BIZSPEC: A BUSINESS-ORIENTED MODEL FOR SPECIFICATION AND ANALYSIS OF OFFICE INFORMATION SYSTEMS

Trent Jaeger and Atul Prakash
Software Systems Research Laboratory
Department of EECS
University of Michigan, Ann Arbor, MI 48109
E-MAIL: jaegert@eecs.umich.edu and aprakash@eecs.umich.edu

Abstract

Normally, domain-independent methods, such as structure charts, data flow diagrams, and entity-relationship diagrams, are used to model the requirements of a business. We propose a model, called BizSpec, in which each logical business unit is defined using business-oriented concepts: *forms*, *unit flows*, and *policy*. This model has three advantages over domain-independent methods: 1) it uses business-oriented concepts to improve communication between the end users and the systems analysts, 2) it represents each logical business unit independently to help the end users examine their part of the system, and 3) its specifications are executable, so the end users and systems analysts can rapidly generate and analyze prototype systems.

1 Introduction

Businesses rely on their application software to gain a competitive advantage. Thus, the ability to rapidly develop software which meets the needs of the business is a key concern. Usually, business systems are specified using domain-independent modeling tools, such as structure charts [13], data flow diagrams [3], and entity-relationship models [6]. Many systems have been built using these models, but they are difficult for end users to assess in detail. We believe that there are several reasons for this limitation. First, the concepts on which these models are based – data dictionaries, entity relationship diagrams, etc. – are not always familiar to end users. End users typically know their domain well, but not the associated computer science terms for concepts in their domain. Second, these models are used to describe the system independent of the way the business is organized, so it's hard for end users to determine if all their tasks are defined. Also, specifications in these models are not executable. The end users cannot check the accuracy of the final system until it is nearly completed.

One common solution to help ease evaluation of specifications by end users is to improve the user-friendliness of the specification process. Several researchers have investigated the use of high-level interfaces to collect user specifications. An example of this research is the development of knowledge-based specification acquisition systems [9, 10, 12]. In these systems, user specifications are captured in natural language statements or context diagrams. Reasoning mechanisms transform the end user specifications into computable specifications. These computable specifications are represented in domain-independent paradigms, such as semantic networks or logic. The knowledge used in the reasoning mechanisms must be powerful enough to transform the user specifications to these internal representations and translate them back, so that the end user can approve the transformations. We believe that this will be very difficult to do in general, so we propose, as an alternative, a formal specification model for system analysts and end users to improve their ability to work together at a formal specification level.

This paper describes the Business-Oriented Model for System Specification and Analysis of Office Automation Systems (BizSpec), a model designed to improve the process of developing business systems in two ways: 1) improve the ability of end users to understand the specifications, so that they can evaluate whether the system meets their business objectives and 2) develop prototype systems quickly, so analyses of the system's performance and correctness can be made. Business-oriented specification consists of two ideas: 1) familiar end user concepts are used as formal specification entities and 2) the specifications are organized according to the way that the business is organized. We believe business-oriented specifications will require less translation to communicate to end users than previous domain-independent models. Prototypes can be

developed quickly because BizSpec specifications are executable. We believe that this is a major advantage of our approach because BizSpec specifications can be executed and shown to an end user before the final system is developed to make sure that specifications match the desired operation of a business.

The remainder of the paper focuses on our definition of a business-oriented specification model which addresses the issues outlined above. Section 2 describes our general approach. Sections 3 through 5 define the business-oriented concepts which are used in our approach. Section 6 discusses the automated analysis of specifications using BizSpec. Section 7 lists the future work, and Section 8 concludes the paper.

2 Our Approach

Our approach has the following properties: 1) business-oriented concepts are used as our formal representation primitives, 2) the procedures and constraints of each work unit are defined separately, and 3) the specification level is computable, so prototype systems can be generated quickly. The term *work unit* refers to a unit of the business (i.e., department).

The business processing is specified using the following business-oriented concepts: *forms*, *unit flows*, and *policy*. A *form* stores and serves as a means to communicate business data. A *unit flow* is a work unit's procedure which uses forms to complete a transaction. A *policy* of a work unit is a constraint which limits how the unit flows are performed.

BizSpec uses forms as the base concept in the model. Forms are used in a wide range of business processes, so they are, by nature, a general representation. Use of forms as a primitive has been proposed in several systems [1, 14, 4]. In these systems, the forms are defined in terms of the fields on the form and their derivations. A *field* is an item on the form for which values are specified. The *derivations* detail how the value can be computed. The derivations are generally based on the values of the other fields in the form. Spreadsheets, which can be considered as a special case of a form-based system, have been popular business-oriented problem-solving tools for both end users and programmers. In BizSpec we also define the ability to specify constraints on the fields in the form.

A limitation of most, current form-based systems is that they do not represent knowledge which is broader than the scope of a form. A form may be used in different ways based on the transaction to be processed. For example, a student's transcript may be checked at several stages during his academic career. He may use it for financial aid applications or the university may use it to ensure that the student is making satisfactory academic progress. The constraints on and the processes which use the transcript are different in these two cases, so they must be specified independently from the form.

Each work unit defines its set of business procedures in *unit flows*. A unit flow consists of a sequence of steps to perform a task in a work unit. A series of unit flows to complete a single transaction involving multiple work units is called a *business flow*. The features of unit flows are that they: 1) include the processing of a single work unit, so each work unit can manage its processes more directly; 2) use forms explicitly just as a normal business process would; 3) provide a procedural representation of the business process; and 4) can include constraints on the process. Unit flows identify the work performed by each work unit and each work units' interaction with other work units. The interaction between work units could be performed either by broadcasting techniques, like blackboards [5], or direct techniques, such as message passing [8]. We prefer message passing because we believe that work units will have a good idea who they want to communicate with.

Unit flows present a procedural representation of the transaction steps from a work unit's viewpoint. Data flow diagrams, on the other hand, show only the data flows, not the procedure which completes the transaction. We believe that it will be easier for the end users to follow the procedural representation. The basis for this belief is analogous to the belief that people have an easier time identifying cases than rules. It is easier outline a complete sequence than to outline each of step in a sequence and the conditions when they apply independently.

Policy represents the rules of the work unit that are used in performing transactions. These constraints can be applicable to a specific form or unit flow or to all forms and unit flows used by the work unit. For example, a work unit might want to make a profit on all sales transactions. Since this constraint applies to all unit flows within the work unit, we want this constraint to exist beyond the lifetime of any specific form or unit flow. The formal representation entity, policy, has been provided to enable each work unit to define its own set of guidelines.

The specifications of a system using BizSpec can be executed, so prototypes can be generated quickly. This enables the systems analysts and end users to evaluate the specifications early in the design pro-

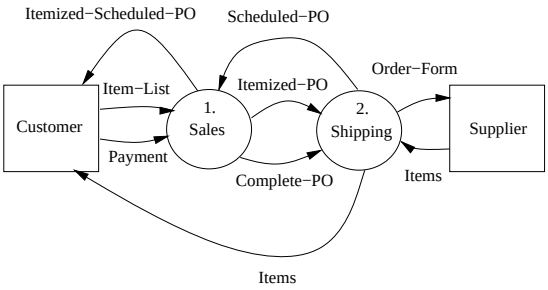


Figure 1: Sales and Shipping Example

Purchase Order Form PO No. 0002

Name: B. Smith

Address: 1111 X St.

Item	Qty	Price/Item	Price	Delivery Dates/Qty
A1000	3	20.00	60.00	9−1−93/2
				10−1−93/1
CZXB3	10	8.75	87.50	4−15−93/10

Approved: AJ Total Price: 147.50 Payment: Credit

Figure 2: Purchase Order Form

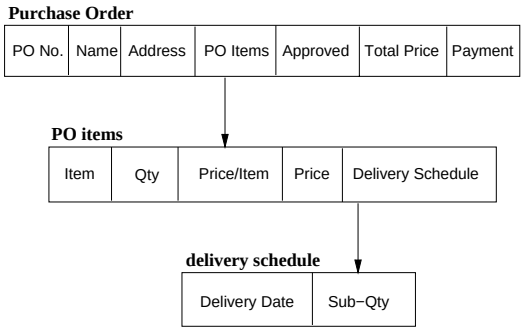


Figure 3: Purchase Order Form – Form Hierarchy

cess. For example, different variations of a policy can be tested to see which is best for the work unit. The types of evaluation we are interested in range from correctness checking to performance improvement. The development of mechanisms which automatically perform these analyses is our future goal.

The remainder of this paper demonstrates the use of the business-oriented concepts to specify a system which solves the problem outlined below. A sales department and a shipping department in a parts distributorship work together to complete sale transactions with customers. A data flow diagram of this process is shown in Figure. First, a salesperson receives a request for a quote for a *item-list* from a customer. The items and their prices are computed which generates the *itemized-PO*. As part of the quote, this distributorship also provides information about the delivery of the items. The shipping department computes the *delivery schedule* and returns a *scheduled-PO* to the sales department. If the customer approves the *itemized-scheduled-PO* and provides *payment* for this order, the salesperson forwards the *completed-PO* to the shipping department. The shipping department sends or orders the items based on their availability.

3 Forms

A *form* in BizSpec corresponds to a form in the business. A person completing a form needs to know: 1) the fields on the form; 2) the way to derive the values for the fields; 3) the relations between the fields; and 4) any constraints between the fields which limit the way they can be used. A system built to use a form must have the same information about it that a person would have.

Consider the *Purchase Order Form* in Figure 2. It has a set of fields for which values can be entered. These include: *PO No.*, *name*, *address*, *item*, and *total-price*. The specification of the form must include these fields. The system needs to know how to derive the values for the fields. For example, we need to know how to compute the *total-price* from the individual *prices* of the items. The system also needs to know which fields are related. It needs to know that the *price/item* is dependent on the *item*. Also, there may be limits on the values which fields can take. An obvious example, is that the *qty* of an item must be greater than 0.

In BizSpec, we use a hierarchy of field aggregations to represent the structure of a form. The field aggregations are called *nodes* and the hierarchy is called a *form hierarchy*. The form hierarchy for the *Purchase Order Form* is shown in Figure 3. There are three nodes (*Purchase Order*, *po-items*, and *delivery schedule*) in this form's structure. The reason that the fields on the form are represented as a hierarchy is that some fields are *non-normalized*. That is, for a specific relationship the one field's value may be entered only once for a set of values of another field. For example in the *Purchase Order Form*, the *PO No.* field will have just one value even though there may be several *items* on the form. The idea of representing a form as a hierarchy of nodes was used previously in [2] as an intermediate point in the derivation of an entity-relationship model from a form.

In BizSpec, each form is defined once for the entire business to ensure consistency in completing busi-

ness flows. For example, when the sales department requests the *delivery schedule* for the *Purchase Order Form* from the shipping department, it needs to tell the shipping department what items to deliver. A good way to do this is to include the *Purchase Order Form* in the message which represents the request. Both the shipping and the sales departments must have a consistent representation of the form, so they can communicate to complete the transaction.

Each form may be defined by information from several departments. In our example, the sales department has no idea how to compute the delivery schedule for an item. Likewise, the shipping department does not care about pricing of items. Therefore, a combination of information from both the shipping and the sales departments is used to define the *Purchase Order Form*. From a representation standpoint, it is only important that we have a single representation of the form in the business. Each department must be able to provide its specification for the parts of the form its responsible for, however. We believe that this is primarily an interface and specification management problem, so we postpone the solution to this problem.

In BizSpec, a form is defined by its constituent nodes in the form hierarchy. Each node is defined separately using the *define-node* macro. Each form is linked to its nodes by giving the topmost node in the form hierarchy the same name as the form. The names of the defined forms are kept in a list. At present, all the specifications are defined in an extended CommonLISP Object System (CLOS) syntax. Conversion to a friendlier interface is part of our future work. An example definition for the *Purchase Order* node is shown in Figure 4.

A form node consists of constraints and field definitions. The *node constraints* limit the values of the fields or the use of fields. In Figure 4, we define the constraint that a purchase for more than $500 made by credit must be approved by a manager. The syntax and use of constraints is detailed in the Policy Section.

A field definition consists of its default value, constraints, and derivations. A default value for a field is specified using the *default* attribute. For example, we set the default *total-price* to 0. Constraints are expressed in two ways: 1) the *value-type-spec* attribute is used to supply the field's data type and 2) the *constraints* attribute is used to declare ad hoc constraints on the field. A typical constraint for the fields in the *Purchase Order* form states that the value of the *total-price* field must be greater than or equal to 0 (see the *total-price* field definition in Figure 4).

```
(define-node purchase-order
  :node-constraints
    ;; This constraint requires that a manager approve
    ;; credit purchases for more than $500
    (when(and (> total-price $500)
              (= payment credit)
    unless(= approved^rank MANAGER)
    then  (surrender "Need manager's approval)))
    ;; A repair unit flow which
    ;; gets a manager's approval goes here
  :fields
  (po-no
    ;; This key is generated by the system
    :value-type-spec integer)
  (name
    :value-type-spec string)
  (address
    :value-type-spec string)
  (po-items
    :value-type-spec po-items)
  (total-price
    :value-type-spec integer
    :derivations
       (iterate
         :for item :in (select po-items)
         :initial (sum = 0)
         :body (sum = (+ sum item^price))
         :return sum)
    :default 0
    :constraints (≥ 0))
  (payment)
  (approved
    :value-type-spec person))
```

Figure 4: Purchase Order Node in the Purchase Order Form

A field's *derivations* define the possible ways that a field's value can be computed. For example, the *total-price* in the *Purchase Order* node view is computed by summing the value of the *price* field in the *po-items* node for each item. This is done using the *select* and *iterate* commands. The *select* command collects all the *po items* in the *Purchase Order Form*. The *iterate* command binds *item* to a new member of *po-items* each time through the loop. Sum is incremented by the *price* of each *item* each time through the loop. The value computed by its derivation is the value of the field.

Derivations imply dependencies between fields. In the style of form-based systems, the syntax of the derivations are structured such that the fields upon which the derivation is dependent can be identified.

The derivation for *total-price* is based on the *po-items* field and the *price* of each *po-items* entry.

To reference the value of a field in a derivation, we traverse the form hierarchy. In the *total-price* derivation, we need to access the *price* field of the *item*. The '^' identifier signifies a chain of fields in the form hierarchy, so it is used to refer to the *price* of *item*. We do not need to use the '^' to get the *po-items* in the select command because *po-items* is defined in this node. Therefore, any of the fields in a node can be used in a derivation in another field in the same node.

4 Unit Flows

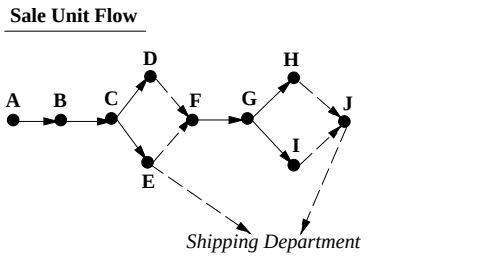
Because a form may be used differently in the context of different transactions, we define *unit flows* to describe how forms are used to complete transactions. Recall the example in the Introduction Section about the student transcripts. Transcripts are used for several purposes. For example, a student's transcript may be used as part of a check for an academic violation or as an application for financial aid. For each transaction, a unit flow is defined which describes the procedure for completing the transaction. Therefore, each of the two uses of a student transcript would be outlined in two separate unit flows.

A *unit flow* is a sequence of steps taken by a work unit to process a transaction. Recall that each work unit defines it own processing specifications, so each work unit defines its own unit flows. Since it is possible that several work units may interact to complete a transaction, a *business flow* contains the set of unit flows which are necessary to complete one transaction.

Recall our example sale transaction from the Our Approach Section. It is detailed graphically in Figure 5. The steps are shown as vertices in the graph, and the directed arcs show the dependencies between steps. In this unit flow, the sales department makes two requests of the shipping department: schedule the items and deliver the items. The communication is performed by the two *send* commands (steps E and J). A separate unit flow in the shipping department would be defined to handle each case. The business flow consists of the execution of all three unit flows.

The unit flows are defined using a macro called *define-unit-flow* which is similar in style to the macros used to define forms. Figure 6 shows an example unit flow definition.

A unit flow is defined by its: 1) *steps* and 2) *constraints*. Constraints in unit flows are added to values of the *constraints* attribute. The *unit flow steps* attribute contains a procedure which describes how to complete the transaction in terms of a sequence of steps. The steps are defined using primitive operations which include: 1) *create* which creates a new instance of a form, 2) *input* which accepts user input, 3) *compute* which propagates values using the form derivations, 4) *send* which sends a message to another work unit, 5) *send-receive* which sends a message and waits for a message to be sent from another work unit before proceeding, and 6) *fork* which declares that the order of computation between sequences of steps is unimportant.



A: Create a new purchase order.
B: Get the information for this order (e.g. PO items, name).
C: Fork because D and E can be accomplished in parallel.
D: Compute the value for the cost field for each item.
E: Ask the shipping department for the delivery schedule for this PO.
F: Input the payment field based on the customer's payment method.
G: Branch: If the total price is above $500 and the purchase is by credit, we need a special approval.
H: Manager approves purchase.
I: Any employee can approve the purchase.
J: Ask the shipping department to deliver the order.

Figure 5: Sale Unit Flow – Process Graph

The progress of the unit flow is maintained by validating that the goals of the steps have been met. Both *input* and *compute* are required to get values for the fields listed in their *fields* argument. Therefore, for the unit flow to make satisfactory progress, the required fields must have values at the completion of the step. The *goal* argument in the *send-receive* command is used for a similar purpose. It ensures that the receiving work unit knows what the result of handling this message must be. In our example, the shipping department must return the *delivery schedule* for the *po-items* in order for the unit flow to continue.

Constraints can either be specified in the form of conditional statements in the unit flow or declaratively by using one of the *constraints* attributes of a form or unit flow. This is demonstrated by the condition that the *branch* step enforces in the unit flow in Figure 6. This condition forces a manager to approve any credit purchase whose value is more than $500. The same condition is enforced by the *node*

```
(define-unit-flow SALE
  :work-unit sales
  :unit-flow-steps
  (create purchase-order)
  (input purchase-order
    :fields name address po-items)
  (fork
    ((compute purchase-order :fields total-price))
    ((send-receive purchase-order :who shipping
      :goal purchase-order^ po-items^ delivery-schedule)))
  (input purchase-order :fields payment)
  (branch
      (and (> purchase-order^ total-price $500)
           (= purchase-order^ payment credit))
    ((input purchase-order :fields approved
      :who manager))
    ((input purchase-order :fields approved)))
  (send purchase-order :who shipping)
  :constraints
  ;; Indicates when a sale is for a loss
  (when purchase-order^ po-items
    if (< purchase-order^ total-price
      (iterate
          :for item1 :in (select purchase-order^ po-items)
          :initial (cost = 0)
          :body (cost = (+ cost
            (* item1^ qty
              (select cost-form^ item^ cost
              :for item = item1))))
          :return cost))
    then (inform "Sale is NOT profitable")
    ;; This repair unit flow records that the policy has
    ;; failed on the unprofitable-sales form
    and do (add unprofitable-sales
      :field (po-no purchase-order^ po-no))))
```

Figure 6: Sale Unit Flow Definition

constraint in the form node definition in Figure 4. Since the node constraint applies to all uses of this form it must met in the execution of this unit flow. Therefore, it is not really necessary to add the conditional to the unit flow. We added the conditional to show how the *branch* command is used.

5 Policy

In our definitions of forms and unit flows, we allow the specification of constraints. There are some constraints which may apply beyond the scope of a single form or unit flow, however. Some other examples of constraints which apply to a wide variety of processes are listed below.

1. Whenever a business transaction is made, make sure it is for a profit.
2. Always let a customer know the conditions of an agreement.
3. Do not ever take more than 2 months to deliver an order.

Rather than having to associate these constraints with all unit flows involved, we would like to supply these constraints only once. In BizSpec, this type of constraint is considered to be part of the policy of the work unit. The concept of *policy* has been created to enable these constraints to be added independently from any unit flow or form.

In general, all constraints in the system are converted to the same representation, so the discussion in this section subsumes the definition of all constraints in BizSpec. For simplicity, this discussion is based on the constraints defined in the forms and unit flows above. Note that policy and constraint have the same meaning. The main example we will use is the constraint in Figure 6 which signals when a sale is not for a profit (see the *constraints* attribute).

A policy is defined is its: 1) *context*, 2) *condition*, and 3) *response*. The context defines the situation in which the condition can be executed in terms of the forms and unit flows which are active, (i.e., instantiated within the business unit flow). The context of a constraint is indicated by the *when* statement in the constraint definition. By this statement, the constraint in Figure 6 does not need to be tested until the *Purchase Order Form* has a value for the *po-items* field.

The *condition* of the constraint is the predicate to be tested. The condition is defined in the *unless* statement. The inputs to that predicate must be available before it can be run. In our constraint, the inputs we need are the *po-items* and the *total-price*. The predicate compares the total price against the total cost of each item. If the total cost is greater than the total price, then the predicate is false and the constraint's response is run.

In a production rule system, such as OPS5 [7], the context and the conditions comprise the antecedent (or left-hand side) of the rule. We separate the state-oriented antecedents from the predicate antecedents for two reasons: 1) to more easily distinguish state-based antecedents from the predicates and 2) to reduce the amount of run-time rule maintenance by "compiling" the rules into the positions in the unit flows where their contexts are active.

The *response* is used to specify an action to take when the constraint is violated (its condition is false). The response consists of two parts: the *severity information* and the *repair unit flow*. The sever-

ity information is contained in the *then* statement and includes a *severity* and a *severity message*. The severity is a symbol (e.g. *surrender*, *inform*, etc.) which designates the importance that the policy not be violated. The severity message is a note which provides some immediate feedback to the user about the violation. The repair unit flow lists the actions which should be taken when a constraint is violated. The repair unit flow is indicated by the *and do* statement. For the case of the sale not being profitable, the unit flow just records the unprofitable sale cases, so that they can be analyzed to determine which prices should be adjusted.

The specification of policy is not typically identified as an individual dimension in the specification of a business system. As we saw in the definition of *sale unit flow*, there was an option for representing the policy as part of the unit flow, a constraint on the unit flow, or as a general constraint in the department. If this constraint is a general constraint in the department, the maintenance of such a constraint will be eased considerably if it is only represented in one place. Defining constraints as policy enables us to do this.

6 Analysis of Specifications

A key goal in the design of BizSpec has been that the specifications should be analyzable for internal consistency and executable for evaluating work unit behavior. Below are several types of analysis that can be done on BizSpec specifications:

- *Communication consistency:* This analysis allows us to answer questions such as: are there classes of messages (forms or requests) that are sent by one work unit to another work unit, but not handled by the receiving work unit? Or are there classes of messages that are expected by a work unit from other specific work units but not sent by those work units?

  For instance, in Figure 6, *Purchase Order Form* is sent by sales department to the shipping department. This form should be handled by some unit flow in the shipping department.

- *Potential communication inconsistency:* Are there classes of messages that are expected by a work unit but no work unit is sending them? This may be an inconsistency, but not always. The work unit that is supposed to send them may be external to the system and thus not specified. In any case, warning should be given so that the analyst is aware of the potential inconsistency.

- *Form completeness:* There should be a way of computing each field on a form – either as an input, by initialization, or by deriving its value from other fields.

While the above types of analyses can give valuable feedback about the specifications, even more important, in our view, is that BizSpec specifications are executable. They can be executed and an end user can see whether the specifications correctly model a work unit or a collection of work units. We have prototyped an execution environment for BizSpec specifications in CLOS that we are currently using to predict whether the proposed system will meet the timing constraints of the business. Typically, businesses have goals for how long transactions should take, so it will be useful if a simulation of the specifications can predict whether the proposed system will meet the desired performance goals. To do that, we extended BizSpec specifications to allow execution time estimates to be attached to individual steps. Concept of *resources* is also added to model resources that are needed to execute steps. A common example of a resource is people (who do a step, etc.). The simulator gives information on average, peak, and standard deviation of utilization of resources, time required to execute an instance of a business flow, and time spent by each step waiting for resources. We believe that such insight from specifications can be very useful in planning.

7 Future Work

In addition to the static and simulation analyses of the specifications mentioned above, the other major areas for future work are listed below.

- *Database design:* Forms provide us with a familiar concept in the business which identifies the fields used in the business and their definitions. Our goal is to define the concept of forms such that a physical data model can be derived. In [2], they demonstrate the issues involved in deriving a data model from specifications.

- *Specification interface:* Higher-level interfaces which support the entry of specifications by both end users and systems analysts is an area of interest. End users may be able to supply some basic information including forms, examples of processing using the forms, and high-level descriptions of policy.

8 Conclusions

The goal of any specification formalism is to be able to collect the specifications of the domain quickly and accurately. In the development of BizSpec, we concentrate on improving the accuracy of the specification process for office information systems. This is done in three ways: 1) by using business-oriented representational concepts as formal representation entities; 2) by making the structure of the specification of unit flows and policy mirror to the structure of the business; and 3) by making the specifications executable. Business-oriented specification will require less translation between system concepts and business concepts. Because the translation process will be simplified the addition of specification interfaces should reduce the time required to capture the specifications as well.

The ability to evaluate and adapt a business system is also becoming a vital part of the business system development process. In BizSpec, the results of changes are immediate because the specified business system is executable. Since business-oriented concepts are used to define the system, we hope that BizSpec will allow end users and systems analysts to work together to evaluate and update their business system as desired.

References

[1] M. Burnett and A. Ambler. Generalizing event detection and response in visual programming languages. Technical report, Michigan Technological University, 1991.

[2] J. Choobineh, M. Mannino, and V. Tseng. A form-based approach for database analysis and design. *Communications of the ACM*, 35(2):108–120, February 1992.

[3] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1979.

[4] W. Du and W. Wadge. A 3d spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, May 1990.

[5] L. Erman, F. Hayes-Roth, V. Lesser, and R. Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 1980.

[6] M. Flavin. *Fundamental Concepts of Information Modeling*. Yourdon Press, 1981.

[7] C. Forgy. Ops5 user's manual. Technical report, Carnegie-Mellon University, July 1981.

[8] L. Gasser, C. Braganza, and N. Herman. Implementing distributed artificial intelligence systems using mace. In *Proceedings of the Third IEEE Conference on Artificial Intelligence Applications*, pages 315–320, 1987.

[9] W. Johnson and M. Feather. Using evolution transformations to construct specifications. In *Automating Software Design*, chapter 4, pages 65–91. AAAI Press, 1991.

[10] V. Kelly and U. Nonnenmann. Reducing the complexity of formal specification acquisition. In *Automating Software Design*, chapter 3, pages 41–64. AAAI Press, 1991.

[11] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1982.

[12] M. Lubars. The rose-2 strategies for supporting high-level software design reuse. In *Automating Software Design*, chapter 5, pages 93–118. AAAI Press, 1991.

[13] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, 1980.

[14] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Language and Computing*, 3(3), September 1992.