A Constraint Solver for Sequences and its Applications

Nikolai Kosmatov
INRIA Lorraine
615 rue du Jardin Botanique
54600 Villers-les-Nancy France
kosmatov@lifc.univ-fcomte.fr

ABSTRACT

Constraint programming techniques are successfully used in various areas of software engineering for industry, commerce, transport, finance etc. Constraint solvers for different data types are applied in validation and verification of programs containing data elements of these types. A general constraint solver for *sequences* is necessary to take into account this data type in the existing validation and verification tools. In this work, we present an original constraint solver for sequences implemented in CHR and based on T. Frühwirth's solver for lists with the propagation of two constraints: generalized concatenation and size. The applications of the solver (with the validation and verification tool BZTT) to different software engineering problems are illustrated by the example of a waiting room model.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic**]: Logic and constraint programming; D.2.4 [**Software/Program Verification**]: Validation

General Terms

Algorithms, verification

Keywords

Constraint solver, sequences, validation, verification

1. INTRODUCTION

Research work of the last ten years has shown the effectiveness and fruitfulness of constraint logic programming in different areas of software engineering. The development of constraint solvers for some data types has already permitted the symbolic evaluation of formal models containing data elements of these types. For example, the tool BZTT [5] uses the solver CLPS-B [4] and allows the animation and test generation for formal models containing integers, sets, functions and relations. BZTT was used to validate and verify software in different industrial projects for industry, transport, commerce and finance [4] (see also references in [4]), e.g. for PSA Peugeot Citroën, Schlumberger, Thales. The model evaluated by BZTT is written in a logic notation with sets like B [1] or Z [15]. Sequences are one of the data types used in these notations and representing finite lists of elements such as stacks, queues, communication channels, sequences of transitions or any other data with consecutive access to elements.

Nowadays there exists no validation and verification tool with an integrated constraint solver for sequences. The main motivation of this work is to develop such a solver, which is necessary to take into account sequences during the validation and verification of software. This solver must treat all different operations on sequences. It will be integrated into the existing constraint solvers for other data types such as CLPS-B [4] and used in validation and verification tools such as BZTT [5].

The problem of constraint solving for sequences is very close to that of words or lists. The fundamental result of Makanin [14] shows that the satisfiability of word equations (where the concatenation is the unique operation) is decidable. Kościelski and Pacholski [11] showed that for a given constant $c > 2$ the problem of the existence of a solution of length $\leq cd$ for an equation of length d is NP-complete. Therefore there does not exist any fast algorithm for word equations in the general case. The decidability of the existential theories of words is close to the borderline of decidability. Durnev [7] showed that the positive $\forall\exists^3$-theory of concatenation is unsolvable. The decidability of word equations with an additional equal-length predicate is still an open problem. We refer the reader to [2, 10] for more detail on the word equations.

The general constraint solving problem for sequences is even more complicated than that for words, because sequences generalize words and are usually considered with more operations. Therefore it is impossible to provide a general and efficient constraint solver for sequences terminating for all constraint problems. Nevertheless, even a partial constraint solving technique of reasonable complexity would be extremely useful for applications.

We know very few results in the general context. Prolog III [6] implements concatenation and size for lists. A representation of sequences by PQR trees is proposed in [3], but this approach is limited to bijective sequences (i.e. permutations of a given finite set E).

Our work aims to study the problem of constraint solving for sequences from the practical point of view. Since we cannot develop an efficient solver for any constraint problem with sequences, it is important to provide at least a partial constraint solving technique for the problems which appear in practice. This paper presents a constraint solving technique which implements 11 basic operations on sequences. It was developed in *Constraint Handling Rules* (CHR) [8] in SICStus Prolog. The CHR allow a very clear and easily modifiable implementation. The complete solver can be consulted and executed from the author's webpage [12], and the essential rules are given in the paper. An application of the solver to model animation and validation shows that even this partial constraint solving technique can be successfully applied in practice.

The paper is organized as follows. In Section 2 we define sequences and operations on sequences appearing in the constraints. Section 3 describes the constraint solver for sequences. The applications of the solver to software validation and verification are illustrated by the specification of a waiting-room system in Section 4. We conclude and present the future work in Section 5.

2. SEQUENCES AND CONSTRAINTS

Definition. Let E be a set. *A sequence* over E is a finite list of elements of E. *The size (length)* of a sequence S is the number of elements of S. The empty sequence (of size 0) is denoted by $[\,]$.

Example. Let $E = \{1,2,3\}$, $S_1 = [\,]$, $S_2 = [3]$, $S_3 = [1,2,3]$, $S_4 = [2,3,1]$, $S_5 = [1,1,2,1]$. Then the S_i are sequences over E.

Let us recall the usual operations on sequences which are used, for example, in the formal notations B [1] and Z [15]. We give in brackets an example with the notation commonly used in B.

1. *the first element* $(\text{first}\,[1,2,2,2,3] = 1)$;
2. *the last element* $(\text{last}\,[1,2,2,2,3] = 3)$;
3. *the front* $(\text{front}\,[1,2,2,2,3] = [1,2,2,2])$;
4. *the tail* $(\text{tail}\,[1,2,2,2,3] = [2,2,2,3])$;
5. *prefix* $(1 \rightarrow [2,2,2,3] = [1,2,2,2,3])$;
6. *append* $([1,2,2,2] \leftarrow 3 = [1,2,2,2,3])$;
7. *the size* $(\text{size}\,[1,2,2,2,3] = 5)$;
8. *take the first n elements* $([1,2,2,2,3] \uparrow 2 = [1,2])$;
9. *remove the first n elements* $([1,2,2,2,3] \downarrow 2 = [2,2,3])$;
10. *the concatenation* $([1,2,2] \cap [2,3] = [1,2,2,2,3])$;
11. *the reverse* $(\text{rev}\,([1,2,3]) = [3,2,1])$.

For the convenience of the reader, we prefer to use this logic notation (e.g. $\text{size}\,(S) = N$) rather than that of Prolog (e.g. `S size N`). Similarly, `S size N, N #>= 5` may be abbreviated by $\text{size}\,(S) \geq 5$.

In this paper, we focus our attention on the resolution of constraints for sequences with the operations 1–11, using an external numerical constraint solver such as CLP(FD) to solve the numerical constraints on the sequence size. The existing constraint solvers can be used for constraints of other data types.

Examples. Let E be a set and $1, 2 \in E$.

1. $\{\ S' \leftarrow 1 = S,\ \text{first}\,(S) = 1,\ \text{tail}\,(S') = [2]\ \}$ has the solution $S = [1,2,1]$, $S' = [1,2]$.

2. $\{\ \text{front}\,(S) = [1,1,1],\ \text{size}\,(S) = 5\ \}$ is unsatisfiable, since the first constraint implies $\text{size}\,(S) = 3 + 1 = 4$.

3. The constraint $\text{front}\,(S) = [1]$ has the solutions $S \in \{\,[1,x] \mid x \in E\,\}$. We have here infinitely many solutions iff E is infinite. A set of constraints on sequences can have infinitely many solutions not only for an infinite set E, but also if E is finite. Set $E = \{1,2\}$. The constraint $[1] \cap S = S \cap [1]$ has infinitely many solutions: $[\,]$, $[1]$, $[1,1]$, $[1,1,1], \ldots$

3. CONSTRAINT SOLVER

In this section we present a technique of constraint solving for sequences. It was implemented in the CHR language [8, 9] in SICStus Prolog, and can be consulted and executed from [12]. The implementation in CHR has the advantage to be very clear and easy to experiment with. We describe the main part of the solver (after simple rewriting of some constraints in terms of others) directly by the corresponding CHR rules.

The algorithm is based on the generalized concatenation. *The generalized concatenation* $S = \text{conc}\,(S_1, S_2, \ldots, S_k)$ is equivalent to $(k-1)$ simple concatenations of the sequences $S_1, S_2, \ldots, S_k$, that is, to $S = S_1 \cap S_2 \cap \cdots \cap S_k$. The first step of the algorithm is rewriting of the constraints 1–6, 8–10 in terms of conc and size according to the following rules, where T and Y denote new variables standing for a sequence and an element respectively. This rewriting is executed at most once for each new constraint and leaves in the constraint store the constraints conc, size and rev only.

$\text{first}\,(S) = X$	$\Leftrightarrow$	$S = \text{conc}\,([X], T)$.
$\text{last}\,(S) = X$	$\Leftrightarrow$	$S = \text{conc}\,(T, [X])$.
$\text{front}\,(S) = S_1$	$\Leftrightarrow$	$S = \text{conc}\,(S_1, [Y])$.
$\text{tail}\,(S) = S_1$	$\Leftrightarrow$	$S = \text{conc}\,([Y], S_1)$.
$X \rightarrow S_1 = S$	$\Leftrightarrow$	$S = \text{conc}\,([X], S_1)$.
$S_1 \leftarrow X = S$	$\Leftrightarrow$	$S = \text{conc}\,(S_1, [X])$.
$S \uparrow N = S_1$	$\Leftrightarrow$	$S = \text{conc}\,(S_1, T),\ \text{size}\,(S_1) = N$.
$S \downarrow N = S_2$	$\Leftrightarrow$	$S = \text{conc}\,(T, S_2),\ \text{size}\,(T) = N$.
$S_1 \cap S_2 = S$	$\Leftrightarrow$	$S = \text{conc}\,(S_1, S_2)$.

The second step of the algorithm is based on Thom Frühwirth's solver for lists [9] and treats these three constraints. The CHR rules for this step are given in Figure 1. Recall that a simplification rule `C1,...,Ci <=> D1,...,Dj` in CHR replaces the constraints `C1,...,Ci` in the constraint store by the list of constraints or Prolog goals `D1,...,Dj`. A guarded rule `C1,...,Ci <=> Guard | D1,...,Dj` is applied if in addition the Prolog goal `Guard` is true. The rule `C1,...,Ci ==> D1,...,Dj` will add (or execute) the constraints (or Prolog goals) `D1,...,Dj` if the constraint store contains the constraints `C1,...,Ci`, which are not deleted. We do not detail passive constraint declarations `#Id ... pragma passive(Id)`, which is just an optimization and is not crucial. We refer the reader to [8] for more detail on the CHR language.

The rule r01 aims to propagate the constraint rev using an additional Prolog predicate `reverse`. The rules r02–r05 propagate the generalized concatenation `Rs conc L` until the first element of `Rs` is a non valuated variable. If it is the case, we can jump over this variable only by deleting empty sequences `[]` or the sequence `L` itself in `Rs` as shown in rules r06–r07. If it is still not sufficient for a constraint $\text{conc}\,[R_1, R_2, ..., R_i] = L$, the rule r08 and `lenPropagate` establish the relation $\text{size}\,(R_1) + \cdots + \text{size}\,(R_i) = \text{size}\,(L)$ between the sizes of the sequences, which can help in propagation. The propagation for the constraint `L size N` (where N can be an arithmetic expression) is provided by the rules r08–r11. To avoid repetitions, the rule r12 replaces the constraint `X size N2` in presence of `X size N1` by `N1 #= N2`. The

```
:- use_module(library(clpfd)).
:- use_module(library(chr)).
handler sequences.

constraints conc/2, size/2, rev/2, labeling/0.
operator(700,xfx,conc).
   % 'List conc Seq' means conc(List)=Seq
operator(700,xfx,size).
   % 'Seq size N' means size(Seq)=N

r01@ rev(R,S) <=> reverse(R,S).
r02@ [] conc L <=> L=[].
r03@ [R] conc L <=> R=L.
r04@ [R|Rs] conc [] <=> R=[], Rs conc [].
r05@ [[X|R]|Rs] conc L
   <=> L=[X|L1], [R|Rs] conc L1.
r06@ Rs conc L <=> delete([],Rs,Rs1) | Rs1 conc L.
r07@ Rs conc L <=> delete(L,Rs,Rs1) | Rs1 conc [].
r08@ R conc L ==> lenPropagate(R,L).
r09@ [] size N <=> N#=0.
r10@ [_|L] size N <=> N#=M+1, L size M.
r11@ L size N
   <=> ground(N) | N1 is N, length(L,N1).
r12@ (X size N1)#Id \ X size N2
   <=> N1=N2 pragma passive(Id).
r13@ labeling, ([R|Rs] conc L)#Id <=> true |
   ( var(L) -> length(L,_) ; true),
   ( R=[], Rs conc L ;
   L=[X|L1], R=[X|R1], [R1|Rs] conc L1 ),
   labeling pragma passive(Id).

reverse([],[]).
reverse(R,L):- R size N, L size N, X size 1,
   [X,R1] conc R, [L1,X] conc L, reverse(R1,L1).

delete( X, [X|L],  L).
delete( Y, [X|Xs], [X|Xt]) :- delete( Y, Xs, Xt).

lenPropagate([], []).
lenPropagate([R|Rs],L) :- R size NR, L size NL,
   L1 size NL1,NL #= NR + NL1,lenPropagate(Rs,L1).
```

Figure 1: The essential part of the solver in CHR

```
MACHINE
   WAITINGROOM
SETS
   STATES = {free, occupied};
   NAMES = {n1, n2, n3, ..., n15}
VARIABLES
   sellerA, sellerB, cashierC,
   clientA, clientB, clientC, qSeller, qCashier
INVARIANT
   sellerA ∈ STATES ∧ sellerB ∈ STATES ∧
   cashierC ∈ STATES ∧ clientA ∈ NAMES ∧
   clientB ∈ NAMES ∧ clientC ∈ NAMES ∧
   qSeller ∈ seq (NAMES) ∧ qCashier ∈ seq (NAMES) ∧
   size(qSeller) ≤ 10 ∧ size(qCashier) ≤ 5
INITIALISATION
   sellerA := free || sellerB := free ||
   cashierC := free || clientA :∈ NAMES ||
   clientB :∈ NAMES || clientC :∈ NAMES ||
   qSeller := [] || qCashier := []
OPERATIONS
new(name) =
   PRE name ∈ NAMES ∧ size(qSeller) ≤ 10
   THEN qSeller := qSeller ← name
   END
callA =
   PRE size(qSeller) > 0 ∧ sellerA = free
   THEN sellerA := occupied || clientA := first (qSeller) ||
      qSeller := tail (qSeller)
   END
callB =
   PRE size(qSeller) > 12 ∧ sellerB = free
   THEN sellerB := occupied || clientB := first (qSeller) ||
      qSeller := tail (qSeller)
   END
endA =
   PRE sellerA = occupied
   THEN sellerA := free || qCashier := qCashier ← clientA
   END
endB =
   PRE sellerB = occupied
   THEN sellerB := free || qCashier := qCashier ← clientB
   END
callC =
   PRE size(qCashier) > 0 ∧ cashierC = free
   THEN cashierC := occupied || clientC := first (qCashier) ||
      qCashier := tail (qCashier)
   END
endC =
   PRE cashierC = occupied
   THEN cashierC := free
   END
END
```

Figure 2: Waiting room specification in B

rule r13 defines the labeling constraint, which may be written at most once at the very end of the constraint list. It tries to find all possible solutions of the constraint problem and does not necessarily terminate.

4. APPLICATIONS TO SOFTWARE ENGINEERING

4.1 Waiting Room Example

This section illustrates the applications of the constraint solver as part of a validation and verification tool such as BZTT to different problems in software validation and verification. Consider the example of a waiting room system specification given in Figure 2 (containing some errors which will be detected and corrected below). This simple specification is written in B notation [1] and models the automatic client queue managing in a shop, travel agency etc. Suppose that each client goes first to one of several agents of the first type (say, sellers), than to one of several agents of the second type (say, cashiers). In this example, we consider two sellers A and B and one cashier C. In addition, seller B is the manager of the shop who takes clients only if the queue is rather long. The clients are represented by names which, for simplicity, are taken in the finite set $NAMES$. A system state is represented by the status $free$ or *occupied* of each agent (variables *sellerA*, *sellerB*, *cashierC*), the name of the last client called by each agent (variables *clientA*, *clientB*, *clientC*), the queue of clients to sellers and the queue of clients to the cashier (variables *qSeller* and *qCashier* whose type is

$$\begin{array}{l}
\mathcal{P}_{new} : \mathcal{I}nv \wedge name \in NAMES \wedge size(qSeller) \leq 10 \wedge \\
\quad qSeller' = qSeller \leftarrow name \\
\mathcal{P}_{callA} : \mathcal{I}nv \wedge size(qSeller) > 0 \wedge sellerA = free \wedge \\
\quad sellerA' = occupied \wedge clientA' = \text{first}\,(qSeller) \wedge \\
\quad qSeller' = \text{tail}\,(qSeller) \\
\mathcal{P}_{callB} : \mathcal{I}nv \wedge size(qSeller) > 12 \wedge sellerB = free \wedge \\
\quad sellerB' = occupied \wedge clientB' = \text{first}\,(qSeller) \wedge \\
\quad qSeller' = \text{tail}\,(qSeller) \\
\mathcal{P}_{endA} : \mathcal{I}nv \wedge sellerA = occupied \wedge sellerA' = free \wedge \\
\quad qCashier' = qCashier \leftarrow clientA \\
\mathcal{P}_{endB} : \mathcal{I}nv \wedge sellerB = occupied \wedge sellerB' = free \wedge \\
\quad qCashier' = qCashier \leftarrow clientB \\
\mathcal{P}_{callC} : \mathcal{I}nv \wedge size(qCashier) > 0 \wedge cashierC = free \wedge \\
\quad cashierC' = occupied \wedge clientC' = \text{first}\,(qCashier) \wedge \\
\quad qCashier' = \text{tail}\,(qCashier) \\
\mathcal{P}_{endC} : \mathcal{I}nv \wedge cashierC = occupied \wedge cashierC' = free
\end{array}$$

Figure 3: Before-After predicates

sequences over the set $NAMES$). Initially, all agents are $free$, the queues are empty sequences [] and the names of the latest called clients are arbitrary elements of $NAMES$. The invariant property must be verified in all system states. In this example, the invariant, denoted below by $\mathcal{I}nv$, defines the possible values of variables and the maximal queue length.

The evolution of the system is described by operations, which are defined by generalized substitutions and may have preconditions, input and output values. A new client *name* first registers himself in the system (or is registered by the receptionist not modelled here) who adds him at the end of *qSeller*, provided that $name \in NAMES$ and the queue is not too long (operation *new*). The seller A can take the first client in the queue *qSeller*, provided that $sellerA$ is $free$ and *qSeller* is not empty (operation *callA*). The seller B can take the first client in the queue *qSeller*, provided that $sellerB$ is $free$ and *qSeller* is rather long (operation *callB*). An *occupied* seller can finish serving his client by putting him at the end of $qCashier$ and changing the status to $free$ (operations *endA*, *endB*). The cashier C can take the first client from $qCashier$, provided that $cashierC$ is $free$ and $qCashier$ is not empty (operation *callC*). While being *occupied*, the cashier C can finish serving his client by changing the status to $free$ (operation *endC*).

4.2 Applications

The following examples briefly illustrate the applications of the solver with the BZTT tool in different areas of software validation and verification.

1. *Representing of the system states and operations by constraints.* Figure 3 shows the $Before$-$After$ predicates corresponding to the operations and describing all possible transitions of the system in terms of constraints. The $Before$ part for each operation contains the invariant $\mathcal{I}nv$ and the precondition of the operation. The $After$ part is the postcondition defining the state after the operation and containing only constraints of the form $X' = \dots$, where X' denotes the new value of the variable X after the operation. For unchanged variables, we omit the trivial constraints $X' = X$ for short. This representation of the system states and operations in terms of constraints allows to group the states and to avoid their enumeration. Note that this enumeration is impossible even in this simplified example with small numbers (card $(NAMES) = 15$) because of the great number of sequences over $NAMES$.

2. *Model animation based on the symbolic evaluation.* Due to the constraint representation of the states and operations ($Before$-$After$ predicates of Figure 3), the animation is not limited to the completely valuated states, but can be also applied to partially valuated ones.

3. *Formal model validation based on the symbolic evaluation: detecting of a too strong invariant, a too weak precondition or an inexecutable behavior.* The simple satisfiability verification of the $Before$ part of each $Before$-$After$ predicate allows to detect that the operation *callB* is inexecutable. Indeed, the solver detects that the $Before$ predicate of *callB*

$$\dots \wedge size(qSeller) \leq 10 \wedge \dots \wedge size(qSeller) > 12 \wedge \dots$$

is unsatisfiable. To correct this specification error, we can replace $size(qSeller) > 12$ by $size(qSeller) > 5$ in the precondition of the operation *callB*.

Another error in the specification is a too weak precondition $size(qSeller) \leq 10$ in $\mathcal{P}_{new}$ or a too strong invariant condition $size(qSeller) \leq 10$ in $\mathcal{I}nv$. Indeed, from a state with $size(qSeller) = 10$, the operation *new* can lead to a state with $size(qSeller') = 11$, which does not satisfy the invariant. To detect such errors, it is sufficient to verify if the invariant property is *always verified* after each operation. In other words, we should check whether the $Before$-$After$ predicate of each operation is incompatible with $\neg\mathcal{I}nv'$, where $\mathcal{I}nv'$ is the invariant condition stated for the variables after the operation. For the operation *new*, the condition $\mathcal{P}_{new} \wedge \neg\mathcal{I}nv'$ rewritten in DNF form, contains the disjunct

$$\dots \wedge size(qSeller) \leq 10 \wedge$$
$$qSeller' = qSeller \leftarrow name \wedge size(qSeller') > 10$$

so the solver detects that it is satisfiable for $size(qSeller) = 10$. To correct this error, we can, for example, replace $size(qSeller) \leq 10$ by $size(qSeller) < 10$ in the precondition of the operation *new*.

4. *Generating of tests satisfying a chosen criterion.* Various test coverage criteria are used to guide the (automatic) test generation or to evaluate a given set of tests. These criteria usually can be expressed in terms of constraints, so a constraint solver can be efficiently used for test generation. To detect domain errors, it is necessary to use boundary coverage criteria, which are useful for other error types as well. A new family of boundary coverage criteria for discrete domains and corresponding generation methods were proposed in [13]. They allow to find tests on the boundary of the domain of possible values of variables. Some of the criteria aim to reach the minimal and maximal values of each variable in the generated tests. These criteria can be easily adapted for sequences by minimizing and maximizing the length of each sequence. For example, the set with the following four tests for the operation *endA*:

$$\begin{array}{c}
qSeller = [],\ qCashier = []; \\
qSeller = [n1, n2, \dots, n9, n10],\ qCashier = []; \\
qSeller = [],\ qCashier = [n1, n2, n3, n4, n5]; \\
qSeller = [n1, n2, \dots, n9, n10], \\
qCashier = [n11, n12, n13, n14, n15]
\end{array}$$

will satisfy the **MD** (multi-dimensional) criterion [13] for sequence lengths (we omit here $sellerA = occupied$ and the values of other variables).

5. *Validation of the formal model by tests.* The execution of the generated tests followed by the invariant verification can be used to validate the model. For example, the execution of the operation $endA$ on the following test (which may be generated according to the **MD** criterion):

$$qSeller = [],\ qCashier = [n1, n2, n3, n4, n5],$$
$$sellerA = occupied,\ clientA = n6$$

leads to the state with $qCashier' = [n1, n2, n3, n4, n5, n6]$, where the invariant condition $size(qCashier) \leq 5$ is not satisfied. This too strong invariant could be also detected as shown in **3** above. To correct this error we can delete the unjustified condition $size(qCashier) \leq 5$ from the invariant.

6. *Testing of an implementation of the formal model.* Due to the constraint representation of the states and operations of the system, in black-box testing the constraint solver allows to obtain *an oracle* predicting the correct state after executing of the tests on the implementation. In white-box testing, the constraint solver gives in addition the possibility *to detect the contradictions* between the model and the implementation by comparing the constraint sets extracted from the model (see Figure 3) and from the implementation.

Integration of the constraint solver for sequences into a validation and verification tool such as BZTT [5] will make these operations automatic or semi-automatic and will provide a convenient graphical interface. We refer the reader to [4] for more detail on the application of constraint solving to different problems of software engineering in the BZTT method.

5. CONCLUSION AND FUTURE WORK

We presented the problem of constraint solving for sequences and proposed a constraint solving technique for the usual operations used in such notations as B and Z.

Our experiments (which are not detailed here for lack of space) show that our method is rather efficient on big constraint sets and gives satisfactory results. The resolution takes several seconds to several minutes on constraint sets with several dozens to several hundreds of constraints. In addition, the resolution time of our solver does not depend very much (compared to other solving techniques we tested) on the ordering of the constraints, which is important to guarantee better efficiency even in the worst case.

It is due to the uniformity of the approach: although we express some simple constraints in terms of a more complicated and more general constraint conc , it minimizes the number of different constraints and the number of constraint handling rules and finally provides a much faster solver for big constraint sets. The example in Section 4 shows various applications of the solver to software engineering in a validation and verification tool such as BZTT. In this example, the solver was called by hand since it has not yet been completely integrated into BZTT. All the other steps (formal model parsing, generating of Before-After predicates, dealing with constraint sets etc.) are implemented and can be done automatically.

The future work will include the following:

1. to integrate the solver into BZTT like it was already done for the existing solver CLPS-B [4];

2. to apply the solver as part of the BZTT to validation and verification of real-sized software with sequences.

Acknowledgment. The author would like to thank Fabrice Ambert, François Fages, Thom Frühwirth, Arnaud Gotlieb and Bruno Legeard for profitable discussions and their interest to this project.

6. REFERENCES

[1] J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996. ISBN 0521496195.

[2] M. Lothaire. Algebraic Combinatorics on Words. Cambridge University Press, 2002. ISBN 0521812208.

[3] L. Berkaoui, B. Legeard. Représentation de séquences définies sur des ensembles non instanciés par arbre PQR partiel. In Actes de JFPLC'98, Nantes, France, 251–266, May 1998. Hermès.

[4] F. Bouquet, B. Legeard, F. Peureux. CLPS-B – Constraint solver to animate a B specification. International Journal on Software Tools for Technology Transfer, 6 (2004), No. 2, 143–157.

[5] The BZ-Testing-Tools web site, http://lifc.univ-fcomte.fr/~bztt, Université de Franche-Comté, Besançon.

[6] A. Colmerauer. An introduction to Prolog III. Communications of the ACM, 33(1990), No. 7, 69–90.

[7] V. G. Durnev. Studying algorithmic problems for free semi-groups and groups. In: S. Adian, A. Nerode (Eds). Logical Foundations of Computer Science (LFCS 97). Lect. Notes Comp. Sci., 1234(1997), 88–101. Springer-Verlag. ISBN 3540630457.

[8] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In: P. Stuckey, K. Marriot (Eds.). Special Issue on Constraint Logic Programming. Journal of Logic Programming, 37(1998), No. 1–3, 95–138.

[9] T. Frühwirth. The CHR web site. http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/chr.html, Universität Ulm.

[10] J. Karhumäki, F. Mignosi, W. Plandowski. The expressibility of languages and relations by word equations. Journal of the ACM, 47(2000), No. 3, 483–505.

[11] A. Koscielski, L Pacholski. Complexity of Makanin's Algorithm. Journal of the ACM, 43(1996), No. 4, 670–684.

[12] N. Kosmatov. Constraint solving for sequences web site. http://lifc.univ-fcomte.fr/ ~kosmatov/sequences, Université de Franche-Comté, Besançon.

[13] N. Kosmatov, B. Legeard, F. Peureux, M. Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In Proc. of the 15th Int. Symp. on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, 139–150, November 2004. IEEE Computer Society Press.

[14] G. S. Makanin. The problem of solvability of equations in a free semigroup. Mat. Sbornik (N.S.), 103 (1977), No. 2, 147–236 (in Russian). English translation in: Math. URSS Sbornik, 32(1977), 129–198.

[15] J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, 2^{nd} edition, 1992. ISBN 0139785299.