Helping Programmers to Adopt Set-Based Specifications

Maximiliano Cristiá[1], Gianfranco Rossi[2], and Claudia Frydman[3]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] Università degli studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it
[3] Aix Marseille Univ., CNRS, ENSAM, Univ. de Toulon, LSIS UMR 7296, France
claudia.frydman@lsis.org

Abstract. Set theory is a key component of formal notations such as B, Z and Alloy. Set-based specifications are short while precise enough as to start the implementation. However, according to our experience, practitioners without a mathematical background find difficulties in using them. In this paper we propose the set-based programming language $\{log\}$ as an aid to teach programmers to write set-based specifications. In one hand, a large class of set-based specifications can be automatically translated into $\{log\}$ programs, which can be used as prototypes; on the other hand, plain $\{log\}$ programs can be used as contracts, which are closer to the implementation. This could help in a widest adoption of set-based specifications since programmers seem to be adopting contracts as a form of specification.

1 Set-Based Specifications

We start by considering a simple requirement and we show three different formal specifications for it. The objective is to informally discuss different aspects of each specification in terms of abstraction level, conciseness, readability and whether programmers would like to use them or not. The ultimate goal is to see whether it is possible to think of a teaching methodology taking advantage of the best of each of the considered approaches.

Consider a bank which offers savings accounts to its clients. Given a savings account anyone can deposit money in it. Say the savings accounts are identified by some account ID's. From a mathematical perspective we can see all the savings accounts of this bank as a partial function from the set of account ID's onto the set of their balances. This can be formalized as $sa \in ID \nrightarrow \mathbb{Z}$, if we consider that balances are integer numbers. Hence, if $a \in \operatorname{dom} sa$ then $sa(a)$ is the balance of account a. sa is a partial function because in any given moment not all of the account ID's are used in the bank. Now, a formalization for the requirement about a person depositing money in an account can be as follows:

$$a? \in \operatorname{dom} sa \land m? > 0 \land sa' = sa \oplus \{a? \mapsto sa(a?) + m?\} \quad \text{(Set)}$$
$$\lor\ (a? \notin \operatorname{dom} sa \lor m? \leq 0) \land sa' = sa$$

where variables decorated with '?' are meant to be input parameters; variables decorated with a prime are meant to be the value of a state variable in the next state; $a?$ is the account where the amount $m?$ is intended to be deposited; and the $\oplus$ operator roughly updates a function [14, page 102]. Note that in this model, partial functions are sets of ordered pairs. For this reason we call it a *set-based specification*.

From now on, the term *set* will include binary relations, partial and total functions, and bags (multisets). In effect, all these structures can be expressed in terms of set theory as shown for example in [14]. So a *set-based specification* will be any specification using sets as the main mathematical structure.

An alternative model, and perhaps closer to a possible implementation, is based on lists instead of partial functions. In this case we can define $sa \in \text{seq}(ID \times \mathbb{Z})$ and the specification for depositing money becomes[1]:

$$\begin{aligned} &\exists\, s_1, s_2 \in \text{seq}(ID \times \mathbb{Z});\ b \in \mathbb{Z} : \qquad \text{(List)} \\ &\quad m? > 0 \land sa = s_1 \frown \langle (a?, b) \rangle \frown s_2 \land sa' = s_1 \frown \langle (a?, b + m?) \rangle \frown s_2 \\ &\quad \lor ((a?, b) \notin sa \lor m? \leq 0) \land sa' = sa \end{aligned}$$

where $\frown$ is is the list concatenation operator. This specification is more complex because there is no easy way of expressing the modification of a list element without having its position.

As can be seen, the list-based specification is harder to understand than the set-based one. We think that the true problem is that the savings accounts of a bank are not a list but, essentially, a partial function. This is a recurring observation in software specification: many real-life entities *are*, essentially, sets, binary relations or partial functions (seen as sets). They *are* not lists, trees or hash tables. Therefore, set-based specifications should be favored over other notations if the goal is to describe the essence of the problem to be implemented.

A third model could be based on design-by-contract (DBC) notations such as the Java Modeling Language (JML) [3], Spec# [2] or the Eiffel contract language [10]. In this case, programmers can give a contract directly as program annotations in terms of the variables and types used in the implementation. Assume the programmer uses some implementation of Java's Map interface [12] to store the savings accounts. Then by declaring sa as, for instance, $\mathsf{HashMap}\langle ID, \mathbb{Z} \rangle$ we can give the following JML contract for the deposit operation:

```
public normal_behavior                                          (JML)
    requires sa.containsKey(a?) && m? > 0
    ensures sa == old(sa).put(a?, old(sa).get(a?) + m?)
public exceptional_behavior
    requires !(sa.containsKey(a?) && m? > 0)
    assignable nothing
```

[1] We are assuming sa has no duplicate elements.

where containsKey, put and get are methods of Map. In a sense, this contract is better than (List) because Map is closer to the notion of partial function.

According to our experience, programmers feel more comfortable with contracts such as (JML) than with specifications such as (Set). The reasons may be in the following facts: a) the contract language is closer to the implementation language thus increasing the learning curve; b) the possibility of using data structures present in the standard library of the programming language and program variables as part of the contract; c) contracts can involve structural properties such as inheritance, information hiding, etc.; d) contracts are given as program annotations rather than as a separate description; and e) some of these DBC notations include tool support for runtime checking, static verification and test case generation [8, 9, 11].

However, the Software Engineering and Formal Methods communities for years have supported the notion that specifications should be more abstract than their implementations. Clearly, each DBC notation is tightly coupled with a given programming language, thereby, necessarily oriented towards specific implementation choices.

Hence, as Software Engineering instructors we are faced with two problems regarding set-based formal specifications: a) specifications should be abstract and set theory provides a sound foundation for this for many systems; and b) contracts are more appealing to programmers but not as abstract as specifications and they seldom use set theory.

In order to tackle these problems from an educational perspective, in this paper we propose to use a combination of set-based specifications, such as (Set), and {*log*} (pronounced 'setlog') programs. {*log*} is a constraint logic programming (CLP) language implementing a very general theory of sets. As such it can determine satisfability for a wide range of set formulas. Therefore, {*log*} can be used both as a tool beneath a set-based specification language such as Z or B, and as the basis of a set-based contract language. In this way, it can be closer to an implementation, like DBC contracts, but it enables set-based specifications, like specification languages.

We believe that {*log*} can help during the teaching process of software specification. It can give an operational, programming-oriented view of set theory to students while forcing them to write set-based specifications.

2 Problems Teaching Set-Based Specifications

The problems listed below were identified by the authors while teaching set theory, programming and formal specification in mandatory courses of several undergraduate degrees in Argentina, France and Italy for many years, and as occasional trainers for practitioners.

From now on *student* means a person enrolled as a student in a university degree as well as a practitioner taking a training course.

We have identified the following problems regarding the use of set theory as the basis for software formal specification.

Students tend to think in terms of the data structures they have already learned and used. This means that it is hard for them to represent a particular concept in the requirements as a set. They hardly "see" a set in the requirements. For example, in the context of information systems, the first data structure they think of is a table. Only after they learn that tables are either relations or partial functions, they start to feel comfortable with them.

Often they think that it is a waste of time to find the best set-based structure for a concept if they are going to end up implementing it as a hash table, a tree or any other implementation-level data structure. They need to see how concise many properties become when they are expressed in terms of sets rather than in terms of implementation-oriented data structures.

This problem may be originated in two sources: the complexity of working at more than one level of abstraction; and the fact that students are used to work with certain kind of languages and data structures.

The lack of control structures is a source of problems rather than an aid. Students, but mostly experienced programmers, find many troubles in formalizing a requirement without control structures. A declarative language seems to be incomplete for them. Even specifications that take the form of state machines pose problems. A typical case is representing a loop as a state transition that is enabled until some precondition becomes false (and thus the transition is disabled) and at that moment another state transition becomes enabled. Another typical case is to make them understand that many times it is not even necessary at all to specify a loop because a (declarative) set formula does the job.

Sometimes they have doubts about the result of a particular set formula. This is a problem that surfaces during the initial classes due to the many set and relational operators they need to learn. Some times you can see them figuring out if a given formula yields the desired result by computing some examples. Frequently they use universal quantifications when a quantification-free formula would work.

They experience problems in reconciling the specification with the implementation. Set-based notations are meant to produce a specification document separated from the implementation. From a Software Engineering perspective we completely agree with this approach. However, from an educational perspective we observe that students do not clearly see the role and implications of the specification document in the development process. Perhaps the major difficulty is to comprehend the relation between the specification and the design document (i.e. the document describing the software components, their functionality and the relationships between them [7, chapter 4]). For instance, they ask questions such as "is a state transition the specification of a method?", "where the state invariant must be implemented?", "should the caller be responsible for checking the precondition or should be the callee?".

We think that a combination of $\{log\}$, DBC and set-based specifications can help students in solving many of these problems making it easier for them to adopt the latter.

3 $\{log\}$ as an Aid for Set-Based Specifications

$\{log\}$ is a CLP language based on Prolog where sets are first-class entities [6, 13]. It supports all the classic set theoretic operators (such as union, intersection, and so forth), user-defined operators, partially specified sets (i.e. sets whose some of its elements are variables), etc. As a CLP language $\{log\}$ can decide the satisfiability of a large class of set formulas. Moreover, it can compute all the solutions of a satisfiable formula one after the other.

For example, (Set) can be translated into the following $\{log\}$ code:

$$\mathsf{dom}(Sa, D)\ \texttt{\&}\ (A\ \mathsf{in}\ D\ \texttt{\&}\ M > 0\ \texttt{\&}\ \mathsf{apply}(Sa, A, B)\ \texttt{\&}\ \mathsf{oplus}(Sa, \{[A, B]\}, NSa))\ \texttt{or}\ ((A\ \mathsf{nin}\ D\ \texttt{or}\ M =< 0)\ \texttt{\&}\ NSa = Sa)$$

which is operationally interpreted as a sequence of calls to predefined procedures that implement the basic operations on sets, partial functions and integers. If the $\{log\}$ interpreter executes this code it will end up in one of three ways (much as SAT or SMT solvers would do): returning a solution to the goal, answering "no", or getting a timeout.

For example, if $\mathsf{oplus}(Sa, \{[A, B]\}, NSa)$ is executed the first answer of the interpreter is[2] `NSa = {[A,B]\Sa}`, where `{[A,B]\Sa}` means $\{[A, B]\} \cup Sa$.

We have shown that a very general fragment of set-based specifications can be automatically translated into $\{log\}$ [5]. Consequently, students could compile their specifications into $\{log\}$ and use the interpreter to check if their formulas are what they meant.

Although $\{log\}$ is a declarative programming language, students may feel more comfortable with it than with plain set-based specifications since many of them are experienced programmers. Thus $\{log\}$ can help in reducing the gap between specifications and programs. Moreover, it could be used to describe contracts. For instance, (JML) can be expressed in $\{log\}$ terms as follows:

($\{log\}$-JML)

public normal_behavior
  requires $\mathsf{dom}(Sa, D)\ \texttt{\&}\ A\ \mathsf{in}\ D\ \texttt{\&}\ M > 0$
  ensures $\mathsf{apply}(Sa, A, B)\ \texttt{\&}\ \mathsf{oplus}(Sa, \{[A, B]\}, NSa)$
public exceptional_behavior
  requires $\mathsf{dom}(Sa, D)\ \texttt{\&}\ A\ \mathsf{nin}\ D\ \texttt{or}\ M =< 0$
  ensures $NSa = Sa$

where a simple naming convention between contract (or specification or $\{log\}$) variables and program variables can be established. Given that the $\{log\}$ predicates present in the requires and ensures clauses are executable programs, this

[2] We have simplified the output generated by the interpreter for presentation issues.

contract could in principle be used for runtime checking as the JML contract, and it can also be used for test case generation [5].

Although (JML) and ({*log*}-JML) may give the impression that contracts written in JML and {*log*} have a similar complexity, it is not true in general. {*log*} contracts will tend to be simpler because they can take advantage of a larger set of operators like domain restriction, relational image, and so forth. These operators are not implemented in the Java classes delivered with the standard library.

In a sense, ({*log*}-JML) is at an intermediate abstraction level between (JML) and (Set), enjoying some of the properties of both ends. Since students feel comfortable with programming languages, then taking them from an imperative programming language to contracts written in terms of sets to set-based specifications may help them in overcoming all the difficulties they find when they are directly introduced into the world of formal specification.

4 Teaching Proposal for Set-Based Software Specification

In this section we set forth a teaching proposal to introduce students into the world of set-based specifications. The proposal is based upon the idea of gradually taking students from DBC to formal notations such as Z and B. This proposal uses {*log*} as the linking tool between the world of DBC and that of set-based specifications.

Our proposal assumes that students:

- Are aware of a programming language such as Java;
- Understand the definition of program correctness. Specifically, they are conscious of: a) the existence of two distinct documents: the specification (S) and the implementation (P); and b) that S is authoritative over P; and
- If they have not had previous exposure to DBC, they will find it natural and intuitive as an extension of their programming skills.

Based on these assumptions, we propose the following teaching process:

1. Teach DBC.
   Pick the notation that best suits the programming skills of your audience. If your goal is to introduce students to set-based formal notations do not go deep into DBC. Recall that the final idea is that students use set-based specifications as contracts.
2. Teach set theory by means of {*log*}.
   We suggest to follow the work plan set forth by Abrial in Z and B [1]. That is, first teach basic set theory, then add Cartesian product and then binary relations and relational operators. Finally, introduce the concept of partial function and function application. Optionally show that sequences and bags (and their operators) can also be expressed in terms of set theory.
   Every time a new set theoretic concept is introduced show how it can be programmed in {*log*}. Do not necessarily go into the details of either Prolog or {*log*} programming; basic features will suffice for students to use {*log*} as a set calculator.

3. Teach how {*log*} can be used to describe program contracts.
   This is perhaps the most difficult step because it implies to force students to think in set terms rather than in implementation-level data structures. The best course of action we have found is to show to them that many implementation-level data structures either: a) obscure the essential properties of the data being represented because they were thought as efficient representations (but efficiency need not to be part of a specification); or b) usually the operations defined on their interfaces complicate the formalization of simple properties.
4. Teach a specification language such as Z or B and its relationship to {*log*} and DBC.
   Most text books on formal specification present specifications as documents with no relation with the software design. In general, a specification is presented as a formalization of the functional requirements, which must not talk about design features—such as information hiding, connectors, inheritance, etc. We agree with this view, although we think that it may be too alien for students used to program non trivial software. These students tend to think in terms of components with interfaces which implement some functionality. Therefore, we consider that the first students' approach to formal specification should be as a complement to a software design. More concretely, formal specification should be introduced as a technique to clearly and concisely document the functionality of design components. Particularly, the fraction of a set-based formal notation that can be automatically translated into {*log*} should be used as the DBC notation.
   The full set-based notation can be used if you do not plan to introduce some form of automatic verification, because it would need to link some tools.
   If the fraction mentioned above is used, our approach allows to automatically build prototypes from a combination of Z and user interface specifications [4]. This and other automatic verification activities may convince students about the added value of formal specifications since they can get for free software artifacts that otherwise cost a lot of resources.

5 Conclusions

As advocates of formal methods we permanently look for ways to make industry to adopt them more widely. One of the reasons we frequently see that impedes a wider use of formal methods is the relatively poor mathematical background of practitioners. This goes against techniques which require the explicit use of some form of formal methods.

In this paper we intend to provide teachers with a more gentle way to introduce students to the world of formal notations like B and Z. We propose to go from DBC, to set theory, to {*log*} and finally to a set-based notation. Consider that in order to fully use this teaching process some tools need to be developed.

However, our proposal might be just a patch. Maybe we should ask ourself whether programming must be taught before specification as we currently do—after all, our community has been advocating for years that, in development

projects, specification must precede implementation. Have we been mistaken all these years? Or are we teaching the other way around?

To formalize or not to formalize that is... not *the question: you are going to write code anyhow. The real question is: how many times are you going to formalize?*

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM 54(6), 81–91 (2011), http://doi.acm.org/10.1145/1953122.1953145
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and esc/java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 342–363. Springer (2005), http://dx.doi.org/10.1007/11804192_16
4. Cristiá, M., Rossi, G.: Rapid prototyping and animation of Z specifications using {*log*}. In: 1st International Workshop about Sets and Tools (SETS 2014). pp. 4–18 (2014), informal proceedings: http://sets2014.cnam.fr/papers/sets2014.pdf
5. Cristiá, M., Rossi, G., Frydman, C.S.: {log} as a test case generator for the Test Template Framework. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 8137, pp. 229–243. Springer (2013)
6. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000)
7. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering (2nd ed.). Prentice Hall (2003)
8. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. 55(1-3), 185–208 (2005), http://dx.doi.org/10.1016/j.scico.2004.05.015
9. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: Müller, P. (ed.) Advanced Lectures on Software Engineering, LASER Summer School 2007/2008. Lecture Notes in Computer Science, vol. 6029, pp. 91–139. Springer (2008), http://dx.doi.org/10.1007/978-3-642-13010-6_4
10. Meyer, B.: Touch of Class: Learning to Program Well with Objects and Contracts. Springer (2009), http://dx.doi.org/10.1007/978-3-540-92145-5
11. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. Computer 42, 46–55 (Sep 2009), http://portal.acm.org/citation.cfm?id=1638584.1638626
12. Oracle: Java^{TM} Platform, Standard Edition 7 – API Specification (1993), http://docs.oracle.com/javase/7/docs/api/, last access: 2014
13. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In: PPDP. pp. 219–229. ACM (2003)
14. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)