ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Antony Bryant and Lesley Semmens (Eds)

Methods Integration

Proceedings of the Methods Integration Workshop, Leeds, 25-26 March 1996

Paper:

A Tale of Two Case Studies: Using Integrated Methods to Support Rigorous Requirements Specification

R.B. France, J. Wu, M.M. Larondo-Petrie, and J.-M. Bruel

Published in collaboration with the
British Computer Society

BCS

Springer

A Tale of Two Case Studies: Using Integrated Methods to Support Rigorous Requirements Specification

R. B. France, J. Wu, M. M. Larrondo-Petrie, and J-M. Bruel*
Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431, USA.
Email: {robert,jie maria bruel}@cse.fau.edu

August 28, 1996

Abstract

Integrated formal and informal specification techniques (FISTs) have been the focus of a number of research projects since the mid-eighties. Research in this area aim at producing specification techniques that integrate concepts and notations used in mature formal specification techniques (FSTs) and popular graphical modeling methods such as *Structured Analysis* (SA) and *Object-Oriented Analysis* (OOA).

In this paper we illustrate, using the results of two case studies, two roles FSTs can play in the context of less formal graphical requirements modeling and analysis techniques. In the first case study discussed an extended Petri Net model is used to prototype a textbook SART (SA/Real-Time) model. In this case, the formal model acts as a prototype, and is used to dynamically validate the requirements expressed in the SART model. In the second case study an integrated OOA method (Fusion) and FST (Z) is used to create requirements models that are graphical and analyzable. In this case, the formal models act as more precise representations of the requirements captured by the graphical models.

1 Introduction

Software requirements engineering is concerned with the systematic analysis and specification of software requirements. One can view requirements engineering as a systematic approach to problem analysis and specification, where the goal is to obtain a precise statement of the problem in terms of goals that must be met by the implemented system.

The graphical model-based techniques of *Structured Analysis* (SA) and *Object-Oriented Analysis* (OOA) (henceforth referred to as informal specification techniques (ISTs)) can provide the flexibility (in terms of ease of change) and modeling constructs needed to explore appropriate abstractions for problem concepts. On the other hand, the lack of firm semantic bases for these methods limits their effectiveness in validation and subsequent verification activities.

The need for precision suggests the use of *formal specification techniques* (FSTs). FSTs utilize mathematical concepts and notation to precisely define theories and models of application behavior. Precise specifications facilitate effective communication among persons with a stake in the development of the software. The ability to reason about properties captured in formal specifications allows developers to rigorously assess the adequacy of their models.

FSTs and ISTs can complement each other in a software development project. For example, variants of the SA method [6] are among the most widely used requirements specification and analysis methods in industry. Their simple, intuitively-appealing specification concepts and notation are major factors behind

*This work was partially funded by NSF grant CCR-9410396.

their popularity. On the other hand, the lack of a firm semantic basis for SA models severely inhibits their use as bases for further development, in particular, their use as major references against which the quality and applicability of implementations can be assessed. The lack of a firm semantic basis also means there is little support for rigorously reasoning about specified properties, thus limiting the use of the models in analyses of desired behaviors. These problems can be alleviated by replacing informally specified parts of SA models (e.g., data descriptions and process specifications) with formal specifications. The inclusion of formal specifications in SA models facilitates a level of rigorous analysis not attainable in the less formal expressions of SA models.

A significant amount of research on the technical aspects of integrating FSTs and ISTs has taken place. We refer to such integrated techniques as ***FISTs***. Surveys of some these research efforts can be found in [11, 15]. In this paper we illustrate, via the results of two case studies, two ways in which FSTs can complement and enhance the application of ISTs in the requirement engineering phase of development. In Section 2 we describe the major results of a case study in which a formal model was used to prototype the required behavior captured by informal, graphical models. We developed a formal operational model, in the form of an extended Petri Net, for a textbook SART (SA/Real Time) [14] model of a vending machine. Using the insights gained from building and exercising the operational model we were able to improve the SART models (i.e., make them more accurate and informative). The development of the formal model was done in a generative style, that is, a rigorously defined relationship between modeling concepts was used to guide the transformation of the SART model to the extended Petri Net model [19].

In Section 3 we describe a case study in which an integrated Fusion [5] and Z [17] FIST was used to develop a rigorous OOA model of a student advising system. The Z models we produced supplemented the OOA models, that is, both the OOA models and the Z models are needed to get a complete picture of the modeled behavior. The FST in this case was used to make more precise the concepts captured by the OOA model. Section 4 contains our conclusions.

2 The SART/Petri Net Case study

The goal of the SART/Petri Net case study was to rigorously analyze an SART model of the vending machine problem taken from the book by Hatley and Pirbhai [14]. The analysis was done by building and exercising a formal operational model of required behavior as modeled by the SART model. The operational model can be viewed as a prototype, or an executable formal model of behavior, and as such, it contains implementation details not present in the SART model. For this reason, the formal and the informal models are at different levels of abstraction. The operational model was created using an extended form of Petri Nets [19].

This case study illustrates how executable formal models can be used to enhance the application of informal requirements modeling techniques. The experience outlined in the following subsections indicate that building and exercising the prototype can yield insights that can be used to make the informal models more accurate and informative.

For this approach to be effective there must be some assurance that the formal model captures the behavior abstractly described in the informal model. We provide a basis for such assurance by imposing an operational semantics on the SART model and defining a relationship, based on the semantics, between the modeling concepts used in the SART and the Petri Net techniques. The dynamic semantics we impose on SART models is based on our experiences with the use of such models in student projects and a particular industrial application. We are not claiming that our semantices reflects the most common interpretation (if one exists!). We suspect that industrial applications of SART techniques are based on a variety of (often implicitly defined) interpretations that may suit the particular domain in which the SART techniques are applied. In this paper, our intent is to illustrate the benefits of explicitly defining a formal semantics for SARTs.

In the subsections that follow we define the relationships and show how it can be used to transform a SART model to an executable extended Petri Net model.

2.1 Modeling requirements with SART

The SART requirements modeling techniques extend the applicability of traditional SA techniques to real-time, reactive systems. An SART requirements model consists of the following major components:

- *Data flow diagrams* (DFDs) model systems in terms of data flow dependencies between processes (also called data transforms). A DFD can be viewed as a structural model of the *functional* aspects of a system.
- *Process specifications* are informal processes descriptions depicted in DFDs.
- *Control flow diagrams* (CFDs) model control dependencies between processes. CFDs depict the control signals that affect the state of a system and the behavior of DFD processes.
- *Control specifications* describe how control flows affect the state and hence the behavior of the system. In this paper, the specifications are expressed in terms of *state transition diagrams*.

The vending machine is constrained to behave as follows [14]:

1. Only nickels, dimes, and quarters are to be accepted as valid contributions to a payment. All other objects are rejected (rejected objects are called *slugs*).
2. Payment computation and product selection can only be activated after a valid coin is detected.
3. Coins must be returned and the customer notified if a selection is not available.
4. Product prices must be changeable.
5. Return the customer's payment on request if he or she decides not to make a selection.
6. A product can only be dispensed if it is available and the payment is sufficient.
7. Disable product selection after the product is dispensed and until the next validated coin is detected.

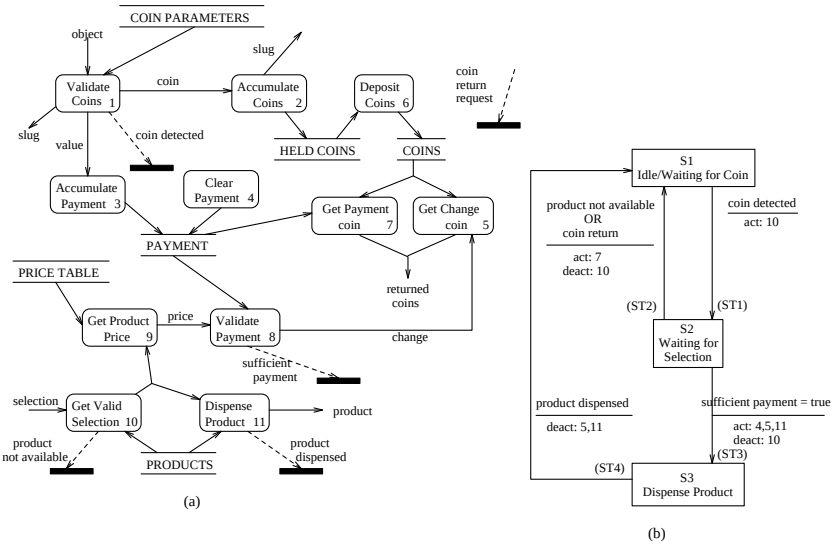

Figure 1: SART model of the vending machine problem

A combined flat DFD and CFD model of the vending machine system, derived from the hierarchical DFD and CFD models given in [14] is shown in Fig. 1(a). The round-edged boxes represent processes, the solid arrows represent data flows, the dashed arrows represent control flows, and the parallel bars represent data stores. Fig. 1(b) shows the state transition diagram describing how the control flows affect system states. In the state transition diagram, states are represented by rectangular boxes, and a state transition from

state *Si* to state *Sk* ($i \neq k$), resulting from the occurrence of an event represented by the control flow *c*, is described by a labeled, directed arrow going from *Si* to *Sk*. The label on the arrow consists of the control flow's name above a horizontal bar and a list of processes that are activated and/or deactivated as a result of the transition below the bar. For example, the generation of the *sufficient payment* signal causes the system to change state from *S*2 (*Waiting For Selection*) to *S*3 (*Dispense Product*), and also causes the activation of the processes *Dispense Change*, *Clear Payment* and *Dispense Product*, and the deactivation of the process *Get Valid Selection* (the process specifications are not given here because of page constraints).

The SART model for the vending machine is typical of the type of description produced by the SA techniques. It utilizes simple graphical constructs to build an intuitively-appealing structure, and is supported by informal descriptions of processing and data details that are not depicted in the diagrams. While the techniques may be appropriate problem structuring aids, the lack of a well-defined semantics for the models produced makes it difficult to analyze specified behavior rigorously.

Development of the formal operational model from the SART model proceeded as follow:

1. With the help of formal transformation rules, an *Extended Petri Net* (EPN) structure was obtained from the SART model.

2. Algebraic type specifications for data elements described in the SART model were defined.

3. Definition of the EPN was completed by defining behavior of processes in terms of transition firing and data transformation rules.

4. The EPN was then converted to a machine executable form called Cabernets [16] and exercised.

2.2 An extended Petri Net model

The *Extended Petri Net* (EPN) model is based on the *Modified Petri Net* model proposed by Yau and Caglayan [19]. The EPN consists of a set of *places*, a set of *data objects*, and a set of *transitions*. Transitions are connected to each other through places and data objects. The EPN incorporates data flow in regular Petri Nets; therefore, it can be used to specify both functional and control aspects of a system.

The function M takes a place and returns its marking (i.e., the number of tokens in the place). A place p is said to be enabled if $M(p) > 0$ and disabled if $M(p) = 0$. If there is a place, p, in an EPN for which $M(p) > 1$ then the EPN is said to be in an *unsafe* state; normal (or safe) behaviors must be modeled in terms of markings of 0 and 1.

Formally, an EPN transition T is represented by the 5-tuple

$$T = (I, O, D, F, G)$$

where I and O are the set of input and output places, respectively, D is the set of data objects, and F and G are the control transfer specification (CTS) and the date transfer specification (DTS) of the transition, respectively.

There are two types of data objects (i.e., elements of D): data *variables* and *stores*. An input (data) variable has its contents removed during transition firing, and an output variable has its contents replaced during transition firing. A variable whose data item has been removed is associated with the value *empty*. A store contains data that persists over the executions of transitions. Execution of transitions may involve reads and/or updates of associated stores.

F consists of an input place specification and an output place specification. An input place specification defines precisely the input place states that initiate the execution (firing) of T. An output place specification defines the change of output places after execution of T. Predicates associated with output places have the same function as those used in *Predicate/Transition Nets* [12], that is, an output place receives a token (as a result of transition firing) only when the predicate associated with the corresponding link is true. An output place associated with a link that is associated with a predicate *true* receives a token as a result of each transition firing.

Data transfer specifications (DTSs) are expressed in terms of operations defined in algebraic specifications [13] of the data types associated with the system being modeled. The specifications abstractly define data types in terms of operations that can create and manipulate instances of the types. Some types are considered

primitive, that is, their definition is provided by the modeling language. Examples of such types are the integer type (denoted by $\mathbb{Z}$), natural numbers (denoted by $\mathbb{N}$), and character.

For convenience, an EPN can be partitioned into two subnets: one for data flow (referred to as the D-EPN) and the other for control flow (referred to as the C-EPN). See Fig. 3 for an example of an EPN.

Execution of an EPN is based on the following transition firing rules:

- If the input place condition specified in a transition's control transfer input specification holds, then the transition is said to be enabled. A transition fires immediately when enabled, and the output result is observable instantaneously. On firing, tokens in the input places of the transition are removed. The effect of the transition's execution on the output places of the transition is determined by the output specification given in the transition's CTS.
- When a transition is executed the effects on the data variables and stores associated with the object are determined by the Data Transfer Specification (DTS).

2.3 Guidelines for transforming SART models to EPNs

Below we give the set of rules we defined to support the systematic transformation of SART models to EPNs (these rules are depicted in Fig. 2):

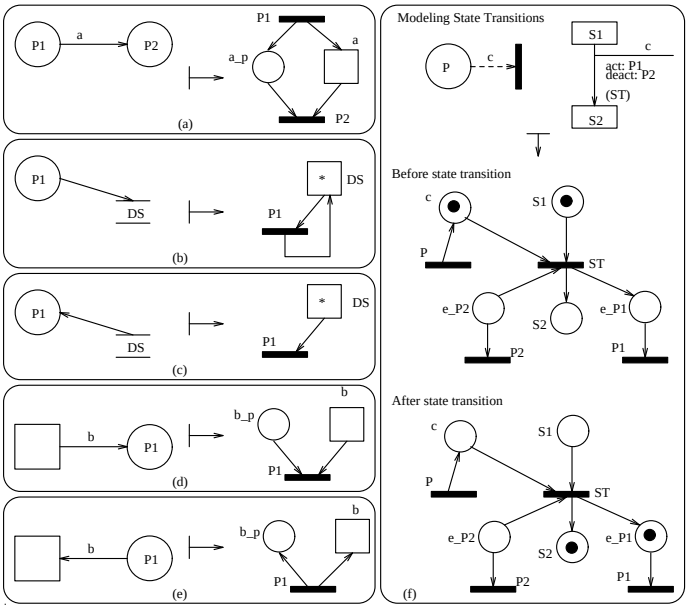

Figure 2: SART to EPN Transformations

1. A data transform is represented by a transition (called a ***process transition***), a data store by a store, and a data flow is associated with a place and a variable (the place indicates the occurrence of the data flow while the variable gives the content). A data flow a from a process $P1$ to another process $P2$ in a DFD is modeled in an EPN by connecting process transition $P1$ to process transition $P2$ via a place (used to indicate the occurrence of data on a) and a variable (holds the contents of a). This relationship is depicted in Fig. 2(a).
2. A data flow from a process $P1$ to a data store DS is represented by an EPN in which transition $P1$ is connected to a store data object as input and output (see Fig. 2(b)). The input arc connecting the store to the transition is needed to access the initial state (state before firing) of the store.

3. A data flow from a data store to a process is represented by an EPN (see Fig. 2(c)) in which the process transition is connected to an input store data object. In this case, the state of the store after execution is the same as the state before execution.

4. A data flow from an external entity to a process is modeled in an EPN as a process transition connected to an input place (used to indicate the occurrence of the flow) and an input variable (see Fig. 2(d)).

5. A data flow from a process to an external entity is modeled as a process transition connected to an output place (used to indicate the occurrence of output on the data flow) and an output variable (see Fig. 2(e)).

6. The EPN model of the effect of control flows on the behavior of a system is obtained using information contained in the CFD and the associated state transition diagram (STD). Each state transition depicted in an STD is represented by an EPN structure. For example, consider a state transition from state $S1$ to $S2$ resulting from the occurrence of a control flow, c, generated by a process P, that causes the activation of process $P1$ and the deactivation of process $P2$ (see Fig. 2(f)). The control flow c is modeled by a place connecting the process transition P (to which it is an output place) to the transition ST representing the state transition (to which it is an input place). The state $S1$ is represented as an input place to ST and the state $S2$ is represented as an output place of ST. The transitions associated with $P1$ and $P2$ are associated with places e_P1 and e_P2, respectively, that are connected to ST. A token in these places indicates that P1 and P2 are enabled. The place e_P1 is connected to ST as an output place, indicating that $P1$ is enabled by the transition, and e_P2 is an input place to ST, indicating that $P2$ is disabled by the transition. The input specification (not shown) indicates that ST will fire whenever there is token in the place c. The result of the firing depends on whether there is a token in $S1$ or not. If there is no token in $S1$ (indicating that the system is not in state $S1$) then only the token in c is consumed (if there is a token in e_P2 it is not consumed) and no output tokens are produced (in other words, signals do not wait for the system to reach a state in which it has an effect; this is an example of an interpretation that may vary across uses of SART - some situations may require signals to wait for the state to occur, in which case this particular firing of the transition will not occur). If there is a token in c and in $S1$ then the transition fires and puts a token in e_P1 and in $S2$, and, if there is a token in e_P2 before firing, it is removed.

Using transformation rules 1 to 5, a D-EPN structure is obtained. The D-EPN structure for the SART vending machine model is shown in Fig. 3. The generation of the D-EPN structure can be automated. The DTSs cannot be generated automatically from informal natural language process specifications and must be created by the modeler. The modeler can use the informal specifications as guides in their development of more formal specifications of data transfer.

The C-EPN is developed using rules 1 to 6. The initial state of the C-EPN obtained for the vending machine problem is shown in Fig. 4.

In the C-EPN, the place label 'dn' is an inhibitor place for transition 'n'. Place labels starting with 'c' in Fig. 4 represent conditions (control flows). Tokens in these places indicate that the associated conditions hold (e.g., a token in $c1$ means that a coin has been detected). Place labels starting with 's' represent states. A token in the following places indicates that the system is in the associated state:

s1: Idle state.

s2: Waiting for selection.

s3: Dispense product.

The transitions labeled 'STn' represent state changes. For example, the state change resulting from the occurrence of the event ***coin detected*** in the Idle state is represented by the state transition *ST1*.

2.4 Analyzing the EPN model

An EPN obtained from an SART model can be viewed as a formal operational interpretation of the SART model. The interpretation can be used to 'test' the model against the stated requirements by simulating

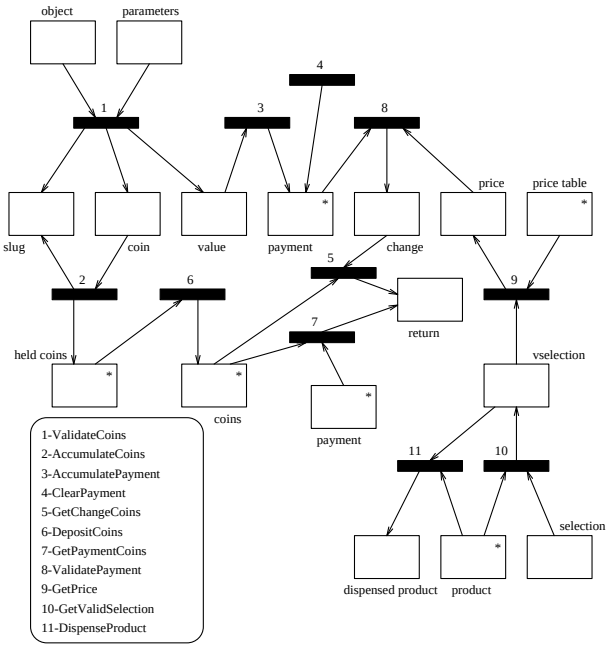

Figure 3: D-EPN model of the vending machine problem

execution of the system according to the firing rules of the EPN (as defined by the CTSs). A validator can develop a set of scenarios ('test cases'), simulate them with the EPN, and observe the results.

As pointed out in the previous subsections, a data flow is associated with both a place and a data variable. The place is used to model the occurrence of data on the flow and the variable holds the value. This requires that the following relationship between data flow variables and places be maintained in an EPN as it moves from state to state: a token in a data flow place requires that a value be present in the associated variable. One way of ensuring that this relationship is maintained is by associating the output data flow places with predicates that are true exactly when the conditions for data transfer to the corresponding variable are true. In cases where data input to a process is from an external entity then each control transfer rule that removes a token from this place must be predicated on the presence of data in the variable.

Before we can analyze the vending machine SART model, we need to define its initial EPN state. The initial EPN state is shown in Fig. 4:

- the state place, $s1$ has a token in it, indicating that the vending machine is in the Idle state; and
- the processes *Get Valid Selection, Get Change Coin, Dispense Product, Clear Payment* and *Get Payment Coin* are all disabled, that is, a token is present in the inhibitors associated with the corresponding process transitions.

In order to execute the EPN on a machine we converted it to an existing machine-executable form called Cabernet [16]. This conversion was done manually. Details of the conversion can be found in a forthcoming technical report.

During development and execution of the EPN, we uncovered a number of problems with the EPN and the SART model. Some of the problems we identified and their resolutions are discussed below:

1. On examining the C-EPN we noted that there was a transition that had an empty set of input and output places (transition 6). This raised a red flag and prompted the question 'under what conditions is this transition fired?'; a question that had to be resolved before a CTS is associated with the transition.

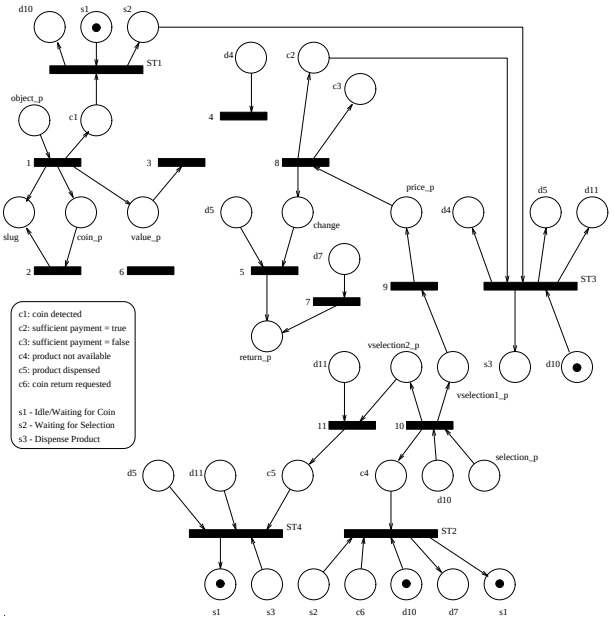

Figure 4: Initial state for vending machine

We looked at the corresponding process ***Deposit Coins*** and noted that the reason for the lack of places is a result of the lack of information related to the conditions under which the process is invoked. The only inputs and outputs of this process are data stores. How does the process know when to read the data store ***Held Coins***? In other words, when do we make the held coins part of the general pool of coins held in the vending machine? An examination of the role of ***Held Coins*** raised more questions. It was not clear why it was needed in the current problem structure; in another structure ***Get Payment Coin*** would need only to recover the coins from ***Held Coins***. Rather than change the structure radically, we chose the simpler approach; we eliminated the data store ***Held Coins*** and the process ***Deposit Coins*** and made ***Accumulate Coins*** direct its output to the data store ***Coins***.

2. In the EPN the coin value is accumulated when a coin is rejected as a slug because of a full repository of coins. We resolved this problem by having ***Validate Coins*** output the valid object (***validobj***) to ***Accumulate Coins***, which puts the physical coin in ***Coins*** and passes the value (***value***) to ***Accumulate Payment*** only after it is determined that the coin repository is not full.

3. ***Payment*** is not cleared whenever payment entered by a customer is returned. We resolved this by activating ***Clear Payment*** after payment is returned. This required the addition of a control flow from ***Get Payment Coin***, called ***payment returned***, indicating that payment was returned.

4. The initial SART model assumed that enough coins for change will always be available. This is an unreasonable assumption. The behavior we specified returns payment (without dispensing product) when there are no coins available for change.

The ability to rigorously 'test' the SART model increases confidence in the structure and properties specified. A weakness of Petri Nets is the complexity problem, that is, Petri Net models tend to become too large for analysis even for a modest-size system. To solve this problem, a Petri Net model can be organized hierarchically into several levels [18]. The lower level specification can be described by replacing each place in the specification with a more detailed Petri Net through a refinement process. The IPTES project [7] has developed a Petri Net formalization of SART that addresses this problem, as well as supports the specification of time constraints.

3 The Fusion/Z Case Study

In the second case study we used an integrated Fusion/Z FIST to develop requirements models for a student advising system. We did not attempt to formalize all the Fusion analysis models, rather, we used formal notations in place of less formal descriptions, and used formal techniques to specify and analyze the object model. The problem tackled was originally stated for a student project class, but addressed a real-world need.

In formalizing the Fusion analysis models we went through the following stages:

Stage 1: Object Modeling In this stage the focus was on modeling the static structure, that is, modeling the static relationships among objects. The object model produced in this stage was formalized using some formal translation rules that expressed relationships between Z constructs and constructs in the object model.

Stage 2: Operation Modeling In this stage the externally observable operations of the application were modeled. In the Fusion method the operations are described using natural language. In our approach we defined the effects of operations using Z. Our Z specifications can completely replace the Fusion models of operations.

Stage 3: Lifecycle Modeling In this stage the constraints on the order in which operations can be executed were defined. We used the Fusion lifecycle expressions; these representations are rigorous so we did not attempt to formalize them.

The following subsections present some of the models we produced in Stages 1 and 2.

3.1 Modeling the Advising System Problem

The following is part of the original requirements statement for the advising system (see also [10]):

> The project is concerned with providing automated support for undergraduate advising in the Department of Computer Science and Engineering. An important component of the department's undergraduate advising system is the student evaluation worksheet, which is used to keep track of courses students have taken, are currently enrolled in, and plan to take. Your specific task concerns developing a system that supports information retrieval and updating activities associated with student worksheets.
>
> The system should allow users to do the following:
>
> 1. Consistently create and destroy student worksheets.
> 2. Consistently update worksheet data. The system should be capable of informing users of updates that violate departmental rules, e.g., the system should disallow the recording of an enrollment when the student does not have the necessary pre-requisites.
> 3. Retrieve worksheets.
>
> ⋮

3.1.1 Stage 1 - Developing the Object Model

In Stage 1 we used an ***explore, formalize, refine*** development style (similar to the 'explore, elaborate, validate' process model described in [9, 10]). In the exploratory phase we modeled the structure using Fusion object modeling concepts; no attempt was made to formalize the modeled concepts during this time. In Fig. 5 we show part of the result of the exploratory task. In Fusion, the label + on an association indicates one or more and the symbol $*$ indicates zero or more. A black box on a relationship indicates that every instance of the adjacent class must participate in the relationship. In the model a student must be assigned an advisor, and must have a worksheet. A worksheet must be associated with one student. An advisor can be assigned to zero or more students (the fact that an advisor is a faculty member is abstracted out of the model). To support the formalization of the object model we defined relationships between Fusion associations and Z concepts. The relationships became part of the Z mathematical toolkit we used to develop a more formal expression of the object model. For more details about the toolkit see [2].

In formalizing the object model we took the following steps:

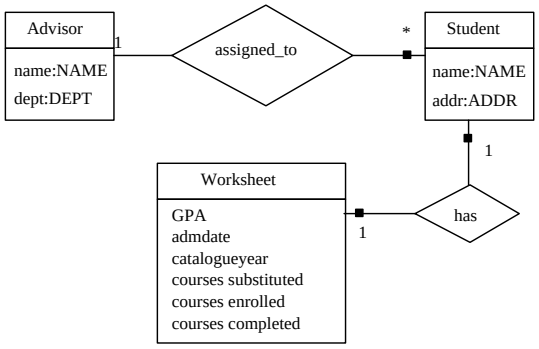

Figure 5: Fusion model of advising system

1. Determine the given sets, constants, and global variables: In this step we considered the types declared in the object model as candidates for given sets, constants, and global variables. We examined each type in the object diagram to determine how they could be modeled. Some attributes did not have types associated with them in which case we had to define suitable types. For some of the other cases we redefined the types in terms of more basic types.

   The result of this set is a Z declaration of given sets, constants and global variables. Some of the specifications that resulted from this step are given below:

$$[NAME, AdvID, StID, ADDR, DEPT]$$
$$[COURSE, REQ, DATE, YEAR, BOOLEAN, TERM]$$

$$GRADE : \mathbb{PN}$$
$$\forall g : GRADE \bullet 0 \leq g \leq 4$$

2. Associate a state schema with each object: The state schema for an object is obtained by treating the attributes of the objects as schema declarations. Each schema also declares a variable, id, that represents the identifier of an instance. In cases where no types are associated with attributes or where the type structure is unclear appropriate types must be defined. The specifications obtained in this step are given below:

Advisor
$name : NAME$
$id : AdvID$
$dept : DEPT$

Student
$name : NAME$
$id : StID$
$addr : DEPT$

Worksheet
$id : WksheetID$
$GPA : GRADE$
$adm\_date : DATE$
$cat\_year : YEAR$
$course\_sub : COURSE \leftrightarrow COURSE$
$course\_enr : COURSE \nrightarrow TERM$
$course\_comp : COURSE \nrightarrow \mathbb{P}(TERM \times GRADE)$

3. Using the mathematical toolkit, create a schema that describes the relationships among objects: In this schema, the schemas defined for objects are used to instantiate the generic definitions of relations in the mathematical toolkit. The specification obtained is given below:

$$
\begin{array}{|l}
\textit{Diagram} \\
\hline
\textit{assigned\_to} : 1 \leftrightarrow * \; [\textit{Advisor}, \textit{Student}] \\
\textit{has} : 1 \leftrightarrow 1 \; [\textit{Student}, \textit{Worksheet}] \\
\hline
\end{array}
$$

4. Create the state schema that ties all the above schemas together and specify any constraints not covered in the above steps: Constraints that cannot be expressed in terms of Fusion graphical notation are usually stated in natural language in the diagrams. In such cases the constraints should be re-expressed in Z. In the state schema produced at this stage each object is modeled as a set of instances in Z. The specification we obtained at the end of this activity is given below:

$$
\begin{array}{|l}
\textit{AdvState} \\
\hline
\textit{Diagram} \\
\textit{advisors} : \mathbb{P}\, \textit{Advisor} \\
\textit{students} : \mathbb{P}\, \textit{Student} \\
\textit{worksheets} : \mathbb{P}\, \textit{Worksheet} \\
\hline
\mathrm{dom}\, \textit{assigned\_to} \subseteq \textit{advisors} \\
\mathrm{ran}\, \textit{assigned\_to} = \textit{students} \\
\mathrm{dom}\, \textit{has} = \mathrm{ran}\, \textit{assigned\_to} \\
\mathrm{ran}\, \textit{has} = \textit{worksheets} \\
\hline
\end{array}
$$

5. Evaluate model: The formal and informal models are examined closely to determine whether they could be improved and/or simplified.

In the latter activity we reexamined our graphical model and decided that a more informative model could be obtained by treating *COURSE* and *TERM* as objects. The relationships among these objects were obtained from the Z relations involving these objects. The result of the restructuring is shown in Fig. 6.

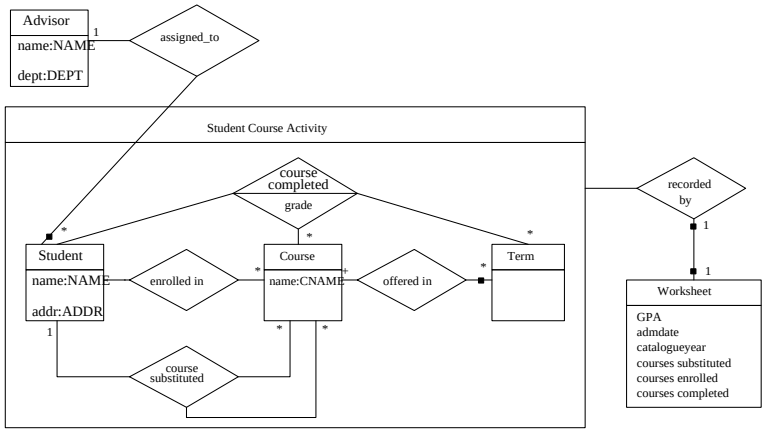

Figure 6: Modified Fusion model of advising system

3.2 Stage 2 - Developing the Operation Model: An example

In this stage, a description of each operation is created and formalized. The descriptions are first given in the Fusion format. Below is a Fusion description of an operation of the advising system, together with its formalization:

Post grade to worksheet operation:

Operation: *update_gr*

Description: Post grade for one class completed.

Reads: supplied $student, course$

Changes: $evaluation\_ws : course\_comp$

Sends: none

Assumes: Student has completed the course.

Result: The student's $evaluation\_ws : course\_comp$ is updated.

update_gr

$$
\begin{array}{l}
id? : ID \\
course? : COURSE \\
grade? : GRADE \\
\Delta AdvState \\
term : TERM \\
\hline
\exists s : Student;\ w, w' : Worksheet \mid has(s) = w \bullet \\
\quad s.id = id? \land \\
\quad course? \in \mathrm{dom}\, w.course\_enr \land \\
\quad term = w.course\_enr(course?) \land \\
\quad (course? \in \mathrm{dom}\, w.course\_comp \land \\
\qquad w'.course\_comp(course?) = w.course\_comp(course?) \\
\qquad \cup \{(term, grade?)\}) \lor \\
\quad (course? \notin \mathrm{dom}\, w.course\_comp \land \\
\qquad w'.course\_comp = w.course\_comp \\
\qquad \cup \{course? \mapsto \{(term, grade?)\}\}) \land \\
\quad w'.adm\_date = w.adm\_date \land \\
\quad w'.cat\_year = w.cat\_year \land \\
\quad w'.course\_enr = w.course\_enr \land \\
\quad w'.course\_sub = w.course\_sub \land \\
\quad w'.GPA = w.GPA \land \\
\quad has' = (has \setminus \{s \mapsto w\}) \cup \{s \mapsto w'\}
\end{array}
$$

4 Conclusion

The results of our case studies are a good indication that the application of FISTs to requirements specification and analysis can bring about the following benefits:

- Support for the analysis and validation of requirements modeling through the use of executable formal models.
- Support for early problem analysis in the form of flexible tools that permit the building of structures that appeal more to intuition than formality, as well as support for more rigorous problem analysis through the formal reexpression of structures and content.
- Use could lead to an understanding of required properties that may not be possible with sole use of formal or informal specification techniques.

- Benefits associated with the use of informal models (for example, ease of use, readability) and benefits associated with formal models (for example, less chance of misinterpretation, support for rigorous analysis and verification activities) are not necessarily impaired by their integration. In fact, formal and informal models can complement each other.

The definedness of the relationship between the formal and informal concepts and notations determines whether one can partially automate the transformation of informal to formal models. If the relationship is well-defined, and can be codified to some extent, then the set of rules can become the basis for a tool that generates portions of the formal model from information available in the informal models. Lack of well-defined rules, though, does not preclude the integration of techniques, as was shown in the Fusion/Z case study. In such cases, the transformation is more dependent on human skill.

There is some work on developing tools to support FISTs (e.g., see [1, 4, 7, 8]). We are currently working on a tool called FuZE that supports the use of a Fusion/Z FIST for requirements modeling and analysis [3, 4]. As the field develops we expect to see more tools appear. In particular, extensions to CASE tools for traditional techniques that allow for the creation of formal models, or that provide formal interpretations of symbols are envisaged.

References

[1] P. Allen and L. Semmens. Integration of structured methods and formal notations. In *ISD'94*, 1994.

[2] B. W. Bates, J.-M. Bruel, R. B. France, and M. M. Larrondo-Petrie. Experiences with Formalizing Fusion Object-Oriented Analysis Models. FAU Technical Report TR-CSE-95-44, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL–33431, USA, Nov. 1995.

[3] J.-M. Bruel and R. B. France. A Formal Object-Oriented CASE Tool for the Development of Complex Systems. In A. H. Seltveit and B. A. Farshchian, editors, *Proceedings of the 7th European Workshop on the Next Generation of CASE Tools (NGCT'96), Crete, Greece*, 20–21 May 1996.

[4] J.-M. Bruel, R. B. France, B. Chintapally, and G. K. Raghavan. A Tool for Rigorous Analysis of Object Models. In *Proceedings of the 20th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS'96), Santa Barbara, California*, July 29–August 2 1996.

[5] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[6] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1978.

[7] R. Elmstrom, R. Lintulampi, and M. Pezze. Giving semantics to SA/RT by means of high-level timed Petri nets. *Real-Time Systems*, 5, 1993.

[8] R. France, T. Horton, M. Larrondo-Petrie, and S. Reeves. Process support for rigorous structured analysis. In *Software Engineering Research Forum '93.* SERF'93, 1993.

[9] R. France and M. M. Larrondo-Petrie. A two-dimensional view of integrated formal and informal specification techniques. In *ZUM'95, Lecture Notes in Computer Science 967.* Springer-Verlag, 1995.

[10] R. France and M. M. Larrondo-Petrie. Understanding the role of formal specification techniques in requirements engineering. In *in Proceedings of The 8th SEI Conference on Software Engineering Education, Lecture Notes in Computer Science 895.* Springer-Verlag, 1995, pages 207-222.

[11] R. B. France and M. M. Larrondo-Petrie. From structured analysis to formal specifications: State of the theory. In *Proceedings of the ACM 1994 Computer Science Conference.* ACM Press, April 1994.

[12] H. J. Genrich. Predicate/transition nets. *Advances in Petri Nets, 1986.* W. Brauer, W. Reisig, and G. Rozenberg, Eds, Ner York: Springer-Verlag, 1987.

[13] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. An initial algebraic approach to specification correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol 4: Data Structuring*. Prentice-Hall, 1978.

[14] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dover Press, 1987.

[15] L. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

[16] S. Silva. *Cabernet user manual*. CEFRIEL and Politecnico di Milano, Italy, 1 edition, May 1994.

[17] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ, 2nd edition, 1992.

[18] J. Wu and E. B. Fernandez. A simplification of conversation design scheme using petri nets. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[19] S. S. Yau and M. U. Caglayan. Distributed software system design representation using modified petri nets. *IEEE Trans. on Software Engineering*. 9, (6), Nov. 1983, 733-744.