Set Constraints: Validation for Set-Based Specification*

Agnès Tellez-Arenas
LIFO, Université d'Orléans (France)
(e-mail: tellez@lifo.univ-orleans.fr)

Abstract

Many powerful specification languages are based on set theory, such as Z ([12]) and B ([1]) derived from VDM([6]). To validate a specification developed along such lines at a formal level, one needs tools to manipulate logical formulas on sets. Behavioral properties of modules specified in this manner can be naturally described using intentional sets and constraints on the global state variables of the specification. In this work we propose a method for such purposes of validation, by constructing a logic program meant as a "synthesis" of the behaviour of the state variables under the operations of the specification. The logic program thus constructed admits as a natural model the set of all reachable values of the state variables, after the execution of any sequence of operations. The main interest of such an approach is that it can allow us to prove the properties of a program; in addition, we can also use the logic program as a "prototype" capable of testing if a given value is reachable by the global state variables of the specified module.

1 Introduction

Software engineering helps to produce quality software, and a major concern in software construction is to try to guarantee correctness of software with respect to certain properties. A first step in software construction is often the writing up of a specification which is a description of a system and its expected properties. We are mainly interested here in the description of the functional behavior, and this is usually done with formal specification methods, employing mathematics-based languages, and methods and techniques allowing to describe the syntax and semantics of a system. Two major approaches are known for the validation of a software. "Model checking" is based on building finite models of the system (on which the expected properties are checked). The specification is usually treated here as an automaton, and the model is expressed in terms of finite state transition systems, or also in terms of an automaton. In the second technique based on "theorem proving", the system and the expected properties are described as logical formulas.

Our work in this paper is mainly oriented towards the B method ([1]), employing the theorem proving approach. States of the system are described using sets and other mathematical structures derived from sets such as functions or relations. Modules are described by *abstract machines*, handling state variables (i.e. data), operations modifying them, and invariants coding expected properties of any state of the module. B allows to specify, design, and also

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

generate code for software. Any initial specification may be successively refined, and and each refinement step has to be proved. Our method described here is meant to help and to complete the description of the initial specification and the validation proofs (the refinement and code generation steps will not be treated in this paper).

In B, the method for validation consists in proving that any operation preserves the invariant. For instance, if x is a state variable and $even(x)$ is the invariant, then we have to show that for any operation applying a substitution σ to x that $even(x) \Rightarrow even(\sigma x)$. Each operation is usually expected thus to be validated independently of other operations. Our approach provides, in such a context, some tools to describe (and then prove) the global properties of the specification, whose proof depends on *all* the operations. This naturally leads us to constrain sets of values reached by state variables after an execution of several operations of an abstract machine. Working in the framework of first-order logic, and a set based specification language, we "intentionally" describe sets : $\{x \mid p(x)\}$ denotes the set of elements x satisfying the property p, where p is generally a predicate defined by logic program. Such a formalism allows us to represent extensional sets (for instance $\{a, b\} = \{x \mid x = a \vee x = b\}$), and also cartesian products ($S \times S' = \{(x, y) \mid x \in S \wedge y \in S'\}$). Furthermore, this way of modelling data using logical formulas allows us to use recursive definitions: we can for instance write $\{zero\} \subseteq \mathcal{S} \wedge \{x \mid x = s(y) \wedge y \in \mathcal{S}\} \subseteq \mathcal{S}$ to describe integers, thus avoiding infinite or too big enumerations. For intentional sets, combinatorial techniques used on extensional sets are in general not appropriate, and we use proof tools coming from first-order logic to prove properties on our sets. Many such are provided with the B environment. Of course we have to develop some particular tactics appropriate to the logical formulas that we obtain in our context. Following the B-philosophy, it is assumed that the usual paradoxes of set theory are avoided by adopting a suitable type theory on our formulas (the type-checking mechanism itself is irrelevant to our work, and will not be presented here).

Our validation approach is based on program transformation techniques, consisting in this context essentially of a mechanism allowing to give a synthesis of the specification (i.e. to say an abstract machine) in terms of a logic program; the set of all reachable values of the global state variables is to constitute a natural model for this logic program. This is a way for defining the formal semantics of a such a set based specification. This approach is different from the "set-based program analysis" ([5]) where the result of the analysis is a system of set constraints whose solution is an *approximation* of the set of values of variables of the program. Besides, the proof of some global properties of the abstract machine (of a module), may need to take into account all the operations of the module under study, and possibly also the global properties of some other modules. For instance, consider an abstract machine for which we want to validate the behavior on a global state variable x, assuming given a set specification: $Set\_of\_values\_of\_x \subseteq \{x \mid even(x)\}$. Assume that an operation of the machine is of the form $x := y + z + x$ where y and z are two other state variables, and that the invariant is $even(x)$. A very local view of the situation would generate a proof-obligation (as in B) $even(x) \Rightarrow even(y + z + x)$. This clearly can not be proved without taking into account other operations affecting y and z. But a logic program synthesizing the behavior of all the operations can very naturally come to our rescue here, and allow us fulfill the proof-obligation; and by adopting such an approach, the need for resorting to "local" assertions - when the invariant is not sufficiently precise - can be considerably reduced.

This paper is organized as follows. In section 2 we succinctly describe a B abstract machine. In section 3 we present a method to describe a specification using intentional set constraints. Section 4 describes the mechanism allowing to give a synthesis of an abstract machine in terms of a logic program and an illustrative example. Section 5 is the conclusion.

2 B-Abstract machines

In the context of the B method, we want to specify the behavior of an abstract machine. We give below the key points of the abstract machines theory. Any abstract machine can be seen as a module, a class or an abstract data type, where we define:

- the *parameters* of the machine and their *constraints*,
- some *constants* and their *properties*,
- the state *variables*,
- the *invariant* which is a logical formula describing the invariant property of the state of the machine,
- some optional *assertions* which are logical formulas to help the proofs
- the *initialization* of the state variables,
- the *operations*.

Table 2.1 : Abstract machine

Constants of an abstract machine are functions, sets, or scalars used in the description of invariant and operations. They may be user-defined by a theory given in addition to the machine, but can also be described by logical formulas. The invariant describes the expected properties of the state variables, given in particular their type, whereas assertions are logical formulas deductible from the invariant used as intermediate lemmas in proofs. Operations may contain some *preconditions* which are used as hypothesis in proofs. To validate the specification, the specifier has to prove that

- constraints$\wedge$ properties $\Rightarrow$ [initialization] invariant
- constraints$\wedge$ properties $\wedge$ invariant $\Rightarrow$ assertions
- constraints$\wedge$ properties $\wedge$ invariant $\wedge$ assertions $\wedge$ precondition $\Rightarrow$ [operation]invariant

Table 2.2 : Proof obligations

3 Set constraints for specification

Set constraints appear naturally when we want to determine the set of states that a given abstract machine can reach after any execution of its operations. We first give the syntax of intentional set constraints, then we present an example.

3.1 Intentional sets constraints

Sets in general can be represented in different ways : by enumeration ([3, 9]) and such sets are extensional, or described using tree automatas ([4, 7]) when they are regular. We are mainly interested in intentional sets $\{\overrightarrow{x} \mid F\}$ which appear naturally in specifying properties along with set constraints. For instance, we might specify that the set VAL of values taken by a given variable is included in the naturals which are even by $VAL \subseteq \{x \mid x \in NAT \wedge even(x)\}$, and if we see by analysis of a specification that this variable can be equal to 1, 2 or 3, then we write the constraint $\{x \mid x = 1 \vee x = 2 \vee x = 3\} \subseteq VAL$. In this case it is easy to notice the inconsistency. In [3, 9, 2], intensional sets are allowed but they are replaced the corresponding extensional set terms, using enumeration techniques.

Assume given to begin with a first order signatures, whose variables are denoted by small roman letters $x, y, \ldots$ This signature is enriched with set constructors among the abstraction $\{\ldots \mid \ldots\}$ is the main one, and then with a new family of variables called *set variables*, denoted by calligraphic letters $\mathcal{X}$, $\mathcal{Y}, \ldots$ Other set constructors are allowed, but they are derived from $\{\ldots \mid \ldots\}$ and are not necessary, though they make easier the description of problems. In order to give a formal meaning of these constructors, we introduce the predicate $\in$ formalizing the membership concept, which is the primitive notion of set theory. The extensionality axiom allows thus to define the equality between two sets. We introduce another predicate formalizing inclusion which will be used to express a specification. A *set expression se* (abstraction, union, intersection, complement, set variable), and a *set constraint* C will be thus built on the following grammar, where we use the syntactic convention that $\overrightarrow{s}$ denotes a tuple $(s_1, ..., s_n)$:

$$\begin{array}{lcl} se & ::= & \{\overrightarrow{x} \mid F\} \mid se \cup se \mid se \cap se \mid \overline{se} \mid \mathcal{X} \\ F & ::= & Atom \mid F \wedge F \mid F \vee F \mid \neg F \mid \overrightarrow{x} \in se \\ C & ::= & se \subseteq se \mid \forall \overrightarrow{x} (se \subseteq se) \mid \overrightarrow{s} \in se \end{array}$$

When we write $\{\overrightarrow{x} \mid F\}$, we have to give a semantics to F. For this purpose we choose to use logic programs with negation defined in the usual manner, using the Künen semantics ([8, 10]). The Künen semantics of a logic program P is defined in term of the so-called "Fitting's immediate consequence operator" Φ_P, iterated upto the first infinite ordinal ω. Therefore, given a logic program P, we will write $\{\overrightarrow{x} \mid F\}$ for denoting the set of all the ground terms $\overrightarrow{t}$ such that there exists a proof of $F[\overrightarrow{x} \leftarrow \overrightarrow{t}]$, where the truth-values of the bound atoms are given by the Künen semantics.

3.2 Constraints on reachable states of an abstract machine

Defining an execution as a finite sequence of operations of a machine, reachable states (see definition 4.1) of a machine which has n state variables $x_1, \ldots, x_n$ are all the tuples $(v_1, \ldots, v_n)$, such that there exists an execution after which $x_i = v_i$ for $i \in [1..n]$. We define reachable values of a state variable in the same way.

We illustrate our purpose on an example, specifying an abstract machine with 3 state variables x_1, x_2, x_3. We denote by the set variable $\mathcal{S}_i$ the set of reachable value of a state variable x_i, and by $\mathcal{E}$ the set of reachable states of the machine. We use the predicate *even* that we suppose to be a constant of the abstract machine. We give a conjunction of set constraints which describes:

• reachable values of state variables	(1) $\mathcal{S}_3 \subseteq \{x \mid even(x)\}$
• a set of counter-example *(states that the abstract machine must not reach)*	(2) $\{(1, 10, 10)\} \subseteq \overline{\mathcal{E}}$
• a set of expected values *(for each triple must exist an execution after which the state is this triple)*	(3) $\{(0, 10, 10), (1, 11, 10)\} \subseteq \mathcal{E}$
• constraints to bind the set variables *(the set of reachable states is included in the cartesian product of set of reachable values of each state variable)*	(4) $\mathcal{E} \subseteq \{(x, y, z) \mid x \in \mathcal{S}_1 \wedge y \in \mathcal{S}_2 \wedge z \in \mathcal{S}_3\}$

Table 3.2: an example of set specification

An algorithm allowing to test the satisfiability of such a specification (i.e. to test if there exists a solution) has been developed ([11, 13]), but we do not present it here. It uses rewriting techniques to transform set constraints in order to get a logical formula without any set variable. The specifier can prove such a goal using the proof tools provided with the B environment. The application of this algorithm on the conjunction of set constraints presented in table 3.2 produces the following logical formula, whose meaning is that there exists some sets $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ and $\mathcal{E}$ satisfying $(1) \wedge (2) \wedge (3) \wedge (4)$ if and only if we can prove:

$$\begin{array}{l} \{(0,10,10),(1,11,10)\} \subseteq \overline{\{(1,10,10)\}} \\ \wedge\ \forall x \forall y(\{z \mid (x=0 \wedge y=10 \wedge z=10) \vee (x=1 \wedge y=11 \wedge z=10)\} \subseteq \{z \mid even(z)\}) \end{array}$$

This first goal can be simplified into

$$\begin{array}{l} \forall x \forall y \forall z(\ (x=0 \wedge y=10 \wedge z=10)\ \Rightarrow\ \neg(x=1 \wedge y=10 \wedge z=10)\ ) \\ \wedge\ \forall x \forall y \forall z(\ (x=1 \wedge y=11 \wedge z=10)\ \Rightarrow\ \neg(x=1 \wedge y=10 \wedge z=10)\ ) \end{array}$$

which is easy to prove, whereas the second goal can be simplified into

$$\begin{array}{l} \forall x \forall y \forall z(\ (x=0 \wedge y=10 \wedge z=10)\ \Rightarrow\ even(z)\ ) \\ \wedge\ \forall x \forall y \forall z(\ (x=1 \wedge y=11 \wedge z=10)\ \Rightarrow\ even(z)\ ) \end{array}$$

which can be proved using the definition of *even*.

We have just shown how to describe a specification using set constraints, and briefly how to check if *there exists* a solution to this specification. We present in the following section a method to check if the set of reachable states of a given abstract machine is a solution to the set specification.

4 Validation of an abstract machine via synthesis

Since a set specification constrains the set of reachable states of a B-abstract machine, the validation step needs a formal description of this set, before checking that it satisfies the constraints. We present now a mechanism whose aim is to construct a logic program admitting this set as a model. We first illustrate this idea with the synthesis of a B-abstract machine whose set specification has been described in the previous section (table 3.2).

4.1 Example

We present in the left-hand column of the following table the operation and the initialization part of the abstract machine, whose invariant could be $x_1 \in NAT \wedge x_2 \in NAT \wedge x_3 \in NAT$, whereas we describe in right-hand column the clauses defining a predicate $synth()$ in such a way that the set of reachable values of x_1, x_2, x_3 and the set or reachable states are:

$$\left\{ \begin{array}{lcl} \mathcal{S}_1 & = & \{x_1 \mid \exists x_2 \exists x_3\ synth(x_1,x_2,x_3)\} \\ \mathcal{S}_2 & = & \{x_2 \mid \exists x_1 \exists x_3\ synth(x_1,x_2,x_3)\} \\ \mathcal{S}_3 & = & \{x_3 \mid \exists x_1 \exists x_2\ synth(x_1,x_2,x_3)\} \\ \mathcal{E} & = & \{(x_1,x_2,x_3) \mid synth(x_1,x_2,x_3)\} \end{array} \right.$$

The table 4.3 gives the method to build this logic program.

INITIALIZATION	
$(x_1, x_2, x_3) := (0, 10, 10)$	$synth(0, 10, 10).$
OPERATIONS	
$op_1(n) \hat{=}$ $IF\ even(n)\ THEN\ x_1 := x_1 + n$	$synth(x_1', x_2, x_3) \Leftarrow x_1' = x_1 + n \wedge even(n)$ $\wedge synth(x_1, x_2, x_3).$
$op_2 \hat{=}$ $(x_1, x_2) := (x_1 + 1, x_2 + 1)$	$synth(x_1', x_2', x_3) \Leftarrow x_1' = x_1 + 1 \wedge x_2' = x_2 + 1$ $\wedge synth(x_1, x_2, x_3).$
$op_3 \hat{=}$ $x_3 := x_1 + x_2 + x_3$	$synth(x_1, x_2, x_3') \Leftarrow x_3' = x_1 + x_2 + x_3$ $\wedge synth(x_1, x_2, x_3).$

Table 4.1 : synthesis of an abstract machine

The predicate *even* has to be declared and typed in the first part (see table 2.1) of this machine. To check that this abstract machine satisfies the set specification given in table 3.2, we have to prove that (in addition to the usual B-proof obligations)

$$
\begin{array}{ll}
& \textbf{(1)}\ \{x_3 \mid \exists x_1 \exists x_2 . synth(x_1, x_2, x_3)\} \subseteq \{x \mid even(x)\} \\
\wedge & \textbf{(2)}\ \{(1, 10, 10)\} \subseteq \overline{\{(x_1, x_2, x_3) \mid synth(x_1, x_2, x_3)\}} \\
\wedge & \textbf{(3)}\ \{(0, 10, 10), (1, 11, 10)\}\ \subseteq \{(x_1, x_2, x_3) \mid synth(x_1, x_2, x_3)\} \\
\wedge & \textbf{(4)}\ \{(x_1, x_2, x_3) \mid synth(x_1, x_2, x_3)\} \subseteq \\
& \qquad \{(x_1, x_2, x_3) \mid \\
& \qquad\qquad x_1 \in \{x_1 \mid \exists x_2 \exists x_3 (synth(x_1, x_2, x_3))\} \wedge \\
& \qquad\qquad x_2 \in \{x_2 \mid \exists x_1 \exists x_3 (synth(x_1, x_2, x_3))\} \wedge \\
& \qquad\qquad x_3 \in \{x_3 \mid \exists x_1 \exists x_2 (synth(x_1, x_2, x_3))\}\}
\end{array}
$$

This proof is decomposed into four goals, and we can reduce the second and the third respectively into:

$$\textbf{(2)} \neg synth(1, 10, 10) \quad \text{and} \quad \textbf{(3)}\ synth(0, 10, 10)\ \wedge\ synth(1, 11, 10).$$

These atomic goals can be proved using the resolution based mechanism of logic programming. Without such a program, we can not check that a given value is reachable by an abstract machine : such a property is not an invariant of the state of the machine. The first goal is more difficult to prove, and we have to use the recursive definition of $synth()$ and $even()$. The fourth goal is easy to check.

4.2 Synthesis of a B-abstract machine

We formally present now the method that we use to build the "synthesis" of the operations of a B-abstract machine. Each operation Σ is described in terms of a so-colled *generalized* substitution. We present below the "grammar" for the main ones, from which others can be defined. The formal meaning of substitutions is given in the B book ([1]) in terms of a predicate $prd_{\overrightarrow{x}}(\Sigma)$, establishing the relation between the tuple of state variables $\overrightarrow{x}$, and the values of $\overrightarrow{x}$ after application of the operation Σ (denoted by $\overrightarrow{x}'$). In the following table an *Expression* is a set, a variable, an element of a set, or a tuple of expressions, and P is a first order formula.

substitution Σ	intuitive meaning	$prd_{\overrightarrow{x}}(\Sigma)$
$\overrightarrow{x}$:= Expression	usual affectation	$\overrightarrow{x}$'=Expression
$\overrightarrow{y}$:= Expression	if $\overrightarrow{x} = \overrightarrow{y}, \overrightarrow{z}$	$\overrightarrow{y}$', $\overrightarrow{z}$'=Expression, $\overrightarrow{z}$
P| Σ	P is a precondition	P$\Rightarrow prd_{\overrightarrow{x}}(\Sigma)$
$\Sigma[]\Sigma'$	apply Σ or Σ'	$prd_{\overrightarrow{x}}(\Sigma) \vee prd_{\overrightarrow{x}}(\Sigma')$
P$\Rightarrow \Sigma$	If P then Σ	P $\wedge prd_{\overrightarrow{x}}(\Sigma)$
@Variable.Σ	for a certain variable, apply Σ	$\exists z.prd_{\overrightarrow{x}}(\Sigma)$

Table4.2 : substitutions vs predicates

Defining an execution as a *finite* sequence of substitutions starting form the initialization step, and the size of an execution by the number of substitutions (i.e. operations) applied, we can now formally define the meaning of reachable state:

Definition 4.1 *We consider an abstract machine M, with n operations $\Sigma_1, \ldots, \Sigma_n$, the initialization being denoted by Σ_0, and whose tuple of state variables is $\overrightarrow{x}$.*

- *Let $E = (\Sigma_{i_j})_{j=0}^{j=m}$ be any execution of M of size m. We define by induction on m the set of reachable states $\mathcal{E}_E^m$ by this execution:*

$$\begin{cases} - \textit{ Set of states reached after initialization: } \mathcal{E}_E^0 = \{\overrightarrow{x}' \mid prd_{\overrightarrow{x}}(\Sigma_0)\} \\ - \textit{ If for all } j < l \leq m, \mathcal{E}_E^j \textit{ is defined, then: } \mathcal{E}_E^l = \{\overrightarrow{x}' \mid \overrightarrow{x} \in \mathcal{E}_E^{l-1} \wedge prd_{\overrightarrow{x}}(\Sigma_{i_l})\} \end{cases}$$

- *The set $\mathcal{E}(M)$ of reachable states of M is the union of all the set of reachable states by an execution.*

We can now give the rules that we use to build for each operation applying a substitution Σ the clause synthesizing its behaviour.

For Σ_0 the clause is	$\mathrm{prd}_{\overrightarrow{x}}(\Sigma) \Rightarrow synth(\overrightarrow{x}')$
For $\Sigma_{i>0}$ the clause is	$synth(\overrightarrow{x}) \wedge \mathrm{prd}_{\overrightarrow{x}}(\Sigma_i) \Rightarrow synth(\overrightarrow{x}')$

Table4.3 : Formulas construction

The formulas thus constructed are not first order clauses, and so not Horn clauses, in general. All the same, we can derive from these formulas a logic program (in general with negation) in a natural way, that we denote by *Synth(M)* and refer to as *behavioral synthesis* of the B-abstract machine *M*. The following proposition establishes a formal semantics for the set of reachable states of a B-abstract machine.

Proposition 4.2 *Let M be a B-abstract machine and $Synth(M)$ the logic program derived as above as its behavioral synthesis, then there is a natural bijective correspondance between $\mathcal{E}(M)$ and the positive part of the Künen semantics of $Synth(M)$.*

Proof sketch :
We prove this result by induction on the size of an execution, and by induction on the number of clauses used in the proof tree for $synth(\overrightarrow{x})$ for a given $\overrightarrow{x}$ □

5 Conclusion

Starting from a B abstract machine, we propose an approach to describe the global properties provable from a specification, using set constraints on reachable states. We give a method consisting in producing a logic program which synthesizes the behavior of the operations of an abstract machine, and gives a logical description of the set of reachable states. This approach allows us to help and to complete the development of a specification, and in the validation proofs for a method such as B. Furthermore, the logic program obtained can be used as a "prototype" of the abstract machine; in particular it can allow us to use the resolution based mechanism of logic programing to check if the machine can reach or not some specific values on its state variables. Please note however that, the logic program produced in general contains negation, so we may not be able to decide *always* on the membership of a given value w.r.t. the set of reachable values of the machine; but such a decision is possible in many cases where, a standard prototype based on top-down traverse of the specification will fail to answer the same question.

The B approach also provides techniques to refine in several steps the initial abstract machine into an executable module. During each refinement step, the abstract machine is reconstructed, and new structures are introduced (loop, sequencing, arrays, ...). Of course the specifier has to prove that the refined machine satisfies its initial specification. The extension of our approach presented here, in order to treat efficiently such refinement problems is part of our projected future work.

References

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

[2] P. Bruscoli and A. Dovier and E. Pontelli and G. Rossi. Compiling Intensional Sets in CLP. In P. Van Entenryck, editor, *Logic Programming :* Proceedings of the 11^{th} International Conference on Logic Programming, pages 647–661. MIT Press, june 1994.

[3] A. Dovier, E. Omodeo, E. Pontelli and G. Rossi. {log} : A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1-44, july 1996.

[4] R. Gilleron, S. Tison and M. Tommasi. Solving systems of set constraints with negated susbset relationships. Technical Report 247, Laboratoire d'Informatique Fondamentale de Lille, 1993.

[5] N. Heintze. *Set Based Progam Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[6] C. B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice Hall International, 1986.

[7] D. Kozen. Set Constraint and Logic Programming. In *Proceedings of CCL95*, 1995.

[8] K. Künen. *Negation in Logic Programming.* J. Logic Programming, Vol 4 1987, pages. 289-308.

[9] B. Legeard and E. Legros. Short overview of the CLPS system. In *Proceedings of the* 3^{rd} *International Symposium PLILP'91*, pages 431-433, 1991.

[10] J.W. Lloyd. *Foundation of Logic Programming.* Springer-Verlag, Symbolic Computation Series, 1987.

[11] G. Richard, F. Saubion and A. Tellez-Arenas. Intentional Set Constraints. In M. Falaschi, M. Navarro and A. Polocritti, editors, *Proceedings of the joint conference on declarative programming APIA-GULP-PRODE'97*, pages 479-489, Grado, Italy, june 16-19 1997.

[12] J. M. Spivey. *The Z notation : A reference Manual.* Series in Computer Science. Prentice Hall International, 1989.

[13] A. Tellez-Arenas. Solving Inclusion Constraints between Intentional Sets. Schloss Hagenberg, Austria, November 1 1997. Third International Workshop on Set Constraints.