REFINEMENT PROPAGATION
Towards Automated Construction of Visual Specifications

Irina Rychkova, Alain Wegmann
School of Communication and Computer Science, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
irina.rychkova@epfl.ch, alain.wegmann@epfl.ch

Keywords: visual specification, model transformation, stepwise refinement, refinement calculus, refinement propagation.

Abstract: Creation and transformation of visual specifications is driven by modeler's design decisions. After a design decision has been made, the modeler needs to adjust the specification to maintain its correctness. The number of adjustments might make the design process tedious for large specifications. We are interested in techniques that will reduce the modeler's obligation to control specification correctness.
Every single transformation of the visual specification can be captured by the notion of *refinement* used in formal methods. In this work we present the technique that supports a stepwise refinement of visual specifications based on calculations. We use *refinement calculus* as a logic for reasoning about refinement correctness. When a design decision is made by the modeler, the necessary adjustments are calculated based on rules of *refinement propagation*. Refinement propagation can automate the specification adjustment and enforce its correctness.

1 INTRODUCTION

It is well accepted by now that visual models play an important role in the information system development. With the growth of system complexity, automated refinement, and refinement verification of visual models is of particular interest.

Stepwise refinement is a well-known paradigm for semantic program constructions originally proposed by Dijkstra (1971) and Wirth (1971). It is based on the idea that a program can be developed through a sequence of refinement steps starting from an abstract specification.

In contrast to techniques where a refinement is first proposed and then *proved* to be correct, some techniques allow *calculation* of refinement step based on the refinement laws (Morgan and Gardiner, 1990). The refinement calculus is an underlying theory. This calculation assures refinement correctness 'by construction', and enables the reduction of proof obligations. We believe that refinement by calculation can be beneficial for the practical application in the context of visual modeling.

In this work we introduce a formal semantics for SEAM visual specifications (Wegmann, 2003) using higher-order logic and Refinement Calculus (Back and von Wright, 1998). Based on this semantics, we define a refinement propagation technique that supports a stepwise refinement of visual specifications. We constrained our discussion to the deterministic specifications. The refinement propagation technique is grounded on the following observation: *An arbitrary refinement may cause a conflict between model elements. To resolve such a conflict and maintain correctness, model adjustment (also considered as a refinement) is usually required.* When the initial refinement can be identified with a design decision that is proposed by a modeler, the adjustment of the entire specification can be calculated based on rules of *refinement propagation*. Propagation means a sequential application of these rules until saturation. We show that sufficient part of calculations can be done without modeler's involvement. We also specify the situations when modeler's decision is required to accomplish the calculation.

This paper is organized as follows. In Section 2 we present SEAM visual language and classify refinements accepted by this language. In Section 3 we define a formal semantics for SEAM using higher-order logic and refinement calculus. This formaliza-

tion allows for reasoning about visual specifications with mathematical precision. In Section 4 we introduce the refinement propagation technique. This technique is formulated as eight rules of refinement propagation. In Section 5 we discuss related works. Section 6 presents our conclusions.

2 REFINEMENT IN 'SEAM' VISUAL MODELING LANGUAGE

SEAM (Systemic Enterprise Architecture Methodology)(Wegmann, 2003) is an approach for modeling general systems, including information systems and enterprises. SEAM epistemological principles are based on General System Thinking (GST) (Weinberg, 1975) and Living Systems Theory (LST) (Miller, 1995). SEAM ontology is grounded on the second part of RM-ODP (1995) specification. Based on this standard, the main modeling concepts such as object, state, action are defined (Wegmann and Naumenko, 2001). Figure 1 illustrates the SEAM visual notation.

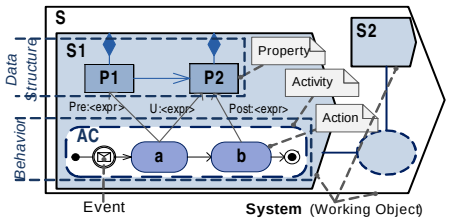

Figure 1: SEAM visual notation.

Any system or system component in SEAM is modeled as a *working object*. The working object may communicate with an environment by *events*. Working object S is modeled as a collaboration of two components $S1$ and $S2$ (also considered as working objects). $S1$ is described by its *observable properties* $P1, P2$, and a *behavior*. Properties constitute the data structure of working object $S1$ and define its state space. The behavior is represented by a set of *actions* a, b organized within *activity* AC.

We focus on the refinement of a *black box* system specification.

Refinement of the state space: (see also data refinement (Börger, 2003; Woodcock and Davies, 1996; Back, 1989) or data structure refinement (Broy, 1993)) deals with the transformation of system data structure. We recognize the following ways to refine a state space:

- Extension: new property is introduced into the system.
- Reduction: some property is eliminated.

Other refinements are used in visual modeling (e.g. substitution of one property (or group of properties) by another property (or group), property renaming, etc.). It is not difficult to show that these refinements can be represented by a combination of extensions and reductions. Refinement of the system state space is illustrated in Figure 2.

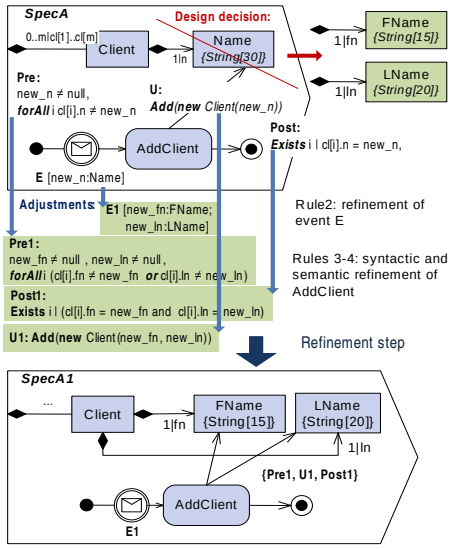

Figure 2: Propagation of refinement of the system state space. Design decision: to substitute *Name* with $\{FName, LName\}$. Required adjustments made by refinement propagation. A correct SpecA1 is calculated as a result of the refinement step.

Refinement of the behavior:

- Syntactic Refinement: a number of action's input and output parameters and their types are changed[1].
- Semantic Refinement: a precondition, an update statement, and/or a postcondition of an action are changed.
- Extension: new behavior (action or activity) is introduced into the system.
- Reduction: some behavior (action or activity) is eliminated.
- Behavior distribution: transition from an action view to an activity (Figure 3).

Aforementioned refinements specify the *basis* refinement types for SEAM visual specifications. Any arbitrary refinement can be represented as a combination of the basis refinement types. Refinement of the state space and refinement of the behavior can appear

[1]This version of behavioral refinement is also called refinement of a syntactic interface (Broy, 1993).

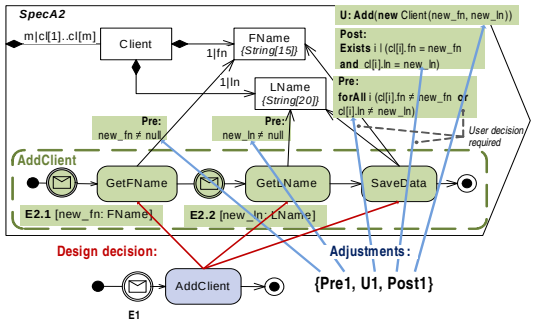

Figure 3: Propagation of behavior distribution refinement. SpecA2 refines SpecA1 (Figure 2) providing a realization of action AddClient.

together in a specification. Also the former can *imply* the latter or vice versa. This effect we call a *refinement propagation*.

3 FORMAL SEMANTICS FOR 'SEAM'

To reason about refinement with mathematical precision, we formalize SEAM modeling concepts using a higher-order logic and refinement calculus (Back, 1978; Back and von Wright, 1998).

In our work, we were inspired by the ideas presented by Mikhajlova and Sekerinski (1997), Back et al. (2000), and Michajlova (1998). In these works, refinement calculus is used for the formalization of object-oriented program development.

We find it necessary to introduce some concepts of refinement calculus in this section.

3.1 Introduction of Refinement Calculus

In this paper we restrict our study to deterministic specifications. Nondeterminism will be addressed in our future work.

A *program state* in refinement calculus is modeled as a tuple of values of all program components. A program *state space* (a type) Σ is defined as a cartesian product $\Sigma = \Sigma_1 \times \Sigma_2 \times ... \times \Sigma_n$ where $\Sigma_1, \Sigma_2, ..., \Sigma_n$ are state spaces of all program components.

A *predicate* over a state space Σ is a boolean function. The set of all predicates defined over the state space Σ is denoted by

$$\mathcal{P}\Sigma \;\hat{=}\; \Sigma \to \{true, false\} \qquad (1)$$

A *relation* between two state spaces Σ and Γ is a function that maps each state in Σ into a predicate in Γ. The set of all relations from Σ to Γ is denoted by

$$\Sigma \leftrightarrow \Gamma \;\hat{=}\; \Sigma \to \mathcal{P}\Gamma \qquad (2)$$

It is equivalent to another, more familiar definition: $\Sigma \leftrightarrow \Gamma \;\hat{=}\; (\Sigma \times \Gamma) \to \{true, false\}$

A *predicate transformer* from Σ to Γ is a function that maps each predicate in Σ into a predicate in Γ. The set of all predicate transformers from Σ to Γ is denoted by

$$\Sigma \mapsto \Gamma \;\hat{=}\; \mathcal{P}\Sigma \to \mathcal{P}\Gamma \qquad (3)$$

Predicate transformer $\langle f \rangle$ is called a *functional update*. It applies the function f to a state σ to yield a new state $f.\sigma$. For all states σ, defined by the precondition p, the functional update will produce the state $f\ \sigma$, for which the postcondition q holds:

$$\forall \sigma \in \Sigma \mid p \bullet \langle f \rangle.q.\sigma \;\hat{=}\; q\ (f\ \sigma) \qquad (4)$$

Program statements in refinement calculus are identified with predicate transformers.

For a *sequential composition* of statements S, T, and predicates p, q, r:

$$p\ \{|S;T|\}\ q \equiv (\exists\, r \bullet p\ \{|S|\}\ r \wedge r\ \{|T|\}\ q \qquad (5)$$

A *refinement ordering* on the predicate transformers is defined as follows. For $S, T : \Sigma \mapsto \Gamma$:

$$S \sqsubseteq T \;\hat{=}\; (\forall q : \mathcal{P}\Gamma \bullet S.q \subseteq T.q) \qquad (6)$$

For refinement ordering the following holds:

$$S_1 \sqsubseteq S_1' \;\wedge\; S_2 \sqsubseteq S_2' \;\Rightarrow\; (S_1 \times S_2) \sqsubseteq (S_1' \times S_2') \qquad (7)$$

$$S \sqsubseteq S' \sqsubseteq S'' \;\Rightarrow\; S \sqsubseteq S'' \qquad (8)$$

3.2 Formalization of SEAM Modeling Concepts

We distinguish between the following views of working objects in SEAM:
- Working object as a whole WO_{whole} - a black box system specification;
- Working object as a composite $WO_{composite}$ - a white box system specification.
In this paper, we focus on the black box specifications.

Working object as a whole describes the system by a number of properties $P_1 \ldots P_m$ that specify data types, and a behavior $\mathcal{B}$.
We declare the working object WO_{whole} as follows:

$$\begin{array}{l} WO_{whole} \;\hat{=} \\ \quad p_{1_1}, .. p_{n_1} : P_1, \\ \quad \ldots, \\ \quad p_{1_m}, .. p_{n_m} : P_m, \\ \quad \mathcal{B} \end{array} \qquad (9)$$

where $p_{1_i}, .. p_{n_i}$ are instances of a property P_i.
Working objects may interact with the environment by receiving inputs $I_1, .., I_k$ and sending outputs $O_1, .. O_l$, also modeled as a part of the system.

We distinguish between **primitive** and **compound properties**. The former can be considered as an alias for an operational data type (e.g. *Int, String, Boolean*, etc.); the latter is defined by a set of *component properties* and *references* to properties using property-property relations.

Example 1. *SpecA* in Figure 2 specifies a working object $SpecA_{whole}$ with its properties *Client* and *Name*. Property *Client* is compound, it has a component property *Name*. *Name* is primitive - an alias for strings of length 30. This representation unifies a declarative specification style:

"..client is identified by his/her name (30 symbols).."

and an imperative style, intrinsic to programming:

```
class Client {
   name: String[30]; } // String of length 30
m: Int;               // number of clients
cl: Array(m) of Client;// list of clients
```

□

A state of the primitive property denotes a value of the corresponding operational type (e.g. $1, "ABC", true$); a state of the compound property is defined by the states of its components and references. A tuple of property instances $p_{1_1}..p_{n_m}$, inputs $i_{1_1}..i_{n_k}$, outputs $o_{1_1}..o_{n_l}$ and their corresponding states defines a **system state** $\sigma \in \Sigma$.

Σ specifies a **system state space** - a set of all possible states of the working object:

$$\Sigma \;\hat{=}\; \Sigma_P \times \Sigma_{In} \times \Sigma_{Out} \qquad \text{where}$$

$$\Sigma_P = \underbrace{(\Sigma_{P_1} \times \ldots \times \Sigma_{P_1})}_{n_1} \times \ldots \times \underbrace{(\Sigma_{P_m} \times \ldots \times \Sigma_{P_m})}_{n_m}$$

$$\Sigma_{In} = \underbrace{(\Sigma_{I_1} \times \ldots \times \Sigma_{I_1})}_{ni_1} \times \ldots \times \underbrace{(\Sigma_{I_k} \times \ldots \times \Sigma_{I_k})}_{ni_k}$$

$$\Sigma_{Out} = \underbrace{(\Sigma_{O_1} \times \ldots \times \Sigma_{O_1})}_{no_1} \times \ldots \times \underbrace{(\Sigma_{O_l} \times \ldots \times \Sigma_{O_l})}_{no_l}$$

$\Sigma_P, \Sigma_{In}, \Sigma_{Out}$ denote state spaces of system properties, inputs, and outputs respectively.

Example2. State space of the working object *SpecA*1 in Figure 2:

$$\Sigma_{SpecA1} = \underbrace{\Sigma_{Client} \times \ldots \times \Sigma_{Client}}_{m} \times \Sigma_{In} \Rightarrow$$

$$\Sigma_{SpecA1} = \underbrace{(String[15] \times String[20]) \times \ldots \times (..)}_{m+1} \;\square$$

Behavior $\mathcal{B}$ of a working object can be seen as an action or as an activity.

Action A is defined by a three-tuple $\{Pre, U, Post\}$. Pre-condition *Pre* and post-condition *Post* define the states of the system $\sigma, \sigma' \in \Sigma$ before and after the action respectively. An update U specifies a transition from pre- state to post- state.

If this action describes a communication with the environment, the input and output events E_{In}, E_{Out} also make part of the action specification[2].

[2]Broy (1993) defines input and output channels and the sort of messages for each channel as a syntactic interface of the black box system view. The causal relationship between input messages and output messages is defined as a semantic interface of this view.

$$A \;\hat{=} \qquad (10)$$
$$E_{In}(I_1, .., I_k), \quad E_{Out}(O_1, .., O_l),$$
$$Pre : \mathcal{P}(\Sigma_P \times \Sigma_{In}),$$
$$U : (\Sigma_P \times \Sigma_{In}) \rightarrow (\Sigma_P \times \Sigma_{Out})$$
$$Post : \mathcal{P}(\Sigma_P \times \Sigma_{Out}),$$

For the deterministic specifications, we consider a functional update $\langle U \rangle : \Sigma \mapsto \Sigma$. U is a function that calculates a post-state from the pre-state. Using the definition of functional update from (4), we can write:

$$\forall \sigma \in \Sigma \mid Pre \bullet \langle U \rangle.Post.\sigma \;\hat{=}\; Post\ (U\ \sigma) \qquad (11)$$

We specify action A with its precondition *Pre*, functional update $\langle U \rangle$ and postcondition *Post* as follows:

$$Pre\ \{|A|\}\ Post \;\hat{=}$$

$$Pre\ \{|\langle U \rangle|\}\ Post \;\equiv Pre \subseteq \langle U \rangle^{-1}.Post \qquad (12)$$

Example3. *SpecA* in Figure 2 defines an action - *AddClient*:

$AddClient =$
$E_{In}(Name),$
$Pre = \exists\ new\_n : Name \wedge \forall\ cl \bullet cl.n \neq new\_n$
$U : \Sigma_{Client} \rightarrow \Sigma_{Client} = Add(newClient(new\_n))$
$Post = \exists\ cl \mid cl.n = new_n$

Pre states that there exists an input parameter *new_n* and that there is no client with the attribute $n = new\_n$ presented in the system. U denotes that a new instance of client $newClient(new\_n)$ is added into the system. The postcondition *Post* specifies the fact that after *AddClient* is carried out, the client with the attribute $n = new\_n$ does exist in the system. $E_{In}(Name)$ specifies the input event E, that transmits a parameter *new_n*. □

Activity *Ac* can be considered as a detailed specification of action *A*: it describes *how* the transition from pre- state to post- state is performed. *Ac* defines a set of component actions and the way they are composed to carry out the transition:

$$Ac \;\hat{=}\; A_1 \bigcirc A_2 \bigcirc \ldots \bigcirc A_t \qquad (13)$$

where $\bigcirc$ stands for component action ordering, defined by a corresponding action-action relation.

Precondition *Pre*, update U, and postcondition *Post* of A are related to those of component actions:

$$\begin{aligned} Pre_A &= \rho_{pre}(Pre_{A_1}, ..Pre_{A_t}); \\ U_A &= \rho_u(U_1, \ldots U_t); \\ Post_A &= \rho_{post}(Post_{A_1}, \ldots, Post_{A_t}); \end{aligned} \qquad (14)$$

$\rho_{pre}, \rho_u, \rho_{post}$ are defined by the component action ordering ($\bigcirc$).

Input and output events E_{In}, E_{Out} of the activity *Ac*, are related to the input and output events of component actions $A_1, ..A_t$:

$$\Sigma_{In/Out} \subseteq \Sigma_{In/Out_1} \times \ldots \times \Sigma_{In/Out_t} \qquad (15)$$

Example4. *SpecA2* in Figure 3 defines an activity - *AddClient* that specifies the realization of the action

AddClient from *SpecA1*, Figure 2. Parameters of an input event E are distributed between input evens $E_{2.1}$ and $E_{2.2}$ of component actions *GetFName* and *GetLName* such that:

$$E(FName, LName) \equiv \{E_{2.1}(FName),\ E_{2.2}(LName)\}$$
$$and\ \ \Sigma_{In} \equiv (\Sigma_{FName} \times \Sigma_{LName}) \quad \square$$

4 REFINEMENT PROPAGATION

In this section we introduce the refinement propagation technique for visual specifications. Exploring the basis refinement types, specified in section 2, we found relations between them in the form *"refinementX implies refinementY"* . This implication we call a *propagation of refinement.*

4.1 Definition of Refinement for Visual Specifications

Refinement calculus specifies the correct refinement between program statements as a refinement ordering on the predicate transformers. *The program statement S is said to be correctly refined by the program statement T, iff T satisfies any specification statement satisfied by S,* (6).

The *specification statement* expresses requirements program has to meet in a given state. Definition of refinement for the programm statements (or programs) can be generalized for visual specification statements (or visual specifications) as follows:

Definition 1. *Specification WO (abstract) is correctly refined by specification WO′ (concrete) $WO \sqsubseteq WO'$, if WO′ satisfies any requirement satisfied by WO.*

4.2 Propagation Rules

In refinement calculus, refinement ordering is *monotonic* with respect to the sequential statement construction. It means that if S is a statement and S_1 occurs as a substatement, $S = S(S_1)$ then

$$S_1 \sqsubseteq S_1' \ \Rightarrow S(S_1) \sqsubseteq S(S_1') \tag{16}$$

We can write WO_{whole} as a compound statement:

$$WO_{whole}(P_1, .., P_m, I_1, .., I_k, O_1, .., O_l, \mathcal{B}) \tag{17}$$

For the substatements, the following can be written:

$$P_i = P_i(.., P_j, ..),\ I_i = I_i(.., I_j, ..), O_i = (.., O_j, ..)$$

where P_i, I_i, O_i may be compound properties; and:

$$\begin{aligned} &\mathcal{B} = A(E_{In}, E_{Out}, Pre, U, Post) \quad | \quad Ac(A_1, .., A_t), \\ &E_{In} = E_{In}(.., I_i, ..), E_{Out} = E_{Out}(.., O_i, ..), \\ &Pre = Pre(.., I_i, .., P_j, ..),\ Post = Post(.., P_i, .., O_j, ..) \\ &U = U(.., I_i, .., P_j, .., O_k, ..) \end{aligned} \tag{18}$$

Properties, events, actions, and activities are related within specification. Thereby a basis refinement of an arbitrary substatement X from (17), (18) may cause a conflict between certain elements, such that

$$X \sqsubseteq X' \ \wedge\ WO_{whole}(X) \not\sqsubseteq WO_{whole}(X')$$

To resolve this conflict, and to preserve monotonicity, refinement X_1' of some other substatement X_1 is usually required. This assertion formalizes our definition of refinement propagation. Within a final number of steps, refinement propagation either results in correct refinement of specification WO_{whole}:

$$(X \sqsubseteq X') \ \Rightarrow\ (X_1 \sqsubseteq X_1') \ \ldots \Rightarrow\ (X_n \sqsubseteq X_n') \ \Rightarrow$$
$$WO_{whole}(X, X_1, .., X_n) \sqsubseteq WO_{whole}(X', X_1', .., X_n')$$

or indicates such a refinement impossible.

Resolving Conflicts between Properties

We distinguish between compound and primitive properties. The elimination of a compound property may cause 'free-floating' properties and make the specification confusing.

RULE 1: If $P_i(P_{i_1}, .., P_{i_k}, P_{i_{k+1}}, .., P_{i_l})$ is a compound property with components $P_{i_1}..P_{i_k}$ and references on properties $P_{i_{k+1}}..P_{i_l}$, and, by reduction of the state space, P_i is eliminated, then its component properties $P_{i_1}, .., P_{i_k}$ have to be also eliminated. Reduction of the state space must be correct.

$$\begin{aligned} &WO_{whole}(.., P_{i-1}, P_i, P_{i_1}, .., P_{i_l}, P_{i+1}, ..) \sqsubseteq_{-P_i} \\ &\qquad WO_{whole}(.., P_{i-1}, P_{i_{k+1}}, .., P_{i_l} P_{i+1}, ..) \end{aligned} \tag{19}$$

We formulate the correctness of *refinement of the state space*:

Definition 2. *Let WO_{whole} be a system specification as defined in* (9)*; WO'_{whole} is its refinement, where some properties have been introduced, eliminated, or substituted; and r is an abstraction relation connecting 'old' and 'new' states:*

$$r : \Sigma_{wo'} \leftrightarrow \Sigma_{wo}$$

The refinement of the state space is correct with respect to r, if any state of the abstract specification has a corresponding state (or group of states) of the concrete specification (i.e. r is a total surjective function).

The notion of r can be generalized. Considering R as a relation between predicates of the abstract and concrete specifications, we can write:

$$R : \Sigma_{wo'} \mapsto \Sigma_{wo} \tag{20}$$
$$WO \sqsubseteq_R WO' \ \hat{=}\ WO \sqsubseteq WO'; R$$

Example5. *SpecA* in Figure 2 is refined by *SpecA1* where the property *Name* is substituted with two other properties *FName* and *LName*. To maintain the refinement correctness, we have to define an abstraction relation between 'old' and 'new' data types as a total surjective function:

$$r : \Sigma_{Client'} \rightarrow \Sigma_{Client} \Leftrightarrow (\Sigma_{FName} \times \Sigma_{LName}) \rightarrow \Sigma_{Name}$$

for example:

```
1. n := substr(fn + ln, 30)
2. n := ln //n in 'old' = ln in 'new' spec
3. ...
```

considering the second expression for *n*, we define:

```
Client_old = {n:Name};
Client_new = {fn:FName, ln:LName};
r(x:FName, y:LName):Name = (Name)y;
```

This relation can be generalized as a predicate transformer:

$$R : (\Sigma_{Client'} \rightarrow \{true, false\}) \rightarrow (\Sigma_{Client} \rightarrow \{true, false\}) \Leftrightarrow (\Sigma_{FName} \times \Sigma_{LName}) \mapsto \Sigma_{Name}$$

```
pred_old(x:Name): Boolean;
pred_new(x:FName, y:LName): Boolean;
R(pred_new(x,y)) = pred_old(r(x,y));
```

This refinement step causes a conflict between refined property *Client'* and the input event E where the 'eliminated' property *Name* occurs as a parameter. Thus the event specification has to be adjusted (refined):

$$Client \sqsubseteq Client' \Rightarrow E \sqsubseteq E'$$

We generalize this observation as a propagation rule.

Resolving Conflicts between Properties and Events

RULE 2: If P' is a refinement of a property P, and $E(..,P,..) \hat{=} E(P)$ is an event, where P occurs as a parameter type, i.e. $\Sigma_P \subseteq \Sigma_E$, then the following holds:

$$P \sqsubseteq P' \Rightarrow E_{In/Out}(P) \sqsubseteq E_{In/Out}(P')$$

Example5 *(continue):* The following elaboration of specification *SpecA1* is deducible:

$$Client \sqsubseteq Client', \text{ where}$$
$$\Sigma_{Client} = \Sigma_{Name},\ \Sigma_{Client'} = \Sigma_{FName} \times \Sigma_{LName},\ \Sigma_{Name} \subseteq \Sigma_{In}$$
$$\Downarrow$$
$$E(Name) \sqsubseteq E_1(FName, LName) \quad \square$$

Resolving Conflicts between Events and Actions: Syntactic Refinement

RULE 3: If E is an input (output) event of an action A, then a refinement of E is a *syntactic refinement* of the action A (by definition of syntactic refinement). Syntactic refinement must be correct.

$$E \sqsubseteq E' \Leftrightarrow A(E) \sqsubseteq_{syn_E} A(E')$$

Definition 3. Let A' be a *syntactic refinement* of an action A, and r_{In} and r_{Out} are abstraction relations for connecting 'old' input and output parameters with the 'new' ones:

$$r_{In/Out} : \Sigma_{wo'} \rightarrow \Sigma_{In/Out}$$

where $\Sigma_{wo'}$ is a state space of the refined working object. R_{In}, R_{Out} are the corresponding predicate transformers:

$$R_{In} : (\Sigma_{wo'} \rightarrow \Sigma_{In}) \rightarrow \{true, false\} = \Sigma_{wo'} \mapsto \Sigma_{In}$$
$$R_{Out} : (\Sigma_{wo'} \rightarrow \Sigma_{Out}) \rightarrow \{true, false\} = \Sigma_{wo'} \mapsto \Sigma_{Out}$$

A is correctly refined by A' with respect to R_{In}, R_{Out} if

$$A \sqsubseteq_{R_{In},R_{Out}} A' \quad \hat{=} \quad R_{In}; A \sqsubseteq A'; R_{Out} \qquad (21)$$

R_{In} and R_{Out} are total and surjective.
For the propagation Rule 3 we write:

$$E_{In} \sqsubseteq E'_{In} \quad \Leftrightarrow \quad R_{In}; A(E) \sqsubseteq A(E') \qquad (22)$$
$$E_{Out} \sqsubseteq E'_{Out} \quad \Leftrightarrow \quad A(E) \sqsubseteq A(E'); R_{Out} \qquad (23)$$

Example6. For the action *AddClient* in Figure 2, syntactic refinement is correct if predicate transformer R_{In} is defined:

$$R_{In} : \Sigma_{SpecA1} \mapsto \Sigma_{Client} = (\Sigma_{FName} \times \Sigma_{LName}) \mapsto \Sigma_{Name}$$

The definition of R_{In} is a modeler's choice. Here we take $R_{In} \equiv R$, as it was specified in Example5.

$$E(Name) \sqsubseteq E1(FName, LName) \Rightarrow$$
$$R_{In}; AddClient(E(Name)) \sqsubseteq AddClient(E1(FName, LName))$$

$\square$

Resolving Conflicts between Events and Actions: Semantic Refinement

RULE 4: If E is an input (output) event of an action A, and E or some of its parameters I_i (O_i) occurs in *Pre*, U, or *Post* of A, then a refinement of E implies a *semantic refinement* of the action A. Semantic refinement must be correct.

$$A \mid Pre(E) \vee U(E) \vee Post(E) \Rightarrow$$
$$E \sqsubseteq E' \Rightarrow A \sqsubseteq_{sem_E} A'$$

Definition 4. Let A be an action, defined by the tuple $\{E, Pre, U, Post\}$; $A' : \{E', Pre', U', Post'\}$ is its *semantic refinement*, where Pre', $Post'$ are new pre- and postconditions, and U' is a new update. A is correctly refined by A' if:
- A' is applicable at least on every state where A is applicable;
- starting at the corresponding initial states, A' and A produce equivalent results.

Providing abstraction relation R, which relates 'old' and 'new' predicates (Definition 2), and using action specification from (12), A is said to be correctly refined by A' with respect to R if

$$R.Pre' \subseteq \langle U \rangle^{-1}.Post \ \ and \ \ Pre \subseteq (\langle U' \rangle^{-1}; R).Post' \qquad (24)$$

For the propagation Rule 4 we write:

$$E \sqsubseteq E' \Rightarrow R; A \sqsubseteq (E'/E)A; R \qquad (25)$$

where $(E'/E)A$ stands for a substitution of each occurrence of E or one of its parameters by E', or its corresponding parameter.

Example7. *SpecA1* illustrates the semantic refinement of the action *AddClient*, specified in Example3:

$$\begin{aligned} &AddClient' = \\ &\quad E_1(FName, LName), \\ &\quad Pre_1 = \exists\, new\_fn : FName, new\_ln : LName \wedge \\ &\qquad \forall\, cl \bullet (cl.fn \neq new\_fn \vee cl.ln \neq new\_ln) \\ &\quad U_1 : \Sigma_{Client} \rightarrow \Sigma_{Client} = Add(newClient(new\_fn, new\_ln)) \\ &\quad Post_1 = \exists\, cl \mid (cl.fn = new\_fn \wedge cl.ln = new\_ln) \end{aligned}$$

New precondition Pre_1 and postcondition $Post_1$ have been calculated by propagation:

$$E(Name) \sqsubseteq E_1(FName, LName) \Rightarrow \text{(rule 4)}$$
$$AddClient(E) \sqsubseteq_{sem_E} AddClient(E_1)$$
$$Pre = Pre(Name), Post = Post(Name) \Rightarrow \text{(rule 5)}$$
$$AddClient(Pre, Post) \sqsubseteq_{sem_E} AddClient(Pre_1, Post_1), \text{ and}$$
$$Pre_1 = \exists\, new\_fn : FName,\ new\_ln : LName \wedge$$
$$\forall\, cl \bullet (cl.fn \neq new\_fn \vee cl.ln \neq new\_ln)$$
$$Post_1 = \exists\, cl \mid (cl.fn = new\_fn \wedge cl.ln = new\_ln)$$

This refinement is correct by Definition 4. For the sake of brevity, the proof of correctness is omitted. The specification adjustment is finished and the refinement propagation is complete.

$$(Client \sqsubseteq Client') \Rightarrow (E \sqsubseteq E_1) \Rightarrow (AddCl.. \sqsubseteq AddCl..') \Rightarrow$$
$$SpecA(Name, E, AddClient) \sqsubseteq$$
$$SpecA1(\{FName, LName\}, E_1, AddClient') \quad \square$$

Resolving Conflicts between Properties and Actions: Semantic Refinement

RULE 5: If property P occurs in Pre, U, or $Post$ of action A, then a refinement of P implies a *semantic refinement* of A. Semantic refinement must be correct.

$$A \mid Pre(P) \vee U(P) \vee Post(P) \Rightarrow$$
$$P \sqsubseteq P' \Rightarrow A \sqsubseteq_{sem_P} A'$$

By definition of correct semantic refinement (Definition 4), we write:

$$P \sqsubseteq P' \Rightarrow R; A \sqsubseteq (P'/P)A; R \qquad (26)$$

where $(P'/P)A$ stands for a substitution of each occurrence of P by P' in the statements of A, and R is an abstraction relation.

Resolving Conflicts between Events and the System State Space

RULE 6: If E is an input (output) event of working object WO_{whole} then an extension of E by some parameter of type T implies an *extension of the system state space* by introduction of a property T.

$$E(..) \sqsubseteq E(.., T, ..) \Rightarrow$$
$$WO_{whole} \sqsubseteq_{+} WO'_{whole} \quad \wedge \quad \Sigma_{wo'} = \Sigma_{wo} \times \Sigma_T \quad (27)$$

Note: If $\Sigma_{E'} \subseteq \Sigma_E$ (a reduction) - there is no conflict with the system state space.

Resolving Conflicts between Pre-, Post-conditions, and Updates

A *semantic refinement* takes place when pre-, post-condition, or update statement of an action is changed. The equation (12) relates pre-, post-condition, and update. Precondition *Pre* can be calculated as a *weakest* precondition $wp(A, Post)$ that guarantees termination of any execution of A ($\langle U \rangle$) in the final state that satisfies *Post*. Postcondition *Post* can be calculated as a strongest postcondition $sp(A, Pre)$ respectively. An update U cannot be resolved by calculation and requires a modeler's decision.

Conflicts between pre-, post-conditions, and updates indicate that semantic refinement is *incorrect*. Correctness of semantic refinement is formulated in Definition 4.

Resolving Conflicts between an Activity and its Component Actions

Behavior distribution refinement stands for an activity definition (Figure 3). This requires a set of component actions and action ordering provided by modeler.

RULE 7: If an activity *Ac* is a behavioral distribution refinement of an action A, and $A_1, ..A_t$ - are component actions, see (13), then:

1. Events E_{In_i} and E_{Out_i} of component actions are defined based on the modeler's decision, providing (15) holds;
2. Preconditions Pre_i of component actions are defined either by propagation or based on modeler's decision, providing ρ_{pre} for the given action ordering (14) is defined;
3. Updates U_i and/or postconditions $Post_i$ are defined based on modelers decision, providing ρ_u, ρ_{post} for the given invocation order (14) are defined, and (12) holds for each component action;

Example8. Figure 3 illustrates the propagation of a *behavior distribution refinement*. Component actions *GetFName*, *GetLName*, and *SaveData*, their ordering, and set of input events are provided by a modeler. By refinement propagation, we define the preconditions for *AddFName* and *AddLName* component actions. Specification of the postcondition is the modeler's decision (here *SaveData*). $\square$

Without loss of generality, any action in SEAM specification can be seen as a component action of some abstract activity Ac_{parent}.[3] Thus, the refinement of the behavior by an *extension* is considered as an introduction a new component to this abstract activity, whereas a *reduction* stands for elimination of some component from it. Rule 6, applied for the Ac_{parent}, specifies the propagation of these refinements.

Both *extension* and *reduction* of the system's behavior have to preserve the semantics of the rest of the specification. Put it in other terms, modeler needs to guarantee that the system will work 'at least as well as before' in presence of new actions/activities or after removing any of them. This follows from the Definition 1.

[3] System *life cycle* is the most abstract activity system performs from the moment of putting in operation (startup) till the end of functioning (shutdown). Any action or activity is a part of the life cycle.

Incorrectness

RULE 8: Refinement propagation is impossible if the correctness of initial refinement is not provable.

Summary

In this section we provide definitions for the refinement correctness and formulate eight rules of refinement propagation. Refinement steps often represent combinations of refinements. For such a combination (7) and (8) hold.

We demonstrate that a significant part of specification elaboration can be done by calculation. We also specify the situations, when intervention of the modeler is necessary. Formal proof of soundness and completeness of the refinement propagation technique is an important issue. This makes a topic of our current research.

For the sake of brevity, some technical details have been omitted. For more explanations, please, contact the authors.

5 RELATED WORK

The foundations in mathematical logic are extensively used to formalize specifications and refinement techniques for program constructions. Woodcock and Davies (1996) present the method of software specification development called Z. Börger and Stärk (2003) introduce the Abstract State Machine method of abstract refinable system specifications. Refinement formalization for object-oriented programs using refinement calculus is presented by Mikhajlova and Sekerinski (1997), Back et al. (2000), and Michajlova (1998). Back(2005) proposes a method of incremental software construction using refinement diagrams. Here refinement calculus is used as a logic for reasoning about software systems and their evolution.

In the domain of visual languages, evolutionary specifications of ADORA are provided with refinement calculus semantic (Xia and Glinz, 2004). The transition between model views requires a representation consistency that is guaranteed by the application of refinement calculus. Muskens, in (Muskens et al., 2005), focuses on the problem of consistency checking between software views, expressed as UML diagrams. The approach in (Muskens et al., 2005) is based on verification of obligation and constraint rules using relation partition algebra. In contrast to these approaches, refinement propagation technique for SEAM black box specifications focuses on preservation of semantic correctness for visual specifications.

Baar and Marcović (2006) introduce a proof technique for the semantic preservation of refactoring rules for UML class diagrams and OCL constraints. Refactoring can be considered as a specific form of refinement. We believe that the refinement propagation technique can be equally used to support automated refactoring of SEAM specifications. In our work, action constraints in SEAM (pre-, post- conditions, and updates) are specified using some meta-language. Alternatively, they can be expressed in OCL.

Pons (2006) presents the OCL-based technique and a tool support for UML and OCL model refinement. Object-Z is an underlying theory for refinement verification. The authors discuss the refinement patterns and formulate the refinement conditions for these patterns in OCL language(OCL, 2003). Similarly, our technique considers several standard refinement types that can be identified with patterns. In addition, we define conflict situations, caused by these refinement types and explore the idea of refinement propagation.

6 CONCLUSION

In this work we formalize the notion of refinement and its correctness for visual specifications using refinement calculus (Back and von Wright, 1998). Based on this formalization, we define the refinement propagation technique for semiautomated specification construction. Initiated by modeler's design decision, correct refinement of the visual specification can be:

- automatically calculated, based on refinement propagation,
- calculated based on supplementary information from the modeler,
- recognized as impossible to calculate.

Eight rules of refinement propagation address possible conflicts between SEAM specification elements, caused by refinements. Figure 4 summaries the applicability of our technique. The leftmost column enumerates the refinements. Each row denotes conflicts between elements, caused by the respective refinement, and solutions for these conflicts. For example,

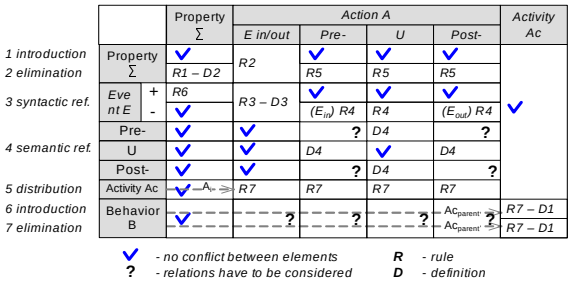

Figure 4: Refinement propagation rules. Summary.

an introduction of a property (row 1) causes no conflict with action preconditions (column 3), whereas property elimination (row 2) may cause such a conflict (column 3). This conflict can be resolved applying a propagation rule 5.

Refinements may cause conflicts by breaking *property-property and action-action relations* in SEAM (Figure 1). Formalization, refinement, and refinement propagation for these relations is out of the scope of this paper. We put a question mark in the summary table to specify these cases.

We consider the refinement propagation technique as an efficient step towards computer-aided construction of visual specifications. Application of this technique in the form of a modeling tool is one of our on-going projects.

REFERENCES

Baar, T. and Marković, S. (2006). A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *Ershov Memorial Conference*, Lecture Notes in Computer Science.

Back, R.-J. (1978). *On the Correctness of Refinement Steps in Program Development*. PhD thesis, bo Akademi, Department of Computer Science, Helsinki, Finland. Report A–1978–4.

Back, R.-J. (1989). Changing data representation in the refinement calculus. In *22nd Hawaii International Conference on System Sciences*, pages 231–242. IEEE.

Back, R.-J. (2005). Incremental software construction with refinement diagrams. In Broy, Gunbauer, H. and Hoare, editors, *Engineering Theories of Software Intensive Systems*, NATO Science Series II: Mathematics, Physics and Chemistry, pages 3–46. Springer, Marktoberdorf, Germany.

Back, R.-J., Mikhajlova, A., and von Wright, J. (2000). Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12(1):18–40.

Back, R.-J. and von Wright, J. (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag. Graduate Texts in Computer Science.

Börger, E. (2003). The asm refinement method. *Formal aspects of computing*.

Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

Broy, M. (1993). Interaction refinement—the easy way. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag.

Dijkstra, E. W. (1971). Notes on structured programming. In *Structured Programming*. Academic Press.

Mikhajlova, A. (1998). Consistent extension of components in presence of explicit invariants. In *Workshop on Component-Oriented Programming (WCOP'98), ECOOP'98*. TUCS General Publication Series.

Mikhajlova, A. and Sekerinski, E. (1997). Class refinement and interface refinement in object-oriented programs. In *FME '97: Industrial Applications and Stengthened Foundations of Formal Metohds*, volume 1313, pages 82–101. Springer.

Miller, J. (1995). *Living Systems*. University of Colorado Press.

Morgan, C. and Gardiner, P. H. B. (1990). Data refinement by calculation. *Acta Informatica*, 27(6):481–503.

Muskens, J., Bril, R. J., and Chaudron, M. R. V. (2005). Generalizing consistency checking between software views. In *WICSA*, pages 169–180.

OCL (2003). *OCL 2.0 Final Adopted Specification*. OMG.

Pons, C. (2006). Heuristics on the definition of UML refinement patterns. In *SOFSEM*, pages 461–470.

RM-ODP (1995). *Reference model of open distributed processing part 1. Draft International Standard (DIS)*. Helsinki, Finland.

UML (2007). *Unified Modeling Language (UML), version 2.1.1*. OMG, www.omg.org.

Wegmann, A. (2003). On the systemic enterprise architecture methodology (seam). In *International Conference on Enterprise Information Systems (ICEIS)*.

Wegmann, A. and Naumenko, A. (2001). Conceptual modeling of complex systems using an rm-odp based ontology. In *EDOC*, pages 200–211.

Weinberg, G. M. (1975). *An Introduction to General Systems Thinking*. New York: Wiley & Sons.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14:221–227.

Woodcock, J. and Davies, J. (1996). *Using Z*. Prentice Hall.

Xia, Y. and Glinz, M. (2004). Extending a graphic modeling language to support partial and evolutionary specification. *apsec*, 00:18–27.