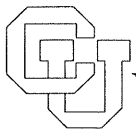
An Example of Formal Specification as an Aid to Design and Development

Mark Terwilliger, Mark J. Maybee and Leon J. Osterweil

CU-CS-399-88 May 1988



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

An Example of Formal Specification as an Aid to Design and Development

Robert B. Terwilliger
Mark J. Maybee
Leon J. Osterweil

Department of Computer Science,
University of Colorado,
Boulder, CO 80309-0430

Abstract

In this paper we describe an experiment which supports the hypothesis that formally specifying a system after a prototype has been created can enhance the development process. We conjectured that describing an existing system using a small number of well understood concepts could increase understanding and enhance communication between project members. We performed an experiment to verify this conjecture using software constructed by the Arcadia project: a consortium developing software environment technologies including "process programming" [19]. At the beginning of the experiment, a prototype requirements building system had been developed and was undergoing redesign and extension. Over the course of three months, the developers met with a specification expert and a formal description was written using PLEASE [23], an Ada®-based executable specification and design language which supports formal methods similar to VDM. Formal specification enhanced the development process by helping the developers to clarify the architecture of the system, separate their concerns, and discuss possible modifications.

1. Introduction

Traditional methods do not ensure the production of correct software. It has been suggested that formal specification can enhance the software development process [7, 28]. Typically, it is proposed that such specifications should either be written before the production of a prototype, or should be executable and also serve as an initial implementation. We believe that formal specifications can enhance the development process in a number of ways. Specifically, we believe that further development can be enhanced by creating a formal specification after a prototype has already been constructed using traditional techniques. We believe that the purpose of prototype development should be to facilitate the acquisition of knowledge about the nature of a problem domain or solution approach. That being the case, it seems clear to us that describing the prototype and the nature of experiences with it more precisely should yield correspondingly more precise and valuable knowledge and insight. It should also increase understanding and enhance communication between project members. Post-prototype formal specification can in this way enhance the remainder of the development process by helping to clar-

---

Ada ® is a trademark of the US Government, Ada Joint Program Office.

This research was supported in part by NSF grant CCR-8705162.

ify system architecture, separate concerns, and provide an "intellectual test bed" for the discussion of changes. In this paper, we describe an experiment performed in the context of the Arcadia project which supports these hypotheses.

The goal of the Arcadia project is to experimentally develop superior software environment technologies, one of which is *process programming* [19]. Advocates of process programming claim that software processes such as requirements specification, design, and testing are themselves software; therefore, they should be specified, designed, and coded. They further believe that process programs should be interpretable by software environments. Thus the Arcadia project is attempting to develop process programs and devise environment architectures capable of interpreting them. Towards this end, Arcadia is developing a series of process programs and supporting environments using a cycle of prototype development, evaluation, and migration. The first phase of this activity is the creation of a prototype process program for the entire software lifecycle called BOPEEP (But One Prototype End to End Process-program). The initial phase of BOPEEP has focussed on the creation of REBUS, a prototype REquirements BUilding System. These prototypes are intended to be used as vehicles for providing important early knowledge about key software processes and languages in which they might be programmed. Hence we conjectured that the use of formalization to describe these prototypes and their components would increase the knowledge acquisition effect of the prototyping activities.

In explaining REBUS and process programming in general, we found informal descriptions to be inadequate. While natural language facilitates the formulation and transfer of general concepts, its ambiguity makes precise communication difficult. We felt we could better analyze the complex issues involved in process programming with the aid of formal descriptions. For systems already constructed, programming language code might serve as a complete, formal description. However, such descriptions often lack abstraction and rigor. Code is written to efficiently utilize machine resources rather than to communicate new ideas. We believe that rigorous higher level descriptions of these systems can increase our understanding of the issues involved. With this in mind, we have attempted to describe some of our key activities using formal methods similar to VDM.

VDM (the Vienna Development Method) supports the top-down development of software specified in a notation suitable for formal verification [2, 6, 13]. In this method, components are first written using a combination of conventional programming languages and predicate logic. A procedure or function may be specified using a *pre-condition*, which states the properties that the inputs must satisfy, and a *post-condition*, which states the relationship

of inputs to outputs. Similarly, an abstract data type may be specified with an *invariant* defining the acceptable states and pre- and post-conditions for the operations. To increase the expressive power of specifications, the high-level types *set*, *list*, and *map* are added to the language. In VDM, specifications are incrementally refined into programs in an implementation language. VDM is used in industrial environments [3, 18, 20] and seems to prove useful in practice; environments to support the method are now being constructed [1].

In one sense, VDM-style specifications can be written in any language which supports predicate logic annotations. For example, ANNA [14, 15] is an annotated version of Ada. PLEASE [22, 23, 26] is a subset of ANNA specifically designed to support formal methods similar to VDM. In PLEASE, the types set, list, and map are predefined and annotations are restricted to Horn clauses: a subset of predicate logic which is also the basis for Prolog [4, 5]. PLEASE specifications can be used in proofs of correctness; they can also be transformed into prototypes which use Prolog to "execute" pre- and post-conditions. PLEASE specifications are effectively exploited by ENCOMPASS [21, 24, 25], an environment which extends VDM with executable specifications, knowledge-based tools, and testing-based verification methods. At present, ENCOMPASS is not robust enough to be used in major development efforts. However, the PLEASE language is stable, and can be used independently of the tools developed for its support.

In the fall of 1987, we performed an experiment in which a high-level design for REBUS was written in PLEASE. At the beginning of the experiment, a prototype version of REBUS had been developed and was undergoing evaluation aimed at redesign and extension. We conjectured that describing the system using a small number of formally defined concepts could increase understanding, enhance communication between project members, and maximize the knowledge acquired though the prototyping activity. Over the course of three months, the developers met with a specification expert and the formal description was created. We found that formal specification enhanced the development process by helping the developers to clarify the architecture of the system, separate their concerns, and discuss possible modifications.

In the rest of this paper we describe this experiment in more detail. In section two we describe the experimental process more completely, including an overview of the original REBUS prototype and the issues raised during the specification process. The discussion centers around the *whiteboard*, a mechanism for controlling the interactions between multiple programmers. In section three we present the formal specification of REBUS; it contains four parts: the object manager, the whiteboard, REBUS itself, and a skeleton of the entire specification process.

Finally, in section five we summarize and draw some conclusions from our experience.

2. An Experiment in Formal Specification

In the Fall of 1987 at the University of Colorado, a prototype version of REBUS had been under construction since the preceding summer by a team of three graduate students. The prototype served as a focal point for integrating a number of pre-existing research efforts. It encorporated both an object-oriented database system [9-12] and a relational language [8] among its major components. Its major objectives included improved understanding of the nature of the requirement specification process and of the linguistic mechanisms needed to support its effective description by executable code. After a four month period of very rapid prototyping, it was decided that a review of the design would enhance further development; we believed that the developers' current perspectives might have been too heavily influenced by implementation considerations. It was suggested that the construction of a formal specification/design could prove to be interesting as well as an aid in future development. By creating an abstract representation of the system, we believed that the underlying issues and concepts would be separated from the implementation. Therefore, over a period of three months, formal specifications of REBUS were written in PLEASE.

During the experiment, weekly meetings were held between a senior graduate student and a faculty member serving as the specification expert. In these meetings, the student presented the goals and design of REBUS to the expert, who questioned the student in order to clarify concepts and devise formal representations for the design. The development of the specifications did not proceed at a uniform rate. Periods of discussion and thought would result in important insights and new formal representations; these would be followed by the rapid generation of specifications. At the beginning of the process a month passed before the specification expert's knowledge of the project was sufficient that any formal representations could be generated. During the second month many insights were reached and the bulk of the specifications were written. An initial version of the specifications were presented to the Arcadia group at the University of Colorado as soon as they were completed. During the third month the specifications were carefully reviewed, discussed, and put into their present form.

To better understand the issues raised and resolved during the experiment, we will consider the REBUS prototype in more detail.

2.1. The REBUS Prototype

REBUS is a requirements building system developed by the Arcadia group as part of BOPEEP, a prototype process program implementation. REBUS attempts to show that requirements specification development can be viewed as a programming activity by developing a system that implements a process program for requirements specification. In REBUS, a requirements specification is viewed as a DAG composed of requirements specification *nodes*, each of which may have *attributes* of various types. Nodes may have *relations* between them, and both nodes and relations may have to satisfy certain *constraints*. In REBUS, the requirements specification process consists of creating and modifying nodes, relating nodes to one another and forming larger aggregates, analyzing groups of nodes to see if they satisfy constraints, and coordinating the efforts of multiple developers.

More specifically, in REBUS a requirements specification is an instance of the type *Reqts_Spec*, which is a DAG of nodes, each of which is of type *Reqts_Elt*. Objects of type *Reqts_Elt* are records whose fields (attributes) are of various types, some of which are more familiar (for example, string or integer) and some of which are more esoteric (for example, predicate or dataflow sequence specification). *Reqts_Elt* objects may have relations between them which are specified in the language APPL/A [8]. APPL/A relations specify constraints that *Reqts_Elt* objects must satisfy and provide mechanisms to trigger appropriate actions when these constraints are violated. Development of an object of type *Reqts_Spec* is therefore a process of creating, evaluating, relating, and altering *Reqts_Elt* objects.

The REBUS prototype, then, implements a process program for requirements specification; this program consists of a number of concurrent sub-processes. For example, each person on a project is expected to specify different sub-requirements, and thus may have a different path of execution through REBUS. This requires different bindings for different sub-processes. The main REBUS process describes the various requirements software objects, attributes, and relations, as well as controlling the ways in which sub-processes can interact. Tasks such as creation, analysis and modification of nodes are represented by coordinated sub-processes; these tasks are concurrent, but may require synchronization at certain points. For example, analysis of a node should be suspended or disallowed while it is undergoing modification. High level direction of these tasks, and the coordination of the people to whom their execution is bound, is effected by the highest level of REBUS, which is viewed as the actual process program, while the lowest-level sub-processes, or possibly their operations, are considered to be the analogs of conventional tools.

In REBUS, the actions of independent sub-processes are coordinated using the *whiteboard*, a communication and coordination mechanism derived from the *blackboard* concept [16, 17]. The REBUS whiteboard was designed specifically to coordinate multiple developers are engaged in the parallel creation of a software object whose partitions may be inter-dependent; requirements development for example. In our concept, a whiteboard holds a group of objects, each of which is in a particular *state*. The state of an object summarizes much of the information available about it; for example its consistency, stage in the lifecycle, work status, and completeness. Objects are moved from state to state to reflect changes in their underlying status. The whiteboard provides access to the objects it holds only through certain operations, the most notable of which modify the object or change its state on the whiteboard. Access and modification permissions may vary with the object, state and user.

To gain a deeper understanding of the whiteboard concept and how it evolved during our experiment, we now describe the whiteboard as it was originally implemented in the REBUS prototype.

2.1.1. The Original Whiteboard Model

Figure 1 shows a simplified version of the *Whiteboard* package specification as it appeared at the beginning of our experiment. The package implements an *abstract data type* (ADT), instances of which are whiteboards with persistent states. Thus, the package provides an interface to a persistent data store, the elements of which are whiteboards. A whiteboard stores and manipulates objects called *items*, each of which represents the state of a unit of a requirements specification object. whiteboard items can be thought of as messages for communicating the state of objects amongst independent processes. In this implementation, there are only three states of interest; therefore, a whiteboard consists of three lists of items: the items currently *posted*, and thus requiring attention; the items presently *locked* by a user; and the list of *completed* items. Posted items are indications of work remaining to be done. As new work is generated, corresponding items are "posted" to the whiteboard. Users select items on which to work by examining the posted items and "locking" them. The list of locked items is actually a list of lists: one for each user working on the project. Locked items represent work in progress. Locks are exclusive so there is no possibility of conflicting changes to the same item. The list of completed items represents work that has reached some reasonable standard of thoroughness and consistency. REBUS automatically subjects items on the completed list to checks for internal consistency and, if the appropriate conditions are not met, automatically re-posts the incorrect items.

```
package Whiteboard is

        --
        --        create, open or close a whiteboard
        --
        procedure create_whiteboard ( Name  : in string    ) ;

        function  open_whiteboard   ( Name  : in string    ) return WB_handle ;

        procedure close_whiteboard  ( Ident : in WB_handle ) ;

        --
        --        access an item on the whiteboard
        --
        function item        ( Board : in WB_handle ; Id : in item_id ) return WB_item ;

        --
        --        get an itemlist off a whiteboard
        --
        function itemlist           ( Ident : in WB_handle ) return IL_handle ;

        function locked_itemlist    ( Ident : in WB_handle ) return IL_handle ;

        function completed_itemlist ( Ident : in WB_handle ) return IL_handle ;
        --
        --        manipulate an itemlist
        --
        procedure get_item   ( Index : in out IL_handle ; Item : out WB_item ) ;

        function end_of_items( Index : in IL_handle ) return Boolean ;

        --
        --        post, lock, delete or complete an item
        --
        procedure post_item     ( Board : in WB_handle ; Contents : in string  ) ;

        procedure repost_item   ( Board : in WB_handle ; Id : in item_id       ) ;

        procedure lock_item     ( Board : in WB_handle ; Id : in item_id       ) ;

        procedure unlock_item   ( Board : in WB_handle ; Id : in item_id       ) ;

        procedure delete_item   ( Board : in WB_handle ; Id : in item_id       ) ;

        procedure complete_item ( Board : in WB_handle ; Id : in item_id       ) ;

end Whiteboard;
```

Figure 1. Simplified Version of Original Whiteboard Header

The *Whiteboard* package provides four types of operations: operations to create, open and close a whiteboard; operations to access items on the whiteboard; operations which manipulate lists of items; and operations to move

items between lists on the whiteboard. The second and third groups of operations are distinguished because it is possible to access items without manipulating lists of items. All operations take parameters of two basic types: *names*, text strings used to refer to objects; and *handles*, which provide more direct and machine-efficient access to objects For example, consider the operations to create, open and close whiteboards. The procedure *create_whiteboard* causes a new whiteboard to be created in the persistent data store with a specified unique name. The function *open_whiteboard* returns a handle to the pre-existing named whiteboard. The procedure *close_whiteboard* notifies REBUS that the user is done with the whiteboard accessed by the indicated handle.

Other operations manipulate lists of items. The function *itemlist* returns a handle to the list of posted items on the whiteboard referenced by *Ident*, while the functions *locked_itemlist* and *completed_itemlist* return handles to the lists of locked and completed items respectively. The procedure *get_item* updates the itemlist handle, returning copies of successive items on an itemlist when called repeatedly, and the function *end_of_items* returns true when the itemlist handle references an empty itemlist. The function *item* provides random access to the items on a whiteboard; given a handle to a whiteboard and a unique identifier for an item it returns a copy of the item. Finally, consider the operations to move items between lists on the whiteboard. The procedure *post_item* creates a new whiteboard item containing *Contents*, and places it on the posted list, while the procedure *repost_item* transfers an item from the completed list to the posted list. The procedure *lock_item* locks the specified item by moving it from the posted list to the locked list, and the procedure *unlock_item* unlocks an item by moving it from the locked to the posted list. Lastly, the procedure *complete_item* takes a locked item and places it in the completed list, while the procedure *delete_item* removes an item from the whiteboard.

After the senior student had presented and explained these operations to the specification expert, discussion of the underlying issues began in earnest.

2.2. Issues Raised by the Specification Process

The first question to arise was how to specify the persistent data store using PLEASE. One solution was suggested by the original package specification; persistence could be specified by declaring the stored objects as static variables local to the *Whiteboard* package. As in the original specification, the package would provide operations to create, access, and modify the whiteboards in the store. Unfortunately, we found that this solution makes the specification of concurrency constraints difficult; the effective unit for concurrency control is a single whiteboard,

not the entire collection. The concurrency constraints can be specified simply if all operations on a whiteboard are *serializable* [27]. The serializability of operations can be easily modeled if each whiteboard is represented as a Unix process, or Ada *task*, with local variables used to store items. Each whiteboard must be able to accept requests to access or modify its contents. We found it easy to model this as message passing; other processes interact with a whiteboard by sending a request message and waiting for a reply. Since a process has only one thread of execution, only one message can be processed at a time.

While this representation makes serialization of whiteboard operations, and thereby specification of concurrency constraints, simple, it leaves some unanswered questions regarding the persistence of whiteboards. If each whiteboard is a process with local variables for its state, then persistence is determined by the lifetimes of processes. To create a new whiteboard a process must be invoked and to destroy a whiteboard a process must be terminated. Fortunately, these situations can be provided for by defining a *manager* for the whiteboard type. The whiteboard manager is itself a process, which controls a sub-process for each whiteboard in existence. Requests to create, open, or close whiteboards are handled by the manager.

The next issued to arise was the meaning of a name, and how it differed formally from a handle. In the prototype version of REBUS, the meanings of these two things were driven by implementation concerns. Handles were implemented as actual pointers to linked lists of items, and *item_ids* were implemented as indices into an array of items. Arbitrary items could be accessed by using the *item* function. An item could be on two whiteboards simultaneously, requiring concurrency control on the item level. Names were not unique identifiers for whiteboards, merely convenient labels which could be bound to new whiteboards as needed. There was no control over the circumstances under which the binding of a name to a whiteboard could change. The same name could refer to different whiteboards at the same time but for different users, and to different whiteboards at different times for the same user.

These characteristics had not been specified or even contemplated. Rather, they emerged from the implementation. Understanding of these characteristics came as the result of grappling with the need to formalize what had been implemented. This understanding caused some dismay and a strong desire to specify and build a conceptually cleaner version of REBUS. In the formal specification, names and handles are both unique identifiers for whiteboards. Each whiteboard has only one name, and the same name refers to the same whiteboard for all users in all situations. Handles are issued by the appropriate manager on an *open* and are necessary to invoke operations on an

object. To make the efficiency aspects of handles explicit, they are formally specified as pointers.

Once these issues had been raised and resolved, formal specifications of REBUS were quickly generated. We also noted that REBUS was then easier to understand and explain. We believe this is leading to a cleaner notion of the requirements specification process.

3. Formal Specification of REBUS

The formal specification of REBUS centers on four components: an object manager, the whiteboard, the REBUS interface, and a top-level showing how the components are combined and interact. The specification does not completely define the objects created and used by REBUS or the policies used to control their development; however, it does define certain minimal characteristics of these objects and the mechanism used to implement these policies. Thus the specification is intended to be sufficiently general that it is reusable. Parts of it are fixed while others are parameters. While REBUS mandates certain methods and mechanisms, such as the use of a whiteboard, instances of REBUS can be created to support different policies and object types. For example, the object manager is defined in such a way that it may be instantiated to manage objects of any type. In REBUS, we instantiate a manager for requirements objects. The whiteboard is also generic in that a whiteboard may be created for any kind of item. REBUS itself is generic in that instances can be created for different types of objects and to implement different management policies. To understand the specification in detail, we will first examine the object manager package.

3.1. The Object Manager

Figure 2 and Figure 3 show the PLEASE specification for the object manager. Its specified as a generic package with parameters to specify the type of objects to be managed, the history information to be kept for each object, and the policies to be used for management. The specification exploits Ada generic packages, which is particularly appropriate as REBUS is written in Ada, but was also found to be semantically appropriate as well. In Ada, a generic construct is instantiated with an actual set of parameters to produce a usable component. In our specification, an instance of *manager_pkg* implements an object manager using a task called *manager*. In Ada, tasks have independent threads of execution and may contain variables private to the task. Inter-task communication is accomplished with *rendezvous* at entry points. When one task calls an entry point and another task accepts the call a rendezvous occurs and the two threads of execution are merged until the call completes. In our specification, the object

```
type object_rec is record
    Name   : name_type  ;
    Object : obj_handle ;
    Info   : info_type  ;
end record ;

type operation_type is (create,destroy,open,close) ;

        •
        •
        •

generic

    type obj_handle is access object_type ;
    --: with predicate is_initial_object( Object : in out obj_handle ) ;

    type info_type  is private ;
    --: no_info : constant info_type ;
    --: with predicate is_okinfo ( Old_info : in out info_type      ;
    --:                            User     : in out user_type      ;
    --:                            Op       : in out operation_type ;
    --:                            New_info : in out info_type    ) ;

    --: with predicate can_create ( User   : in out user_type    ;
    --:                             Name   : in out name_type  ) ;
    --: with predicate can_destroy( User   : in out user_type    ;
    --:                             Object : in out object_rec ) ;
    --: with predicate can_open   ( User   : in out user_type    ;
    --:                             Object : in out object_rec ) ;
    --: with predicate can_close  ( User   : in out user_type    ;
    --:                             Object : in out object_rec ) ;

package manager_pkg is

    task type manager is

        •
        •
        •

    end manager ;
    type manager_handle is access manager ;

end manager_pkg ;
```

Figure 2. Specification of generic manager package

manager task has an entry for each operation it provides; other tasks can call the manager to *create* a new object, *destroy* a pre-existing object, *open* an object for further manipulation, or *close* an open object.

For the purpose of specification, the manager task has a *virtual state* consisting of a set of object records. Each item of type *object_rec* has three fields: the name of the object, a pointer to the object, and the information on the object's history being kept by the manager. The specification of the set of object records incorporates an invariant stating that no two names refer to the same object, and no two objects have the same name. The invariant makes use of *predicates* which are defined elsewhere. In PLEASE, predicates are similar to Boolean functions and are used to specify logical properties. For example, the predicate *member(Elmt,Set)* is true if *Elmt* is currently a member of *Set*. In PLEASE, a *predicate* definition syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. Predicate definitions describe acceptable tuples in a format which has a simple translation into Prolog procedures.

The parameters used to instantiate *manager_pkg* define the properties of the resultant manager. The types *object_handle* and *object_type* and the predicate *is_initial_object* define the objects to be manipulated by the manager. For example, *is_initial_object* defines the initial or "empty" instance of *object_type*; *is_initial_object(Object)* is true only if *Object* is "empty". The type *info_type*, the constant *no_info*, and the predicate *is_okinfo* define the history information to be kept for each object. For example, *is_okinfo(Oldinfo,User,Op,Newinfo)* is true only if *Newinfo* is the correct history information for an object which had history *Oldinfo* before *User* performed *Op*. The predicates *can_create*, *can_destroy*, *can_open*, and *can_close* define the policy implemented by the manager.

The manager uses a simple mechanism to implement the policy: each entry has a pre-condition, which must be true for a rendezvous to begin, and a post-condition, which states the conditions which must hold when the rendezvous is complete. In the specification, the state before execution begins is denoted by *in(...)*, while the state after execution has completed is denoted by *out(...)*. For example, the pre-condition for *create* states that there is no record for an object called *Name*[1] and that *User* has the proper permissions. The post condition for *create* states that an initialized object called *Name* has been created with the proper history information. Similarly, the pre-condition for *destroy* states that there is an object with the specified name and *User* is authorized to destroy it. The post-condition for *destroy* states that the object has been deleted from the set of object records[2]. As *manager_pkg* is generic, managers can be created which implement different policies using the same mechanism.

---

[1] *cnot* is a closed world not, which in this case is acceptable.

[2] Since *Obj0* is a constant, it must be the same in the input and output states.

```
task type manager is

    --: Objects : set of object_rec ;
    --|           where for all Obj1,Obj2 : object_rec =>
    --|                 member(Obj1,Objects) and member(Obj2,Objects) and
    --|                 ( Obj1.Name = Obj2.Name or Obj1.Object = Obj2.Object )
    --|                     -> Obj1 = Obj2  ;

    --: Obj0 : constant object_rec ;

    entry create( User : in user_type ; Name : in name_type ) ;
        --| where in ( cnot member((Name,_,_),Objects) and can_create(User,Name) ),
        --|       out( is_okinfo(no_info,User,create,(Info:info_type))
        --|            and is_initial_object((Obj:object_handle)) and
        --|            Objects = insert((Name,Obj,Info),in(Objects)) ) ;

    entry destroy( User : in user_type ; Name : in name_type ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_destroy(User,Obj0) ),
        --|       out( Objects = delete(Obj0,in(Objects)) ) ;

    entry open( User : in user_type ; Name : in name_type ; Handle : out obj_handle ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_open(User,Obj0) ),
        --|       out( (Obj:object_rec).Name = Name and
        --|            Obj.Object = Obj0.Object = Handle and
        --|            is_okinfo(Obj0.Info,User,open,Obj.Info) and
        --|            Objects = insert(Obj,(delete(Obj0,in(Objects)))) ) ;

    entry close( User : in user_type ; Name : in name_type ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_close(User,Obj0) ),
        --|       out( (Obj:object_rec).Name = Name and Obj.Object = Obj0.Object and
        --|            is_okinfo(Obj0.Info,User,close,Obj.Info) and
        --|            Objects = insert(Obj,(delete(Obj0,in(Objects)))) );

end manager ;
```

Figure 3. Specification of generic manager

The object manager ultimately controls the objects produced and used in the requirements specification process; however, users must access these objects through a whiteboard.

3.2. The Whiteboard

Figure 4 and Figure 5 show the PLEASE specification of the whiteboard. Like the object manager, the whiteboard is specified as a generic package. Parameters to the package define the type of objects to be manipulated using the whiteboard, the states these objects move between, the history information to be kept for the objects, and

```
type item_type is record
    Object : obj_handle  ;
    State  : state_type  ;
    Info   : info_type   ;
end record ;
type item_set is set of item_type ;

type operation_type is (add,move,remove,getobjects) ;

        • • •

generic

    type obj_handle is access object_type ;
    type obj_set    is set of object_handle ;
    type state_type is (<>) ;

    type info_type is private ;
    --: no_info : constant info_type ;
    --: with predicate is_okinfo ( Old_info : in out info_type      ;
    --:                            User     : in out user_type      ;
    --:                            Op       : in out operation_type ;
    --:                            State    : in out state_type     ;
    --:                            New_info : in out info_type )    ;

    --: with predicate can_add       ( User   : in out user_type    ;
    --:                                Object : in out obj_handle   ;
    --:                                State  : in out state_type ) ;
    --: with predicate can_move      ( User   : in out user_type    ;
    --:                                Item   : in out item_type    ;
    --:                                State  : in out state_type ) ;
    --: with predicate can_remove    ( User   : in out user_type    ;
                                      Item   : in out item_type  ) ;
    --: with predicate can_getobjects( User   : in out user_type    ;
                                      State  : in out state_type ) ;

package whiteboard_pkg is

    task type whiteboard is

        • • •

    end whiteboard ;
    type whiteboard_handle  is access whiteboard ;

end whiteboard_pkg ;
```

Figure 4. Specification of generic whiteboard package

the management policies to be followed. As in the case of the object manager, an instance of the whiteboard package implements a whiteboard as an Ada task with an entry for each operation provided. In the specification, the

whiteboard task has a virtual state consisting of a set of items. Each item contains a pointer to an object, the current state of the object and some history information. An invariant specifies that there can be only one item for any object. The whiteboard task provides four entries: *add*, *move*, *remove* and *getobjects*. Each entry has a pre-

```
task type whiteboard is

      --: Items : item_set := {} ;
      --|         where for all I1,I2 : item_type =>
      --|               member(I1,Items) and member(I2,Items) and
      --|               I1.Object = I2.Object -> I1 = I2 ;

      --: Item0 : constant item_type ;

      --: entry predicate is_state( Object : in out obj_handle   ;
      --:                           State  : in out state_type ) ;

      entry add( User   : in user_type    ;
                 Object : in obj_handle   ;
                 State  : in state_type ) ;
          --| where in ( cnot member((Object,_,_),Items) and
          --|             can_add(User,Object,State) ),
          --|       out( is_okinfo(no_info,User,add,State,(Info:info_type))
          --|            and Items = insert((Object,State,Info),in(Items)) ) ;

      entry move( User   : in user_type    ;
                  Object : in obj_handle   ;
                  State  : in state_type ) ;
          --| where in ( Item0.Object = Object and member(Item0,Items)
          --|            and can_move(User,Item0,State) ),
          --|       out( is_okinfo(Item0.Info,User,move,State,(Item:item_type).Info)
          --|            and Item.Object = Item0.Object and Item.State = State
          --|            and Items = insert(Item,(delete(Item0,in(Items)))) ) ;

      entry remove( User : in user_type ; Object : in obj_handle ) ;
          --| where in ( Item0.Object = Object and member(Item0,Items)
          --|            and can_remove(User,Item0) ),
          --|       out( Items = delete(Item0,in(Items)) ) ;

      entry getobjects( User     : in user_type     ;
                        State    : in state_type    ;
                        Sobjects : out object_set ) ;
          --| where in ( can_getobjects(User,State) ),
          --|       out( for all Item : item_type =>
          --|              member(Item,Items) and Item.State = State ->
          --|                  member(Item.Object,Sobjects) ) ;

end whiteboard ;
```

Figure 5. Specification of whiteboard task

condition stating the conditions necessary for the operation to be invoked and a post condition stating the requirements for successful execution. The operation *add* puts a pre-existing object onto the whiteboard while *remove* takes an object off. The operation *move* puts on object already on the whiteboard into a different state while *getobjects* allows the user to examine all the objects in a certain state.

To be precise, users do not access the whiteboard directly, but only interact with the system through the REBUS interface.

3.3. REBUS Interface

Figure 6 and Figure 7 show the specification of the REBUS interface. Like the object manager and whiteboard, the REBUS interface is specified as a generic package. Parameters to the interface package specify the type of objects to be manipulated, the states these objects move between, the history information to be kept by REBUS, the management policies to be followed, and the analyses that can be performed by the user. Like the object manager and whiteboard, an instance of the REBUS interface package implements REBUS as a task containing an entry for each operation provided to the user. In the specification, the REBUS task has a virtual state consisting of a set of records, each of which contains a pointer to an object as well as history information. An invariant specifies that there can be only one record for each object. In the specification, the interface provides five operations: *create*, *edit*, *move*, *getobjects* and *analyze*. The operation *create* allows the user to create a new object while *edit* allows objects to be modified. The *move* command transfers an object to the specified state; when an object is created its state is not defined, so *move* must be used to initialize the state. The *get_objects* operation allows the user to examine all the objects in a particular state, while the *analyze* command may be used to invoke an analyzer on a set of objects.

The object manager, whiteboard, and REBUS interface (as well as the users) are the principal elements in the specification process. We can now examine how these elements are combined and interact.

3.4. The Specification Process

Figure 8 shows a skeletal specification of the requirements specification process. The process contains multiple users who create, modify and combine objects to construct complete specifications. The objects are controlled by an object manager and the users interact via the whiteboard. The objects used by, as well as the policies used to control, the process are specified separately from the structure of the process and the mechanism used to implement

```
type node_type is record
    Object : obj_handle  ;
    Info   : info_type   ;
end record ;
type node_set is set of node_type ;

type operation_type is (create,edit,move,get_objects,analyze) ;

        • • •

generic

    type obj_handle is access object_type ;
    type obj_set    is set of obj_handle ;
    --: predicate is_initial_object( Object : in out obj_handle ) ;

    type state_type  is (<>) ;

    type info_type is private ;
    --: no_info : constant info_type ;
    --: with predicate is_okinfo    ( Old_info : in out info_type      ;
    --:                               User     : in out user_type      ;
    --:                               Op       : in out operation_type ;
    --:                               State    : in out state_type     ;
    --:                               New_info : in out info_type    ) ;

    --: with predicate can_create     ( User  : in out user_type    ) ;
    --: with predicate can_edit       ( User  : in out user_type      ;
    --:                                 Node  : in out node_type    ) ;
    --: with predicate can_move       ( User  : in out user_type      ;
    --:                                 Node  : in out node_type      ;
    --:                                 State : in out state_type   ) ;
    --: with predicate can_get_objects( User  : in out user_type      ;
    --:                                 State : in out state_type   ) ;

    type analyzer_name is (<>) ;
    analyzers : array(analyzer_name) of analyzer_type ;
    --: with predicate can_analyze( User     : in out user_type       ;
    --:                            Analyzer : in out analyzer_name ) ;
    type analysis_type is private ;
    --: with predicate is_okanalysis ( Objects  : in out obj_set         ;
    --:                                Analyzer : in out analyzer_name    ;
    --:                                Analysis : in out analysis_type ) ;
package rebus_pkg is

        • • •

end rebus_pkg ;
```

Figure 6. Specification of generic rebus package

```
task type rebus is

      --: Nodes : node_set := {} ;
      --|         where for all N1,N2 : node_type =>
      --|               member(N1,Nodes) and member(N2,Nodes) and
      --|               N1.Object = N2.Object -> N1 = N2 ;

      --: N1,N2 : constant node_type ;

      entry create( User : in user_type ; Object : out obj_handle ) ;
          --| where in ( can_create(User) ),
          --|       out( is_okinfo(no_info,User,create,_,(Info:info_type)) and
          --|            is_initial_node(Object) and
          --|            Nodes = insert((Object,Info),in(Nodes)) ) ;

      entry edit( User : in user_type ; Object : in obj_handle ) ;
          --| where in ( N1.Object = Object and member(N1,Nodes) and
          --|            can_edit(User,N1) ),
          --|       out( is_okinfo(N1.info,User,edit,_,N2.Info) and
          --|            N2.Object = Object and
          --|            Nodes = insert(N2,delete(N1,in(Nodes))) ) ;

      entry move( User   : in user_type    ;
                  Object : in obj_handle   ;
                  State  : in state_type ) ;
          --| where in ( N1.Object = Object and member(N1,Nodes)
          --|            and can_move(User,N1,State) ),
          --|       out( is_state(Object,State) and N2.Object = Object and
          --|            is_okinfo(N1.Info,User,move,State,N2.Info) and
          --|            Nodes = insert(N2,delete(N1,in(Nodes))) ) ;

      entry get_objects( User     : in user_type   ;
                         State    : in state_type  ;
                         Sobjects : out obj_set  ) ;
          --| where in ( can_get_nodes(User,State) ),
          --|       out( for all N : node_type =>
          --|              member(N,Nodes) and is_state(N.Object,State) ->
          --|                  member(N.Object,Sobjects) ) ;

      entry analyze( User     : in user_type     ; Sobjects : in obj_set          ;
                     Analyzer : in analyzer_name ; Analysis : out analysis_type ) ;
          --| where in ( can_analyze(User,Analyzer) ),
          --|       out( is_okanalysis(Sobjects,Analyzer,Analysis) ) ;

end rebus ;
```

Figure 7. Specification of rebus task

```
task body specification is

        -- Specification of types

        type object_type     is ... ;
        type state_type      is ... ;
        type obj_info        is ... ;
        type whiteboard_info is ... ;
        type rebus_info      is ... ;

                • • •

        -- Specification of policy predicates

        predicate can_create_node ( ... ) ;
        predicate can_destroy_node( ... ) ;
        predicate can_open_node   ( ... ) ;
        predicate can_close_node  ( ... ) ;

                • • •

        -- Specification of users (who can invoke rebus)

        task type user is

                • • •

        end user ;
        Users : array( user_names ) of user ;

        package obj_pkg is new manager_pkg    ( ... ) ;
        package wb_pkg  is new whiteboard_pkg( ... ) ;
        package rbs_pkg is new rebus_pkg      ( ... ) ;

        object_manager : obj_pkg.manager   ;
        whiteboard     : wb_pkg.whiteboard ;

begin

        whiteboard     ;                    -- start the whiteboard,
        object_manager ;                    --   object manager,

        for I in user_names loop
                User(I) ;                   --   and users
        end loop ;

end specification ;
```

Figure 8. The specification process

the policies. To describe the specification process we must first define the type of objects from which the specification is to be created, the states that these objects can be in, and the history information to be kept for objects

at each level of the system. This is accomplished by the types *object_type*, *state_type*, *obj_info*, *whiteboard_info*, and *rebus_info* respectively. Next, we specify the policies to be followed during the specification process. These policies are defined by the predicates *can_create_node*, *can_destroy_node*, *can_open_node*, *can_close_node*, etc. These predicates are then used in the instantiation of the object manager, whiteboard and REBUS interface. It is interesting that some policies can be conveniently described at any of the three levels (object manager, whiteboard or REBUS interface), while others may be easily specified only at a particular level. Finally we describe what actions the users can perform and when. This is accomplished by the task type *user*.

The formal specification of REBUS is now concluded. We have not completely defined all of the objects created and used by REBUS or the policies used to control their development. However, we have defined certain minimal characteristics of these objects and the mechanisms used to implement these policies, as well as showing how they could be completely defined. We can view the specification as a process program specification: parts of the process are fixed while other aspects are parameters. We can see this as static process programming: while REBUS mandates certain methods and mechanisms, such as the use of a whiteboard, instances of the system can be created to support different policies and object types. We feel the creation of this specification has greatly increased our understanding of REBUS and the requirements specification process.

4. Summary and Conclusions

We believe that formal specification can enhance the software development process in a number of ways. Specifically, we believe that software development can be enhanced by creating a formal specification after a prototype has already been constructed using traditional techniques. We have performed an experiment which supports this hypothesis in the context of the Arcadia project, an effort to experimentally develop superior software environment technologies. One of Arcadia's current goals is the creation of REBUS, a REquirements BUilding System. In REBUS, a requirements specification is viewed as an attributed DAG which may have to satisfy certain constraints. In REBUS, multiple developers interact via objects stored on the whiteboard, a communications and coordination mechanism.

In the Fall of 1987, a prototype version of REBUS had been constructed and was undergoing redesign and extension. We conjectured that describing the system using a small number of formally defined concepts could increase understanding and enhance communication between project members. Over the course of three months,

the developers met with a specification expert and a formal description was written in PLEASE, an Ada-based, wide-spectrum, executable specification and design language which supports formal methods similar to VDM. Several issues surfaced during construction of the specification, including persistence and concurrency control for whiteboards, as well as the meanings of names and handles. Construction of the formal specification enhanced further development by helping the developers to clarify the architecture of the system, separate their concerns, and discuss possible modifications. We feel the use of similar methods can enhance other software development efforts.

5. References

1. Bjorner, D., T. Denvir, E. Meiling and J. S. Pedersen, "The RAISE Project - Fundamental Issues and Requirements", RAISE/DDC/EM/1, Dansk Datamatik Center, 1985.
2. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
3. Bloomfield, R. E. and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 988-993.
4. Chang, C. and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
5. Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
6. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.
7. Gehani, N. and A. D. McGettrick, eds., *Software Specification Techniques*, Addison Wesley, Reading, Massachusetts, 1986.
8. Heimbigner, D., Stanley Sutton, Jr. and L. J. Osterweil, "APPL/A: A Language for Managing Relations", Report No. CU-CS-374-87, Department of Computer Science, University of Colorado at Boulder, 1987.
9. Hudson, S. and R. King, "CACTIS: A Database System for Specifying Functionally-Defined Data", *Proceedings of the Workshop on Object-Oriented Databases*, September 1986, 26-37.
10. Hudson, S. E. and R. King, "Object-Oriented Database Support for Software Environments", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1987, 491-503.
11. Hudson, S. and R. King, "The Cactis Project: Database Support for Software Engineering", *IEEE Transactions on Software Engineering*,, June 1988.
12. Hudson, S. and R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM Transactions on Database Systems*, to appear.
13. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
14. Luckham, D. C. and F. W. Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software 2, 2* (March 1985), 9-22.
15. Luckham, D. C., D. P. Helmbold, S. Meldal, D. L. Bryan and M. A. Haberler, "Task Sequencing Language for Specifying Distributed Ada Systems, TSL-1", Report No. CSL-TR-87-334, Computer Systems Laboratory, Stanford University, July 1987.
16. Nii, H. P., "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures", *The AI Magazine 7*, 2 (Summer 1986), 38-53.
17. Nii, H. P., "Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective", *The AI Magazine 7*, 3 (Conference 1986), 82-106.
18. Oest, O. N., "VDM From Research to Practice", *Information Processing*, 1986, 527-533.
19. Osterweil, L. J., "Software Processes Are Software Too", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 2-13.
20. Shaw, R. C., P. N. Hudson and N. W. Davis, "Introduction of A Formal Technique into a Software Development Environment (Early Observations)", *Software Engineering Notes 9, 2* (April 1984), 54-79.
21. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the *Journal of Systems and Software*), September 1986.
22. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the *Journal of Systems and Software*), September 1986.
23. Terwilliger, R. B. and R. H. Campbell, "PLEASE: a Language for Incremental Software Development", *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987, 249-256.

24. Terwilliger, R. B., "An Example of Knowledge-Based Development in ENCOMPASS", *Proceedings of the Third Annual Conference on Artificial Intelligence & Ada*, George Mason University, October 1987, 40-55.

25. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 344-354.

26. Terwilliger, R. B., "PLEASE: a Language Combining Imperative and Logic Programming", *SIGPLAN Notices 23*, 4 (April 1988), 103-110.

27. Ullman, J. D., *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1980.

28. *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.