifiers.
3. It provides for binding of Petri net transitions with functions, which should be called at firing of these transitions. The rules of binding input and output tokens of a transition with input and output parameters of the respective function can be defined. Such bindings are necessary for modeling of information system as a whole, what is not taken into account in more abstract models [10], [2].

The declaration of any entity in the SYNTHESIS language (for example, types, classes, functions) syntactically is given by means of a *frame*. Generally frame may be considered as a structured symbolic model of some entity or concept, used to represent their instances. Syntactically a frame is separated with

braces. 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.

Scripts are defined applying a generic script type in the SYNTHESIS language. They form a subtyping hierarchy. Each instance of a script type corresponds to an execution of the process, defined by the script. An example of script specification is given in the following section.

2.2 Generic Types as Extensions of the Canonical Model Kernel

Extensions of the canonical process model kernel were constructed [16] for workflow patterns defined in [1]. Here we consider the extension technique of the canonical model kernel for the *discriminator* pattern as an example.

This pattern describes a situation when a completion of one of (concurrent) branches is expected. After that the subsequent transition is activated, and all remaining branches are cancelled.

In canonical model each pattern corresponds to a generic script type, treated as an extension of the kernel. This type defines rules, according to which process control flow is organized. Various script elements (such as functions and data types) can be used as parameters of the type.

Graphically scripts are represented by a bipartite graph with two sorts of nodes – places (represented by circles) and transitions (represented by squares). Nodes of different sorts are connected by the incidence relation (arrows). Places can accumulate tokens of various types. Transitions may fire when a certain conditions are met, consuming tokens from input places of a transition and producing tokens in its output places.

The *discriminator* pattern is represented in the canonical model with the generic script type `discriminator` (fig. 1). Without loss of generality we consider the pattern to be a junction of two branches. In the figure the `Trunk` transition is connected with a dashed line to a dashed rectangle with rounded corners. This graphically denotes that at firing of the transition `Trunk` all tokens are to be removed from the marked area. In this way the cancellation of the remaining (concurrent) branches is realized.

```
{ discriminator; in: script;
  params: {branch1/function, branch2/function, trunk/function,
           entrance1TokenType/type, entrance2TokenType/type,
           auxPlaceTokenType/type, exitTokenType/type };
```

Here `discriminator` – is a name of the script type. The formal parameters of a type are set in the optional `params` slot. So, each transition (for example, `Trunk`) is parameterized with the function type (in this case – `trunk`), which is called at transition firing. Each place (for example, `entrance1`) is parameterized with the token type, admissible for the given place (here – `entrance1TokenType`).

```
states:
 {entrance1; token: entrance1TokenType},
```

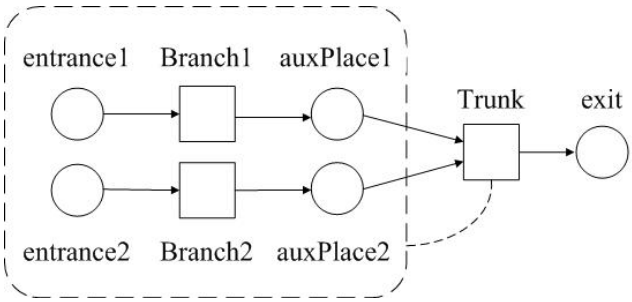

Fig. 1. The discriminator pattern

```
 {entrance2; token: entrance2TokenType},
 {auxPlace1; token: auxPlaceTokenType},
 {auxPlace2; token: auxPlaceTokenType},
 {exit; token: exitTokenType};
```

The `states` slot defines the set of places of a specific net. Each place is characterized by a name and an admissible token type.

```
transitions:
{ Branch1;
  from: entrance1;  bind_from: {entrance1, in};
  to: auxPlace1;    bind_to: {auxPlace1, out};
  activity: {in: function;
   params: {+ in/entrance1TokenType, -out/auxPlaceTokenType};
    {{branch1(in,out)}};
  }
}
```

In the `transitions` slot a set of transitions of a specific net is defined. Each transition is characterized by name (for example, `Branch1`), list of input places (`from`), list of output places (`to`), optional list of conditions (`conditions`), function, which is called at firing of the transition (`activity`), and also binding lists of input and output parameters of this function (`bind_from`, `bind_to`). The body of the function for transition `Branch1` consists of respective function call with a correct set of parameters (`branch1(in,out)`). The given script fragment works as follows. A token of type `entrance1TokenType` is selected in place `entrance1`. If a token is found, the transition fires – the token is passed into the input parameter `in` of the `branch1` function (according to the slot `bind_from`). A token at the output of the function (`out` parameter) is passed into `auxPlace1` place by virtue of the binding, defined in `bind_to` slot.

The specification for `Branch2` transition looks similarly. Transition `Trunk` concludes specification of the generic script type.

```
{ Trunk;
```

```
  from: auxPlace1, auxPlace2;
  bind_from: {auxPlace1, in}, {auxPlace2, in};
  to: exit;
  bind_to: {exit, out};
  activity: {in: function;
   params: {+ in/auxPlaceTokenType, -out/exitTokenType};
   {{
     trunk (in, out) &
     isempty (entrance1') & isempty (entrance2') &
     isempty (auxPlace1') & isempty (auxPlace2')
   }};
  }
}
```

A function, substituted as a script parameter (for example, `trunk`), should have, besides other, certain number of input and output parameters of definite types (in our case these are two input parameters of types `auxPlace1TokenType` and `auxPlace2TokenType`, and one output parameter of a type `exitTokenType`). Primed state names denote post-states of operations. The body of the function also contains formula, according to which certain places are cleared of tokens.

In this work we use the following notation. Input and output places of a pattern are named as `entrance` and `exit`, respectively (with addition of index if required). Additional places are named `auxPlace1`, `auxPlace2`, etc. Transitions are named either `Trans`, or, if the pattern deals with branching, "trunk" transition is named `Trunk`, and transitions in branches as `Branch`. If required, an additional indexing can be added. Such notational agreement is not impose any limitations on the way patterns are used or implemented. They are introduced for readability.

2.3 Semantics of the Canonical Process Model in AMN

The synthesis of the extensible canonical process model is realized on the basis of the formal system (Abstract Machine Notation (AMN)) [3], [4]. In this section we consider a part of the semantics of the kernel and semantics of the extension, required for understanding of the example below.

AMN, as a model-theoretic notation, allows to consider specifications of state space and behavior (defined by operations on states) in an integrated way . The specification of a machine state is introduced by state variables together with invariants – constraints, which should always be satisfied. Operations are defined on the basis of the extended formalism of the Dijkstra's guarded commands [9].

The *refinement* as a key concept of AMN, provides for correlation of system specifications of various levels of abstraction. A refining specification can be significantly more detailed, than a refined one. The refining specification is constructed on the basis of the algorithmic and data refinement [3]. The refinement is formalized in AMN by formulation of a number of theorems of special sort, so-called *proof obligations*. Such theorems are formulated automatically by tools supporting B-technology (e.g., B-Toolkit, AntelierB) on the basis of *gluing*

invariants – the invariants, correlating states of refined and refining systems. The theorems can be proved with the help of tools supporting automatic and (or) interactive proof.

Due to [7], [18], [19], [8], [17], [5], common understanding of how one should interpret process models in AMN has been formed. This understanding, in its turn, has allowed to develop a method of script model mapping into AMN.

The main idea of mapping scripts into AMN consists in modeling of system state (i.e. places of a script) as variables of AMN, and modeling of transitions through AMN operations bodies of which are expressed in guarded substitutions. Such idea is characteristic for all approaches of representation of process models in AMN.

The `discriminator` script described above is represented in AMN as REFINEMENT with a name *DiscriminatorScript*.

REFINEMENT *DiscriminatorScript*
SETS *Obj*
CONSTANTS $ext\_entrance1\,TokenType, ext\_auxState1\,TokenType, \ldots$
PROPERTIES
$ext\_entrance1\,TokenType \in \mathbb{P}(Obj) \wedge$
$ext\_auxState1\,TokenType \in \mathbb{P}(Obj) \wedge \ldots$
VARIABLES $entrance1, auxState1, \ldots$
INVARIANT
$entrance1 \in \mathbb{P}(ext\_entrance1\,TokenType) \wedge$
$auxState1 \in \mathbb{P}(ext\_auxState1\,TokenType) \wedge \ldots$
INITIALISATION $entrance1 := \varnothing \parallel auxState1 := \varnothing \parallel \ldots$
OPERATIONS ...

REFINEMENT is the most universal AMN construction, since it can be used both as refined and as refining construction. Therefore this construction is most preferable for homogeneous representation of scripts in AMN.

In AMN specifications we shall represent types as extents, which are defined in the section of constants. For example, an extent *ext _entrance1TokenType* is introduced for place *entrance1*. Then in the section of properties the types are represented as subsets of the deferred set *Obj* , which is interpreted as the union of extents of all object types. Each place of the script, defined in `states` slot, is represented in AMN as an individual variable which is typed in the INVARIANT section appropriately. In the INITIALISATION section variables are initialised with empty sets, corresponding to initial absence of tokens in the script places.

Each transition of the script, defined in `transitions` slot, is represented in AMN by an operation of the machine *DiscriminatorScript* .

The operations of abstract machines are based on the generalized substitutions. We shall use operations of sort

$$op = S$$

Here *op* is an operation name, S – substitution, defining the effect of the operation on the state space.

The Generalized Substitution Language (GSL) provides for description of transitions between system states. Each generalized substitution S defines a Predicate transformer, linking some postcondition R with its *weakest* precondition $[S]R$. This guarantees preservation of R after the operation execution. In such case we say that S *establishes* R. We shall use substitutions given in the table 1. Here S, T, S_1, S_2 stand for substitutions, x, y, t are variables, E, F denote expressions, G, G_1, G_2, P are predicates, $P\{x \to E\}$ denotes predicate P having all free occurrences of variable x replaced by E.

The generalized substitution S	$[S]P$
$X := E$	$P\{x \to E\}$
$X := E \parallel y := F$	$[x, y := E, F]P$
SELECT G_1 THEN T_1 WHEN G_2 THEN T_2 END	$(G_1 \Rightarrow [T_1]P) \wedge (G_2 \Rightarrow [T_2]P)$
ANY t WHERE G THEN T END	$\forall t \bullet (G \Rightarrow [T]P)$
$S\,;\ T$	$[S][T]P$

Table 1. The Generalized substitutions and their semantics

Now we proceed to the OPERATIONS section of the machine.

The SELECT substitution, with guarding predicate representing a condition of the transition firing, taken from bindings of the slot `bind_from`, constitutes the body of the script operation. For example, for operation *Branch1* such condition requires an occurrence of some token t in place *entrance1*. For operation *Trunk* the condition requires an occurrence of a token in one of the input places *auxState1*, *auxState2*, and absence of tokens in the others.

$Branch1 =$
SELECT
  $\exists\, t \bullet (t \in entrance1)$
THEN
  ANY t WHERE $t \in entrance1$
  THEN
    $entrance1 := entrance1 - \{t\} \parallel$
    ANY r WHERE $r \in ext\_auxState1\,TokenType$
    THEN
      SELECT $TRUE = TRUE$
      THEN $auxState1 := auxState1 \cup \{r\}$
      END
    END
  END
END

If the conditions of transition firing are met, appropriate tokens are taken from places defined by bindings of the slot `bind_from`. These tokens are passed as input parameters to the transition function. Selected tokens are removed from the input places. For operation *branch1* a token t from place *entrance1* is removed: $entrance1 := entrance1 - \{t\}$. For operation *trunk* a token t, satisfying the guarding predicate of substitution SELECT, is removed from the place it occupied. Simultaneously (that is specified by simultaneous substitution ||) another substitution is executed, consisting of sequential composition of two substitutions. The first of them represents the transition function, the second represents correct allocation of an output token of the transition. In our example the transition functions are absent, since we consider the generic type, so the sequential composition is degenerated. In case of operation *branch1* only token allocation is interpreted: with the help of *ANY* substitution a token r is selected from the extension of a type *auxState1TokenType*. This token is passed into place *auxState1*, an output place for the transition. In case of operation *trunk* an allocation of a token r of admissible type *exitTokenType* in an *exit* place of the transition is interpreted. After that with the help of sequential substitution $\langle\,;\,\rangle$ a cancellation of concurrent branches is realized so that tokens are removed from the respective script places.

$Trunk =$
SELECT
$\quad \exists\, t_1 \bullet (t_1 \in auxState1 \wedge auxState2 = \varnothing) \vee$
$\quad \exists\, t_2 \bullet (t_2 \in auxState2 \wedge auxState1 = \varnothing)$
THEN
$\quad$ ANY t WHERE $t \in Obj \wedge$
$\quad\quad (t \in auxState1 \wedge auxState2 = \varnothing \vee$
$\quad\quad t \in auxState2 \wedge auxState1 = \varnothing)$
$\quad$ THEN
$\quad\quad auxState1 := auxState1 - \{t\} \;||\; auxState2 := auxState2 - \{t\} \;||$
$\quad\quad$ ANY r WHERE $r \in ext\_exitTokenType$
$\quad\quad$ THEN
$\quad\quad\quad$ SELECT $TRUE = TRUE$ THEN $exit := exit \cup \{r\}$ END
$\quad\quad$ END ;
$\quad\quad (entrance1 := \varnothing \;||\; entrance2 := \varnothing \;||$
$\quad\quad\quad auxState1 := \varnothing \;||\; auxState2 := \varnothing)$
$\quad$ END
END

3 Construction of Refining Extensions of the Canonical Process Model Kernel

Two process models, source and target, are required for the refining extensions construction technique. The canonical model stands for the target model. In

section 2 it was shown how extensions of the canonical model kernel are specified by means of generic types for workflow patterns (for the discriminator example). Constructing an extension, it is necessary to show, that the extension of the kernel is refined by the source workflow pattern model. Here it is convenient to use the source models of workflow patterns given in YAWL (Yet Another Workflow Language) [2]. This language has constructions sufficient for expression of all workflow patterns [1]. At construction of the extension of the process model kernel it is necessary to prove that the source model refines the target one. In this section we show this for the discriminator pattern.

The workflow process specification in YAWL is a set of the Extended Workflow Nets (EWF-nets) [2], forming a tree-like structure. In this paper for simplicity we shall not go beyond YAWL facilities sufficient for description of the the discriminator pattern example. The specification will be given as a single EWF-net.

In general a data model mapping requires to construct 1) mapping of a source model M_j into an extension of a target model M_i; 2) AMN semantics for M_j; 3) AMN semantics for extended M_i. After that the B technology is applied to prove a) state-based properties of the mapping (commutativity of the data type state mapping diagrams); b) behavioral properties of the mapping for all types, defined for the source data model. This leads to a proof that M_j is a refinement of the extension of M_i.

Thus, for our example nothing else left but to construct AMN semantics of the discriminator pattern defined in YAWL.

The discriminator as the EWF-net is defined as the following 8-tuple:

$$Discriminator = \langle C, i, o, T, F, join, split, rem \rangle$$

$$\begin{aligned}
&C = \{enter1, enter2, auxState1, auxState2, exit\}\\
&T = \{branch1, branch2, trunk\}\\
&F = \{enter1 \mapsto branch1, enter2 \mapsto branch2,\\
&\quad branch1 \mapsto auxSatate1, branch2 \mapsto auxSatate2,\\
&\quad auxSatate1 \mapsto trunk, auxSatate2 \mapsto trunk,\\
&\quad trunk \mapsto exit\}\\
&join = \{trunk \mapsto XOR\}\\
&split = \varnothing\\
&rem = \{trunk \mapsto \{enter1, enter2, auxState1, auxState2, branch1, branch2\}\}
\end{aligned}$$

Here C is a set of places (in Petri nets terminology). T is a set of tasks. For the purpose of this paper it is sufficient to consider a *task* as a subnet showed in the Fig. 2.

If the place $exec_t$ contains a token then t is said to be *executed*.

F is an incidence relation, i.e. a set of the ordered pairs of nodes. Such pairs define a possibility of moving tokens through the net. *join* is a function describing a mode of consuming tokens by a transition. There are three possible modes: *XOR* (only one token is consumed from all input places), *AND* (one to-

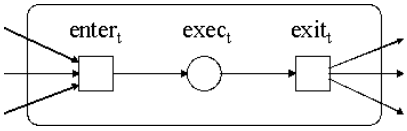

Fig. 2. EWF-net task structure

ken is consumed from each of the input places) and *OR* (one token is consumed from each of the several input places). If a transition has more than one input place (for example, incidence relation has two pairs related to *trunk* transition: $auxSatate1 \mapsto trunk, AuxSatate2 \mapsto trunk$), we should define join-behavior (in ur case as $trunk \mapsto XOR$). *split* is a function describing split-behaviour of transitions, i.e. a mode of token appearance in output places of the transitions at their firing. This function is reciprocal to the function *join* and is described similarly. *rem* is the function associating transitions with places, which should additionally be removed of tokens at firing of these transitions (thus cancellation is realized). In our case firing of the *trunk* transition leads to removing tokens from *enter1, enter2, auxState1, auxState2, branch1, branch2* places.

Such net is represented in AMN by a construction

REFINEMENT *Discriminator*
SETS $States = \{state\_enter1, state\_enter2,$
$state\_auxState1, state\_auxState2, state\_exit\}$
VARIABLES $States, Exec\_branch1, exec\_branch2, exec\_trunk$
INVARIANT $states \in States \rightarrow NAT \wedge exec\_branch1 \in NAT \wedge$
$exec\_branch2 \in NAT \wedge exec\_trunk \in NAT$
INITIALISATION
ANY $states1$ WHERE
$states1 \in States \rightarrow NAT \wedge \forall\, st \bullet (st \in \mathrm{dom}\,(states1) \Rightarrow states1(st) = 0)$
THEN
$States := states1$
END ||
$exec\_branch1 := 0 \,||\, exec\_branch2 := 0 \,||\, exec\_trunk := 0$
OPERATIONS ...

In the SETS section the *States* set is defined, which represents the set of a string constants conforming to names of places. In general, each task $t_i \in T$ is represented in AMN with the two operations $enter\_t_i$, $exit\_t_i$, and with a variable $exec\_t_i$ of the respective machine. Variable $exec\_t_i$ is typed in the *INVARIANT* section by the type of natural numbers. A natural number stored in $states(state\_c_i)$ reflects the number of tokens contained in the c_i place. The variable *states* and all variables corresponding to places are initialised in the INITIALISATION section as zero values, so that in the initial moment the net had no tokens.

In the OPERATIONS section an arbitrary operation $enter\_t_i$ is defined as follows, where a kind of predicate P_{enter} and a kind of substitution S_{enter} depend on join-behavior of task t_i (value of $join(t_i)$):

$enter\_t_i =$
SELECT $exec\_t_i = 0 \wedge P_{enter}$
THEN $S_{enter} \mid\mid exec\_t_i = exec\_t_i + 1$
END

For example, for operation $enter\_trunk$ of task $trunk$, P_{enter} looks as follows (what is characteristic for the XOR-join behavior):

$(states(state\_auxState1) > 0 \wedge states(state\_auxState2) = 0 \vee$
$(states(state\_auxState1) = 0 \wedge states(state\_auxState2) > 0)$

For the same operation a substitution S_{enter} realizes consumption of a single token from one input place.

IF $states(state\_auxState1) > 0$
THEN
  $states(state\_auxState1) := states(state\_auxState1) - 1$
ELSIF $states(state\_auxState2) > 0$
THEN
  $states(state\_auxState2) := states(state\_auxState2) - 1$
END

An arbitrary operation $exit\_t_i$ is defined so that a kind of substitution S_{exit} depends on split-behavior of task t_i (of the value of $split(t_i)$).

$exit\_t_i =$
SELECT $exec\_t_i > 0$
THEN $(S_{exit} \mid\mid exec\_t_i = exec\_t_i - 1);\ R_{exit}$
END

For example, for the operation $exit\_trunk$ the S_{exit} is defined as $State\_exit := state\_exit + 1$. R_{exit} for the same operation looks as follows.

$states :=$
  $States \lhd\!\!\!\!- \{st \mapsto num \mid num = 0 \wedge$
    $st \in \{state\_enter1, state\_enter2,$
      $state\_auxState1, state\_auxState2\}\} \mid\mid$
  $exec\_branch1 := 0 \mid\mid exec\_branch2 := 0$

Here tokens are removed from places *enter1, enter2, auxState1, auxState2* (respective variables are set to zero) and execution of tasks *branch1, branch2* is cancelled. $r_1 \lhd r_2$ denotes a relation r_1 overridden by a relation r_2.

The extension of the canonical process model for the *Discriminator* pattern has been defined by means of the generic script type described in the previous section.

The last stage of the proof, that the extension of the canonical model by the generic script type is correct, consists in applying of the automation facilities of B-Technology to prove that machine *DiscriminatorScript* is refined by machine *Discriminator*. For this purpose it is necessary to *conform* machines *DiscriminatorScript* and *Discriminator*. The conformance process consists of the following steps:

- point the refinement direction adding the REFINES section in the *Discriminator* machine.

  REFINES *DiscriminatorScript*

- conform operation names in refined and refining machines: for each transition t of the script rename the respective operation t of *DiscriminatorScript* machine into the operation *exit_t*; add the empty operation *enter_t* to *DiscriminatorScript* machine.

  $enter\_t = skip$;

- add of a *refinement invariant* in section INVARIANT of *Discriminator* machine. The invariant should describe a relationship between states of refined and refining machines.

  $\mathrm{card}\,(entrance1) = states(state\_entrance1) + exec\_branch1 \wedge$
  $\mathrm{card}\,(entrance2) = states(state\_entrance2) + exec\_branch2 \wedge$
  $(\,\mathrm{card}\,(auxState1) = states(state\_auxState1) + exec\_trunk \wedge$
  $\mathrm{card}\,(auxState2) = states(state\_auxState2) \vee$
  $\mathrm{card}\,(auxState2) = states(state\_auxState2) + exec\_trunk \wedge$
  $\mathrm{card}\,(auxState1) = states(state\_auxState1)) \wedge$
  $\mathrm{card}\,(exit) = states(state\_exit)$

  $\mathrm{card}\,(s)$ denotes a cardinality of a state s as a set.

At the end of the conformance process the tool for the automated AMN refinement proof has been applied (B-Toolkit 5.1.4). It had automatically formulated 65 theorems, expressing the fact that machine *DiscriminatorScript* is refined by machine *Discriminator*. Large number of theorems is explained by automatically subdividing complex theorems by the tool into simpler ones to prove them independently. For example, the theorem of initialisation refinement was subdivided into 6 theorems. 38 theorems were proved automatically by the tool,

the others were proved interactively. In the table 2 total number of theorems formulated and number of theorems automatically proved are shown.

	Number of theorems	Number of automatically proved theorems
The theorem of the unified state non-emptines	1	0
Theorems of the initialisation refinement	6	6
Theorems of refinement for operation *enter_branch*1	7	5
Theorems of refinement for operation *exit_branch*1	7	4
Theorems of refinement for operation *enter_branch*2	7	5
Theorems of refinement for operation *exit_branch*2	8	5
Theorems of refinement for operation *enter_trunk*	16	11
Theorems of refinement for operation *exit_trunk*	13	2
Total number of theorems	65	38

Table 2. The number of theorems

4 Conclusion

The provable synthesis of the canonical process model shown here appeared to be feasible due to: 1) the recently discovered possibility of interpretation of concurrent events, characteristic for process models, in logics and 2) the reduction of the workflow models diversity to a relatively small number of workflow patterns. These premises have served as necessary precondition for the canonical process model synthesis, following the principles proclaimed earlier. The methods developed constitute necessary basis for reaching semantic interoperability, reuse and composition of heterogeneous process components in distributed information systems.

At the same time, due to the exploding growth of number and variety of information representation models, it is difficult to cope with such variety of information models manually preserving the basic principles of mapping and synthesis of canonical models applying the refinement techniques. Therefore the methods developed should be supplemented with a compositional approach to the canonical models synthesis. This approach in general consists in registering of data types of each source data model in the canonical model so that they could serve as refinements of types or compositions of types already included into the canonical model. If there are no appropriate types in the current canonical model, we should construct its extension. Specifications of components (data types) of the canonical and source models are stored in a repository. Special tools are needed to discover necessary components, to match them, to eliminate structural and behavioral discrepancies of components, to form their compositions, and to prove commutativity of the resulting mappings. This idea is planned to be

developed on the basis of the existing approach for compositional development of information systems [6].

References

1. W.M.P. van der Aalst, et al. Workflow Patterns. – Distributed and Parallel Databases, 14(3):5-51, 2003.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language (Revised version). – QUT Technical report, FIT-TR-2003-04. Brisbane, 2003.
3. Abrial J.-R. B-Technology. Technical overview. – BP International Ltd., 1992.
4. J. -R. Abrial. The B-Book. – Cambridge University Press, 1996.
5. Jean-Raymond Abrial. B# : Toward a synthesis between Z and B. – In Proc. of the International Conference of Z and B Users ZB'2003. Springer, 2003.
6. Briukhov D., Kalinichenko L. Component-based information systems development tool supporting the SYNTHESIS design method. – In Proc. of the Second East European Conference ADBIS'1998. Springer, 1998.
7. M. Butler. csp2B: A Practical Approach to Combining CSP and B. – Formal Aspects of Computing, Vol. 12, 2000.
8. M. Butler, C. Snook. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. – In Proc. of the UML 2000 Workshop Dynamic Behaviour in UML Models: Semantic Questions.
9. Edsger W. Dijkstra. A discipline of programming. – Prentice Hall, 1976.
10. K. Jensen. Coloured Petri Nets: a High Level Language for System Design and Analysis. – Springer, 1991.
11. Kalinichenko L.A. Data model transformation method based on axiomatic data model extension. – Proc. of the 4th International Conference on Very Large Data Bases, 1978.
12. Kalinichenko L.A. Methods and tools for equivalent data model mapping construction. – Proc. of the International Conference on Extending Database Technology EDBT'90. Springer, 1990
13. Kalinichenko L.A. SYNTHESIS: the language for desription, design and programming of the heterogeneous interoperable information resource environment. – Moscow, 1995.
14. Kalinichenko L.A. Method for Data Models Integration in the Common Paradigm. – In Proc. of the First East-European Conference, ADBIS'97. St.Petersburg, 1997.
15. Kalinichenko L.A. Workflow Reuse and Semantic Interoperation Issues. – Advances in workflow management systems and interoperability; A.Dogac, L.Kalinichenko, M.T. Ozsu, A.Sheth (Eds.). NATO Advanced Study Institute, 1997.
16. L.A. Kalinichenko, S.A. Stupnikov, N.A. Zemtsov. Canonical models synthesis for heterogeneous information sources integration. – Moscow, 2005.
17. H. Ledang, J. Souquieres. Contributions for Modeling UML State-Charts In B. – In Proc. of the Third International Conference on Integrated Formal Methods IFM 2002. Springer, 2002.
18. Stupnikov S.A.,Kalinichenko L.A., Jin Song DONG. Applying CSP-like Workflow Process Specifications for their Refinement in AMN by Pre- existing Workflows. – In Proc. of the Sixth East-European Conference on Advances in Databases and Information Systems ADBIS'2002. Slovak University of Technology, 2000.
19. H. Treharne, S. Schneider. How to Drive a B machine. – In Proc. of the First International Conference of Z and B Users ZB'2000. Springer, 2000.