NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-56040-1

A Case Study of the Formal Development of an Object Manager

Patrice Chalin

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

December 1989

ABSTRACT

A Case Study of the
Formal Development of an Object Manager

Patrice Chalin

In this thesis we motivate a formal approach to software development by discussing the use of formal notations in software engineering. The formal development of software consists of producing a series of formal specifications, where each specification contains slightly more detail than the one that precedes it. All specifications but the first have an associated proof of correctness that demonstrates that the specification satisfies the one that precedes it. A memory manager for a LISP-like interpreter is chosen as a sample problem for formal development. This leads to the formal development of an object manager consisting of a requirements specification, two levels of 'design specification' and a detailed specification of one of the allocation operations. All specifications are expressed in the Z notation. We conclude by indicating possible improvements and extensions to the object manager specifications and the Z notation.

Acknowledgements

I would like to thank my thesis supervisor, Dr. Peter Grogono, for his continued support and guidance during the period of research that has lead to the present thesis and during my undergraduate studies. Dr. Grogono has given me the opportunity to complement my theoretical studies by the next two, most important ingredients — *practice* and *experience*. From the beginning of my undergraduate studies, my involvement in the development of BIAS (a Scheme-like interpreter) has given me the practice and experience which are necessary for the appreciation of software engineering techniques. I am also thankful for the many stimulating and varied discussions which we have had.

I am grateful to the Natural Sciences and Engineering Research Council of Canada, and Concordia University for their financial support during my graduate studies and research.

Finally, I thank my wife, Sylvie, for her encouragement and support.

Table of Contents

1 Introduction

One of the goals of software engineering is the production of quality software. One of the most important software quality attributes is correctness. A software product is said to be *correct* if it meets its requirements.

Software engineering arose out of a need. It was recognized that the development techniques used for small programs could not be scaled up to be used in the production of larger software systems [Sommerville89]. Due to the size and complexity of larger software projects, special attention to management issues and to the process of software development is required [Fairley85]. Software process models, also called life-cycle models, have been developed which represent the development process as a series of phases where each phase is associated with certain activities having a specific purpose and with the goal of producing one or more deliverables [Fairley85, Sommerville89].

In section 2 we take a brief look at the current day use of formal methods in the various life-cycle phases. The discussion sets the background for a formal software development technique.

2 The Use of Formal Notations in Software Engineering

Most life-cycle models identify stages in software development which correspond to the following general phases:

1. Requirements analysis and definition.
2. Design.
3. Implementation.
4. Maintenance.

In the subsections which follow we consider the activities that are performed in each phase, the deliverables that are produced, and whether formal notations are used. We also discuss the verification and validation activities used to assure the quality of the deliverables produced. When possible, references are given to illustrate the use of formal notations in the deliverables relevant to the phase.

Life-cycle models differ in the amount of development effort that is allocated to each phase. In some models the phases may be performed more that once and in various orders (other than the ordering presented above). In practice the phases are generally not conducted one after the other but with a certain amount of overlap. For example, a prototype model may have implementation as an initial phase during which a prototype would be built. Prototype implementation would be performed during the early part of — and hence concurrently with — the requirements analysis and definition phase.

2.1 Requirements Analysis and Definition

2.1.1 Software Requirements Definition

The software development process is begun by the need for a system. That is, a customer identifies a *problem* for which a software solution is desired. The customer and supplier cooperate in the process of making explicit those environmental constraints and customer needs which are relevant to the problem [IEEE84]. These needs and constraints are documented in the form of *requirements*. A requirement is an unambiguous statement of a single environmental constraint or customer need which is required of any system which is

to solve the problem. The requirements are brought together in a *software requirements definition document* (SRDD) [Sommerville89].

The purpose of the SRDD is to describe completely and unambiguously (1) the problem to be solved and (2) the properties or constraints which any system must satisfy in order to be an acceptable solution to the problem. The desired quality attributes of the SRDD are: completeness, consistency, unambiguity, maintainability, traceability, and verifiability. The SRDD is the first formulation of requirements to be produced and it must be understandable to both the customer and the supplier [IEEE84, Sommerville89]. Thus it is generally agreed that the natural language used by experts from the problem domain is the most appropriate language for expressing requirements in the SRDD. Another argument in favor of expertise natural language is that most problem domains are too complex to be completely formalized. Thus, the rich and expressive vocabulary of expertise natural language is the preferred language for requirements definition [Abbott81, Sommerville89].

It has been recognized that a natural language statement of requirements cannot be used as a basis for design since the requirements tend to be ambiguous and incomplete [Meyer85a, Sommerville89]. Thus the requirements are subject to formal analysis [Sommerville89].

2.1.2 Software Requirements Specification

The next activity is the formal analysis and definition of some of the requirements in the SRDD. In general, only the functional requirements can be subject to formal analysis and definition [Sommerville89]. In this activity, one or more formal notations are chosen and an attempt is made to express the requirements using these notations. The act of formalizing the natural language requirements will generally bring to light ambiguities and omissions in the requirements [Meyer85a]. During this process, as errors are found in the requirements, the SRDD is reviewed and changed to correct the errors.

Formal notations permit precise analysis of the requirements through proofs. Theoretical issues such as consistency can be considered. A set of requirements is said to be *consistent* if there exists at least one system which satisfies those requirements. Requirements may also be *validated*, that is, they may be checked against actual customer needs and environmental constraints. Validation may be performed by establishing proofs of properties which the customer expects to be true. Research is being conducted in the area

of animation of specifications [Jones88a] which will provide another means by which requirements may be validated.

The activity of requirements specification serves as a development aid for the formulation of the natural language statement of requirements of the SRDD [Meyer85a]. This activity helps one achieve the quality attributes of completeness, consistency and unambiguity. Although completeness and consistency are theoretically achievable they are generally impossible to achieve in practice [Rich88]. This should not discourage us from trying to come as close as possible to achieving them. The results of the formal requirements specification are brought together in a *software requirements specification document* (SRSD). This document provides a formal basis from which a software design may be derived [Sommerville89].

Formal specification languages such as Z [Spivey89] and VDM [Jones86] can be used for the formal definition of software requirements. A collection of software requirement specifications written in the Z notation for various systems is presented in [Hayes87].

Although we have described the SRDD and SRSD as being two separate documents they are sometimes merged to form a *software requirements document* (SRD) [Sommerville89]. One may ask why it is important to put so much effort into the production of the software requirements. It is because the later in the life-cycle an error is caught the more expensive it is to correct. Also, the SRD acts as a contract between the customer and supplier [Sommerville89]: the customer agrees to accept any system which satisfies the SRD — that is, the requirements set forth in the SRD — and the supplier agrees to produce a system satisfying the SRD.

Note that the SRDD and SRSD (and hence the SRD) are descriptions of a *problem*. In the phases that follow requirements analysis and definition, the deliverables which are produced are descriptions of a *solution* to the problem identified in the SRD.

2.2 Design

2.2.1 Architectural Design

The software requirements, as captured by the SRD, are used as a basis for design. A preliminary design, called the *architectural design*, is conceived which identifies the general structure of the software to be developed [Pressman82]. The system structure is expressed as a collection of modules where each module has a specific purpose. Although the functional requirements determine what the modules will do, other requirements (such as performance constraints) constrain the ways that can be used to do it.

Architectural design is generally not expressed in a formal notation, possibly because there has been little research on the development of specialized formal notations for architectural design. General notations such as Z and VDM can be used.

Regardless of whether a formal notation is used, the architectural design must be shown to be adequate with respect to the SRD. The functional adequacy of the architectural design may be established by showing that the combined functionality of the design components meets the desired system functionality as expressed by the SRD. At this level of design it is not possible to determine if all requirements have been satisfied (e.g. performance requirements). Verification of the adequacy of the architectural design is established by design reviews and these are conducted using informal methods [Pressman82].

During architectural design, errors may be found in the SRD which may require the SRDD and consequently the SRSD to be reviewed and changed. Hence a certain amount of the project development effort may have to be reallocated to the requirements analysis and definition phase. While the SRD is being updated, work on the architectural design may continue or it may be halted. Some life-cycle models do not permit reiteration of the requirements analysis and definition phase. In most situations it is practical to reiterate to this phase only a small number of times [Sommerville89].

2.2.2 Intermediary Design

The architectural design may be refined in zero, one or more design steps. Each step results in the production of an intermediary design document. Each intermediary design document builds upon the design document which precedes it by adding more detail [Jones86, Sommerville89].

Although some authors admit no intermediary design steps [Pressman82], it seems impractical for most software projects to proceed from architectural design directly to detailed design. There is too large a gap between the two levels of design. Such an attitude is reminiscent of the times when it was thought that the design activity could be omitted altogether [Pressman82]— that is, implementation would commence immediately after requirements definition.

Much research has been devoted to the development of formal notations for intermediary design [Abbott81]. Some of these notations include: algebraic specification notations, grammars, and state-based specification notations. Examples of algebraic specification notations include Clear [Burstall80], OBJ [Goguen79] and NUSL [Jiang88]. These are simple notations which can be executed directly and are generally used to express abstract data types and simple interfaces [Sommerville89].

Grammars are formal notations which are used to describe the syntax of languages. Parsers for certain kinds of languages can be constructed automatically from a grammar for the language. One popular parser generator is YACC [Aho86]. Language semantics can be expressed in part by attribute grammars [Aho86]. Attribute grammars are also being used as a notation for the formal specification of structure editors [Ritchie88]. Tools now exists which generate a structure editor for a specific language given an attribute grammar for the language.

The general state-based notations Z and VDM can be used to express intermediary design although VDM appears have been developed specifically for system design.

As is illustrated in [Meyer85b], it is not necessary to restrict oneself to the use of formal notations that exist or that have a large user community. It is possible to invent a formal notation based on, as in [Meyer85b] for example, first order predicate logic.

Generally only part of the design is formalized, thus formal verification of intermediary designs is not possible for the entire design. The first intermediary design is verified against the architectural design and subsequent intermediary designs are verified against the intermediary design which precede it. Verification is done by design reviews and is generally conducted using informal methods.

A presentation of the formalization of the design of a display-oriented text editor written in Z is given in [Sufrin82].

2.2.3 Detailed Design

The last intermediary design or the architectural design (in the case where no intermediary design was produced) is refined into a detailed design. The detailed design is a generalized image of the source modules which will eventually be produced. Thus, all data structures are defined and all algorithms are explicitly expressed in a detailed design [Fairley85].

A semi-formal approach is generally taken to detailed design. There exist graphical notations for detailed design but these are equivalent to program design languages (PDL). A PDL is usually a derivative of a programming language. There are PDL's based on Pascal or ALGOL-like languages [Pressman82], Ada [Sommerville89], C, and Eiffel [Meyer88]. In any case, the detailed design language is usually different from the language(s) used in the intermediary design.

The detailed design is verified against the design from which it was derived. Again, this is done by design reviews and is conducted using informal methods such as inspections and walkthroughs [Pressman82, Fairley85].

- During the intermediary or final design stages errors may be found in the previous designs or in the SRD. This may require that the previous designs or the SRD be reviewed.

2.3 Implementation

The process of implementation consists of translating a detailed design into source modules written in one or more programming languages [Fairley85]. Much research has been done in the area of programming language semantics [Abbott81, Schmidt88]. Most common

programming languages have a formally or informally defined semantics [Ghezzi87]. Thus, programming languages can be considered as formal notations and source modules as formal documents. Hence, potentially, one can prove that a program unit (e.g. function or procedure) meets it specification. Although proofs are the ideal verification technique it is practical to prove the correctness of only small program units. The difficulty of such proofs is in the large amount of detail involved [Pressman82]

An implementation can be verified against the detailed design by review teams which conduct source code inspections and walkthroughs [Fairley85, Sommerville89, Pressman82]. Since programming languages are executable, partial verification of the implementation is also possible by unit testing and (after system integration) system testing [Fairley85, Sommerville89, Pressman82]. Verification by testing is partial since, as Dijkstra has said, testing can show the presence of errors but it can not establish their absence.

Similarly, since programming languages are executable, validation of the system or of its components is possible [Pressman82].

As was the case in the previous phases, errors may be found during implementation which have their origin in the SRD or any one of the design documents. If the error originates from the SRD then the SRD will need to be reviewed and any changes which are made may induce changes in the design documents and finally the implementation.

2.4 Maintenance

Maintenance involves a possible reiteration of the phases of requirement analysis and definition , design and implementation due to errors found in the software or changes in customer needs or environmental constraints [Fairley85, Sommerville89].

3 The Formal Development of Software

From the exposition above on the use of formalism in the software development process we see that formal notations exist which can be used in the production of deliverables for each of the phases. Thus there is a potential for the formal derivation of software.

An approach to software development which we will call *formal development* consists of using a formal notation for the derivation of a software system from the software requirements specification through to detailed design or implementation [Jones86, Spivey89]. The research described in this thesis is a case study of the formal development of an object manager using the Z notation.

The approach of formal development consists in producing a series of formal specifications $S_1, S_2, \ldots, S_n$. The first specification serves the same purpose as the software requirements specification document. It is a formal description of the functional requirements of a desired system. The last specification is a detailed design or implementation. Each specification S_{i+1} in the series contains slightly more detail that the specification S_i that precedes it. Along with each specification S_{i+1} there is a proof obligation: the specifier must prove that the specification S_{i+1} *satisfies* the specification S_i, that is, the system's functional behavior as expressed by the specification S_i is preserved by S_{i+1} [Jones86].

The method of formal development has the potential for guaranteeing the functional correctness of the implementation [Sommerville89]. The first specification is *the* definition of the behavior of the desired system. Each subsequent specification is proven to respect the functional behavior as described in the specification which precedes it. Thus, by transitivity, the implementation must behave according to the initial system specification (provided, of course, that all of the proofs are correct). The correctness proofs that arise are of manageable size and complexity since each specification is a slight refinement of its predecessor [Sommerville89, Jones88a]

The formal development approach also has the potential for reducing the cost of development and maintenance. The refinement of an inadequate design is a common cause of the wastage of development effort. The situation arises because refinement of the design is done before the adequacy of the design has been proven. Formal development

encourages developers to do proofs of correctness of a specification before proceeding to its refinement [Jones86, Jones88a]. Thus, errors in functionality should not get beyond the specification in which they were introduced. A product may be correct without being *valid*, that is, without satisfying actual customer needs. Such a situation arises because the software requirements do not correspond to actual user needs. The formal development approach is technically no better than other development methods with respect to the production of valid requirements. The proofs of correctness of the specifications establishes a dependency graph between the information contained in the specifications. It is therefore possible to know what effect a change in a specification S_i will have on the specifications S_j for $j \geq i$ [Jones88a].

The reuse of source code has not been as successful as anticipated [Meyer88]. This is mainly due to the application-specific nature of program source. Although object-oriented languages seem promising in this respect [Meyer88], it seems that the formal development of software is most likely to produce deliverables that are reusable. As formal methods become more popular we should see the emergence of abstract theories which provide useful properties for sets, sequences, bags, trees, stacks, etc [Jones88a].

Formal development is not widely used and probably will not be used by industry for some time. The main reasons are the large amount of proofs generated by the approach and the lack of automated tools to assist developers in the construction and management of proofs. Providing computerized assistance for formal software development methods is an active area of research [Jones88a]. Prototype systems have already been developed and major projects are near completion but tools are not yet available [Jones88a, Jones88b].

4 A Choice of Specification Language

For formal development to be possible there must exist a formal notation which can be used to express everything from a system's requirements to its detailed design. Two general purpose notations which can be used are VDM [Jones86] and Z [Spivey89]. In this section I explain my choice of Z for the case study.

Z is a mathematical specification notation based on typed set theory [Spivey88, Spivey89]. It is a general notation which can be applied to a wide range of problems at varying levels of abstraction — e.g. [Delisle89, London89, Woodcock89]. The Z notation is centered around the concept of *schema*. A schema combines the declaration of variables and predicates over these variables. The conjunction of the predicates defines a *property*. Schemas are used to describe both the static and dynamic aspects of a software system. The static aspects include the system state-space and initial states. Dynamic aspects include operations which transform the system states. Schemas are also used to describe the relationship between an abstract system state and a concrete implementation of the system state [Spivey88]. A Z specification consists of a series of paragraphs of formal text interleaved with informal prose which introduces and explains the content of the formal paragraphs.

Another general notation for writing specifications is the notation used for the Vienna Development Method (VDM) [Jones86]. The VDM notation is quite similar to Z but there are important differences.

For example, these two notations differ in their approach to undefined terms that occur in predicates. Consider the predicate $3/0 = 1 \wedge 3 < 2$ in which the term $3/0$ is undefined. Predicates in Z which contain undefined terms are called *undetermined* predicates [Spivey89]. Undetermined predicates have a truth value — they are either true or false — but it may not be possible to determine which value it is. This approach permits the use of classical two-valued logic. Thus, the predicate $3/0 = 1 \wedge 3 < 2$ is false because its second conjunct is false, regardless of the (undetermined) truth value of $3/0 = 1$.

VDM uses a three-valued logic. In addition to the usual truth values of true and false a new 'non-value' is used [Jones86]. The usual logical connectives, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$, must be redefined over these three values. The result is a logical system — called the logic of

partial functions (LPF) — in which classical tautologies, such as the law of excluded middle, no longer hold. LPF has its advantages; in particular, it is complete [Jones86]. Being most familiar with classical logic I prefer the two-valued logic of Z.

Operation specification in VDM tends to be more explicit than in Z. The VDM notation for operation specification requires the specifier to declare explicitly which components of the system state are being used along with the pre- and post-conditions of the operation. In Z, this information is usually implicit and can be derived if needed. For example, operations in Z are defined by a single property (i.e. conjunction of predicates). From this property is it possible to determine which components of the system are being used and what the pre- and post-conditions are [Spivey88].

I do not believe that a specification notation should *force* one to express information that can be derived. Being explicit tends to be the safer approach to specification, but only at the appropriate level of abstraction. One can be as explicit in Z as in VDM but one is free to be only as explicit as is deemed necessary.

The use of schemas allows Z specifications to be developed incrementally in a natural way [Spivey89]. For example, schemas can be used to define a complex state-space or operation as separate components and subsequently combine them. VDM does not have similar capabilities. This has prompted some authors [Sommerville89] to choose Z over VDM.

5 An Object Manager

In this section we present the formal specification of an object manager. This is the first step in its formal development.

5.1 Introduction

An important part of interpreters for LISP-like languages is the memory management unit. LISP systems are avaricious consumers of memory. Generally, the most complex subsystem of such a memory manager is the garbage collector. A garbage collector is concerned with reclaiming previously allocated but no longer needed memory so that it can be reused. Although memory reclamation is an effective means of managing memory, it is generally insufficient since repeated allocation of memory will cause fragmentation: at some point, an allocation request may fail because there is no single contiguous block of memory which is large enough, even though the total amount of free memory exceeds the amount requested. Thus, another activity of the garbage collector is memory compaction. The purpose of compaction is to bring the recycled memory into one contiguous block. The process will require the relocation of memory blocks and consequently the pointers referring to these blocks must also be changed.

Thus, LISP memory managers are non-trivial systems. I was responsible for the development of an interpreter for a Scheme-like language called BIAS [Grogono87] and was faced with the problem of implementing a memory manager. Following a traditional development method did not seem appropriate since this would mean that system validation could only be done by testing and adequate testing is impossible for a LISP memory manager — not to mention debugging! Thus the problem of memory management seemed like an excellent candidate for a case study of the formal development of a software system.

Grogono proposed a general mark-and-sweep memory allocation algorithm which was to be used in BIAS. The algorithm has three general phases which can be briefly described as follows:

1. Attempt allocation from free memory. If this is not possible then
2. recycle (i.e. collect) unused memory and retry allocation. If allocation is not possible then

3. compact all free memory into one block and retry allocation. If allocation is still not possible then allocation fails.

A memory manager controls the use of memory. Memory can be partitioned into *active* and *inactive* memory. Active memory is the memory that is currently in use by the LISP system. Inactive memory is the rest of memory and it is from this memory that new allocation requests are satisfied. Inactive memory can be partitioned into *free* and *garbage* memory. Free memory is the inactive memory that is readily available for allocation. Garbage memory is the allocated memory which is no longer needed but has not been recycled yet. Garbage memory, once recycled, becomes free memory.

A LISP interpreter uses memory to build S-expressions and other internal structures. These objects are made of discrete elements called *nodes* which are linked together by pointers. Nodes may contain *values* (such as integers, reals, strings, etc) and *pointers*. Conceptually, a LISP interpreter uses memory to construct a representation of a graph. The only information that is conveyed by a graph is the relationship that exists between the nodes of the graph. Consequently, the arcs that relate the nodes have no meaningful value. Said another way, the value of a pointer has no meaning; its only purpose is to link one node to another.

As mentioned above, memory must be periodically scanned for garbage that can be recycled. The memory manager needs a means of distinguishing active memory from garbage memory. For this purpose, memory managers usually contain a *pointer stack*. A node is considered to be *active* if and only if it is directly or indirectly accessible from a pointer on the pointer stack.

Memory must also be periodically compacted. Compacting involves the relocation of nodes. The relocation must be done is such a way that the graph, as seen by the LISP system, remains the same. This implies that when an active node is moved, all pointers to it must be updated to reflect the new location that the node will be moved to. Thus, for the purpose of compaction, the memory manager must be able to identify all pointers to active nodes. In many implementations it is assumed that the pointer stack is the only structure to contain pointers to active nodes. In those cases where the memory manager is called as a subroutine (as opposed to a coroutine) it is 'safe' to make copies of pointers from the pointer stack to local variables provided these variables are only used in *between* calls to the memory manager. If a pointer is kept in a local variable across a call to the memory

manager the variable may be left with a pointer to arbitrary memory. Therefore, use of memory and the pointer stack must be carefully controlled.

5.2 Evolution of the Specifications

Initially my objective was to specify formally and derive an implementation of a memory manager with a single operation for allocation. Time was spent searching for an appropriate model for the memory manager state and trying different formulations for an allocation operation. I continued with the refinement of the allocation operation which was to resemble the algorithm outlined above. The specifications went through numerous revisions as I learned the Z notation and found concise means of expressing important concepts.

As mentioned above, a very controlled use of both memory and the pointer stack must be made if the memory manager is to function properly. Thus, I realized that the memory manager specification needed to be extended to include operations which would allow for a disciplined use of memory and the pointer stack. At the same time I was also looking for an abstract model that would describe the behavior of the memory manager without concern for memory. The resulting model is that of an object manager which is presented in the sections which follow.

5.3 Informal Description of the Object Manager

At the heart of the matter is the definition of an *object*. An object is an entity which contains data. Two kinds of data are of concern: *values* and *references**. A reference is a link which relates one object to another. Any other data that is not used as a reference is a value. Without loss of generality, we assume that an object contains a single value and an ordered collection of zero or more references.

The purpose of the object manager is to permit an application to construct arbitrary labeled and ordered multidigraphs — a multidigraph is a directed multigraph [Roberts84] — without concern for memory management issues. We will use the term *graph* to mean labeled and ordered multidigraph. Each vertex in the graph corresponds to an object and the

---

* At this level of abstraction we avoid the use of the term 'pointer' and use the term 'reference' instead.

vertex label corresponds to the value contained in the object. An arc exists from an object A to an object B if and only if A contains a reference to B.

The object manager does not permit the direct manipulation of references. Instead it provides operations that permit the manipulation of an ordered collection of rooted subgraphs called the *reference stack.*

To summarize, the object manager maintains a graph which is constructed, altered and inspected by the intermediary of a reference stack. The object manager provides the following operations:

1. Push onto the reference stack:
   (a) by creating a new object
   (b) a reference already on the stack
   (c) a reference contained within another object.
2. Pop the reference stack.
3. Change the contents of the reference stack by assigning a reference from one location in the stack to another.
4. Change
   (a) the value of or
   (b) a reference contained in an object.
5. Get the value of an object.
6. Get the number of references contained in an object.
7. Compare references for equality.

5.4 Formal Specification of the Object Manager

5.4.1 Objects

An object contains a value and an ordered collection of zero or more references. The object manager specification can be expressed independently of the representation of the values or references contained in objects. We therefore assume that we are given two sets named VALUE_0 and $\mathrm{REFERENCE}_0$ which represent the set of all values and the set of all references, respectively, which an object can contain. In Z, sets like these are called *given*

sets (or *basic types*) and they are introduced into a specification by listing their names in between square brackets.

$[VALUE_0, REFERENCE_0]$

By the following schema we define an object as containing two components: a value and a sequence of references.

OBJECT

$value : VALUE_0$

$next : seq\ REFERENCE_0$

When used as an expression, the schema name OBJECT represents the schema type OBJECT which is the set of *bindings* which have two components named value (of type $VALUE_0$) and next (of type seq $REFERENCE_0$). Bindings can be thought of as the values of Pascal-like record types. If b is a binding of the schema type OBJECT then b.value denotes the value of its value component and b.next denotes the value of its next component.

5.4.2 The Object Manager State Space

The object manager will maintain a reference stack (represented by a sequence of references) and a graph (represented by a partial function from references to objects). Given a reference r, if the object referenced by r contains a reference s, then we say that s is a *next of kin* of r. The object manager state space is defined by the schema OBJ_MAN:

OBJ_MAN

$m : REFERENCE_0 \pfun OBJECT$

$stack : seq\ REFERENCE_0$

$Next\_of\_kin : REFERENCE_0 \leftrightarrow REFERENCE_0$

$Next\_of\_kin^*(ran\ stack) \subseteq dom\ m$

$\forall\ r,s : REFERENCE_0 \bullet Next\_of\_kin(r,s) \Leftrightarrow$

$\quad \{r,s\} \subseteq dom\ m \land s \in ran\ (m\ r).next$

A schema has two parts: a declaration part which is above the dividing line and a predicate part which is below the dividing line. The declaration part contains the declaration of one or more identifiers. The predicate part contains zero or more predicates separated by semicolons or line breaks. The predicates describe a property of the declared variables of the schema. For schemas representing a system state space, this property is called the system state *invariant*. If the predicate part is empty then the property is considered `true`, by convention.

The expression $\texttt{REFERENCE}_0$ `-+>` `OBJECT` denotes the set of all partial functions of references into objects. In the schema `OBJ_MAN` the variable `m` is declared to be one of these partial functions. The expression $\texttt{REFERENCE}_0 \leftrightarrow \texttt{REFERENCE}_0$ denotes the set of all binary relations over references. `Next_of_kin` is declared to be a relation over references.

We say that a reference `r` is *active* if `r` is on the stack or if `r` is the next of kin of an active reference. An object referred to by an active reference is called an *active object*. The expression $\texttt{Next\_of\_kin}^*$ denotes the reflexive-transitive closure of `Next_of_kin`. If `R` is a relation between `X` and `Y` and `S` is a subset of `X`, then the relational image of `S` through `R` is the set of all `y`'s related by `R` to some `x` in `S` and is denoted by `R(S)`. Thus, $\texttt{Next\_of\_kin}^*\texttt{(ran stack)}$ is the set of active references. The first predicate of the schema `OBJ_MAN` constrains the partial function `m` to be defined over at least the set of active references.

As part of the object manager specification we must identify the initial states of the system. This is accomplished by the schema `InitOBJ_MAN`.

InitOBJ_MAN

```
OBJ_MAN
---------
stack = ⟨⟩
```

The initial states of the system have an empty reference stack and hence contain no active objects.

5.4.3 Object Manager Operations

To create a new object one must specify the value it will contain and the objects it will be related to (that is, its next of kin). The operation `Push_new` creates a new object and pushes the reference to this object onto the reference stack.

```
Push_new
ΔOBJ_MAN
value? : VALUE0
next? : seq(INDEX ∪ {0})
---
ran next? ⊆ dom stack ∪ {0}
∃ r : REFERENCE0 \ (dom m) •
  stack' = ⟨r⟩ ^ stack ∧
  m' = m ⊕ {r ↦ μ OBJECT |
        value = value? ∧
        next = stack' ∘ succ ∘ next?}
```

The next of kin of the object to be created are identified by their position (or index) in the reference stack. The set `INDEX` is defined as an abbreviation for the domain of sequences, that is, the set of natural numbers — see appendix A.

The schema `ΔOBJ_MAN` in the declaration section alerts us to the fact that the `Push_new` operation changes the object manager state. This Δ-schema introduces the identifiers `m`, `stack`, `Next_of_kin`, `m'`, `stack'`, and `Next_of_kin'`. The first three identifiers are observations of the object manager state *before* the `Push_new` operation and the last three are observations of the object manager state *after* the operation. Both sets of identifiers are constrained to satisfy the object manager state invariant. It is a convention in Z to decorate input variables names with a question mark.

In Z, relations are modeled by their graphs. That is, a relation is represented by a set of ordered pairs. A function is a special kind of relation which relates each element in its domain to a single element in its range. The set `{ (1, 2), (2, 3), (3, 4) }` represents a function — it is a subset of the successor function — and can also be denoted as `{1 ↦ 2, 2 ↦ 3, 3 ↦ 4}`. The expression `x ↦ y` is called a *maplet* and it is an

abbreviation for the ordered pair (x,y). A sequence is modeled by a partial function from the set of positive integers into an element set. The function presented above is a sequence and can be written as ⟨2, 3, 4⟩. The symbol '^' is the infix operator for sequence concatenation. The domain (dom) and range (ran) of a sequence are the set of indices over which it is defined and the set of elements it contains, respectively. Thus dom⟨2, 3, 4⟩ = {1, 2, 3} and ran⟨2, 3, 4⟩ = {2, 3, 4}.

If f and g are functions then the expression f ⊕ g is a function which has as domain the union of the domains of f and g. Over the elements in the domain of g, f ⊕ g has the same value as g. Over the elements in the domain of f that are not in the domain of g, f ⊕ g has the same value as f. Therefore the predicate m′ = m ⊕ {r ↦ obj} says that the function m′ has as domain (dom m) ∪ {r}. For each element s in this domain, m′(s) is obj if s = r and m(s) otherwise. The expression μ OBJECT | value = v ∧ next = n denotes the unique binding of the schema type OBJECT whose value component is equal to v and whose next component is equal to n. Finally, the symbol '\' is the set difference operator and '∘' denotes the functional composition operator.

The following operation pushes a reference to an object that already exists. The reference must be somewhere in the reference stack. 'which?' is the stack index of this reference.

```
Push_old
ΔOBJ_MAN
which? : INDEX
---
which? ∈ dom stack
stack′ = ⟨stack which?⟩ ^ stack
m′ = m
```

If f is a function then f(x) and f x both denote the function f applied to an argument x. The expression f(x) actually denotes the application of the function f to the argument (x) and (x) is simply x.

The Push_kin operation pushes a reference which is the next of kin of a reference in the reference stack. If an object has next of kin, then each of its next of kin has a specific

position in the sequence of references `next`, contained in the object. The *kinship* of a next of kin is the index of the next of kin in the sequence `next`.

```
Push_kin
ΔOBJ_MAN
which?, kinship? : INDEX
---
which? ∈ dom stack
kinship? ∈ dom (m(stack which?)).next
stack' = ⟨(m(stack which?)).next kinship?⟩ ^ stack
m' = m
```

The `Pop` operation is used to remove the top-most reference from the reference stack. The empty sequence is denoted by ⟨ ⟩. If a sequence is not empty then its *head* is the first element of the sequence and its *tail* is the sequence that results when the head is removed.

```
Pop
ΔOBJ_MAN
---
stack ≠ ⟨⟩
stack' = tail stack
m' ⊆ m
```

The following operation is used to assign a reference from one location (`from?`) in the reference stack to another location (`to?`).

```
Assign
ΔOBJ_MAN
from?, to? : INDEX
---
from? ≠ to?
{from?, to?} ⊆ dom stack
stack' = stack ⊕ {to? ↦ stack from?}
m' = m
```

Given the index (`which?`) of a reference in the reference stack and a value (`value?`), the operation `Change_value` changes the value of the object referenced by `stack which?` to `value?`.

```
Change_value
ΔOBJ_MAN
which? : INDEX
value? : VALUE0
---
which? ∈ dom stack
m' = m ⊕ {stack which? ↦ μ OBJECT |
        value = value? ∧
        next = (m(stack which?)).next}
stack' = stack
```

The `Change_kin` operation is used to change a next of kin of an object. To make the operation specification more readable we introduce the local variable `obj` as a shorthand for `m(stack which?)`.

```
Change_kin
ΔOBJ_MAN
which?, kinship?, new_kin? : INDEX
obj : OBJECT
---
{which?, new_kin?} ⊆ dom stack
obj = m(stack which?)
kinship? ∈ dom obj.next
m' = m ⊕ {stack which? ↦ μ OBJECT |
        value = obj.value ∧
        next = obj.next ⊕
              {kinship? ↦ stack new_kin?}}
```

The Get_value operation is used to extract the value contained within an object. The Z convention is to decorate output variable names with an exclamation mark.

```
Get_value
ΞOBJ_MAN
which? : INDEX
value! : VALUE0
---
which? ∈ dom stack
value! = m(stack which?)).value
```

The Get_num_kin operation is used to obtain the number of kin that an object has.

```
Get_num_kin
ΞOBJ_MAN
which? : INDEX
num_kin! : N
---
which? ∈ dom stack
num_kin! = #(m(stack which?)).next)
```

The Compare operation is used to compare two references on the reference stack for equality. The result of the operation is of type COMPARISON which consists of the finite set {Same, Different}.

$$COMPARISON ::= Same \mid Different$$

Compare

$$\Xi OBJ\_MAN$$
$$i?, j? : INDEX$$
$$result! : COMPARISON$$

$$\{i?, j?\} \subseteq \mathrm{dom}\ stack$$
$$stack\ i? = stack\ j? \Rightarrow result! = Same$$
$$stack\ i? \neq stack\ j? \Rightarrow result! = Different$$

It is important to notice that the manipulation of references is confined to the object manager: none of the object manager operations accept references as input or return references as output. It is this controlled use of references that permits a practical object manager to be built. Since Push_new is the most difficult operation to implement, subsequent development is concerned with the refinement of this operation.

6 A Memory Manager

In this section we present a series of specifications. Each specification builds upon the previous one and gradually introduces a memory manager which will satisfy the object manager specifications. The sequel does not contain proofs of correctness. Instead the specifications are carefully derived — a method that is called *design by decomposition* [Jones86] — and emphasis is placed on proving properties of the specifications.

Section 6.1 presents basic concepts and terminology for the formalization of memory and the most elementary specification of the memory manager state space. Section 6.2 builds upon the specifications of section 6.1 by introducing the concepts of active and inactive memory which leads to a more detailed specification of a memory manager state and the definition of an allocation operation. Section 6.3 introduces free and garbage memory as two kinds of inactive memory. This permits allocation to be decomposed into three separate steps: allocation from free memory, garbage collection, and compaction of free memory. The operation of 'allocation from free memory' is refined in section 6.4.

6.1 Formalizing Memory

A memory manager controls the use of memory. Thus, our first objective is to describe what memory is. We provide an abstract model of memory which has those properties which are of interest to us in the description of the memory manger.

Memory is modeled by a finite partial function. The domain of the function is a set of elements called *references*. We have chosen to represent references by natural numbers.

```
REFERENCE == ℕ
```

The paragraph above introduces the identifier REFERENCE into the specification as an abbreviation for the set of natural numbers (ℕ). References are mapped into elements called *nodes*. A node contains a value and an ordered collection of zero or more references.

```
[VALUE]
NODE ≙ [value : VALUE; next : seq REFERENCE]
```

The form of the values contained in nodes is unimportant, hence VALUE is introduced as a basic type. The definition of the schema NODE is an example of the use of the *horizontal form* for schema definitions. Like OBJECT, NODE is defined to have the two components named `value` and `next`.

```
MEMORY == REFERENCE -#-> NODE
```

The identifier MEMORY is used as an abbreviation for the set of finite partial functions from references into nodes.

If `r` is a reference, `n` is a node and `m` is a memory such that `m r = n` then we say *`n` is the node referenced by `r` under `m`.* A reference is *valid* with respect to a given memory if and only if it is in the domain of the memory.

```
valid_reference : MEMORY ↔ REFERENCE
─────────────────
valid_reference(mem,ref) ⇔ ref ∈ dom mem
```

It will be convenient to have a type which groups together a reference and a memory such that the reference is valid with respect to the memory. The following schema will serve this purpose.

```
REF
mem : MEMORY
ref : REFERENCE
─────────────
valid_reference(mem, ref)
```

If `r` is a binding of type REF then we may say *`n` is the node referenced by `r`* when we mean *`n` is the node referenced by `r.ref` under `r.mem`.* The Z notation does not include syntax for constants of schema types, hence we define a constructor function for bindings of type REF.

```
mk_REF : MEMORY × REFERENCE -+> REF
node : REF → NODE
─────────────────
mk_REF = {REF • (mem, ref) |→ θREF}
node r = r.mem r.ref
```

The use of the schema name REF inside the set expression is an abbreviation for the text of the schema. Thus

```
{REF • (mem, ref) |→ θREF} = {mem : MEMORY; ref : REFERENCE |
        valid_reference(mem, ref) • (mem, ref) |→ θREF}
```

The expression θREF denotes a binding whose component values are taken from values of the corresponding variables in the surrounding scope. Hence the binding mk_REF(m, r) can also be expressed as μ REF | mem = m ∧ ref = r, that is, the unique binding of type REF whose mem component is m and whose ref component is r. 'node r' is defined to be the node referenced by r.

A memory can be used to store only a finite number of nodes. This is because memories are of finite size and each stored node occupies part of the available space. We assume that every node has a unique positive integer as its size and that any two nodes containing the same value and the same number of references have the same size. These assumptions are captured by the following function.

```
size : NODE → ℕ1
─────────────────
∀ n1, n2 : NODE |
  n1.value = n2.value ∧
  #n1.next = #n2.next •
        size n1 = size n2
```

$\mathbb{N}_1$ denotes the set of positive integers. The predicate ∀ x : T | P • Q is true if and only if for all x of type T such that the predicate P is true, the predicate Q is also true. Thus ∀ x : T | P • Q is logically equivalent to ∀ x : T • P ⇒ Q. Similarly the predicate ∃ x : T | P • Q is true if and only if there exists an x of type T such that P and Q hold. This predicate is equivalent to ∃ x : T • P ∧ Q.

The memory space occupied by a node will be represented by an interval of references called a *block*. The *lower bound* and *upper bound* of a block are the smallest and largest values of the interval, respectively. The set of blocks is BLOCK and the set of nonempty blocks is BLOCK1.

```
BLOCK, BLOCK1 : ℙ(ℙ REFERENCE)
─────────────────────────────
BLOCK = {r,s : REFERENCE • r..s}
BLOCK1 = BLOCK\{∅}
```

ℙ X denotes the power set of X. If a and b are integers then a . . b is the set of all integers between a and b inclusive.

Given a binding r of type REF, we define the block (of memory) occupied by the node referenced by r to be the interval r.ref . . (r.ref + size(node r) - 1). The function block maps bindings of type REF to their blocks and the functions lower and upper map bindings of type REF to the lower bound and upper bound of their blocks, respectively.

```
block : REF → BLOCK1
lower, upper : REF → REFERENCE
──────────────────────────────
block r = lower r .. upper r
lower r = r.ref
upper r = lower r + size(node r) - 1
```

Two bindings of type REF are said to *share* memory if their blocks overlap.

```
share : REF ↔ REF
─────────────────
share(r,s) ⇔ block r ∩ block s ≠ ∅
```

Two bindings of type REF are said to be *adjacent* if their blocks are side-by-side.

```
adjacent : REF ↔ REF
```

```
adjacent(r,s) ⇔
 ¬share(r,s) ∧
 block r ∪ block s ∈ BLOCK1
```

The functions `block`, `lower`, and `upper` and the relations `share` and `adjacent` accept arguments of type `REF`. Recall that a binding of type `REF` has as components a reference and a memory over which the reference is valid. As part of the schema describing the memory manager state space (to be defined below) there will be a memory represented by the variable mem. In most situations (e.g. when defining memory manager operations or in lemmas and proofs), bindings of type `REF` will be used as arguments to the above mentioned functions and relations, and we will want the memory component of the bindings to be equal to mem. Hence we define the schema `MEM` which has as components a memory called `mem` and functions and relations (corresponding to `block`, etc.) which accept arguments of type `REFERENCE`.

MEM

```
mem : MEMORY
Block : REFERENCE -+> BLOCK1
Lower, Upper : REFERENCE -+> REFERENCE
Size : REFERENCE -+> N1
Share : REFERENCE ↔ REFERENCE
Adjacent : REFERENCE ↔ REFERENCE
```

```
dom Block = dom Lower = dom Upper = dom Size = dom mem
∀ r : REFERENCE •
 Block r = block(mk_REF(mem,r)) ∧
 Lower r = lower(mk_REF(mem,r)) ∧
 Upper r = upper(mk_REF(mem,r)) ∧
 Size r = size(mem r)
∀ r,s : REFERENCE •
 Share(r,s) ⇔ share(mk_REF(mem,r), mk_REF(mem,s)) ∧
 Adjacent(r,s) ⇔
       adjacent(mk_REF(mem,r), mk_REF(mem,s))
```

For example, `Block` maps a reference `r`, which is valid with respect to `mem`, into the block occupied by the node referenced by `r` under `mem`. `Block`, `Lower`, and `Upper` are partial functions since they are defined only over those references which are valid with respect to `mem` — that is, they are defined over the domain of `mem`. The schema `MEM` is a key schema of this section and those that follow. Lemma `MEM_1` in Appendix B identifies expected properties of the functions and relations defined in `MEM`: for example, `Block r = Lower r .. Upper r`.

6.1.1 The State Space

In the first description of the memory manager state space the following observations are captured:

1. The content of memory depends on an interpretation. The interpretation (meaning) function is `mem`. It identifies which references are valid and which nodes are related to the valid references.

2. Memory is finite. The variable `memory_z` represents the size of memory and `mem` is a *finite* partial function.

3. Memory consists of one block. The block is represented by the variable `mem_block`. The bounds of the block are given by the variables `lower_bound` and `upper_bound`.

4. For every node, the block occupied by the node is completely contained within the memory bounds.

5. No two nodes can share memory.

The state space schema is `MM`:

```
MM
MEM
memory_z : ℕ1
lower_bound, upper_bound : REFERENCE
mem_block : BLOCK
---
upper_bound = lower_bound + memory_z - 1
mem_block = lower_bound .. upper_bound
dom mem ⊆ mem_block
∀ r : dom mem • Block r ⊆ mem_block
∀ r,s : dom mem | r≠s • ¬Share(r,s)
```

Let S and T be schemas such that T includes the schema name S in its declaration section. This has the same effect on the declaration section of T as textually including the declarations of S in T. The property of T is the conjunction of the predicates in S and T. Thus, the schema MM implicitly declares components Block, Lower, etc and their behavior is described by the property of the schema MEM.

In initializing the memory manager we are presently only concerned with assigning a definite value to the memory block.

```
InitMM
MM
memory_z? : ℕ1
lower_bound? : REFERENCE
---
memory_z = memory_z?
lower_bound = lower_bound?
```

6.1.2 Properties

One of the advantages of formal notations is that we can ask precise questions and (in most cases) get precise answers. In this section we present properties — and their proofs — of the functions defined above and of the formalization of the memory manager given so far.

There is a large gap between a concept and a statement which is meant to express the concept. It is for this reason that it is important to state properties of a specification which are expected to be true and to prove them. By proving properties of a specification we can increase our confidence in its correctness: *proofs are an indispensable validation tool.*

It would seem obvious, for example, that the size of the block occupied by a node should be the same as the size of the node because this corresponds to our intuitive understanding of a node's size. Thus the following property — stated formally as a lemma — should be true:

```
Lemma Block_size.
  ∀ r : REF • #(block r) = size(node r)
Proof
  Let r be a binding of type REF — that is, r : REF.
1.  1 ≤ size(node r)                                   [ran size = N1]
2.  lower r ≤ lower r + size(node r) - 1                [1,arithmetic]
3.  lower r ≤ upper r                          [2,definition of upper]
4.  #(lower r .. upper r)
        = upper r - lower r + 1                  [3,property of _..._]
5.      = lower r + size(node r) - 1 - lower r + 1 [def'n of upper]
6.      = size(node r)                                   [arithmetic]
7.  #(block r) = size(node r)                          [6,def'n block]
QED
```

The given proofs are rigorous rather than formal. Proof lines are followed by comments in brackets which justify them. If a proof line is dependent on another, then the line number of the latter will be part of the comments which justify the former.

The above lemma confirms the relationship that should exist between the functions `block` and `size`. Corollary Block_size demonstrates that the same relationship holds between the corresponding functions `Block` and `Size`.

```
Corollary Block_size.
  MEM ⊢ ∀ ref : REFERENCE | valid_reference(mem, ref) •
            #(Block ref) = Size ref
```

```
Proof
  Assume   MEM; ref : REFERENCE | valid_reference(mem, ref).

  valid_reference(mem, ref)
  ⇒ ref ∈ dom mem                                  [def'n valid_reference]
        = dom Block                                                [MEM]
        = dom Size                                                 [MEM]

  Thus Block ref and Size ref are well-defined expressions.

  #(Block ref)
    = #(block mk_REF(mem,ref))                                     [MEM]
    = size(node mk_REF(mem,ref))                    [Lemma Block_size]
    = size(mem ref)                                [def'n node, mk_REF]
    = Size ref                                                     [MEM]
QED
```

As was mentioned in section 4, it is possible for an expression to be undefined and hence for a predicate to be undetermined — i.e. have an unknown truth value. As is demonstrated in the proof above, care must be taken when writing proofs to insure that expressions are well defined.

The following lemma confirms that the size of the memory block is equal to the memory size.

```
Lemma Mem_block_z.
  MM ⊢ #mem_block = memory_z
Proof
  memory_z ≥ 1                                                     [MM]
  ⇒ 0 ≤ memory_z - 1                                          [arith.]
  ⇒ lower_bound ≤ lower_bound + memory_z - 1                   [arith.]
  ⇒ lower_bound ≤ upper_bound                       [def'n upper_bound]
  ⇒ #(lower_bound..upper_bound)
        =  upper_bound - lower_bound + 1             [property of _..._]
        =  memory_z                                 [def'n of memory_z]
```

```
  ⇒ #mem_block = memory_z                          [def'n of mem_block]
QED
```

A property which will be used often in subsequent proofs is that fact that the function Block is injective. The expression REFERENCE >+> BLOCK1 is the set of partial injections from references into nonempty blocks.

```
Lemma Block_injective.
  MEM ⊢ Block ∈ REFERENCE >+> BLOCK1
```

The notation 'p.116#9' used in the proof of lemma Block_injective indicates that the ninth 'law' on page 116 of [Spivey89] is used to justify the proof line. A collection of 'laws', such as the one found in [Spivey89] was found to be very helpful in the development of proofs. Some lemmas used in this section are given in Appendix B.

```
Proof of Lemma Block_injective.
  Block ∈ REFERENCE >+> BLOCK1
                ⇔
  Block ∈ REFERENCE -+> BLOCK1 ∧
1.  ∀ r,s : dom Block • Block r = Block s ⇒ r = s        [def'n >+>]

  Since MEM ⊢ Block ∈ REFERENCE -+> BLOCK1
  we need only prove the other conjunct (1).

2.  Assume r,s : dom Block | Block r = Block s

  ran Block ⊆ P1 Z = dom min so that min(Block r) and min(Block s) are
  well defined. Therefore,

  min(Block r) = min(Block s)                                    [2]
  ⇒ min(Lower r .. Upper r) = min(Lower s .. Upper s)[def'n Block]
  ⇒ Lower r = Lower s                        [property of min:p.116#9]
  ⇒ r = s                                [property of Lower:Lemma MB_1]
QED
```

The inverse of an injective function is also injective. The relational inverse of `Block` is `Block~`.

```
Corollary Block_injective.
  MEM ⊢ Block~ ∈ BLOCK1 >+> REFERENCE                    [p.109#10]
```

An important property to be verified is that no two distinct blocks in memory can overlap. The predicate `disjoint_sets S` (see Appendix A), which is used in the next lemma, holds if and only if all pairs of distinct elements of S are mutually disjoint sets.

```
Lemma Disjoint_Blocks.
  MM ⊢ ∀ S : P(dom mem) • disjoint_sets(Block(S))
Proof
1.  Let S : P(dom mem), that is, S ⊆ dom mem.

    disjoint_sets(Block(S)) ⇔
2.        ∀ b,c : Block(S) | b ≠ c • b ∩ c = ∅ [def'n disjoint_sets]
    Thus we prove (2).
3.  Assume b,c : Block(S) | b ≠ c.
4.  Since {b,c} ⊆ ran Block = dom Block~,
    Block~ b and Block~ c are well defined.
5.  Block~ is an injective function.          [corollary Block_injective]
6.  b≠c                                                          [3]
7.  Block~ b ≠ Block~ c                                       [5,6]
8.  ran Block~ = dom Block = dom mem,
    so Block~ b and Block~ c are elements of dom mem.
9.  ∀ r,s : dom mem | r≠s • ¬Share(r,s)                        [MM]
10. ¬Share(Block~ b, Block~ c)                             [7,8,9]
11. ¬(Block(Block~ b) ∩ Block(Block~ c) ≠ ∅)     [10,Lemma MEM_1]
12. Block(Block~ b) ∩ Block(Block~ c) = ∅                     [11]
13. (Block ∘ Block~) b ∩ (Block ∘ Block~) c = ∅       [12, p.97#7]
14. Block ∘ Block~ = id(ran Block)                      [p.105#1]
15. id(ran Block) b ∩ id(ran Block) c = ∅                 [13, 14]
16. b ∩ c = ∅                                                 [15]
QED
```

As an immediate corollary we have:

```
Corollary Disjoint_Blocks.
 MM ⊢ disjoint_sets(Block(dom mem))
```

We next verify that all blocks in memory are contained within the memory block.

```
Lemma Mem_block_contains_blocks.
 MM ⊢ ∪(Block(dom mem)) ⊆ mem_block
Proof
 Assume b : Block(dom mem)
 Block~ is a function.                          [corollary Block_injective]
 b ∈ ran Block = dom Block~ thus                          [property of ~]
 Block~ b is well defined and
 ran Block~ = dom Block = dom mem                                  [MEM]

 ∀ r : dom mem • Block r ⊆ mem_block                               [MM]
 thus in particular for Block~ b ∈ dom mem we have
 Block(Block~ b) ⊆ mem_block
 ⇒ (Block ∘ Block~) b
       = id(ran Block) b                                        [p.105#1]
       = b
   ⊆ mem_block

 Therefore ∀ b : Block(dom mem) • b ⊆ mem_block
 hence ∪(Block(dom mem)) ⊆ mem_block                              [p.94#5]
QED
```

6.2 Formalizing Allocation

We build upon the specification of the previous section and define a new state space schema for a memory manager. By introducing the concepts of active and inactive memory into the specification, it is possible to express a memory allocation operation.

Inspired by the discussion of section 5.1, we identify two kinds of memory: *active* and *inactive* memory. Active memory is the memory that is currently in use. Inactive memory is the rest of memory. For any given memory mem, we define `active` to be the set of active references, `active_r` to be active memory, and `active_z` to be the size of active memory.

Active

```
MEM
active,
active_r : P REFERENCE
active_z : N
-----------
active ⊆ dom mem
active_r = U(Block(active))
active_z = #active_r
```

If S is a set of sets then US is the union of the sets contains in S. If S is empty then so is US.

For any given memory mem, we define `inactive` to be the set of inactive references, `inactive_r` to be inactive memory, and `inactive_z` to be the size of inactive memory.

Inactive

```
MEM
inactive,
inactive_r : P RERERENCE
inactive_z : N
-----------
inactive ⊆ dom mem
inactive_r = U(Block(inactive))
inactive_z = #inactive_r
```

Guided by the definition of next of kin in section 5.4.2, we define the following. Given a memory m and a reference r which is valid with respect to m, if the node referenced by r under m contains a reference s which is also valid with respect to m, then we say s is a *next of kin* of r.

```
next_of_kin : MEMORY → REFERENCE ↔ REFERENCE
─────────────────────────────────────────────
next_of_kin mem (r,s) ⇔
        {r, s} ⊆ dom mem ∧
        s ∈ ran (mem r).next
```

The *relatives* of a reference r are its next of kin, and the next of kin of its next of kin, etc. A reference r is said to be *related* to a reference s if s is a relative of r.

```
related : MEMORY → REFERENCE ↔ REFERENCE
relatives : MEMORY → REFERENCE → P REFERENCE
─────────────────────────────────────────────
∀ mem : MEMORY •
  related mem = (next_of_kin mem)+ ∧
  ∀ r : REFERENCE •
        relatives mem r = (related mem)(|{r}|)
```

The relations Next_of_kin and Related behave like next_of_kin and related respectively except that the former use the memory mem defined in scope.

```
MEM1
─────────────────────────────────────────────
MEM
Next_of_kin : REFERENCE ↔ REFERENCE
Related : REFERENCE ↔ REFERENCE
─────────────────────────────────────────────
Next_of_kin = next_of_kin mem
Related = related mem
```

6.2.1 The State Space

Observations captured by the formulation of the memory manager state space presented in this section are:

1. The active references are the references on the reference stack and their relatives.

2. Every valid reference (with respect to mem) is either an active reference or an inactive reference (but not both).

3. Active memory and inactive memory accounts for all of memory.

The new state space schema is MM1:

```
MM1
MEM1
MM
Active
Inactive
roots : seq REFERENCE
-----------
active = (Next_of_kin*)(ran roots)
⟨active, inactive⟩ partition dom mem                    (1)
active_z + inactive_z = memory_z                        (2)
```

A reference on the reference stack can be thought of as a root of a connected component of the graph represented by the memory mem. Thus the reference stack has been named roots. Notice that active could have been defined as

```
active = ran roots ∪ Related(ran roots)
```

The predicates (1) and (2) of MM1 are independent — there exist situations in which only one of the predicates is satisfied. Both predicates are necessary to ensure that the third observation mentioned above is respected.

Initial memory manager states have no active memory since the reference stack is empty.

InitMM1

InitMM

---

roots = ⟨⟩

The state change schema ΔMM1 is made more strict than is usual for a Δ-schema: the operations which change the memory manager state can not resize or move the memory block.

ΔMM1

MM1

MM1'

---

mem_block' = mem_block

The ΞMM1 schema has its usual definition:

ΞMM1

ΔMM1

θMM1 = θMM1'

6.2.2 The Allocation Operation

It is now possible to define an allocation operation which behaves like the `Push_new` object manager operation. As a first step towards this goal we define the schema `Allocate0`. Let `R` be a relation and `S` a set then `S <| R` is the largest subrelation of `R` whose domain is contained in `S`. '<|' is the domain restriction operator.

```
Allocate0
ΔMM1
value? : VALUE
next? : seq(INDEX ∪ {0})
ref : REFERENCE
---
ran next? ⊆ (dom roots) ∪ {0}
ref ∉ active
roots' = ⟨ref⟩ ^ roots
(active'\{ref} ◁ mem') = (active ◁ mem)
mem' ref = μ NODE |
  value = v? ∧ next = roots' ∘ succ ∘ next?
```

The memory which was active before the `Allocate0` operation is not affected by the operation. Therefore, due to fragmentation of the memory space, `Allocate0` may fail even though the total amount of inactive memory exceeds the amount needed for allocation. In situations where fragmentation occurs it may be possible to satisfy the allocation request if nodes are moved.

Since the use of references is confined to the memory manager, it is possible to change systematically the references in the memory manager without any perceptible effect to the memory manager state from the point of view of a system using the memory manager. Our choice of a graph as an abstraction for active memory provides a simple way to formalize this: two memories are *equivalent* if the graphs they represent are isomorphic. If `A` and `B` are sets then `A >-->> B` denotes the set of all bijections from `A` into `B`.

```
equiv_mem : MEMORY ↔ MEMORY
---
equiv_mem(m1,m2) ⇔
  ∃ f : REFERENCE >-->> REFERENCE |
    f ∈ dom m1 >-->> dom m2 •
        ∀ r : dom m1 •
              (m1 r).value = (m2 (f r)).value ∧
              f ∘ (m1 r).next = (m2 (f r)).next
```

If a reference is considered as a root to a connected component of a graph, then we may say that two references are equivalent if the subgraphs they identify are equivalent.

```
equiv_ref : (MEMORY × REFERENCE) ↔ (MEMORY × REFERENCE)
─────────────
equiv_ref((memr, r), (mems, s)) ⇔
  equiv_mem(relatives memr r ◁| memr,
        relatives mems s ◁| mems)
```

Memory manager states MM1 and MM1′ are *equivalent* if they have the same number of (references to) rooted subgraphs on the reference stack and if the corresponding subgraphs are equivalent.

```
EQUIV_ACTIVE
ΔMM1
─────────────
#roots′ = #roots
∀ i : dom roots •
  equiv_ref((mem, roots i), (mem′, roots′ i))
```

We now define an operation Allocate_MM1 which first does allocation like Allocate0. If Allocate0 fails, the memory manager relocates active blocks and then tries Allocate0 again.

```
Allocate1 ≙ Allocate0\(ref)
Allocate_MM1 ≙ Allocate1 ∨ (EQUIV_ACTIVE ; Allocate1)
```

The operation Allocate_MM1 corresponds to the object manager operation Push_new. Allocation is the most difficult operation to implement. Memory manager operations corresponding to the other object manager operations can be easily derived and are omitted.

6.2.3 Properties

The proof of Allocate0 ⊢ active_z′ = active_z + Size′ ref > active_z helped us discover that ref ∉ active is a necessary precondition to Allocate0.

This is an example of how proving apparently trivial properties is a vital part of specification validation.

Remark Active_inactive_r_1.

Recall that

active_r = U(Block(active)) and [Active]

inactive_r = U(Block(inactive)) [Inactive]

Hence by lemma Partition_union_block (in Appendix B),

⟨active_r, inactive_r⟩ partition U(Block(active ∪ inactive))

Under MM1, active and inactive partition dom mem,

hence active ∪ inactive = dom mem. Therefore

⟨active_r, inactive_r⟩ partition U(Block(dom mem))

hence also

#active_r + #inactive_r = #U(Block(dom mem))

End

Although we might expect U(Block(dom mem)) = mem_block, this is not immediately obvious. In particular, the next lemma — which proves a slightly more general property — shows that this equality depends on predicate (2) in the memory manager state schema MM1 of section 6.2.1.

Lemma Partition_mem_block.

MM1 ⊢ Block(dom mem) partitions mem_block

Proof

1. Elements of Block(dom mem) are disjoint. This follows from lemma Disjoint_Blocks.
2. Next we must show that

   U(Block(dom mem)) = mem_block.

   It has already been established that

   U(Block(dom mem)) ⊆ mem_block

   (by lemma Mem_block_contains_blocks). To prove equality it is sufficient to show that

   #(U(Block(dom mem))) = #mem_block

   #mem_block

   = memory_z [Lemma Mem_block_z]

   = active_z + inactive_z [MM1]

```
  = #active_r + #inactive_r                              [def'n α_z]
  = #(U(Block(dom mem)))                  [Remark Active_inactive_r_1]
QED
```

As was claimed above, active and inactive memory account for all of memory.

```
Lemma Active_inactive_r_partition.
  MM1 ⊢ ⟨active_r, inactive_r⟩ partition mem_block
Proof
1.  ⟨active_r, inactive_r⟩ partition
        U(Block(dom mem))                 [Remark Active_inactive_r_1]
2.  U(Block(dom mem)) = mem_block          [Lemma Partition_mem_block]
3.  ⟨active_r, inactive_r⟩ partition mem_block                  [1,2]
QED
```

We next show that equiv_mem is an equivalence relation.

```
Lemma Equivalence_of_equiv_mem.
  equiv_mem is an equivalence relation.
Proof
1.  Reflexivity

2.  Let m : MEMORY and
3.  f = id REFERENCE
4.  f ∈ REFERENCE >->> REFERENCE                          [3,p.109#2]
5.  dom m ⊆ REFERENCE
6.  f ∈ dom m >->> dom m                                      [4,5]

7.  Let r : dom m then
8.  (m r).value = (m (f r)).value                              [3]

9.  f ∘ (m r).next
        = (m r).next                                       [p.97#5]
        = (m (f r)).next                                       [3]
10. Thus equiv_mem(m, m).
```

11. Symmetry

12. Assume m_1, m_2 : MEMORY | equiv_mem(m_1, m_2)

13. ∃ f : REFERENCE >->> REFERENCE | f ∈ dom m_1 >->> dom m_2 •
    ∀ r : dom m_1 •
    (m_1 r).value = (m_2 (f r)).value ∧
    f ∘ (m_1 r).next = (m_2 (f r)).next [12]

We must show that

14. ∃ g : REFERENCE >->> REFERENCE | g ∈ dom m_2 >->> dom m_1 •
    ∀ r : dom m_2 •
    (m_2 r).value = (m_1 (g r)).value ∧
    g ∘ (m_2 r).next = (m_1 (g r)).next [12]

15. We will do so by assuming that there exists a bijection f satisfying the quantified predicate [13].

16. Let g = f~. Since f is a bijection,

17. g ∈ REFERENCE >->> REFERENCE [15]

18. g ∈ dom m_2 >->> dom m_1 [15]

Let r : dom m_2.

19. (m_2 r).value
    = (m_2 (id REFERENCE r)).value
    = (m_2 ((f ∘ f~) r)).value [p.107#2]
    = (m_2 (f (g r))).value
    = (m_1 (g r)).value [13]

20. (m_1 (g r)).next
    = id REFERENCE ∘ (m_1 (g r)).next [p.97#5]
    = (f~ ∘ f) ∘ (m_1 (g r)).next [p.107#2]
    = g ∘ (f ∘ (m_1 (g r)).next [associativity of ∘]
    = g ∘ (m_2 (f (g r))).next [13]
    = g ∘ (m_2 ((f ∘ f~) r)).next
    = g ∘ (m_2 r).next [p.107#5]

21. Thus equiv_mem(m_2, m_1).

22. Transitivity

23. **Assume** m_1, m_2, m_3 : MEMORY | equiv_mem(m_1, m_2) ∧ equiv_mem(m_2, m_3)

24. ∃ f : REFERENCE >->> REFERENCE | f ∈ dom m_1 >->> dom m_2 •
    ∀ r : dom m_1 •
    (m_1 r).value = (m_2 (f r)).value ∧
    f ∘ (m_1 r).next = (m_2 (f r)).next [23]

25. ∃ g : REFERENCE >->> REFERENCE | g ∈ dom m_2 >->> dom m_3 •
    ∀ r : dom m_2 •
    (m_2 r).value = (m_3 (g r)).value ∧
    g ∘ (m_2 r).next = (m_3 (g r)).next [23]

26. We must show that

27. ∃ h : REFERENCE >->> REFERENCE | h ∈ dom m_1 >->> dom m_3 •
    ∀ r : dom m_1 •
    (m_1 r).value = (m_3 (h r)).value ∧
    h ∘ (m_1 r).next = (m_3 (h r)).next

We will do so by assuming that there exists bijections f and g satisfying the quantified predicates [24] and [25].

28. Let h = g ∘ f then

29. h ∈ REFERENCE >->> REFERENCE [28]

30. h ∈ dom m_1 >->> dom m_3 [28]

Let r : dom m_1.

31. (m_3 (h r)).value
    = (m_3 ((g ∘ f) r)).value [28]
    = (m_3 (g (f r))).value [p.97#7]
    = (m_2 (f r)).value [25]
    = (m_1 r).value [24]

32. h ∘ (m_1 r).next
    = (g ∘ f) ∘ (m_1 r).next [28]
    = g ∘ (f ∘ (m_1 r).next) [associativity of ∘]
    = g ∘ (m_2 (f r)).next [24]
    = (m_3 (g (f r))).next [25]
    = (m_3 ((g ∘ f) r)).next [associativity of ∘]
    = (m_3 (h r)).next [28]

33. Thus equiv_mem(m1, m3).
QED

6.3 Refining Allocation

In the previous section we noted that it may be necessary to move nodes in memory in order to be able to satisfy an allocation request. In this section we refine the allocation operation by defining when and how nodes will be moved in memory.

Again, following the discussion of section 5.1, we identify two kinds of inactive memory: *free* and *garbage* memory. Free memory is the inactive memory which is readily available for allocation. Garbage memory is the rest of inactive memory. Intuitively, garbage is memory that has been active, is no longer active, but has not yet been reclaimed. For any given memory mem, we define free to be the set of references to free nodes, free_r to be free memory, and free_z to be the size of free memory. Notice that there is a special constraint on free nodes: no two distinct free nodes can be adjacent.

Free

MEM

free,

free_r : P REFERENCE

free_z : N

---

free ⊆ dom mem

free_r = U(Block(free))

free_z = #free_r

∀ r,s : free | r≠s • ¬Adjacent(r,s)

Given any memory mem, garbage is the set of references to garbage nodes, garbage_r is garbage memory, and garbage_z is the size of garbage memory.

```
Garbage
MEM
garbage,
garbage_r : P REFERENCE
garbage_z : N
---
garbage ⊆ dom mem
garbage_r = U(Block(garbage))
garbage_z = #garbage_r
```

6.3.1 The State Space

The memory manger state MM2 expresses the fact that inactive memory is partitioned into free and garbage memory.

```
MM2
MM1
Free
Garbage
---
⟨free, garbage⟩ partition inactive
free_z + garbage_z = inactive_z
```

The initial states of the memory manager contain no garbage memory:

```
InitMM2
InitMM1
---
garbage = ∅
```

Since the initial memory manager states are also constrained not to have any active memory (by the InitMM1 schema) this implies that all of memory is free memory.

The state change schema ΔMM2 still requires that the memory block remain unaltered:

```
ΔMM2 ≙ MM2 ∧ MM2′ ∧ ΔMM1
```

6.3.2 Allocation Revisited

Recall the proposed memory allocation strategy given in section 5.1:

1. Try to allocate space from the free memory.
2. If step 1 fails then collect all garbage memory and try step 1 again.
3. If step 2 fails then compact the free memory into one block and try step 1 again
4. If step 3 fails then allocation fails.

Step 1 is stated formally as the operation `Alloc_free0`. This operation extends `Allocate0` by ensuring that garbage memory is not affected by the operation.

```
Alloc_free0
Allocate0
ΔMM2
─────────
garbage′ = garbage
(garbage′ ◁| mem′) = (garbage ◁| mem)
```

The operation `Collect` specifies garbage collection: it ensures that there is no garbage memory after the operation is complete and that active memory is not affected.

```
EQUAL_ACTIVE
ΔMM1
─────────
roots′ = roots
(active′ ◁| mem′) = (active ◁| mem)
```

```
Collect
EQUAL_ACTIVE
ΔMM2
---
garbage' = Ø
```

Given that there is no garbage memory, compaction consists of bringing all inactive (free) memory into one block.

```
Compact
EQUIV_ACTIVE
ΔMM2
---
garbage = Ø = garbage'
free_r' ∈ BLOCK
```

The memory allocation strategy described above is captured by the schema Allocate_MM2.

```
Alloc_free ≙ Alloc_free0\(ref)
Allocate_MM2 ≙ Alloc_free ∨
               (Collect ; Alloc_free) ∨
               (Collect ; Compact ; Alloc_free)
```

6.3.3 Properties

After allocation the amount of free memory must decrease by an amount equal to the size of the allocated node.

```
Lemma Free_z_dec.
  Alloc_free0 ⊢ free_z' = free_z - Size' ref
```

Under MM2 free and garbage memory constitute all of inactive memory.

```
Lemma Free_garbage_r_partition.
  MM2 ⊢ ⟨free_r, garbage_r⟩ partition inactive_r
Proof
1.  free ∩ garbage = ∅                                              [MM2]
2.  free_r = ∪(Block(free))                                         [Free]
3.  garbage_r = ∪(Block(garbage))                                   [Garbage]
4.  free ∪ garbage = inactive                                       [MM2]
5.  inactive_r = ∪(Block(inactive))                                 [Inactive]
6.  inactive_r = ∪(Block(free ∪ garbage))                           [4,5]
7.  ⟨∪(Block(free)), ∪(Block(garbage))⟩ partition
        ∪(Block(free ∪ garbage))      [Lemma Partition_union_block]
8.  ⟨free_r, garbage_r⟩ partition inactive_r                        [2,3,6,7]
QED
```

The above result is needed for deriving a refinement for `Alloc_free`.

6.4 Refinement of `Alloc_free`

In this section we examine the consequences of the definition of `Alloc_free` and use these to guide us in the formulation of a refinement for the operation. We begin by proving that after allocation, the free memory prior to allocation is partitioned into the block that was allocated and the free memory that is left.

```
Lemma Block_and_free_parition_free.
  Alloc_free ⊢ ⟨Block' ref, free_r'⟩ partition free_r
Proof
  Assume ΔMM2.
1.  ⟨active_r, inactive_r⟩ partition mem_block          [MM1, Lemma
                                        Active_inactive_r_partition]
2.  ⟨free_r, garbage_r⟩ partition inactive_r            [MM2, Lemma
                                      Free_garbage_r_partition]
3.  ⟨active_r, free_r, garbage_r⟩ partition mem_block            [1,2]
    Similarly
4.  ⟨active_r', free_r', garbage_r'⟩ partition mem_block'
5.  mem_block' = mem_block                                      [ΔMM2]
```

```
6.  free_r = mem_block \ active_r \ garbage_r                          [3]

7.  ⟨{ref}, active⟩ partition active'                                  [Allocate0]
8.  ⟨Block' ref, active_r⟩ partition
        activ:_r'                          [7,Lemma Partition_union_block]
9.  garbage' = garbage                                                 [Alloc_free]
10. garbage_r' = garbage_r                                             [9,Garbage]
11. ⟨Block' ref, active_r, free_r', garbage_r⟩ partition
        mem_block                                                      [4,8,10]
12. ⟨Block' ref, free_r'⟩ partition
        (mem_block \ active_r \ garbage_r)                             [11]
13. ⟨Block' ref, free_r'⟩ partition free_r                             [6]
QED
```

No two distinct free blocks are adjacent, therefore if a block is allocated from free memory it must be allocated from a single free block. If allocation succeeds then there must be a free block whose size is equal to or larger than the size of the block to be allocated and the reference to the allocated block will be contained in the free block. That is,

```
Alloc_free0 |-
  ∃ f : free | Size f ≥ Size' ref •
        ref ∈ Block f
```

If the size of the free block and the block to be allocated are equal then the reference to the allocated block must be equal to the reference to the free block. Otherwise the allocated block must be chosen as a subblock of the free block. Since we want to reduce fragmentation we choose to divide the free block into two parts. If the allocated block does not entirely consume the free block from which it is allocated, then under the `Alloc_free0_1` schema, it will be allocated as either the first or the last part of the free block.

Alloc_free0_1

Alloc_free0

$$
\begin{aligned}
&\exists\ f : free \mid Size\ f \geq Size'\ ref \bullet \\
&\quad (Size\ f = Size'\ ref \Rightarrow \\
&\qquad ref = f) \wedge \\
&\quad (Size\ f \neq Size'\ ref \Rightarrow \\
&\qquad \langle Block'\ f, Block'\ ref \rangle\ partition\ Block\ f)
\end{aligned}
$$

In the final refinement we include the effects of the allocation operation on `free` and we (arbitrarily) choose, in those cases where the free block must be divided, to make the allocated block the last part of the free block.

Alloc_free0_2

Alloc_free0

$$
\begin{aligned}
&\exists\ f : free;\ z : \mathbb{N}_1 \mid z = Size\ f - Size'\ ref \geq 0 \bullet \\
&\quad (z = 0 \Rightarrow \\
&\qquad ref = f \wedge \\
&\qquad free' = free \setminus \{f\}) \wedge \\
&\quad (z \neq 0 \Rightarrow \\
&\qquad ref = f + z \wedge \\
&\qquad free' = free \wedge \\
&\qquad Size'\ f = z)
\end{aligned}
$$

7 Conclusion

Whether formal or informal methods are used to develop software, experience shows [Knuth89] that there will always be errors to discover and improvements to make to the software product (as well as to the software development process).

Software may be correct without being valid, that is, without satisfying customer needs or environmental constraints. It is accepted that it is inherently impossible to *prove* that a system meets its informal requirements [Jones86]. Thus, although we may someday be able to produce correct software, we may never be quite sure whether the produced software is valid.

Based on my experience with the development of the object manager, I suggest improvements that could be made to the specifications and to the Z notation. Also discussed are the possibility of reuse of the specifications and various extensions.

7.1 Immediate Improvements

Early in the specification of the memory manager the schema type REF is introduced. A binding of type REF has as components a memory and a reference which is valid with respect to the memory. The REF type allows us to define `block`, `lower`, and `upper` as *total* functions and `share` and `adjacent` as relations. These functions and relations are practically never used after the definitions of `Block`, `Lower`, etc in MEM. In fact, the definitions of the components of MEM were motivated by the awkward statements that resulted with the use of `block`, `lower`, `upper`, `share` and `adjacent`. Thus, in retrospect, the specification would clearly be more readable and understandable if the type REF was not introduced and if the global variables `block`, `lower`, etc were replaced by the definitions in MEM.

A second improvement would involve significant modifications to the memory manager specification. This change has been inspired from the object manager specification and from my attempts at refining the operations `Collect` and `Compact`. Conceptually memory is used for two different purposes. Most importantly, part of memory is used to hold nodes which make up the graph that the object manager requires. The rest of memory is the free pool out of which nodes are built. Let us call these two kinds of memory

allocated and *free* memory, respectively. Note that the use which is made of allocated memory is independent of the manner in which free memory is managed. The use which is made of allocated memory determines the structure of nodes and the manner in which free memory is managed determines the structure of free memory blocks. Consequently, memory is used for two different and independent purposes and this fact should be reflected in the memory manager model.

A possible solution would be to have two 'memories' as part of the memory manager state space. These memories would be represented by partial functions from references into nodes, but the node types would differ. The blocks belonging to these two memories would still partition (and hence account for all of) the memory block.

The present formulation of the memory manager specification is certainly adequate for pursuing with the refinements of the `Collect` and `Compact` operations but the resulting refinements would be more readable and understandable if the specification were changed as described above.

7.2 Extensions and Reuse

The object manager specification is in essence the requirements specification for the memory manager, and section 6.1 is a collection of formal definitions of various concepts relevant to memories. Thus, the specifications which precede section 6.2 are independent of any particular allocation strategy and could be reused to describe other garbage collection algorithms.

In the sections which follow we discuss a simple extension that could be made to the memory manager specification and why the concept of alignment was not introduced into the specification.

7.2.1 A Heterogeneous Memory Manager

Let us say a node is *homogeneous* with respect to a given memory if it contains only valid references (with respect to the memory) and say it is *heterogeneous* otherwise. Formally

```
homogeneous_node : MEMORY ↔ NODE
─────────────────────────────────
homogeneous_node(mem, node) ⇔ ran node.next ⊆ dom mem
```

We say a memory manager is *homogeneous* if it only permits the construction of homogeneous nodes. Otherwise we say it is *heterogeneous*.

The memory manager presented in section 6 is homogeneous, but its specification has been carefully designed to allow it to be easily made into a heterogeneous system. This is possible because of the definitions of `Active`, `next_of_kin` and `equiv_mem`.

Care must be taken in making the change. In particular, one should not permit nodes to contain references which are within the memory bounds and not valid with respect to the memory. The reason for this is not that the resulting specification would be inconsistent but that it would be impractical to implement.

Hence we introduce a set of references called `special_refs`. By changing the definition of `Active` to the one of `Hetero_active` presented below, the resulting memory manager is heterogeneous.

```
Hetero_active
MM
special_refs,
active,
active_r : P REFERENCE
active_z : N
─────────────────────────────────
disjoint ⟨special_refs, mem_block⟩
active ⊆ dom mem ∪ special_refs
active_r = U(Block(active))
active_z = #active_r
```

Why a heterogeneous memory manager is desirable becomes apparent when we consider the use of special pointer values such as **`nil`** in Pascal or `0` (often denoted by `NULL`) in C.

7.2.2 Memory Alignment

It is possible to incorporate the concept of alignment in the memory manager specification. This can be done by constraining the 'values' which references may have. For example, the addition of a single predicate to the MM schema ensures that nodes are aligned at even boundaries.

```
MM_with_alignment
MM
---
dom mem ⊆ {r : mem_block | (r mod 2) = 0}
```

This additional constraint implicitly imposes restrictions on the function `size`.

For the development of the BIAS memory manager I had decided to use C as implementation language. It is possible to write portable C code which handles alignment [Kernighan78]. Hence there was no need to complicate the specification with a concept that can be handled (and should be left to) the implementation language.

7.3 Remarks about Z

The Z notation has proven to be both expressive and practical but there are certain aspects of the notation which need to be improved.

Before one can consider the meaning of a Z specification one must verify that it is well formed. In particular, for a specification to be well formed it must conform to the type rules of the Z notation. The type rules are separate from the underlying mathematical logic. For example, in $\forall x : T \bullet P$ the variable x is declared to be of type T — that is, x is an element of the set T — but this is *type information* and it is not part of the logical calculus. Even though it may seem intuitively obvious that if x is of type T ($x : T$) then x must be an element of T ($x \in T$), there are no inference rules in Z which permit us to make this deduction. Hence separate 'inference rules' for types are necessary to permit reasoning about types within predicates. As an example of what such a rule could be, consider:

$$\frac{\forall\ x:T \bullet P}{\forall\ x:T \mid x \in T \bullet P}$$

These type 'inference rules' are an apparent omission from [Spivey89], the Z language reference manual.

Z lacks a notation for defining specifications as separate modules that can be combined and reused, like classes in object oriented languages, for example. To be practical it should be possible to write generic specifications. A simple solution would be to view the basic types of a specification as generic parameters. Given that a specification can be named, one could subsequently instantiate it and the instantiation would bind the actual parameters to the basic types. This simple scheme would not handle name clashes for which some renaming mechanism would be required. If Z is to be applicable to large-scale development then the notation will need to be adapted to allow for specifications to be built as separate modules.

7.4 Further Work

Even if specifications are produced only during the first stages of software development, the benefits will certainly be a clearer and more complete understanding of the problem to be solved and of the solution which has been chosen [Jones86]. The formal development of the object manager includes a requirements specification, two levels of 'design specification' and the detailed specification of one of the allocation operations. Several important properties of the specifications are proven. The object manager needs to be strengthened with proofs of correctness. The construction and maintenance of such proofs is a major undertaking — especially by hand. Thus, I intend to continue my research in the areas of development of notations for formal specification and automated support for formal reasoning as related to software development.

References

[Abbott81] Abbott, R. J. and D. K. Moorhead. Software requirements and specifications: a survey of needs and languages. *The Journal of Systems and Software* , *2*, 297-316 (1981)

[Aho86] Aho, A. V. et al. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Burstall80] Burstall, R. M. and J. A. Goguen. The Semantics of Clear, a Specification Language. In *Abstract Software Specification*, Springer, LNCS 86, 292-322, 1980.

[Delisle89] Delisle, N. and D. Garlan. Formally Specifying Electronic Instruments. *ACM SIGSOFT Software Engineering Notes*, *14*, 3, 242-248 (1989)

[Fairley85] Fairley, R. E. *Software Engineering Concepts*. McGraw-Hill, 1985.

[Ghezzi87] Ghezzi, C. and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1987.

[Goguen79] Goguen, J. A. and J. J. Tardo. An introduction to OBJ, a language for writing and testing software specifications. In *Specification of Reliable Software*, IEEE, 1979.

[Grogono87] Grogono, P. *BIAS User Manual*. Technical Report, PLSG-9, Concordia University, 1987.

[Guttag82] Guttag, J., J. Horning and J. Wing. Some notes on putting formal specifications to productive use. *Sci. Comput. Programming*, 2, 53-68 (1982)

[Hayes87] Hayes, I. (editor). *Specification Case Studies*. Prentice-Hall International, 1987.

[IEEE84] IEEE Std 830-1984. *IEEE Guide to Software Requirements Specification.*

[Jiang88] Jiang, X. and X. YongSen. NUSL: an executable specification language based on data abstraction. In Bloomfield et al., editors, *VDM'88: VDM — The Way Ahead*, Springer, LNCS 328, 124-138, 1988.

[Jones86] Jones, C. B. *Systematic software development using VDM*. Prentice-Hall International, 1986.

[Jones88a] Jones, C. B. and P. A. Lindsay. A support system for formal reasoning: requirements and status. In Bloomfield et al., editors, *VDM'88: VDM — The Way Ahead*, Springer, LNCS 328, 139-152, 1988.

[Jones88b] Jones, C. B. and R. Moore. Muffin: a user interface design experiment for a theorem proving assistant. In Bloomfield et al., editors, *VDM'88: VDM — The Way Ahead*, Springer, LNCS 328, 337-375, 1988.

[Kernighan78] Kernighan, B. W., and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Knuth89] Knuth, D. E. The errors of TEX. *Software Practice and Experience, 19*, 2, 607-685 (1989)

[London89] London, R. L., and K. R. Midsted. Specifying Reusable Components Using Z: Realistic Sets and Dictionaries. *ACM SIGSOFT Software Engineering Notes, 14*, 3, 242-248 (1989)

[Meyer85a] Meyer, B. On formalism in specifications. *IEEE Software, 2*, 1, 6-26 (1985).

[Meyer85b] Meyer, B., J. Nerson and H. K. Soon. Showing programs on a screen. *Sci. Comput. Programming, 5*, 2, 111-142 (1985)

[Meyer88] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Pressman82] Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1982.

[Rich88] Rich, C. and R. C. Waters. Automatic programming: myths and prospects. *IEEE Computer, 21*, 8, 40-51 (1988)

[Ritchie88] Ritchie, B. *The Design and Implementation of an Interactive Proof Editor*. Ph.D. thesis CST-57-88, Dept. of Comp. Sci., University of Edinburgh.

[Roberts84] Roberts, F. S. *Applied Combinatorics*. Prentice-Hall, 1984.

[Schmidt88] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, 1988.

[Sommerville89] Sommerville, I. *Software Engineering*. Addison-Wesley, 1989.

[Spivey88] Spivey, J. M. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[Spivey89] Spivey, J. M. *The Z Notation: A Reference Manual*. Prentice-Hall International, 1989.

[Sufrin82] Sufrin, B. Formal specification of a display-oriented text editor. *Sci. Comput. Programming, 1*, 3, 157-202 (1982)

[Woodcock89] Woodcock, J. C. P. Calculating properties of Z specifications. *ACM SIGSOFT, 14*, 5, 43-54 (1989).

Appendix A. Nonstandard Generic Constants

This appendix contains nonstandard generic constants that are used in the specifications.

$$Property\ X == \mathbb{P}\ X$$

The standard predicates `disjoint` and `partition` are defined over indexed collections of sets. Here are definitions for disjointness and partition for sets of sets.

$$[X]$$

$$disjoint\_sets : Property(\mathbb{P}(\mathbb{P}\ X))$$

$$\_\ partitions\ \_ : \mathbb{P}(\mathbb{P}\ X) \leftrightarrow \mathbb{P}\ X$$

$$disjoint\_sets\ SS \Leftrightarrow \forall s,t : SS \bullet s \neq t \Rightarrow s \cap t = \emptyset$$

$$SS\ partitions\ S \Leftrightarrow disjoint\_sets\ SS \land \bigcup SS = S$$

$$INDEX == \mathbb{N}$$

The identifier `INDEX` is defined as an abbreviation for the domain of sequences, that is, the set of natural numbers.

Appendix B. Selected Lemmas

The following lemma identifies alternative formulations of the functions Lower, Upper, Share and Adjacent.

```
Lemma MEM_1.
 MEM |- ∀ r : REFERENCE •
        Lower r = r  ∧
        Upper r = Lower r + Size r - 1
     ∧
        ∀ r,s : REFERENCE •
        Share(r,s) ⇔ Block r ∩ Block s ≠ ∅  ∧
        Adjacent(r,s) ⇔
              ¬Share(r,s) ∧
              Block r ∪ Block s ∈ BLOCK1
```

Lemma Block_partition_union is used in the lemma Partition_union_block which follows it.

```
Lemma Block_partition_union.
 MM |- ∀ A,B : P(dom mem) | A ∩ B = ∅ •
        ⟨Block(A), Block(B)⟩ partition Block(A ∪ B)
Proof
 1. Disjointness

 Let A,B : P(dom mem) | A ∩ B = ∅

 Assume b ∈ Block(A) ∩ Block(B) then
 b ∈ Block(A)
 b ∈ Block(B)
 b ∈ ran Block = dom Block~ hence Block~ b is well defined.
 Block~ b ∈ A
 Block~ b ∈ B
 Block~ b ∈ A ∩ B = ∅
 false
```

Thus our assumption was false and hence

Block(A) ∩ Block(B) = ∅

2. Covers

Block(A ∪ B) = Block(A) ∪ Block(B) [p.101#5]

QED

Lemma Partition_union_block is used in sections 6.2.3, 6.3.3, 6.4.

Lemma Partition_union_block.

MM |- ∀ A,B : P(dom mem) | A ∩ B = ∅ •
⟨U(Block(A)), U(Block(B))⟩
partition U(Block(A ∪ B))

Proof

1. Disjointness

U(Block(A)) ∩ U(Block(B))
= U{a : Block(A) • a ∩ U(Block(B))} [p.92#5]
= U{a : Block(A) •
U{b : Block(B) • a ∩ b}} [p.92#5]
= (*)

Since A and B are disjoint, Block(A) and Block(B) are disjoint (by lemma Block_partition_union). Hence a ≠ b for any a in Block(A) and b in Block(B). By lemma Disjoint_Blocks we know that any two distinct blocks are disjoint, hence

(*) = U{a : Block(A) • U{b : Block(B) • ∅}}
= U{a : Block(A) • ∅}
= ∅

2. Covers

U(Block(A ∪ B))
= U(Block(A) ∪ Block(B)) [Block_partition_union]
= U(Block(A)) ∪ U(Block(B)) [p.91#1]

QED