An Approach to Declaring Data Types for Formal Specifications

Xi Wang(✉) and Shaoying Liu

Department of Computer Science, Hosei University, Tokyo, Japan
xi.wang.y2@stu.hosei.ac.jp, sliu@hosei.ac.jp

Abstract. Data type declaration is an important activity in formal specification construction, which results in a collection of custom types for defining variables to be used in writing formal expressions such as pre- and post-conditions. As the complexity of software products rises, such a task will become more and more difficult to be handled by practitioners. This paper proposes an approach to facilitate the declaration of data types based on a set of *function patterns*, each designed for guiding the description of one kind of function in formal expressions. During the application of these patterns, necessary data types will be automatically recognized and their definitions will be gradually refined. Meanwhile, formal expressions will be modified to keep their consistency with the type definitions. A case study on a banking system is presented to show the validity of the approach in practice.

1 Introduction

Formal specification serves as the foundation of many software verification techniques, such as formal specification-based testing and inspection. It documents software behaviors in formal expressions, such as pre- and post-conditions, with a set of state variables of the envisioned system. These state variables needs to be formally defined with custom data types. Therefore, declaring appropriate data types is the first and important step for formal specification construction.

As the complexity of software rises, data type declaration becomes more difficult to manage and more likely to result in defected data types. Type checking technique and model transformation have been introduced to facilitate such an activity [1–3]. The former detects static type errors to prevent erroneous formal descriptions while the latter allows data to be described in certain intermediate language easier to use and provides a method for transforming the data model into formal data types. Unfortunately, they fall short of meeting practitioners' demand. First, relations between types and functions to be described is not considered, i.e., type definitions incapable of or unsuitable for describing the intended functions are not able to be identified. Secondly, no guidance or automated assistant is provided during the declaration process. Lastly, the consistency between formal expressions and type definitions cannot be guaranteed. In a formal specification f, if a type definition t is changed into t', all the formal

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2013, LNCS 8332, pp. 135–153, 2014.
DOI: 10.1007/978-3-319-04915-1_10,

expressions involving state variables defined with t need to be manually modified to be consistent with the new definition t'.

To deal with the above problems, this paper puts forward an approach to supporting data type declarations for formal specifications. Its underlying principle is that types should be defined to meet the need of correctly and concisely describing related functions. Type definitions will evolve as function description proceeds until all the expected functions are properly represented in formal expressions.

In this approach, *function pattern* is adopted to assist the writing of formal expressions [4]. Each *function pattern* provides a framework for formalizing one kind of function through interactions. Describing functions in formal expressions is to select appropriate patterns and apply them. During the application process, necessary data types can be automatically recognized and their definitions will be refined. Specifically, when applying each selected pattern, we use *function-related declaration* to guide the refinement of the related data types. It consists of two steps for different stages of the application process: *property-guided declaration* and *priority-guided declaration*.

We also give a method for updating formal expressions as their involved data types are refined to keep the consistency. When a type definition is modified after the application of a pattern, the formal expressions affected by such modification will be fully explored. For each formal expression, the method first retrieves the pattern applied for writing it and the application process of the pattern. Based on the retrieved information and the modified type definition., the formal expression is automatically updated.

Since we believe that object identification is an intelligent activity that cannot be manipulated by machines, the approach is not expected to be total automatic and requires human effort when creative decisions need to be made.

It should be noted that the proposed approach is language-independent. We choose SOFL as an example formal notation to illustrate the approach because of our expertise. A formal specification in SOFL comprises a set of modules in a hierarchical structure where lower level modules describe the detailed behavior of their upper level modules. Each module is an encapsulation of processes, which describe functions in terms of pre- and post-conditions, within the specific specification context of the module. Relation between these processes are reflected by a CDFD (Condition Data Flow Diagrams) which specifies interactions between them via data flows and stores. The independency of each module allows us to discuss the production approach on the module level and a complete set of data types will be achieved after applying the approach to all the modules included in the specification. For more details in SOFL, one can refer to [5,6].

The remainder of this article is organized as follows. Section 2 summarizes the related work. To facilitate the understanding of the proposed approach, some fundamental concepts are first introduced in Sect. 3, including data context and function pattern. Based on these concepts, the declaration approach is explained in detail in Sect. 4 and a case study is presented in Sect. 5 to illustrate the approach. Finally, Sect. 6 concludes the paper and points out the future works.

2 Related Work

We know of no existing approach that provides assistance throughout the whole data type declaration process, although some researches have been concerned with certain aspects of the problem.

To ensure type compliance and absence of erroneous descriptions, typecheckers are designed and implemented for various formal specification languages with different type systems. Jian Chen et al. [2] develop a simple but useful set of rules for type checking the object-oriented formal specification language Object-Z and an earlier version of the type checker for Z is given in [7]. For the Vienna Development Method (VDM), the most feature-rich analytic tool available is VDMTools which includes syntax- and type-checking facilities [1,8]. Syntax checking results in positional error reporting supported by an indication of error points. Typechecking can be divided into static type-checking and dynamic type-checking. The former checks for static semantics errors of specifications including incorrect values applied to function calls, badly typed assignments, use of undefined variables and module imports/exports, while the latter aims at avoiding semantic inconsistency and potential sources of run-time errors. As one of the major components in the Rodin tool for Event-B, static checker analyses Event-B contexts and Event-B machines and generates feedback to the user about syntactical and typing errors in them [9,10]. Prototype Verification System (PVS) extends higher order logic with dependent types and structural and predicate subtypes. In addition to conventional type-checking, it returns a set of proof obligations TCCs (Type Correctness Conditions) as potent detectors of erroneous specifications and provides a powerful interactive theorem prover that implements several decision procedures and proof automation tools [11,12]. Tan et al.[13] presents a type checker for formal specifications of software systems described in Real-Time Process Algebra, which is able to handle three tasks: identifier type compliancy, expression type compliancy and process constraint consistency. Xavier et al.[14] defines the type system of formal language Circus which combines Z, CSP and additional constructors of Morgan's refinement calculus, and describes the design and implementation of a corresponding type checker based on the typing rules that formalize the type system of Circus.

The quality of the declared data types can be significantly improved by the supporting tools listed above, unfortunately practitioners are still complaining about the difficulties in identifying real objects by formal data type definitions, the lack of effective guidelines throughout the declaration process and everlasting appearance of errors implicitly explained. Despite the use of "semantic analysis" in some of these tools' underlying theories, it refers to the semantics of the embedded type system that is part of the built-in mechanism, rather than the semantics of ideas in users' mind. By contrast, our approach tries to connect the semantics of specifications with the corresponding system behaviors through data types and evaluate the appropriateness of the declared types on the real semantic level. Moreover, the given systematic guidance in the overall declaration process specifies how to reach the appropriate data types step by step while

checking the correctness of the result of each step, which alleviates burdens of manual design.

There are also some researches done for transforming models in intermediate languages to formal data type definitions. These intermediate languages provide accessible visualization of object relation models and therefore simplify the object identification process. In [3], entity relationship models are treated as the basis for producing VDM data types in specifications. Colin Snook et al. [15] propose a formal modeling technique that emerge UML and B to benefit from both languages where the semantics of UML entities is defined via a translation into B. Anastasakis et al.[16] presents an automated transformation method from UML class diagrams with OCL constraints to Alloy which is a formal language supported by a tool for automated specification analysis. The problem, however, lies in the fact that identifying and defining objects are separated from functions to be described in these methods and totally depending on the developer's initial understanding of the real system. Hence our approach would be more reliable in declaring data types for function description and practitioners can utilize models in graphical representations as supplementary materials.

3 Preliminaries

3.1 Data Context

Constants and variables compose a data context under which formal expressions in formal specifications can be written and become analyzable. The formal definition of *data context* is given as follows.

Definition 1. *A data context is a 4-tuple* (C, T, V, vt) *where* C *is the set of constants,* T *is the set of custom data types,* V *is the set of variables and* $vt : V \rightarrow T$ *is the type function that determines the data type of each variable in* V*.*

To facilitate automated analysis and improve specification readability, each variable in the data context is required to be defined as a custom type in our approach, i.e., for each $v \in V$, there exists a type t in T that satisfies $vt(v) = t$. For example, when describing an ATM system, a password should be defined as a variable of a custom type declared as *string*. Although the built-in type *string* itself is capable of representing the nature of password and one can define the required password as a variable of *string* type, it fails to distinguish the object from others that are also defined as *string*, such as error messages. In addition, modification on the definitions of all the password entities in the specification can be easily manipulated by modifying the definition of the corresponding custom type.

3.2 Function Pattern

A function pattern provides a framework for formalizing one kind of function. Different from traditional ones, function pattern is designed to be applied in a full

automated way. For each unit function intended to be described, the developer will be first guided to select a proper function pattern and then a formal expression will be generated by automatic application of the pattern. The definition of function pattern is given as follows.

Definition 2. *A function pattern p is a 6-tuple $(id, E, PR, \Delta, \Phi, \Psi)$ where*

- *id is a unique identify of p written in natural language*
- *E is a set of elements that needs to be specified to apply p, which can be assigned with 3 kinds of values: choice value (CV) generated by choosing from several candidate items, variable value (VV) composed of constants and system variables, and property value (PV) that specifies properties of certain objects*
- *PR is a property set including properties of the pattern p or properties of the elements in E*
- *$\Delta : PR \rightarrow \mathcal{P}(E \cup \{p\})$ indicates the objects involved in each property $pr \in PR$ which may include p and elements in E.*
- *$\Phi : SN \times (E \cup \mathcal{P}(PR)) \rightarrow E \cup \mathcal{P}(PR)$ indicates a set of rules including element rules and property rules where*
  - *each $sn : SN$ denotes the sequence number of the rule it associated with*
  - *$\exists_1 x : E \cup \mathcal{P}(PR) \cdot \Phi(1, null) \rightarrow x$*
  - *$\forall i : SN, e : E \cdot e' = \Phi(i, e) \Rightarrow e' \in E \wedge e' \neq e$(e' = \Phi(i, e) denotes an* element rule *where e' should be specified after e)*
  - *$\forall i : SN, PR_i : \mathcal{P}(PR) \cdot PR_j = \Phi(i, PR_i) \Rightarrow PR_j \in \mathcal{P}(PR) \wedge (\forall pr : PR_i \cdot pr \notin PR_j)$($PR_j = \Phi(i, PR_i)$ denotes a* property rule *where properties in PR_j will be hold if each property in PR_i is satisfied)*
  - *$\forall PR_i, PR_j : \mathcal{P}(PR) \cdot (\exists i : SN \cdot (i, PR_i) \in dom(\Phi) \wedge (i, PR_j) \in dom(\Phi)) \Rightarrow ((\forall pr_i : PR_i \cdot pr_i = true) \Rightarrow (\exists pr_j : PR_j \cdot pr_j = false))$*
- *$\Psi : \mathcal{P}(PR) \rightarrow exp$ produces a formal result exp when certain properties in PR are satisfied iff*
  $\forall PR_i, PR_j : dom(\Psi) \cdot (\forall pr_i : PR_i \cdot pr_i = true) \Rightarrow (\exists pr_j : PR_j \cdot pr_j = false)$

Since patterns are categorized and each pattern holds a distinguishable *id* that reflects, on an abstract level, the function it is able to describe, developers can easily select the most suitable pattern. After the selection decision of certain pattern p is made, elements in E_p will be required to be specified under the guidance produced by the rules in Φ_p. The obtained element information is then analyzed within the context of Ψ_p to determine its corresponding formal result.

But such a formal result may still contain informal statements composed of pattern *id* and element information, which indicates that further formalization needs to be conducted by applying the reused patterns with the attached element information. For example, suppose a formal result involving informal statement "$p'(v_1, v_2)$" is achieved where $E_{p'} = \{e_1, e_2, e_3\}$ and $\exists_{i,j \in SN_{p'}} \cdot (i, e_1) \rightarrow e_2 \wedge (j, e_2) \rightarrow e_3$, it should be further formalized by replacing the informal statement with the formal result generated by applying p' with e_1 and e_2 set as v_1 and v_2 respectively. This procedure will not be terminated until reaching a formal expression.

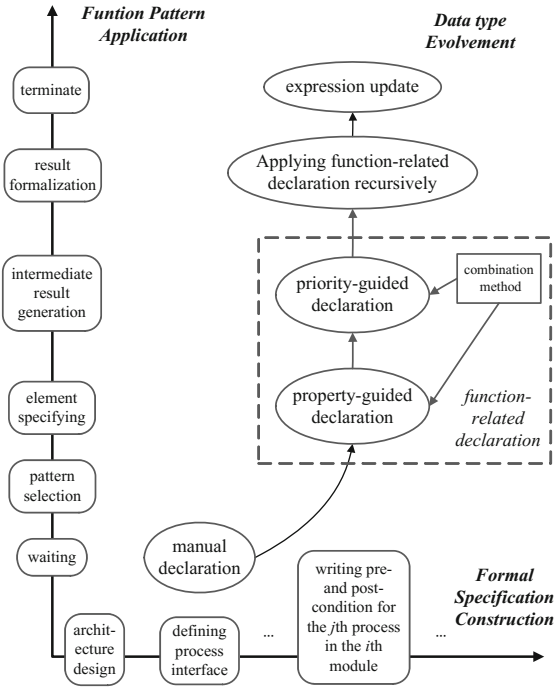

Fig. 1. The outline of the data type declaration approach

4 The Approach to Declaring Data Types

4.1 Approach Outline

The proposed approach regards data type declaration as an evolution process along with the writing of formal expressions based on function patterns. This evolution process starts with a modulized formal specification and terminates when the detailed behavior of each module is precisely given. Figure 1 shows the outline of the approach where x-axis and y-axis indicate the pattern application process and the formal specification construction process respectively.

On the assumption that specification architecture is already established where modules are organized in a hierarchical structure and processes of each module are connected by their interfaces, developers will first be required to manually declare data types for defining these interfaces. Since process behaviors is not considered in this stage, the declared data types only reflects the initial idea of the intended functions and will be refined as the function details are clarified.

Then the description of individual processes is started where each process should be attached with a pair of pre- and post-condition. For each pre-/post-condition, a pattern suitable for describing the expected function will first be selected. The selected pattern is then applied. Step 1 is to guide the specifying of its elements and step 2 is to generate an intermediate formal result based on the

specified elements. During these two steps, *function-related declaration* is carried out to declare new types and refine the existing type definitions where *property-guided declaration* is carried out on step 1 and *priority-guided declaration* is carried out on step 2. The former guides the refinement of type definitions under the principle that all the properties inferred from the specified elements should be satisfied while the latter provides suggested definition of certain types according to the priority attribute associated to Ψ of the selected pattern. These two techniques share a type combination method that refines the existing type definitions by combining different definitions of the same type. For example, suppose pattern p is selected to write a formal expression and type t is initially declared as definition def_1 for specifying element e_1 of p. When specifying element e_2, *property-guided declaration* leads to a suggestion that t should be defined as definition def_2 to enable the correct representation of the value assigned to e_2. If def_1 is not equal to def_2, the combination method will be applied to refine def_1 with def_2 by combining them into a new definition for declaring t.

If the generated intermediate result contains informal expressions, formalization of the result is needed. Since it is performed by applying the patterns indicated by the informal expressions, *function-related declaration* can be repeatedly manipulated to further refine the data types of the specification. When the formalization process terminates with a formal expression, a refined data context is obtained. Finally, *expression update* is carried out where all the formal expressions that are inconsistent with the refined data context are updated.

Serving as the critical techniques in the described declaration approach, *function-related declaration* and *expression update* will be presented in details respectively.

4.2 Function-Related Declaration

Function-related declaration guides the refinement of data types to enable the application of the selected function patterns. It adopts *property-guided declaration* and *priority-guided declaration* in declaring data types for specifying element and generation intermediate result, respectively. Before presenting the detailed techniques in *function-related declaration*, some necessary concepts are introduced first.

Definition 3. *Given a data context dc and a pattern p,* $es_p^{dc} : E_p \rightarrow \mathcal{P}(Choices) \cup Exp_{dc} \cup \mathcal{P}(Props_{dc})$ *is an* element state *of p under dc revealing the value of each element* $e \in E_p$ *where*

- $Choices$ *denotes the universal set of choice values*
- Exp_{dc} *is the universal set of formal expressions within context dc and each* $exp_{dc} \in Exp_{dc}$ *is a sequence:* $N^+ \rightarrow C_{dc} \cup V_{fsc} \cup Operator$ *where* $Operator$ *is the set of operators in formal notations*
- $Props_{dc}$ *denotes the universal set of property values within context dc and for each* $prop \in Props_{dc}$, $inVar(prop)$ *is adopted to denote the variables involved in prop.*

It should be noted that es_p denotes all the possible element states of p, i.e., set $\{es_p^{dc_1}, ..., es_p^{dc_i}, ...\}$ where $\{dc_1, ..., dc_i, ...\}$ is the universal set of data contexts.

Definition 4. *Given a data context dc and a pattern p, function* $satisfy_p^{dc} : PR_p \times ES_p^{dc} \rightarrow$ *boolean denotes satisfaction relations between properties and element states where each* $es_p^{dc} \in ES_p^{dc}$ *is a possible element state of p under dc and* $satisfy_p^{dc}(pr, es_p^{dc})$ *indicates* $\forall_{e \in \Delta(pr)} \cdot es_p^{dc}(e) \neq \varnothing \wedge pr$ *is satisfied by* es_p^{dc}.

Definition 5. *Given a data context dc and a pattern p,* $condSatisfy_p^{dc} : es_p \rightarrow \mathcal{P}(\ominus_p)$ *is a* conditional satisfaction *function iff*

- $es_0 \in es_p \wedge \forall_{e \in dom(es_0)} \cdot es_0(e) = \varnothing \Rightarrow$
  $condSatisfy_p^{dc}(es_0) = \ominus_p$
- $condSatisfy_p^{dc}(es_p^{dc}) = R \Rightarrow$
  $\forall_{PR_i \in dom(R)} \cdot \forall_{pr \in PR_i} \cdot (satisfy_p^{dc}(pr, es_p^{dc}) \vee$
  $((\exists_{e \in \Delta(pr)} \cdot es_p^{dc}(e) = \varnothing) \wedge$
  $(pr, es') \notin dom(satisfy_p^{dc})))$
  $where\ es' \subset es_p^{dc} \wedge \forall_{e \in dom(es_p^{dc} - es')} \cdot$
  $es_p^{dc}(e) = \varnothing \wedge (\forall_{e' \in dom(es')} \cdot es(e') \neq \varnothing)$

Due to the fact that the type combination method is employed in both *property-guided declaration* and *priority-guided declaration*, it is first introduced.

Type Combination. Type combination is an operation that combines two different definitions of the same type into an appropriate new definition for declaring that type. The result of the operation is determined by certain properties held by the definition pair. Considering that it is impossible to combine all kinds of definition pairs automatically, the strategy of the operation is to deal with syntactic issues by machines and ask the developer to handle the semantic problems.

In order to precisely describe various properties of definition pairs, the concept of *subtype* is introduced and formally defined as follows.

Definition 6. *Given a custom type ct,* $subType(ct)$ *denotes the subtype of ct where*

- *ct is basic type* $\Rightarrow subType(ct) = \varnothing$
- *ct is composite type with each field* f_i *defined as type* $t_i \Rightarrow subType(ct) = \{(f_1, t_1), ..., (f_n, t_n)\}$
- *ct is product type with the ith field defined as type* $t_i \Rightarrow subType(ct) = \{1 \rightarrow t_i, ..., n \rightarrow t_n\}$
- *ct is set or sequence type with each element defined as type* $t \Rightarrow subType(ct) = t$
- *ct is mapping type with domain defined as type* t_i *and range defined as* $t_j \Rightarrow subType(ct) = (t_i, t_j)$

Table 1. Solution table for type combination

Property of definition pair			Combination solution
$buildIn(d) = buildIn(d')$	basic		human effort
	set/sequence		$sol(subType(d), subType(d'))$
	composite	$subType(d) \subset subType(d')$	$def = d'$
		$dom(subType(d)) = dom(subType(d')) \wedge \exists_{(f,t)\in subType(d),(f',t')\in subType(d')} \cdot f = f' \wedge t \neq t'$	$\forall_{(f,t)\in subType(d)} \cdot \exists_{(f,t')\in subType(d')} \cdot t \neq t' \Rightarrow sol(t, t')$
		.....	.....
	product	$rng(subType(d)) \subset rng(subType(d'))$	$def = d'$
		.....	.....
	map	$dom(d) = dom(d') \wedge rng(d) \neq rng(d')$	$sol(rng(d), rng(d'))$
		$dom(d) = rng(d') \wedge rng(d) = dom(d')$	human effort
		.....	.....
$buildIn(d) \neq buildIn(d')$	$buildIn(d)$ and $buildIn(d')$ are basic types		human effort
	$buildIn(d) = composite \wedge buildIn(d') = set$	$\exists_{(f,t)\in subType(d)} \cdot t = d'$	$def = d$
		.....	.....
	$buildIn(d) = composite \wedge buildIn(d') = mapping$	$d = dom(d') \vee d = rng(d')$	$def = d'$
		.....	.....
	.....	.....	.....

Based on the definition, we try to summarize possible properties of definition pairs and figure out the corresponding combination solutions. Table 1 (with formal notations in SOFL) shows part of the work where $buildIn(t)$ denotes the built-in type that type t belongs to and def indicates the result definition of the combination operation. For each pair of type definition d and d' where $d \neq d'$, a combination solution $sol(d, d')$ can be found by matching the definition pair against the properties listed in the table.

It can be seen from the table, properties of definition pair are classified into two categories: properties where d and d' belong to the same built-in type and properties where d and d' belong to different build-in types. The first category is further divided into five sub-categories that cover all the built-in types (in SOFL) and a solution is provided for each specific property within each built-in type. For example, the first "basic" denotes the property that d and d' belongs to the same basic type and its corresponding solution "human effort" indicates the combination of such kind of definition pair needs intelligent decision and the developer will be asked to give the operation result based on d and d'. More specific properties are provided with combination solutions if both d and d' are composite types. The second property within "composite" category and its corresponding solution mean that if d and d' owns the same fields and some of them are declared as different types, the combination method should be conducted on each pair of different types to achieve $sol(d, d')$. Within the second category, all combinations of different built-in types are considered and only parts of them are listed in the table for the sake of space. For instance, if d and d' are declared as different basic types, only human effort is able to figure out the proper definition. In case d is a composite type and d' is a set type, the combination result should be d if one of the fields in d is defined as d'.

Property-Guided Declaration. Figure 2 shows the main procedure of *property-guided declaration* for each selected pattern p within data context dc where cE

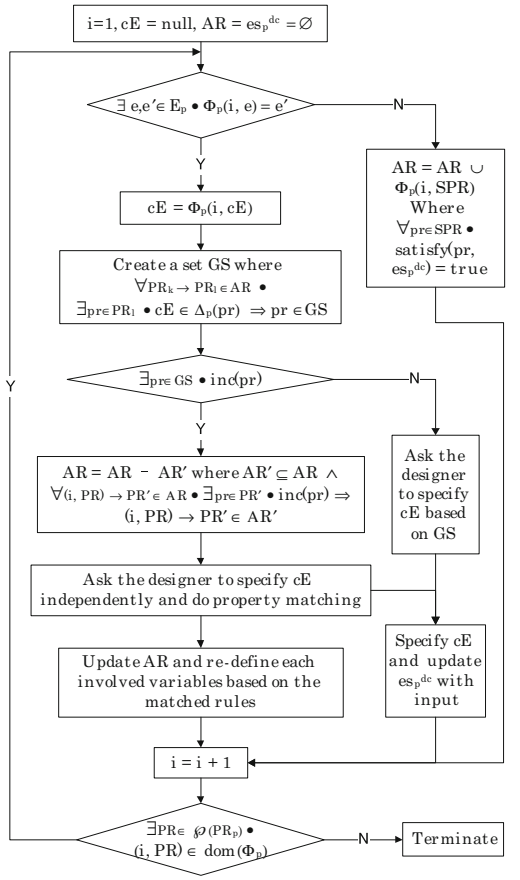

Fig. 2. The main procedure of the property-guided declaration

denotes the element currently being specified, AR denotes the set of activated rules and $inc(pr)$ denotes that the developer identifies the property pr as being inconsistent with the expected function.

Rules in Φ_p are applied sequentially according to their attached sequence numbers and when dealing with those who own the same number, the one with its required conditions satisfied will be activated. For each $i(0 < i \leq maximum\ sequence\ number)$, if i corresponds to a set of property rules $R \subset \Phi_p$, the rule $(i, SPR) \rightarrow SPR' \in R$ will be identified where all the properties in SPR can be satisfied. Meanwhile, a set of new properties SPR' will be obtained and added to AR. If i corresponds to an element rule, cE will be set as $\Phi_p(i, cE)$ which is the next element waiting to be specified. To assist the value assignment to cE, activated rules that lead to properties of cE will be extracted from set AR and these properties form a property set GS. After confirming that all the properties in GS are consistent with the desired function, the developer needs to assign a

value to cE based on GS. In case that certain properties in GS violate the expected function, the activated rules that lead to these properties form a set AR' and will be deleted from AR. Then the developer will be required to specify cE manually and property matching will be carried out to obtain the rules that match the given value.

In addition to the value v assigned to cE by the developer, set $CR : \mathcal{P}(\mathcal{P}(\Phi_p))$ serves as another critical participant in property matching which satisfies:

$$\begin{aligned}&\forall_{R\in CR}\cdot(\forall_{(m,x),(n,y)\in dom(R)}\cdot m = n\wedge\\ &\qquad\forall_{R'\in CS-\{R\}}\cdot\forall_{(k,x')\in R,(l,y'\in R')}\cdot k\neq l)\\ &\wedge\forall_{(no,Pr)\in dom(AR')}\cdot\exists_{R\in CR}\cdot\forall_{(no',Pr')\in dom(\Phi_p)}\cdot\\ &\qquad(no = no'\wedge\exists_{pr\in\Phi_p(no',Pr')}\cdot cE\in\Delta(pr))\end{aligned}$$

Each set $R \in CR$ comprises all the candidate rules for substituting one of the rules that lead to properties violating the expected function. With the given v, dc will be updated accordingly and property matching can be by the following algorithm where RS denotes the set of rules that match the given v:

$RS = temp = \{\}$;
for each $R \in CR\{$
  for each $Pr \rightarrow Pr' \in R$
    if $(satisfy_p^{dc}(Pr', es_p^{dc}) = true)$
      $temp = temp \cup \{Pr \rightarrow Pr'\}$;
  if $(|\ temp\ | = 1)$
    for the only element sr $\quad RS = RS \cup \{sr\}$;
  else$\{$
    $tempP = \{\}$;
    for each $Spr \rightarrow Spr' \in temp$
      $tempP = tempP \cup \{Spr\}$;
    display all the items in $tempP$ *and*
    ask the developer to choose
      the most appropriate one "item;"
    $RS = RS \cup \{r\}$ *where*
      $r \in temp \wedge \exists_{y\in\mathcal{P}(\Phi_p)}\cdot r = item \rightarrow y;\}\}$
return RS;

This algorithm helps explore a set RS containing all the rules in $\mathcal{P}(\Phi_p)$ consistent with the function intended to be described which is reflected by the values assigned to elements. These rules will then be added into set AR and for each rule $Spr \rightarrow Spr'$, data context dc will be updated according to Spr.

Priority-Guided Declaration. The main idea of priority-guided declaration is to provide suggested definition of concerned types based on Ψ after assigning values to pattern elements. Rules in each Ψ are attached with priority attributes which help select a most appropriate one when elements are incompletely specified or no rule can be applied according to the specified elements.

Definition 7. *Given a pattern p,* $PS_p : \mathcal{P}(\mathcal{P}(\ominus_p))$ *is the* priority set *of p iff*

- $\forall_{ps_i \in PS_p} \cdot \exists_{es_p^{dc} \in es_p} \cdot condSatisfy_p^{dc}(es_p^{dc}) = ps_i$
- $\forall_{R \in \mathcal{P}(\ominus_p)} \cdot \exists_{es_p^{dc} \in es_p} \cdot condSatisfy_p^{dc}(es_p^{dc}) = R \Rightarrow R \in PS_p$

Definition 8. *Given a pattern p,* $\tau_p : \ominus_p \times PS_p \rightarrow N^+$ *determines the priority of each rule in* $\ominus_p$ *where* $\tau_p(r, ps_i) = n$ *means that* $r \in \ominus_p$ *is ranked as the* n*th rule in set* ps_i*.*

Based on the definition, priority-guided declaration is conducted as the following steps for each selected pattern p within formal specification context fsc.

1. Ask the developer to provide element information, and define types and variables when necessary, which results in an element state es_p^{fsc}.
2. Analyze priority set PS_p and extract the item $ps \in PS_p$ that satisfies $condSatisfy_p^{fsc}(es_p^{fsc}) = ps$.
3. Sort set ps into a sequence $psSeq$ where
   $\forall i, j : int \cdot 0 < i < j \leq \mid psSeq \mid \Rightarrow$
   $\tau_p(psSeq(i), ps) > \tau_p(psSeq(j), ps)$
4. Set $rule = psSeq(k)$ where k is initialized as 1. Provide the properties involved in $rule$ for the developer to assist the declaration of relative types and variables.
5. If the suggestion is not accepted and $k \leq \mid psSeq \mid$, set $k = k + 1$ and repeat step 4-5. Otherwise terminate.

4.3 Expression Update

In contrast to the traditional formal specification construction method that requires formal expressions to be written manually, function patterns enables automatic generation of formal expressions based on the given values of necessary elements. Therefore, instead of grammar checking, the essential idea of expression update in our approach is to record the element values specified during the pattern application process and reuse that information to update the original formal expression. For an expression exp generated through the application process ap of the pattern p, if exp becomes erroneous under the refined data context, it will be replaced by a new expression generated by applying p again based on ap.

Definition 9. *Given a pattern* p_0*, sequence* $(p_0, es_{dc_0}^{p_0}, exp_0, p_1, es_{dc_1}^{p_1}, exp_1, ..., p_n, es_{dc_n}^{p_n}, exp_n)$ *is the application process of* p_0 *where*

- $p_1, ..., p_n$ *are the reused patterns*
- *each* $es_{dc_i}^{p_i}$ *denotes the element state after all the elements in* E_{p_i} *are specified*
- exp_0 *denotes the intermediate formal result produced by applying* p_0 *with specified elements in* $es_{dc_0}^{p_0}$*, which can be represented as* $exp_0 = p_0(es_{dc_0}^{p_0})$

- *each* $exp_i (0 < i \leq n)$ *denotes the intermediate formal result generated by replacing certain informal part in* exp_{i-1} *with* $p_i(es^{p_i}_{fsc_i})$*, which can be represented as* $exp_i = exp_{i-1} \oplus p_i(es^{p_i}_{dc_i})$ *where* $exp_i = p_i(es^{p_i}_{dc_i})$ *if* $exp_{i-1} = \varnothing$
- exp_n *is the resultant formal expression*

Definition 10. *Given a data context* dc*,* $vdept_{dc} : T_{dc} \rightarrow V_{dc}$ *reveals dependent relations between types and variables where* $vdept_{dc}(t) = V$ *indicates that for each variable* $v \in V$*, the definition of* $vt_{dc}(v)$ *involves type* t*.*

Definition 11. *Given a data context* dc *and a pattern* p*,* $sdept^p_{dc} : T_{fsc} \rightarrow \mathcal{P}(es^{dc}_p)$ *reveals dependent relations between types and element values where*

$sdept^p_{dc}(t) = Es^{dc}_p \Rightarrow$
$\forall_{e \rightarrow vl \in Es^{fsc}_p} \cdot (vl \in Exp^{dc}_p \wedge$
$\exists_{i \in N^+, v \in V_{dc}} \cdot (i, v) \in vl \wedge v \in vdept_{dc}(t)$
$\vee (vl \in Props_{dc} \wedge \exists_{v \in inVar(vl)} \cdot v \in vdept_{dc}(t))$

Assume that the data context dc has been modified into dc', the update of each formal expressions exp previously written through application process $ap = (p_0, es^{p_0}_{dc_0}, exp_0, p_1, es^{p_1}_{dc_1}, exp_1, ..., p_n, es^{p_n}_{dc_n}, exp_n)$ is conducted as the following algorithm where $def_{dc}(t)$ denotes the definition of type t under dc.

$if(\exists_{(i,v) \in exp} \cdot (vt_{dc}(v) \neq vt_{dc'}(v)) \vee$
$(\exists_{t \in T_{dc}} \cdot t \in T_{dc'} \wedge v \in vdept_{dc}(t) \wedge$
$v \in vdept_{dc'}(t) \wedge def_{dc}(t) \neq def_{dc'}(t))\{$
$exp_{-1} = \varnothing;$
for each p_i in $ap\{$
$if(\exists_{t \in T_{dc}, e \rightarrow vl \in es^{p_i}_{dc}} \cdot t \in T_{dc'} \wedge$
$def_{dc}(t) \neq def_{dc'}(t) \wedge e \rightarrow vl \in vdept_{dc}(t))$
$exp'_i = exp'_{i-1} \oplus p_i(es^{p_i}_{dc'});$
$else$
$exp'_i = exp'_{i-1} \oplus temp$ where $exp_i = exp_{i-1} \oplus temp\}$
$exp = exp'_n;\}$

The algorithm first checks whether there exist variables or types used in exp with definitions being modified. If so, the application of pattern p_0 will be restarted with element information $es^{p_0}_{dc}$ and further formalization will be conducted by applying the rest of the reused patterns in ap with their element information sequentially. Before generating formal expression for each pattern p_i, the value of each element indicated by $es^{p_i}_{dc}$ will be analyzed to determine its change caused by the update of the data context. Expression exp_i can be directly used to formalize the current formal result exp'_{i-1} if no difference is found between $es^{p_i}_{dc}$ and $es^{p_i}_{dc'}$. Otherwise, $p_i(es^{p_i}_{dc'})$ will be produced to replace the corresponding informal part of exp'_{i-1}.

5 Case Study

A case study on a banking system is presented to show the feasibility and effectiveness of the proposed approach in practice. The system allows for the

management of various currency types and mainly provides four services for authorized customers: *deposit*, *withdraw*, *account information display* and *currency exchange*. Since architecture design is not discussed in this paper, we assume that it has already been done and the result is a CDFD shown in Fig. 3 where rectangles drawn with input and output ports are processes and other three are datastores. It can be seen from the figure, neither type definitions nor relation between the input and output of each process is provided in the CDFD, and it only specifies the interfaces of the included processes and demonstrates their relation with data flows represented as solid lines and control flows represented as dotted lines. For example, process $Id\_confirm$ owns one input port and two output ports, and when receiving data flow $inputInf$, it will be activated and generate data flow inf or $warning$ when terminated. If inf is generated, it will reach process $Selection$ which produces one of the four possible outputs according to the available control flow.

Based on the CDFD, necessary data types can be declared to meet the need of accurately describing the behavior of each enclosed process sequentially. Due to space limitation, we take process $Id\_confirm$ and $Withdraw$ as examples. For the process $Id\_confirm$, manual declaration is first required for defining its inputs and outputs. According to the expected behavior of the process, one can easily response with the following definitions:

$$
\begin{aligned}
&Num = string, Psd = string, Msg = string,\\
&Inf = composed\ of\\
&\qquad inf\_num : Num\\
&\qquad\ inf\_psd : Psd\\
&\qquad end\\
&inputInf : Inf, warning : Msg, inf : Inf
\end{aligned}
$$

No pre-condition is needed in the process and the informal idea of the post-condition is that if the provided ID information can be found in the datastore *account_store*, data flow inf will be produced. Otherwise, error message *warning* will be displayed to the customer. Such idea leads us to the selection

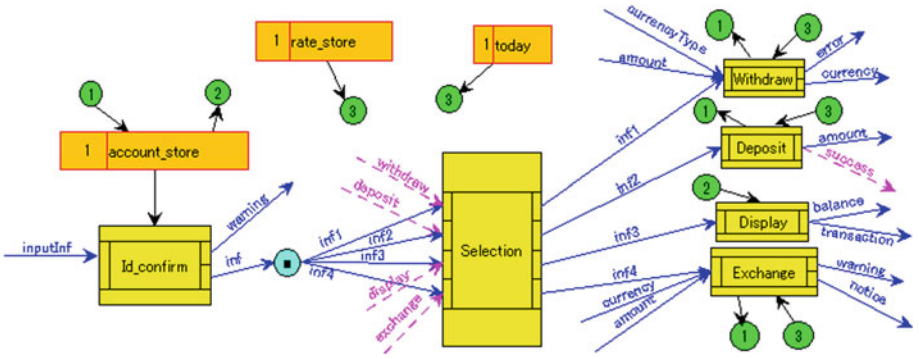

Fig. 3. The CDFD of the example banking system

Table 2. Pattern "belongTo"

id	belongTo
expl	A relation between 2 objects where one is part of another
E	$\{element, container : VV, specifier : null \mid PV\}$
PR	$\{dt(element) = T, dt(container) = set\ of\ T,$ $specifier = f_i, specifier = null, ...\}$
Ψ	$\{\{dt(element) = T, dt(container) = set\ of\ T\}$ $\underrightarrow{a}\ element\ inset\ container,$
	$\{dt(element) = T, dt(container) = seq\ of\ T\}$ $\underrightarrow{b}\ element\ inset\ elems(container),$
	$\{dt(element) = set\ of\ T, dt(container) = T \rightarrow T'\}$ $\underrightarrow{c}\ belongTo(elemetn,\ dom(container)),$
	$\{dt(element) = set\ of\ T, dt(container) = composite$
	$specifier = f_i\}\ \underrightarrow{d}\ element\ inset\ container.f_i,$
	$\{dt(element) = T \rightarrow T', dt(container) = T \rightarrow T',$
	$specifier = null\}\ \underrightarrow{e}\ element\ subset\ container,$ $\{dt(element) = seq\ of\ T, dt(container) = set\ of\ product,$ $specifier = null\}\ \underrightarrow{f}\ exists[e : container] \mid$ $forall[i : N^+] \mid element(i) = e(i), ...\}$

of pattern *belongTo* as shown in Table 2 where $dt(v)$ indicates the data type of the element e in E. It is used to describe a relation where one object is part of another. There are three elements in the pattern: *element* denoting the member object, *container* denoting object that *element* belongs to and $specifier$ denoting constraints on their relations which can be assigned with either *null* or a property value. The application of pattern *belongTo* starts from the requirement of specifying these three elements. Apparently, *element* is $inputInf$ and container is *account_store* which has not been defined. In case that $specifier$ is not decided yet, the generation of an intermediate result begins and *priority-guided declaration* will be carried out according to the priority knowledge given in Table 3. Suppose the developer uses "*AccountFile*" to represent its type, priority set $ps_1(\cup)$ is then selected and rule a is first suggested which indicates that the type *AccountFile* should be defined as $set\ of\ Inf$. Assume that the suggestion is accepted, the formal expression for describing the "belongTo" relation is automatically generated and the post-condition of process $Id\_confirm$ will be written as:

$if\ (inputInf\ inset\ account\_store)$
$then\ inf = inputInf$
$else\ warning =$"$Invalid\ user.$"

Table 3. Priority in pattern "belongTo"

Rule	**Priority set**				
	$ps_1(\cup)$	$ps_2(a,b,c,d)$	$ps_3(a,b,e)$	$ps_4(a,b,f)$	...
a	1	3	2	2	
b	2	4	3	3	
c	3	1	-	-	
d	4	2	-	-	
e	5	-	1	-	
f	6	-	-	1	
...					

Table 4. Pattern "alter"

id	alter
expl	To describe the altering of certain system variable
E	$\{obj : VV, decompose : Boolean,$ $specifier, onlyOne : Boolean, new\}$
PR	$\{the\ pattern\ is\ reused,\ dt(obj) = basic,$ $dt(obj) = set\ of\ composite(with\ each\ field\ as\ f_i),$ $dt(obj) = T \rightarrow T', decompose = false, ...\}$
Φ	$\{(1, null) \xrightarrow{a} obj, (2, \{dt(obj) = basic\}) \xrightarrow{b}$ $\{decompose = false\}, ..., (3, obj) \xrightarrow{c} decompose,$ $(4, \{dt(obj) = basic\}) \xrightarrow{d} \{specifier = null\},$ $(4, \{dt(obj) = set\ of\ composite(with\ each\ field\ as\ f_i)$ $, decompose = true\}) \xrightarrow{e} \{specifier : \mathcal{P}(\{f_i\})\},$ $(4, \{dt(obj) = T \rightarrow T', decompose = true\}) \xrightarrow{f}$ $\{specifier : (\{constraints(dom), constraints(rng),$ $constraints((dom, rng))\}, dom \mid rng),$ $..., (5, decompose) \rightarrow specifier, ...\}$
Ψ	$\{\{the\ pattern\ is\ not\ reused, dt(obj) = T \rightarrow T'$ $decompose = true, specifier = (dom = x, rng),$ $new = alter\} \rightarrow alter(obj(x)), ...\}$

Notice that no formal expression was written before the application of pattern *belongTo*, expression update is therefore not needed.

Since the data types and functions involved in the process *Selection* are simple enough to be manually written and the data context will not be affected after the description, data type declaration during the construction of process $Withdraw$ is presented based on the type definitions declared for the process $Id\_confirm$. Process $Withdraw$ takes the intended currency type and amount

as inputs and currency or error messages as outputs, which can be manually defined as:

$$CurrencyType = string, Amount = real,$$
$$currencyType : CurrencyType, amount : Amount$$
$$currency : Amount, error : Msg$$

The pre-condition is also true and the post-condition should clarify how the account information in *account_store* is altered when the withdraw operation is successfully done. Therefore, pattern *alter* will be selected to describe such function, which is shown in Table 4 where $constraints(x)$ denotes certain constraints on object x. It contains four elements for depicting the altering of system variables: *obj* denoting the object to be altered, decompose meaning to replace the whole given *obj* by a new value if it is designated as true and to modify parts of the given *obj* if it is designated as false, $specifier$ denoting the description of the parts to be altered within *obj*, *onlyOne* meaning there exists only one part consistent with the description in $specifier$ if it is designated as true and new indicates the new values for replacing the corresponding parts to be altered. Figure 4 reveals the property-guided declaration process during the application of the pattern.

The above application process results in an definition "$Inf \rightarrow AccountInf$" for type $AccountFile$ that is more appropriate for describing process $Withdraw$.

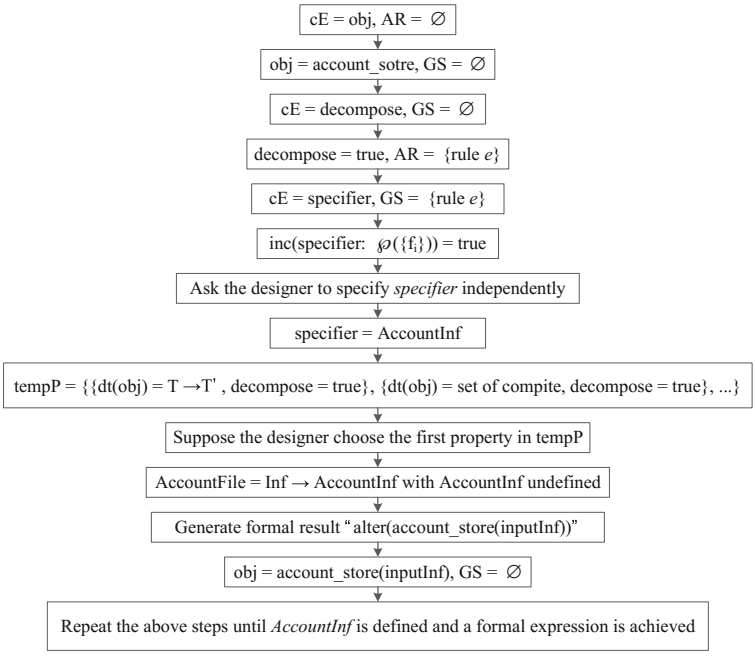

Fig. 4. The priority-guided declaration process during the application of the pattern "alter"

Thus, the original definition $set\ of\ Inf$ needs to be refined by applying the combination method. According to the solution table for type combination, the definition of type $AccountFile$ should be refined as: $Inf \rightarrow AccountInf$.

Due to the refinement of the definition of type $AccountFile$ and the use of the type in the post-condition of process $Id\_confirm$, formal expression previously generated by applying the pattern $belongTo$ needs to be updated accordingly. The application process of the pattern $belongTo$ for the post-condition of process $Id\_confirm$ can be described as:

$$(belongTo, \{element \rightarrow inputInf, \\ container \rightarrow account\_store, \\ specifier = null\}, \\ \text{“}inputInf\ inset\ account\_store\text{”})$$

According to the algorithm for expression update, formal expression “$inputInf\ inset\ account\_store$” will be transformed into:

$$belongTo(\{element \rightarrow inputInf, \\ container \rightarrow account\_store, \\ specifier = dom\}, newExp)$$

where $account\_store$ is defined as a map type and element $specifier$ is modified into “dom” in the refined data context. By analyzing the above expression in the context of the Ψ of the pattern $belongTo$, formal expression “$inputInf\ inset\ dom(account\_store)$” will be generated as the value of $newExp$ to replace the original one.

Following the similar procedures for describing process $Withdraw$, the data context can be gradually refined while the pre- and post-conditions of the rest processes $Deposit$, $Display$ and $Exchange$ are specified. After completing the description of the last process $Exchange$, a set of appropriate data types for the example banking system is established.

6 Conclusion

This paper proposes an approach to assist the declaration of data types for formal specifications along with the function description in formal expressions. After the architecture of the formal specification is determined, the approach helps adjust the type definitions to fit the expected functions captured and formally described by applying function patterns. Besides, as the data types are refined, their consistency with the written formal expressions will be maintained by applying the involved patterns again based on their history application information.

In order to investigate the performance of the approach when being applied to more complicated systems, an empirical case study is intended to be held in the future. For example, as complexity rises, the update of expressions in accordance with specification context will be more difficult and less likely to be automatically done.

Furthermore, only tool implementation could bring the proposed approach into practice and allow practitioners to benefit from the assistance expected to be provided, which is also part of our future work.

Acknowledgement. This work has been conducted as a part of "Research Initiative on Advanced Software Engineering in 2012" supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA).

References

1. Gorm, L.P., Nick, B., Miguel, F., John, F., Kenneth, L., Marcel, V.: The overture initiative integrating tools for vdm. SIGSOFT Softw. Eng. Notes **35**(1), 1–6 (2010)
2. Chen, J., Durnota, B.: Type checking classes in object-z to promote quality of specifications (1994)
3. Vadera, S., Meziane, F.: From English to formal specifications. Comput. J. **37**(9), 753–763 (1994)
4. Wang, X., Liu, S., Miao, H.: A pattern system to support refining informal ideas into formal expressions. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 662–677. Springer, Heidelberg (2010)
5. Liu, S.: Formal Engineering for Industrial Software Development. Springer, Heidelberg (2004)
6. Liu, S., Offutt, A., Ho-Stuart, C., Sun, Y., Ohba, M.: Sofl: a formal engineering methodology for industrial applications. IEEE Trans. Softw. Eng. **24**(1), 24–45 (1998)
7. http://spivey.oriel.ox.ac.uk/mike/fuzz/
8. John, F., Gorm, L.P., Shin, S.: Vdmtools: advances in support for formal modeling in vdm. SIGPLAN Not. **43**(2), 3–11 (2008)
9. Abrial, J.R.: Modelling in Event-B: System and Software Design. Cambridge University Press, Cambridge (2010)
10. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. Int. J. Softw. Tools Technol. Transf. (STTT) **12**, 447–466 (2010). doi:10.1007/s10009-010-0145-y. http://dx.doi.org/10.1007/s10009-010-0145-y (Online)
11. Owre, S., Shankar, N.: A brief overview of PVS. In: Ait Mohamed, O., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 22–27. Springer, Heidelberg (2008)
12. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: predicate subtyping in pvs. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998)
13. Tan, X., Wang, Y., Ngolah, C.: A novel type checker for software system specifications in rtpa. In: Canadian Conference on Electrical and Computer Engineering, vol. 3, May 2004, pp. 1549–1552 (2004)
14. Xavier, M., Cavalcanti, A., Sampaio, A.: Type checking circus specifications. Electr. Notes Theor. Comput. Sci. **195**, 75–93 (2008). http://dx.doi.org/10.1016/j.entcs.2007.08.027 (Online)
15. Snook, C., Butler, M.: Uml-b: Formal modeling and design aided by uml. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006). http://doi.acm.org/10.1145/1125808.1125811 (Online)
16. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)