Animation of Object-Z Specifications with a Set-Oriented Prototyping Language

W. Hasselbring
Dept. of Computer Science (Software Technology), University of Dortmund
D-44221 Dortmund, Germany

Abstract

Within the computer science community, it is a well-known fact that the cost to correct an error in a computer system increases dramatically as the system life cycle progresses. The cost of correcting an error increases by orders of magnitude as the system moves from the development stages of analysis and design, to become most expensive during the maintenance and operation phase. Formal specification and prototyping help to eliminate many of these errors in the very early stages of a project before any production-level code has been written.

Formal methods are used to diminish ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during the costly testing and debugging phases. Even when using a formal specification one is left with the problem of validating the specification against the informal requirements. Consequently, after specifying the formal semantics of a proposed system a prototype should be built rapidly to validate the formal specification. Such a prototype enables us to *test* the specification with respect to its adequacy.

A common problem with software products is that the users of the system may not be fully aware of what they require and they may be unable to communicate their desires to the development team. Using a prototype, the user can interact with the system and can discover requirement deficiencies early, enabling rapid correction of the requirements.

The original goal of our research is to design a language for prototyping parallel algorithms to make parallel program design easier. We construct this tool in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken. In this paper we present the Object-Z specification of our parallel programming language and a prototype implementation with the set-oriented language ProSet. Note, therefore, that this paper presents more a case study than a fully developed methodology for the combination of a formal specification language and a prototyping language. The specified parallel programming language combines ProSet with Linda.

1 Introduction

Specification of the requirements is the phase in software construction which is concerned with the analysis of the tasks to be performed by the intended software system. Specifications and prototypes are the medium of communication

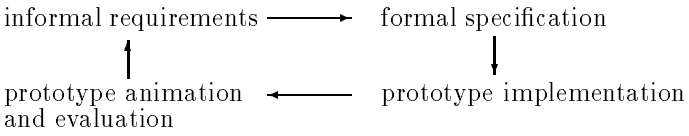
between developer and user of a proposed system. Formal specifications can help expose ambiguities and contradictions, because they force the specifier to describe features of the problem to be solved precisely and rigorously.

Let us take a closer look at the definition and implementation of programming languages, since our intention is to design and implement a language for parallel programming. Usually, the syntax (not the semantics) of programming languages is formally described with syntax diagrams or with Backus Naur Form. Therefore, they appear to be good candidates for a formal treatment. To discuss the advantages of formal specifications in the development of a new language, we briefly evaluate the development of the parallel programming language C-Linda [4].

The definition of C-Linda has been presented informally [4] and has included several ambiguities. [23] summarizes four basic types of process creation used in implementations of C-Linda's `eval` operation. These are different interpretations of the informal specification of the `eval` operation. Additional discussions of problems with the semantics of the `eval` operation may also be found in [17]. Such a situation demands a more precise definition. However, informal descriptions are very valuable because it is easy to obtain general understanding of the semantics with little effort.

To overcome the problems caused by the informal specification, several formalizations of the coordination language Linda have been undertaken. A comparative study of some approaches may be found in [5]. A Z specification has been presented in [3]. To avoid the problems which may be caused by an inconsistent informal specification, we present the formal semantics of our approach to parallel programming by means of the formal specification language Object-Z [10].

We *test* the specification against its adequacy through evaluation of a prototype. The prototype allows immediate *validation* of the specification by execution. The execution and presentation of the prototype is then called *animation* of the specification [6]. It is not possible to check the correspondence between informal requirements and formal specifications formally by *verification*. This situation suggests the following cyclic process for requirements analysis:



When the requirements reach a stable state, the next phases in the process of software production may begin. The next phases should be based on the formal specification and on the prototype implementation. When a system's requirements change after delivery, it is more appropriate to base changes in the maintenance phase on the flexible prototype instead on the optimized implementation, because optimizing transformations often introduce conceptual dependencies that increase the fraction of the code affected by a change in the requirements. In these cases, prototyping can reduce uncertainty and the number of times the production program must be changed before a satisfactory result is obtained. Refer to [12, Chapter 7] for an introduction to software production process models. The subject of this paper is the construction of prototypes from formal specifications and not the transformation of prototypes

into production-quality programs. Therefore, we consider here only the early phases in the process of software construction.

This paper presents an approach to animation of Object-Z specifications with ProSet, a procedural, set-oriented prototyping language [9]. There exist some other approaches to animation of Z and VDM specifications with PROLOG [6, 33], with functional languages such as Miranda [6] or ML [24], and with the database query language SQL [19]. It has also been proposed to make subsets of Z executable [13, 32]. For a debate on whether specifications themselves should be executable see [11, 18]. It should be noted that our prototype has been hand-coded from the formal specification without tool support. However, the construction of the prototype from the formal specification is straightforward (as we will see in Sect. 6). Before we develop tools supporting the construction of ProSet prototypes from Object-Z specifications, we would like to gain wider practical experience with this approach.

We discuss the relationship between formal specifications and prototyping in Sect. 2. Section 3 presents a brief introduction to the prototyping language ProSet and Sect. 4 specifies ProSet's features for parallel programming (ProSet-Linda) informally. The formal specification of ProSet-Linda and the prototype implementation are then presented in Sects. 5 and 6, respectively. Section 7 presents our conclusions.

2 Formal Specifications and Prototyping

Traditionally, most large software development projects are conducted along the lines of so-called life-cycle plans [1]. In life-cycle plans, the principle of proceeding "from the general to the concrete" or "from the problem to the solution" is applied to the organization of a software project. Each activity is viewed as an input-process-output step. Only minimal provision is made for feedback cycles. We refer to [2, 7] for critical assessments of the traditional life-cycle approach. Some of the reported problems are:

- Users are excluded from system development.
- Maintenance is unplanned system development.
- Life-cycle plans are unsuitable for project control.

To avoid these problems, we do not try to construct a production-quality program directly from our formal specification. Instead we intentionally build a prototype first.

Richard Kemmerer calls this "testing formal specifications" [21]. It is necessary to test specifications early to develop systems which meet their critical requirements and provide the desired functionality. Some of the functional requirements may not be known at design time. In fact, some functional requirements may only arise while evaluating the prototype.

First, a prototype helps the specification writer to debug the specification. It also helps a potential user to experience the capabilities of the system. It is often only through this type of experience that the necessary functional requirements can be discovered. Furthermore, it is better to have the user discover needs early in the production process, and not after the system has

been completely implemented and delivered. The prototype provides the user with a vehicle which can be exercised to see if it meets the (sometimes fuzzy) requirements. Users usually do not understand formal specifications, but they know how they want to use (the prototype of) the system.

Prototyping is used for requirements engineering, risk reduction, specification validation, increased user acceptance, and simplified maintenance of software systems [2, 7]. We want to note that a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer. Prototyping has been developed as an answer to deficiencies in the waterfall model, but it should not be considered as an alternative to this model. It is rather optimally useful when it complements the waterfall model. It is plausible that prototyping may be used during the early phases of the design.

The idea of prototyping is being adopted in software engineering for different purposes: prototypes are used *exploratively* to arrive at a feasible specification, *experimentally* to check different approaches, and *evolutionary* to build a system incrementally. Our approach to prototyping is an evolutionary development in versions. The prototype evolves in accordance with the changing environment. The ordering of development steps in the traditional life cycle model is mapped here into successive development cycles. Users are involved in the system development process which supports the communication between users and developers.

Prototypes should be built in very high level languages to make them rapidly available. To be useful, prototypes must be built rapidly, and designed in such a way that they can be modified rapidly. Consequently, a prototype is usually not a very efficient program since the language should offer constructs which are semantically on a very high level, and the runtime system has a heavy burden for executing these highly expressive constructs.

3 The Prototyping Language ProSet

The procedural, set-oriented language ProSet [9] is a successor to SETL [27]. ProSet is an acronym for Prototyping with Sets. This section will present a brief introduction to data and control structures of the language and a short example. The high-level structures that ProSet provides qualify the language for prototyping. For a full account to prototyping with set-oriented languages we refer to [7]. A case study for prototyping using SETL is documented in [22]. The application of SETL for prototyping algorithms for parallelizing compilers is described in [25].

ProSet provides data types for atom, integer, real, string, Boolean, tuple, set, function, and module values. It is a *higher-order* language, because functions and modules have first-class rights. ProSet is weakly typed, i.e., the type of an object is in general not known at compile time. Atoms are unique with respect to one machine and across machines. They can only be created and compared. The unary `type` operator returns a predefined type atom corresponding to the type of its operand. Tuples and sets are compound data structures, which may be heterogeneously composed. Sets are unordered collections while tuples are ordered. ProSet only supports finite sets and tuples, whereas Z also supports infinite sets. There is also the undefined value `om` which

$$
\begin{array}{|l}
n : \mathbb{N} \\
abs : \mathbb{Z} \longrightarrow \mathbb{N} \\
\hline
n \geq 2 \\
\forall\, z : \mathbb{Z} \bullet \\
\quad abs(z) = \textbf{if } z \geq 0 \textbf{ then } z \textbf{ else } -z
\end{array}
$$

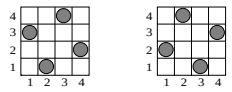
$$fields == (1 \,..\, n) \times (1 \,..\, n)$$

$$
\begin{array}{|l}
NonConflictingPositions : \mathbb{P}(\mathbb{P}\, fields) \\
\hline
\forall\, position : \mathbb{P}\, fields \bullet \\
\quad position \in NonConflictingPositions \Leftrightarrow \\
\qquad \#position = n \land \\
\qquad (\forall f1, f2 : position \bullet \\
\qquad\quad f1 \neq f2 \Rightarrow \\
\qquad\qquad first\, f1 \neq first\, f2 \land second\, f1 \neq second\, f2 \land \\
\qquad\qquad abs(first\, f1 - first\, f2) \neq abs(second\, f1 - second\, f2))
\end{array}
$$

Figure 1: Specification of the Queens Problem in Z.
abs yields the absolute value from a number. For $n = 4$ queens we have:

$$NonConflictingPositions = \{\{[1,3],[2,1],[4,2],[3,4]\}, \{[3,1],[1,2],[2,4],[4,3]\}\}$$

which corresponds to these positions:



indicates undefined situations. As an example consider the expression `[123, "abc", true, {1.4, 1.5}]` which creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals. This is an example of what is called a *tuple former*. As another example consider the set forming expression `{2*x: x in [1..10] | x>5}` which yields the set `{12, 14, 16, 18, 20}`. Sets consisting only of tuples of length two are called maps. There is no genuine data type for maps, because set theory suggests handling them this way. Maps are called *relations* in Z.

The control structures show that the language has ALGOL as one of its ancestors. There are `if`, `case`, `loop`, `while`, and `until` statements as usual, and the `for` and `whilefound` loops which are custom tailored for iteration over the compound data structures. The quantifiers ($\exists$, $\forall$) of predicate calculus are provided.

In Fig. 1, a specification of the so-called *Queens Problem* is given in Z. Informally, the problem may be stated as follows: "Is it possible to place n queens on an $n \times n$ chess-board in such a way that they do not attack each

```
program Queens;
   constant n := 4;
begin
   fields := {[x,y]: x in [1..n], y in [1..n]};
   put({Pos: Pos in npow(n,fields) | NonConflict(Pos)});

   procedure NonConflict (Position);
   begin
      return forall F1 in Position, F2 in Position |
                  (F1 /= F2) !implies
                    (F1(1) /= F2(1) and F1(2) /= F2(2) and
                    (abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2))));

      procedure implies (a, b); begin
         return not a or b;
      end implies;
   end NonConflict;
end Queens;
```

Figure 2: Solution for the Queens Problem in ProSet.
The predefined function `npow(k,s)` yields the set of all subsets of the set `s` which contain exactly `k` elements. The predefined function `abs` returns the absolute value of its argument. `NonConflict` checks whether the queens in a given position do not attack each other. It is possible to use procedures with appropriate parameters as user-defined infix operators by prefixing their names with the "`!`" symbol. This is done here with the procedure `implies`. `T(i)` selects the `i`th element from tuple `T`.

other?". Anyone familiar with the basic rules of chess also knows what *attack* means in this context: in order to attack each other, two queens are placed in the same row, the same column, or the same diagonal. In Fig. 2, a solution for the *Queens Problem* is given in ProSet. Our program does not solve the above problem directly. It prints out the set of all positions in which the n queens do not attack each other (the Z specification in Fig. 1 specifies this set of positions). If it is not possible to place n queens in non-attacking positions, this set will be empty. We denote fields on the chess-board by pairs of natural numbers for convenience (this is unusual in chess, where letters are used to denote the columns). `[1,1]` denotes the lower left corner.

Note that there are no explicit loops and that there is no recursion in the program. All iterations are done implicitly. Apart from syntactical differences, both, the Z specification and the ProSet program, seem to have roughly the same level of expressiveness. Therefore, one may regard this program also as an executable specification of the Queens Problem. However, note that we do not propose to start with an executable specification, because one may be induced to optimize the prototype instead of concentrating on the problem specification.

4 Informal Specification of ProSet-Linda

Prototyping means constructing a model. Since applications which are inherently parallel should be programmed in a parallel way, it is most natural to incorporate parallelism into the process of model building. Opportunities for automatic detection of parallelism in existing programs are limited and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Most systems in real life are of a parallel nature, thus the intent for integrating parallelism into a prototyping language is not only increased performance. If one wants to model an inherently parallel system, it is reasonable to have features for specifying (coarse-grained) processes that communicate and synchronize via a simple communication medium, and not to force such inherent parallelism into sequences. Our work intends to provide a tool for prototyping parallel algorithms and modeling parallel systems.

Linda is a coordination language concept for explicitly parallel programming in an architecture independent way [4]. Communication in Linda is based on the concept of *tuple space*, i.e., a virtual common data space accessed by an associative addressing scheme. Its access unit is the tuple, similar to tuples in ProSet (Sect. 3). Tuples live in tuple space which is simply a collection of tuples. It may contain any number of copies of the same tuple: it is a multiset, not a set. Tuple space is the fundamental medium of communication. Process communication and synchronization in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly. Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. We refer to [4] for a full account to programming with Linda.

ProSet supports multiple tuple spaces. Atoms are used to identify tuple spaces. As mentioned in Sect. 3 atoms are unique for one machine and across machines. They have first-class rights. Several library functions are provided for handling multiple tuple spaces dynamically. The function `CreateTS(limit)` creates a new tuple space and returns its identity (an atom). Since one has exclusive access to a fresh created tuple-space identity, `CreateTS` supports information hiding. The integer parameter `limit` specifies a limit on the expected or desired size of the new tuple space. This size limit denotes the total number of passive and active tuples, which are allowed in a tuple space at the same time. `CreateTS(om)` would instead indicate that the expected or wanted size is not limited. The function `ExistsTS(TS)` yields `true`, if `TS` is an atom that identifies an existing tuple space; else `false`. The function `ClearTS(TS)` removes all active and passive tuples from the specified tuple space. The function `RemoveTS(TS)` calls `ClearTS(TS)` and removes `TS` from the list of existing tuple spaces.

ProSet provides three tuple-space operations. The `deposit` operation deposits a new tuple into a tuple space, the `fetch` operation fetches and removes a tuple from a tuple space, and the `meet` operation. The `meet` operation is the same as `fetch`, but the tuple is not removed and may be changed.

5 Formal Specification of ProSet-Linda

The formal specification language Object-Z was chosen as a means for presenting the formal semantics of ProSet-Linda for several reasons. Firstly, Object-Z has many similarities with ProSet: both languages are based on set theory and predicate calculus. This alleviates the access to the formal specification for readers who are familiar with ProSet. Consequently, we capitalize on the similarities when constructing prototypes in ProSet from Object-Z specifications. Furthermore, there are some tools available to support the construction of specifications in Z [26].

A preliminary specification of ProSet-Linda using plain Z has been presented in [16]. Inferring the operation schemas that may affect a particular state schema requires examining the signatures of all operation schemas in plain Z. In large specifications this is impracticable. This problem and the absence of temporal logic notation in plain Z led us to consider the use of object-oriented extensions of Z to improve our preliminary specification. However, note that the small extract of the formal specification of ProSet-Linda, which we can present in this paper, cannot depict the advantages of Object-Z over plain Z.

There exist several object-oriented extensions of plain Z. [30] is a collection of papers describing various object-oriented approaches for Z — for example Hall's Style, ZERO, the Schuman & Pitt Approach, MooZ, OOZE, ZEST, Z^{++}, and Object-Z — mainly written by the methods' inventors, and all specifying the same two examples. Hall's Style and ZERO provide conventions for writing object-oriented specifications in plain Z. The Schuman & Pitt Approach is more concerned with fundamental issues of composition of schemas and reasoning about the resulting composition than with specifying object-oriented systems, or specifying systems in an object-oriented way.

Object-Z [10] extends Z by introducing a class structure which encapsulates a single state schema with the operations which may affect that state. Object-Z uses the class concept to incorporate a description of an object's state with related operations. Classes, and hence state operations, can be inherited by other classes. The model for a class is based on the idea of a history which captures the sequences of operations and state changes undergone by an instance (object) of the class. One of the advantages of Object-Z is that it enables such constraints to be incorporated directly within the model. Within plain Z, to specify liveness properties, histories must be explicitly defined separately from the state and operation schemas as it has been done in [16].

Z^{++} is very similar to Object-Z. Z^{++} also provides history predicates using temporal logic and therefore is an interesting candidate for our purposes. However, Z^{++} does not have a graphical display for schemas and classes. The idiosyncratic syntax is unfamiliar for readers familiar with the Z notation. This drawback leads us to the use of Object-Z. MooZ, OOZE and ZEST are other object-oriented extensions of Z, but they do not support history predicates with temporal logic. As the use of history predicates simplifies our specification significantly, we actually preferred Object-Z. To save space, we must refer the interested reader to [17] for a detailed discussion on this subject. We cannot present the entire formal specification in this paper

We use the following notational conventions: components of ProSet programs are displayed in `typewriter` font to set them apart from Z specifications, which are displayed in *slanted* font. We display identifiers in sans serif font when

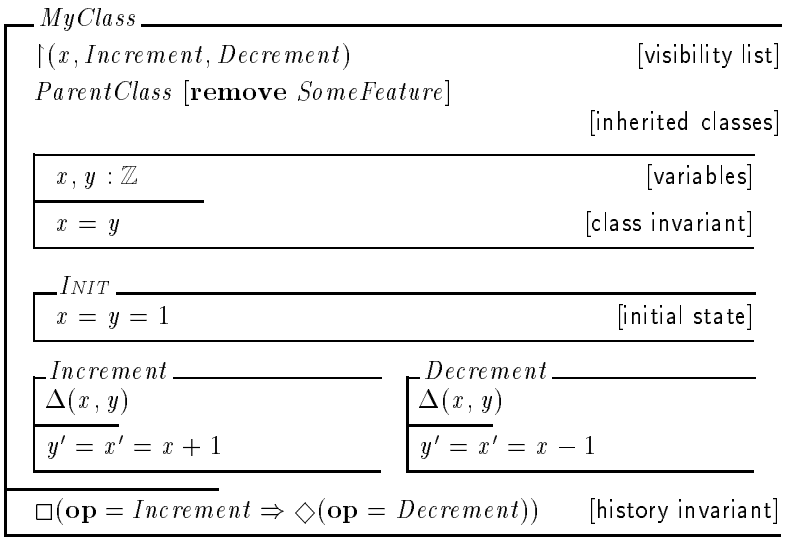


Figure 3: A simple class specification in Object-Z.

we use them as infix relation or operation symbols.

5.1 Object-Z

Refer to [6, 29] for an account of plain Z. In Object-Z, state variables and related operations are encapsulated into classes [10]. Inheritance facilitates the construction of complex specifications by allowing components to include the states and operations from simpler components. A simple example class specification in Object-Z is given in Fig. 3.

The [visibility list] gives those features externally visible to the clients of an object (users via instantiation). If none is given, the default is that all operations are visible. The symbol ↾ is used to indicate visibility. In our example, the state variable x, and the operations *Increment* and *Decrement* are externally visible to clients, whereas the state variable y is hidden.

The attributes (constants and state variables) and the operations are collectively called the *features* of a class. The [inherited classes] are the names of super-classes to be inherited; a subclass incorporates all the features of its super-classes, including their operations and invariants. Visibility is not inherited so that a derived class may nominate any inherited feature as visible. It is not possible in Object-Z to indicate which features of an object are available to its children (users through inheritance). Therefore, children always have access to all the features of their parents. However, children may optionally restrict their access via the keyword **remove** while inheriting a class, as it has been done in the example with *SomeFeature*. Under multiple inheritance, features having the same name are merged (semantic identification). The values of the

[variables] are constrained by the [class invariant]. The initial state schema is distinguished by having as its name the keyword *INIT*.

The operations are defined using schemas, in a way very similar to plain Z, defining a relation between before and after state. An operation's Δ-list contains a subset of the variables which are declared, either implicitly or explicitly, in both unprimed and primed form in the operations signature. The understanding is that when the operation is applied to an object of the class, those variables not in the list are unchanged. By convention, not indicating a Δ-list is equivalent to specifying an empty Δ-list.

The set of all possible histories is restricted by history invariants. Such history invariants are liveness and fairness properties, which explicitly restrict the set of allowable histories by means of temporal logic. The set of possible histories of a class is initially determined by the class state (including the initial state) and the allowable operations, and can then be further restricted by incorporating history invariants. In history invariants, the keyword **op** denotes the name of an operation in the history. The temporal logic notation can be used within history invariants. The history invariant in our example means that *always* after an *Increment* operation, *eventually* a *Decrement* operation must occur.

5.2 Basic Definitions

We specify generative communication in ProSet (ProSet-Linda) and not the entire language. Therefore, we need for the embedding in the computation part interfaces to some basic concepts of ProSet. The basic, given types for our specification are *Process*, *Statement*, *Expression*, *LValue*, and *Value*. We have to know a few specific things about types and values in our specification:

$$atom, boolean, integer, real, string, tuple, set, function, module : Value$$
$$TRUE, FALSE : Value$$
$$om : Value$$
$$ValuesOfType : Value \rightarrowtail\!\!\!\!\!\!+\!\!\!\rightarrow \mathbb{P}\ Value$$

$$ValuesOfType\ boolean = \{TRUE, FALSE\}$$
$$\mathrm{dom}\ ValuesOfType = \{atom, boolean, integer, real, string, tuple, set, function, module\}$$
$$\mathrm{dom}\ ValuesOfType \subset ValuesOfType\ atom$$
$$\mathsf{disjoint}\ \langle\{TRUE\}, \{FALSE\}\rangle$$
$$\mathsf{disjoint}\ \{\, t : \mathrm{dom}\ ValuesOfType \bullet t \mapsto \{t\}\,\}$$
$$(\{\, om \mapsto \{om\}\,\} \cup \{\, t : \mathrm{dom}\ ValuesOfType \bullet t \mapsto ValuesOfType\ t\,\})\ \mathsf{partition}\ Value$$

The Boolean values are `true` and `false` as usual. We use capital letters for *TRUE* and *FALSE* in our specification, because *true* and *false* are predefined predicates in Z. Every value in ProSet, except for `om`, belongs to exactly one type set (defined by the last property). Each type atom is mapped by *ValuesOfType* to the set of values which belong to the type it denotes. The type atoms themselves are atoms. The sets of values for the types are disjoint (see the definition of partition in [29, page 122]). The undefined value `om`, which

indicates undefined situations, has no type. The unary operator `type` yields a predefined type-atom according to the type of its operand. Applying the unary operator `type` to `om` is undefined, and thus yields the undefined value (`type om = om`). The corresponding function is *Type*:

$$
\begin{array}{|l}
Type : Value \longrightarrow Value \\
\hline
boolean = Type\ TRUE = Type\ FALSE \\
atom = Type\ atom = Type\ boolean = Type\ integer = Type\ string = \\
\qquad Type\ real = Type\ tuple = Type\ set = Type\ function = Type\ module \\
om = Type\ om \\
\forall x : Value \mid x \neq om \bullet \\
\qquad x \in ValuesOfType\,(Type\ x) \\
\end{array}
$$

For our purposes it is not necessary to specify the types of ProSet through an additionally given, basic type. It is sufficient to specify the semantics of the `type` operator.

Accordingly, we define tuples (with type *Tuple*) and the other basic concepts of ProSet-Linda. A tuple space then consists of an identity, its specified limit, and a bag of tuples:

TupleSpace

$$
\begin{array}{|l}
Id : Value \\
Limit : Value \\
Tuples : \mathrm{bag}\ Tuple \\
\hline
(Type\ Id = atom) \land (Id \notin \mathrm{dom}\ ValuesOfType) \\
Type\ Limit \in \{integer, om\} \\
\end{array}
$$

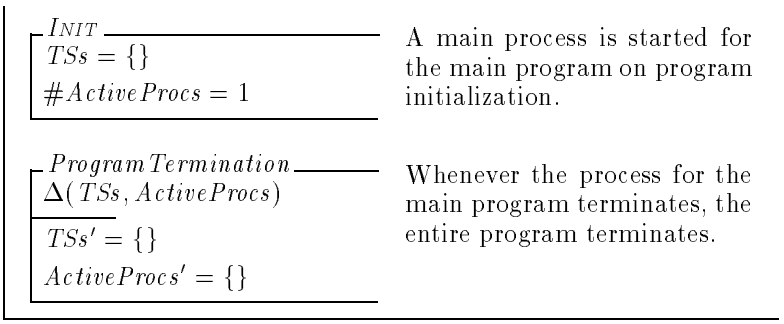
Tuple-space identities are atoms (not including the predefined type-atoms). The limit for the number of simultaneously deposited tuples in tuple space has to be an integer or the undefined value. A negative limit is equivalent to 0 (no tuples may be deposited into such a tuple space). The undefined value indicates that no limit has been specified on creation of the tuple space. This limit is a parameter to the function `CreateTS` (see Sects. 4 and 5.3). The main part of a *TupleSpace* it the bag of *Tuple*s.

Up to this point in our specification, we only used plain Z for our specification. Now we start with the object-oriented specification of program states. We view the state of a program as the state of a finite set of tuple spaces, and a finite set of active processes:

ProgramState

$$
\begin{array}{|l}
TSs : \mathbb{F}\ TupleSpace \\
ActiveProcs : \mathbb{F}\ Process \\
\hline
\forall ts1, ts2 : TSs \bullet \\
\quad ts1 \neq ts2 \Rightarrow ts1.Id \neq ts2.Id \\
\end{array}
$$

The class property asserts the uniqueness of tuple-space identities.



For a specification of the entire ProSet language, additional components would be necessary to specify the program state.

Initialization, unlike other operations, can only occur as the first operation and merely determines an initial state (there are no pre-conditions). Semantically, *Init* is interpreted as an operation for obtaining a uniform treatment of histories as sequences of events [10].

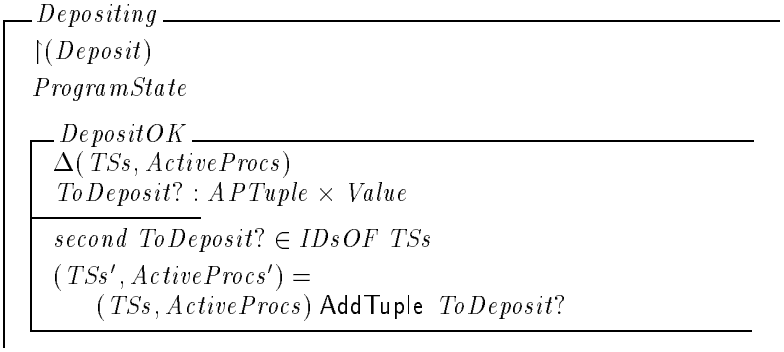
In principle, we could have defined classes earlier in our specification, for example a class for tuple spaces. Then, we would use object-instances of such a tuple-space class in our program state. But then we could not use the *f*uzz type-checker, which only accepts plain Z specifications [28]. In our specification in [17], we could not type-check the object-oriented extensions, in particular the class hierarchy, but all defined schemas with their formulas. If a type-checker for Object-Z were available, we would have used the object-oriented features of Object-Z more resolutely.

5.3 Handling Multiple Tuple Spaces

The library function `CreateTS` creates a new tuple space and returns the corresponding tuple-space identity, provided that the given limit is an integer or the undefined value. It is specified in Fig. 4. It inherits the features of *ProgramState*. Accordingly, we define the other library functions in [17].

5.4 Tuple-space operations

Depositing of tuples is defined as follows:



DepositInvalid

$ToDeposit? : APTuple \times Value$
$Exception! : Statement$

$second\ ToDeposit? \notin IDsOF\ TSs$
$Exception! =$ `'escape ts_invalid_id();'`

$Deposit \mathrel{\widehat=} DepositOK \lor DepositInvalid$

AddTuple has been defined in [17] to add a tuple to the program state. Accordingly, we define the other two tuple-space operations. We had to define some fairness properties for the tuple-space operations. The temporal logic operators of Object-Z are very helpful for this task. We refer the interested reader to [17] for details.

Concurrency is described by arbitrary interleavings of the atomic actions of the participating processes. However, nothing in the semantics given here prevents causal independent actions to occur in parallel. The concurrency of programs is modeled by the nondeterministic interleaving of atomic actions, i.e., an asynchronous model. Atomic transitions happen one after another in an arbitrary order.

6 A Prototype Implementation

After specifying the formal semantics of a proposed system a prototype should be build rapidly to validate the formal specification. We present in this section a prototype implementation of the runtime system for ProSet-Linda in ProSet itself.

ProSet is compiled and not interpreted. The compiler construction system Eli [14] is the central tool for implementing ProSet. Eli integrates off-the-shelf tools and libraries with specialized language processors to provide a system for generating complete compilers quickly and reliably. Our compiler compiles ProSet into C. We do not intend to present the compiler implementation here. This has been done in [8]. Instead we present the prototype implementation for the runtime system of the tuple-space operations and the library functions for handling multiple tuple spaces: the tuple-space management.

The prototype implementation for the tuple-space management is implemented in ProSet and has been translated by the compiler. Therefore, tuple-space operations in ProSet programs are translated by the compiler into calls to those procedures which are part of the tuple-space management. Fig. 5 displays the main part of the tuple-space management module. The program state is represented by a set of tuple spaces corresponding to the formal specification of the class *ProgramState* in Sect. 5.2. The set of active processes (*ActiveProcs*) is not represented in our prototype implementation, because the kernel of ProSet does not support parallel execution. Variables, which are declared as `visible` on the top level of modules, are *static variables* for instances of these modules. These variables are visible to each procedure within the corresponding module, and they are only visible to the encapsulated procedures. See [9] for an introduction to modules in ProSet.

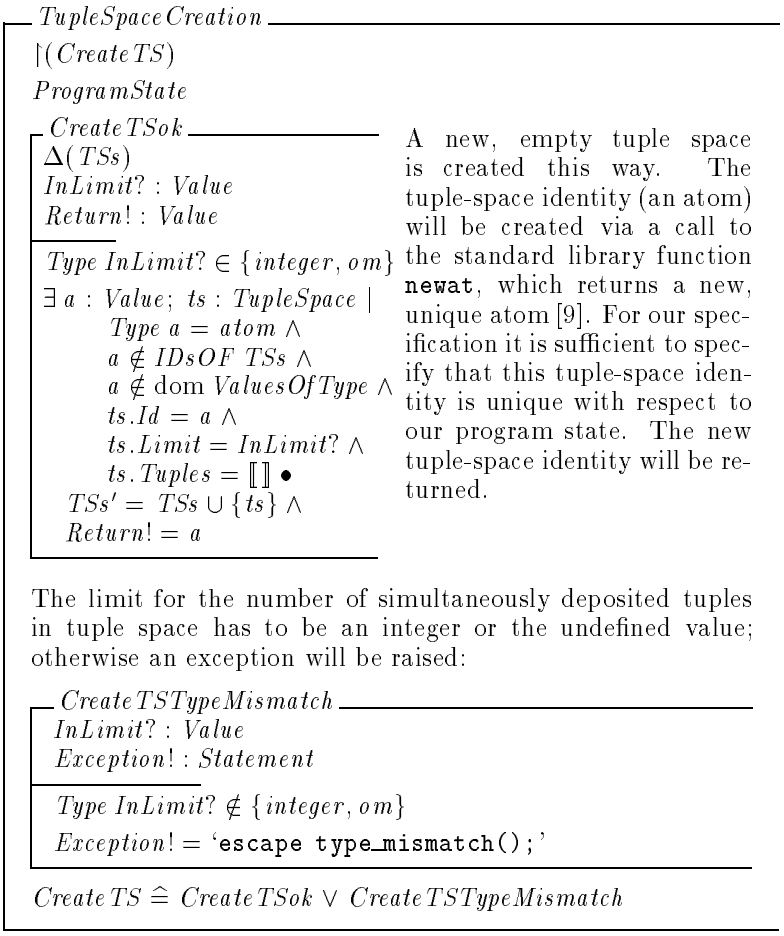


Figure 4: Specification of tuple-space creation.
IDsOF has previously been defined as a function which yields the set of tuple-space identities from a given set of tuple spaces.

6.1 Handling Multiple Tuple Spaces

The function for creating tuple spaces is displayed in Fig. 6. It belongs to the tuple-space management module in Fig. 5. The function `CreateTS` first checks the given `limit`. If the `limit` is not an integer or the undefined value, the exception `type_mismatch` will be raised via `escape`. We refer to [9] for a discussion of exception handling in ProSet.

Tuple spaces are implemented through tuples with three components. The position of these components corresponds to the position of the components in the schema *TupleSpace* in Sect. 5.2. The first component is the tuple-space

```
module TupleSpaceManager
  export CreateTS, ExistsTS, ClearTS, RemoveTS,
         Deposit, Fetch, Meet;

  visible TSs;
begin  -- the initialization of module instances:
  TSs := {};

  ...
end TupleSpaceManager;
```

Figure 5: The main part of the ProSet implementation for the tuple-space management.

identity. The second component is the specified limit. The third component of a tuple space implements the multiset of tuples in our prototype implementation. As we see in Fig. 6, this multiset is initially empty. See also the definition of *CreateTS* in Sect. 5.3. Because ProSet does not directly provide multisets, we model multisets of tuples through maps from tuples to counts. Z supports multisets (or bags) in the same way.

6.2 Tuple-Space Operations

The tuple-space operations are mapped to the respective procedures `Deposit`, `Fetch`, and `Meet` which belong to the tuple-space management. We only present `Deposit` here, and refer to [17] for a detailed discussion of the entire prototype implementation. Fig. 7 displays the procedure `Deposit`. First, the given tuple and the tuple-space identity are checked. If no exceptional situations are discovered, the tuple is added to the tuple space. See Fig. 7 for details and Sect. 5.4 for the definition of *Deposit*.

7 Conclusions

The goal of our research is to design a tool for prototyping parallel algorithms to make parallel program design easier. The high level of ProSet's constructs for parallel programming enables us to rapidly develop prototypes of parallel programs and to experiment with parallel algorithms. We construct this tool in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken.

A formal specification of ProSet-Linda has been presented. The specification of the formal semantics of generative communication in ProSet led us to the recognition of several omissions and imperfections in our previous informal specification, which has been presented in [15]. The main advantage of using a formal specification lies in subsequent development steps for the implementation. A formal development process is more expensive in terms of time and

```
procedure CreateTS (limit);
begin
  if limit /= om and type limit /= integer then
    escape type_mismatch();  -- raise an exception
  end if;
  newTS := [newat(), -- the new tuple-space identity
            limit,   -- the given limit
            {}       -- the bag of tuples
           ];
  TSs with:= newTS;
  return newTS(1);  -- the tuple-space identity
end CreateTS;
```

Figure 6: The library function for creating tuple spaces.
The predefined function `newat` returns a fresh atom. The `with` operator adds an element to a set or to the end of a tuple.

```
procedure Deposit (tup, tsid);
begin
  if not (exists ts in TSs | (ts(1) = tsid)) then
    escape ts_invalid_id(); -- not a valid tuple-space identity
  end if;

  myts := arb {ts: ts in TSs | ts(1) = tsid};

  TSs less:= myts; -- remove the tuple space
  myts(3)(tup) := if myts(3)(tup) = om then 1
                                      else myts(3)(tup)+1
                  end if;
  TSs with:= myts; -- insert the tuple space with the added tuple
end Deposit;
```

Figure 7: Depositing tuples.
The unary operator `arb` returns an arbitrary element from a set. The `less` operator removes a specified element from a set.

education, but much cheaper in terms of maintenance.

We implemented a prototype from the formal specification. The prototype allows immediate validation of the specification by execution. It is not possible to check the correspondence between informal requirements and formal specifications formally by verification. The prototype enables us to avoid the large time lag between specification of a system and its validation in the traditional model of software production using the life cycle approach.

The main part of the implementation work for the prototype was not the implementation of the tuple-space management. It was quite easy to write the ProSet procedures with the formal specification on the desk. The main work was the implementation of the compiler. Experience with using the compiler construction system Eli [14] for implementing the ProSet compiler is documented in [8].

We do not present proofs for the correspondence between the formal specification and the prototype implementation. Even textbooks on Z do not provide proofs when executable prototypes are developed from Z specifications. For instance, in [6] prototypes for Z specifications were constructed in Miranda and PROLOG. There, the Z specification is "straightforwardly translated into Miranda" (page 218) and the "PROLOG animation of it should be fairly clear" (page 223). No formal proofs are given for the translation of Z specifications into Miranda and PROLOG. The reasons for not providing proofs for the development into code are that such proofs are usually very complex and do not essentially increase the confidence in the correctness of the development. Since the implementation in a very high-level language — such as ProSet — is straightforward, we already have high confidence in the correspondence between the formal specification and the prototype. Additionally, programming languages like ProSet are not defined with formal semantics for programs in the sense that formal specification languages are defined with formal semantics for specifications. This difference makes it hard (and even impossible in practice) to formally specify the relations between programs and specifications.

However, while constructing the presented formal specification we recognized an important drawback of using Object-Z or Z with the *f*uzz package [28]: the principle of *definition before use*, which leads to a bottom-up development of the specification. It would be more natural to write and explain the specification in a top-down manner. The important point with Z is just that any specification must be written in a way such that its definitions can be ordered to satisfy the principle of definition before use [29, page 47]. This avoids recursive definitions in which a schema includes itself. Therefore, it should be possible to develop a tool for Z that allows the introduction of paragraphs in any order and ensures that the principle of definition before use can be satisfied. We propose a *f*uzz directive, which *announces* a forthcoming definition. The existing *f*uzz directives allow preliminary, invisible definitions, but the later final definitions cannot be type-checked because they are redefinitions of global names. One could copy the formulae of the final definition into the preliminary, invisible definition, but then it would become impractical to change the specification.

Additionally, it is not possible in Object-Z to indicate which features of a class are available to its children (users through inheritance). Therefore, children always have access to all the features of their parents. However, children may restrict their access while inheriting a class. Our proposal for enhancing

Object-Z is to split the definition of a class in private and public parts, as one can do in C++ [31]. This way one could hide auxiliary definitions from children and also from clients. We think that children should not be responsible for restricting their access while inheriting a class, but clients should be able to restrict their access to an instantiated object. This philosophy seems to be somewhat the opposite to the principles applied in Object-Z.

ProSet and Object-Z (or plain Z) appear to be a good combination for software engineering in general. However, there exist some significant differences between ProSet and Object-Z resp. Z:

- ProSet is weakly typed, whereas Z is strongly typed.
- ProSet programs are executable prototypes, whereas Z specifications are not executable.
- ProSet only supports finite sets, whereas Z also supports infinite sets.

An executable language is by definition more restricted in expressive power than a non-executable one, because its functions must be computable and are defined over domains with finite representations.

Prototypes for Z specifications are often constructed with functional or logic languages, but set-oriented programming techniques may be a more adequate choice for constructing prototypes from Z specifications than techniques from functional or logic programming. ProSet is a procedural language which also contains a Pascal-like subset that facilitates prototyping by allowing a program to be refined into successively finer detail while staying within the language: it is a wide-spectrum language. These features allow us to systematically transform prototypes into production-quality products. For functional or logic languages there is a somewhat wider gap to bridge to arrive at a production-level program. Production-level programs are usually written in procedural languages like C.

In [20], guidelines for the manual conversion of Object-Z specifications into C++ are proposed. This approach is somewhat similar to the approach presented in the present paper, but the differences in expressiveness are greater for the combination of Object-Z and C++ than for the combination of Object-Z and ProSet. However, to be successful in practice, appropriate tool support for the construction of ProSet prototypes from Object-Z specifications would be necessary.

Acknowledgements

The discussion with Keld Kondrup Jensen on formal semantics of Linda and the comments on a preliminary formal specification by Stephen Gilmore were very helpful. The comments on drafts of this paper by Ernst-Erich Doberkat, Kelvin Ross and the anonymous reviewers are gratefully acknowledged.

References

[1] Boehm B. Software engineering. IEEE Trans. Comput., 25(12):1226–1241, 1976.

[2] Budde R, Kautz K, Kuhlenkamp K, Züllighoven H. Prototyping — An Approach to Evolutionary System Development. Springer-Verlag, 1992.

[3] Butcher P. A behavioural semantics for Linda-2. Software Engineering Journal, 6(4):196–204, 1991.

[4] Carriero N, Gelernter D. How to write parallel programs. MIT Press, 1990.

[5] Ciancarini P, Jensen K, Yanklevich D. The semantics of a parallel language based on a shared dataspace. Technical Report 26/92, University of Pisa, Pisa, Italy, 1992.

[6] Diller A. Z: An introduction to formal methods. Wiley, 1990.

[7] Doberkat EE, Fox D. Software Prototyping mit SETL. Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989.

[8] Doberkat EE, Franke W, Gutenbeil U, Hasselbring W, Lammers U, Pahl C. A First Implementation of ProSet. In Kastens U, Pfahler P (eds), International Workshop on Compiler Construction CC'92 (Poster Session), pp 23–27. University of Paderborn, Informatik-Bericht Nr. 103, 1992.

[9] Doberkat EE, Franke W, Gutenbeil U, Hasselbring W, Lammers U, Pahl C. ProSet — A Language for Prototyping with Sets. In Kanopoulos N (ed), Proc. Third International Workshop on Rapid System Prototyping, pp 235–248, Research Triangle Park, NC, 1992. IEEE Computer Society Press.

[10] Duke R, King P, Rose G, Smith G. The Object-Z Specification Language: Version 1. Technical Report 91-1, University of Queensland, Software Verification Research Center, Queensland, Australia, 1991.

[11] Fuchs N. Specifications are (preferably) executable. Software Engineering Journal, 7(5):323–334, 1992.

[12] Ghezzi C, Jazayeri M, Mandrioli D. Fundamentals of Software Engineering. Prentice-Hall, 1991.

[13] Gimnich R, Ebert J. Zur Definition und Interpretation ausführbarer Spezifikationen. In Boley H, Furbach U, Lippe WM (eds), Sprachen für KI-Anwendungen — Konzepte, Methoden, Implementierungen, pp 150–160. Münster (Schriftenreihe), 1992.

[14] Gray R, Heuring V, Levi S, Sloane A, Waite W. Eli: A complete, flexible compiler construction system. Commun. ACM, 35(2):121–131, 1992.

[15] Hasselbring W. Combining SETL/E with Linda. In Wilson G (ed), Proc. Workshop on Linda-Like Systems and Their Implementation, pp 84–99. Edinburgh Parallel Computing Centre TR91-13, 1991.

[16] Hasselbring W. A Formal Z Specification of ProSet-Linda. Informatik-Bericht 04-92, University of Essen, 1992.

[17] Hasselbring W. Prototyping Parallel Algorithms in a Set-Oriented Language. PhD thesis, University of Dortmund, 1994. (in preparation).

[18] Hayes I, Jones C. Specifications are not (necessarily) executable. Software Engineering Journal, 4(6):330–338, 1989.

[19] Jabry Z, Austin S. An experiment in VDM to SQL translation. NPL Report DITC 193/91, National Physical Laboratory, Teddington, UK, 1991.

[20] Johnston W, Rose G. Guidelines for the Manual Conversion of Object-Z to C++. Technical Report 93-14, University of Queensland, Software Verification Research Center, Queensland, Australia, 1993.

[21] Kemmerer R. Testing formal specifications to detect design errors. IEEE Trans. Softw. Eng., 11(1):32–43, 1985.

[22] Kruchten P, Schonberg E, Schwartz J. Software prototyping using the SETL programming language. IEEE Software, pp 66–75, 1984.

[23] Narem J. An informal operational semantics of C-Linda V2.3.5. Technical Report 839, Yale University, New Haven, CT, 1989.

[24] O'Neill G. Automatic translation of VDM specifications into Standard ML programs. The Computer Journal, 35(6):623–624, 1992.

[25] Padua D, Eigenmann R, Hoeflinger J, Petersen P, Tu P, Weatherford S, Faigin K. Polaris: A new-generation parallelizing compiler for MPPs. CSRD Report No. 1306, University of Illinois at Urbana-Champaign, Urbana, IL, 1993.

[26] Parker C. Z tools catalogue. Technical Report ZIP/BAe/90/020, British Aerospace, Warton, UK, 1991.

[27] Schwartz J, Dewar R, Dubinsky E, Schonberg E. Programming with Sets – An Introduction to SETL. Springer-Verlag, 1986.

[28] Spivey J. The *f*UZZ Manual. Computing Science Consultancy, Oxford, UK, 2nd edition, 1992.

[29] Spivey J. The Z Notation: A Reference Manual. Prentice-Hall, 2nd edition, 1992.

[30] Stepney S, Barden R, Cooper D (eds). Object Orientation in Z. Springer-Verlag, 1992.

[31] Stroustrup B. The C++ Programming Language. Addison-Wesley, 1986.

[32] Valentine S. Z^{--}, an executable subset of Z. In Nicholls J (ed), Z User Workshop, York 1991, Workshops in Computing, pp 157–187. Springer-Verlag, 1992.

[33] West M, Eaglestone B. Software development: two approaches to animation of Z specifications using Prolog. Software Engineering Journal, 7(4):264–276, 1992.