# Переписування Семантики Історій Користувача

Сьцібор Собіескі, Бартош Зієлінскі, Павел Масьлянка
Кафедра комп'ютерних наук,
факультет фізики та прикладної інформатики,
Університет Лодзь,
Лодзь, Польща
{scibor.sobieski, bzielinski, pmaslan}@uni.lodz.pl

# Rewriting Semantics of User Stories

Ścibor Sobieski, Bartosz Zieliński, Paweł Maślanka
Department of Computer Science,
Faculty of Physics and Applied Informatics,
University of Łódź,
Łódź, Poland
{scibor.sobieski, bzielinski, pmaslan}@uni.lodz.pl

***Анотація*—Ми описуємо виконувану семантику, засновану на мультизмісній перезапису для користувацьких історій - популярного формату для опису вимог користувача. Семантика зосереджена на захопленні основних операцій (CRUD) з передачею даних та потоків даних між учасниками, що беруть участь у розповідях, і абстракції від складних операцій як незрозумілі терміни (кодування залежностей даних). Семантика достатня для аналізу доступності. Переклад історій користувача, доповнений моделлю даних у систему перезапису, досі є посібником, але ця робота є основою для майбутнього компілятора історії користувачів.**

***Abstract*—We describe an executable semantics based on multiset rewriting for user stories — a popular format for describing user requirements. The semantics focuses on capturing the basic (CRUD) operations on data and data flow between actors involved in the stories and abstracts away complex operations as uninterpreted terms (encoding data dependency). The semantics is sufficient for reachability analysis. Translation of user stories augmented with data model into a rewriting system is manual so far, but but the present work is a foundation for a future user story compiler.**

***Ключові слова—користувацькі історії; багатозаписний перепис; виконувана семантика; вимоги техніки;***

***Keywords—user stories; multiset rewriting; executable semantics; requirements engineering;***

## I. Introduction

The two established formats for describing user requirements — *user stories* and *use cases* — are intended mainly for human consumption. They are written in natural (though usually constrained) language, easily understandable by non-programmer stakeholders. While specifications in the form of user stories or use cases are not necessarily machine unreadable (cf. [1]), automated verification and testing requires some kind of formal semantics.

In this paper, we attempt to provide an executable semantics for user stories (in the Csee format [2]). More precisely, we associate to a collection of stories augmented with a data model a multiset rewriting system, representable as a coloured Petri Nets with inhibitor edges and ability to generate fresh nominal values (cf. [3]). Currently the association is done manually, however, we are working on a ``compiler'' for user stories (written in a constrained English parsable without natural language processing tools, cf. [4]).

Creating the executable semantics we focused on capturing the basic (CRUD) operations on data and data flow between actors involved in the stories. The result bears similarity to specification of artifact-centric business process (see e.g., [5], [6]). Complex operations are abstracted away as uninterpreted terms (encoding data dependency). The semantics is detailed enough to perform reachability analysis to prove that desirable final states of a case are attainable through legal user actions, and undesirable data transformations and accesses are not.

When presenting the semantics, we decided to dispense with Petri Net pictures which would be too complex and less susceptible to piecewise introduction. Instead we use actual multiset rewriting rules. Note that such rules are directly translatable into rewriting language and system Maude [7] often used to express and test formal specifications (see e.g., [8], [9], [10] and also to simulate a variety of Petri Nets [11], [12].

## II. Facts and multiset rewriting

### A. Values

All values we use are typed. Types can be divided into abstract and concrete ones. Concrete ones contain defined operations and relations (usually through equations). Abstract types may contain constructors, but no defined operators and relations. We allow matching (e.g., in rewrite rules) only on

fully reduced terms which may contain constants, variables and constructors but no defined operators. Thus, e.g., in case of natural numbers $\mathbb{N}$ (a concrete type) we can match on 0, *s(m)*, *s(s(m))* or 3:=*s(s(s(0)))*, where *m*: $\mathbb{N}$ is a variable, and *s*: $\mathbb{N} \to \mathbb{N}$ and 0: $\to \mathbb{N}$ are natural numbers constructors (successor and zero, respectively), but we cannot match on *m*+*n* as "+" is a defined operator.

Abstract types are further divided into nominal ones (for which the only constructors are constants) and non-nominal ones (with non-constant constructors). Nominal types serve as identifiers of objects, and for each nominal type we can generate fresh values of this type. Non-nominal abstract types serve as representations of computed values where we want to keep the computation abstract.

### B. Facts

Facts are instances of base predicates (one can think of them as rows in a database table, if each row stored also a table name). A fact $P(a_1, \ldots, a_n)$ consists of a predicate name $P$ and a list of values $a_1, \ldots, a_n$ (fully reduced terms serving as predicate arguments). Non-ground facts may contain variables among (or inside) arguments $a_i$.

A predicate signature is a tuple of types. We write $P{:}T_1, \ldots, T_n$ if $P$ is a predicate name with signature $T_1, \ldots, T_n$. In this case all facts $P(a_1, \ldots, a_m)$ with $P$ as predicate name are such that $m=n$ and $a_i{:}T_i$, $i \in \{1, \ldots, n\}$.

### C. Multisets of facts

We represent state of the system as a multiset of ground facts. Changes to this state are represented by multiset rewrites. We assume usual multiset operations $\Gamma \cup \Delta$ (union) $\Gamma \cap \Delta$ (intersection), $\Gamma - \Delta$ (difference). $\varnothing$ is the empty multiset. When writing rules we often identify facts with single-element multisets and consider ``," (comma) to be an associative and commutative multiset constructor corresponding to multiset union. We denote by Var(Γ) the set of variables occurring in the multiset Γ.

Multisets of facts, apart from representing state of the system may also appear as arguments of abstract aggregate functions.

### D. Domain conditions

A domain condition is a quantifier free, first order formula which can refer to domain predicates (such as ordering on numbers) but not to base predicates stored in the multiset database.

### E. Rewrite rules

A rewrite rule $\lambda=(p, C, \Gamma, \Delta, \Delta')$ consists of term p of type Person denoting agent of the action, domain condition $C$ and three (non-ground) multisets Γ, Δ, and Δ' such that $\mathrm{Var}(C) \cup \mathrm{Var}(\Gamma) \cup \mathrm{Var}(p) \subseteq \mathrm{Var}(\Delta)$ and any $x \in Var(\Delta') - Var(\Delta)$ is of nominal type.

We say that a ground multiset of facts Ψ rewrites to a ground multiset Ψ' with rule λ, agent a, and ground substitution σ (i.e., $\sigma(x)$ is a ground term for all $x \in Var(\Delta') \cup Var(\Delta)$, which we denote by $\Psi \to_{\lambda,\sigma,a} \Psi'$, if and only if

- $\sigma(C)$ is true,
- $\sigma(p)=a$,
- $\sigma(\Gamma) \subseteq \Psi$ and $\sigma(\Delta) \subseteq \Psi$,
- $\Psi' = (\Psi - \sigma(\Delta)) \cup \sigma(\Delta')$,
- for all $x \in Var(\Delta') - Var(\Delta)$, $\sigma(x)$ is fresh.

Let $\Psi_0 \to_{\lambda 0,\sigma 0,a0} \Psi_1 \to_{\lambda 1,\sigma 1,a1 \ldots} \to_{\lambda i,\sigma i,ai} \Psi_{i+1}$, where $\Psi_0$ is some initial multiset of facts. A value $c$ of nominal type is fresh at $i \geq 0$ if it did not occur in $\Psi_0 \cup \ldots \cup \Psi_{i-1}$.

To improve readability we use an alternative syntax for rules. Namely, we write

$$C; \neg\boldsymbol{\alpha}, \boldsymbol{\beta}; \boldsymbol{\gamma} \Rightarrow_p \boldsymbol{\delta},$$

where $\neg\boldsymbol{\alpha} = \neg\alpha_1, \ldots, \neg\alpha_n$, $\boldsymbol{\beta} = \beta_1, \ldots, \beta_m$, etc., and $\alpha_i$'s, etc., are facts, to denote the rule

$$(p, C, \boldsymbol{\alpha}, \{\boldsymbol{\beta}\} \cup \{\boldsymbol{\gamma}\}, \{\boldsymbol{\beta}\} \cup \{\boldsymbol{\delta}\}).$$

### F. Rewrite systems

A rewrite system R is a set of rewrite rules. We write $\Psi \to_{R,a} \Psi'$ if there exists some $\lambda \in R$ and a ground substitution σ such that $\Psi \to_{\lambda,\sigma,a} \Psi'$.

## III. From User Stories to Rewriting System

In this section we show how to construct from a colection of user stories augmented with data model a multiset rewriting system. We will use as a running example an actual specification of a system supporting national selection of candidates for study programmes at the universities in New Guinea. Because of pecular conditions of life in New Guinea, candidates do not apply to a particular university. Instead, they submit their applications to the Department of Higher Education, Research, Science and Technology (DHERST). Higher education institutions (HEI) submit information about study programmes. Then, during national selection, candidates are matched with programmes and universities based on plethora of factors, which include student abilities, level of education and personal preferences.

### A. From entities to predicates

As remarked above, our rewriting semantics does not attempt to capture all of requirements contained in user stories. Instead, it focues mostly on CRUD operations on data. Thus, we need to augment the user stories with data model of business entities referred to in the stories, perhaps partially extracted from stories themselves using techniques described in [1]. If entities are modeled using E/R technique, then the E/R model can be converted to definitions of predicates using usual techniques for transforming E/R model into relational schema. That is, key and obligatory single-valued attributes of an entity

or relationship become a arguments of a predicate with the same name as the entity/relationship, and nullable and many-valued attributes give rise to separate predicates with the same name as the corresponding attribute. As an example consider the entities corresponding to HEI and the list of HEI:

entity HEI(**id_T** id, **string** name, **string** street, **person** chief),

entity HEIList(**id_T** id, **year** year, **id_T multival** hei).

In the second entity, the year attribute says for which year the list was composed. The first entitity gives rise to a single predicate

HEI : **id_T, string, string , person**

The second entity gives rise to two predicates, the second of which corresponds to the multivalued attribute hei containing identifiers of higher education institutions taking part in national selection:

HEIList : **id_T, year,** hei : **id_T, id_T**.

Thus, hei(x,y) means that a higer education institution x belongs to the list y.

*B. User stories and rewriting rules*

Each user story gives rise to one or more rewriting rules. We assume that each story corresponds either to some CRUD operation (creation or update of entities) or to using data to compute or view something. The latter is abstracted as creation of special facts, e.g., of the form view(x, y), i.e., person x viewed entity with identifier y.

*C. Tokens*

Tokens are special facts. While the usual facts encode business relevant data, tokens encode information related to control flow. Names of tokens start with "#". The basic token is argument-less and named #next. This token symbolizes the choice of the next story. If the user story is implemented with more than one rewriting rule, the story's "entry" rule replaces #next with story-specific token. The rule ending execution of a given story recreates the #next token. This token can be thought of as corresponding to the non-deterministic choice of an applicable story, and sometimes of value from the active domain. "Internal" tokens of the given story often have a natural argument corresponding to the maximal re-creation count to prevent infinite executions, as well as other arguments which store values which must be the same across all rewrites within execution of the same user story, such as the agent.

*D. Example rules*

Consider the following user story:

As a DHERST admin I would like to **create** list of HEI if it doesn't exist for the current year *that would take part in National Selections*.

The italicized part above is an explanation for humans. What is important for our semantics is the agency description "As a DHERST admin), and main verb "create" followed by object "list of HEI for the given year" and additional condition "if it doesn't exist". Thus, from the point of view of our semantics, the story describes creation of the new HEIList entity by a person assigned role of DHERST admin. The "current year being x" and "x having the role DHERST admin" is described by predicates with obvious names. The list is created empty (default interpretation of creation is that many-valued and optional attributes are unassigned). Thus, the semantics of the above story is captured by the following single rule (x, y, z, p are variables):

true; ¬HEIList(x, y), DHERSTAdmin(p), year(y) ; #next $\Rightarrow_p$ HEIList(z, y), #next.

Above, z is a variable not bound by the left hand side of the rule, and thus it will be assigned a fresh identifier of the HEIList entity when this rule is executed. Note that the rule consumes the #next token but then it recreates it. Let us now consider the example of a story which compiles into multiple rules:

As a DHERST admin I want to **update** list of HEI for the current year.

Here the main verb is "update". In this context it means deleting some HEI from the list and adding new ones. We assume that the information about a given higher education institution already exists in the system, so "adding a new one" means picking the HEI entity and adding it to the list. We utilize additional token predicate #upd : ℕ, Person, id_T. The token #upd(n, p, l) stores an agent (with role DHERST admin) who is the subject of this story, and the identifier l of the list of HEI which is being updated, so that all the rules update the same list. The number in the first argument limits the number of possible recreations of the token so that a given user story execution makes a limited number of elementary update operations and we avoid infinite loops when simulating. The first rule simply picks up the agent and the list to be updated and stores them in the token:

true; DHERSTAdmin(p), HEIList(x,y); #next $\Rightarrow_p$ #upd(N, p, x).

In the above rule, p, x, y are variables, however N is not a variable but some fixed number describing a maximal number of recreations of the #upd token. Observe that the #next token is not recreated. This ensures rewritings corresponding to different user stories cannot occur until this story is finished.

The next two rules add an existing HEI to the list (if it is not there already, or deletes some HEI from the list. Note that both rules require the #upd token with first argument greater than 0 (which is ensured by matching with s(0)) and that both rules recreate the #upd token with first argument decreased by 1:

true; ¬hei(h, x), HEI(h, n, a, c); #upd(s(m), p, x) $\Rightarrow_p$ hei(h, x), #upd(m, p, x).

true; ; hei(h, x), #upd(s(m), p, x) $\Rightarrow_p$ #upd(m, p, x).

Note that instead of matching against s(m) we could have instead use the condition part of the rule, e.g.,

m > 0; ; hei(h, x), #upd(m, p, x) $\Rightarrow_p$ #upd(m-1, p, x).

The last rule serves to non-deterministically finish execution of the present story by recreation of the #next token. Observe that because we do not limit variable m, the story ends after no more, but possibly less then N elementary update steps, where N was the first argument of the #upd token created in the first rewrite rule:

$$\text{true};;\#\text{upd}(m,p,x) \Rightarrow_p \#\text{next}.$$

In the last example of converting user story to rewrite rules we show how to abstract non-trivial computation. The story we want to implement is as follows:

As a DHERST admin I can run Gale Shapley algorithm to find best matching of available candidates and programs.

Clearly, we do not want to model or specify Gale Shapley algorithm, as it is out of the scope of our executable semantics. However, we can model the associated data flow. To make the presentation more readable in the small space available, we simplify significantly the data model. Namely, we suppose that we have the following two two-argument predicates:

$$\text{Cand} : \text{Cand\_T}, \qquad \text{Prog} : \text{Prog\_T}.$$

We assume that elements of types Cand_T and Prog_T store all the data about candidates and study programmes, respectively. We also assume that we have constructor ga-sh : FactMultiset → Matching, and a predicate match : Matching, where Matching is the abstract type corresponding to the results of Gale-Shapley algorithm, and FactMultiset is a type of multisets of facts used when defining abstract aggregate functions such as ga-sh. The story is implemented with the following four rules:

$$\text{true; DHERSTAdmin(p); \#next} \Rightarrow_p \#\text{dh(p), match(ga-sh}(\emptyset))$$

$$\text{true; ; \#dh(p), Cand(c), match(ga-sh(S))} \Rightarrow_p \#\text{dh(p), match(ga-sh}(S \cup \{ \text{Cand(c)} \})),$$

$$\text{true; ; \#dh(p), Prog(c), match(ga-sh(S))} \Rightarrow_p \#\text{dh(p), match(ga-sh}(S \cup \{ \text{Prog(c)} \})),$$

$$\text{true; } \neg \text{Cand(c)}, \neg \text{Prog(c')}, \text{match(ga-sh(S)) ; \#dh(p)} \Rightarrow_p \#\text{next, S}$$

The first one picks DHERST admin and stores the choice in a token, as well as creates an empty matching term. The next two gather candidates and study programs, the last one finishes story execution when all candidates and programs have been aggregated. Note that it recreates facts about candidates and programs temporarily removed from the multiset database.

## Conclusion

We outlined how to assign an executable semantics to a collection of user stories. Executable semantics we describe is based on multiset term rewriting, and it tries to capture the basic CRUD operations. Currently the assignement is done manually, but we are working on a user story compiler.

## References


[1] M. Landhausser, A. Genaid, "Connecting user stories and code for test development," in Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop, pages 33-37, June 2012.

[2] Connextra. Connextrastorycard.

[3] M. Montali, A. Rivkin. "Model checking petri nets with names using data-centric dynamic systems," Formal Aspects of Computing, 28(4):615-641, 2016.

[4] Ś. Sobieski, B. Zieliński, "User stories and parameterized role based access control," in Model and Data Engineering, pp. 311-319, Cham, 2015. Springer International Publishing.

[5] R. Hull, "Artifact-centric business process models: Brief survey of research, results and challenges," in On the Move to Meaningful Internet Systems: OTM 2008, pp 1152-1163, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] P.A. Abdulla, C. Aiswarya, M.F. Atig, M. Montali, O. Rezine. "Recency-bounded verification of dynamic database-driven systems," in Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '16, pp.195-210, New York, NY, USA, 2016. ACM.

[7] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. "The Maude 2.0 system," in Rewriting Techniques and Applications (RTA 2003), LNCS pp. 76-87. Springer-Verlag, June 2003.

[8] G. Denker, J. Meseguer, C. Talcott, "Protocol specification and analysis in Maude," in Proc. of Workshop on Formal Methods and Security Protocols, 1998.

[9] Ś, Sobieski, B. Zieliński, "Using Maude rewriting system to modularize and extend SQL," in Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 853-858. ACM, 2013.

[10] Ś, Sobieski, B. Zieliński, "Modularisation in Maude of parametrized RBAC for row level access control," in Advances in Databases and Information Systems, pp. 401-414. Springer, 2011.

[11] M.O. Stehr, J. Meseguer, P.C. Olveczky. "Rewriting logic as a unifying framework for Petri nets," in Unifying Petri Nets: Advances in Petri Nets}, pp. 250-303, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] J. Padberg. A. Schulz, "Model checking reconfigurable Petri nets with Maude," in Graph Transformation, pp. 54-70, Cham, 2016. Springer International Publishing.