An Approach to Specifying and Verifying Safety-Critical Systems with Practical Formal Method SOFL*

Shaoying Liu*, Masashi Asuka†, Kiyotoshi Komaya†, Yasuaki Nakamura*
*Hiroshima City University, Japan
†Mitsubishi Electric Corporation, Japan

Abstract

One of the primary concerns in developing computer embedded safety-critical systems is how to develop quality software. Software must fulfill its functional requirements and must not contribute to the violation of safety properties of the entire system. To this end, capturing error free and satisfactory functional requirements is crucial before proceeding to the subsequent development phases. We describe an approach to specifying and verifying software for safety-critical systems with the practical formal method SOFL (Structured-Object-based-Formal Language). Requirements specification focuses on the functionality of the software, but with the consideration of safety constraints and its interaction with the surrounding operational environment. The verification of specifications can be carried out using three techniques: *data flow reachability checking*, *specification testing*, and *rigorous proofs*, respectively. We apply this approach to a realistic railway crossing controller for a case study and analyzes its result.

Keywords: Safety-critical systems, formal methods, functional requirements, safety requirements, formal specification, verification, railway crossing controller.

1 Introduction

The quality of safety-critical systems is extremely important as their failure may cause catastrophic lose of life and/or properties. By quality we mean both functionality and safety. Ideal safety-critical systems must exactly and accurately function as required and guarantee the safety of the system during its operation. If computers are used to control part or the entire system, software must fulfill its functional requirements and must not contribute to the violation of safety properties of the entire system. To this end, capturing error free and satisfactory functional requirements is crucial before proceeding to subsequent development phases.

It is worth noting that erroneous software may cause safety problem only when it interacts with its surrounding hardware components. For this reason, interactions between software and its surrounding hardware components must be taken into account when requirements for software is specified. That is, interactive activities should be modeled as part of the requirements specification for software. To ensure that software does not contribute to the violation of safety properties of the entire system, it is desirable to obtain error free and precise functional requirements. To verify this property, functional requirements must be well documented and safety requirements must be properly expressed.

Formal methods offer an approach to the construction and verification of precise, consistent and correct specifications using mathematical notation and well defined calculus. Evidences have suggested that formal methods can be effective in capturing requirements, identifying errors, transforming specifications to programs, and reducing cost of overall development [1]. However, there also exist barriers in applying formal methods such as Z [2], VDM [3], and RAISE [4] in practice because they require high skills for abstraction and their notations have a poor readability for most software engineers in industry. In particular, their formal refinement and formal proof techniques are extremely difficult to apply in practice [5].

We describe a practical approach to specifying and verifying software for safety-critical systems using the practical formal method SOFL (Structured-Object-based-Formal Language). We take the view that all the expected functional behaviors related to the safety of the system must be captured correctly by the requirements

*Work is supported in part by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440) and by Mitsubishi Electric Corporation under a joint research grant

specification because a desirable safety-critical system is expected both to behavior satisfactorily and to maintain the safety of the entire system. To achieve this goal, we believe that two activities are necessary. One is to build a formal but comprehensible functional requirements specification, and the other is to ensure the consistency and correctness of the specification through verification.

We employ SOFL for specification and verification. Since SOFL integrates data flow diagrams, Petri nets and VDM-SL to provide a graphical and textural formal notation for specifications and supports both structured and object-oriented methodologies, it is user-friendly and practical [6, 7]. In this paper we take a realistic railway crossing controller as an example to describe how to specify requirements specification for safety-critical systems and how to conduct the verification of the specification for the assurance of safety properties. We suggest three techniques for verification, each checking the specification at one level. The first is to check automatically the *data flow reachability* to ensure that a safety related functional behavior is really specified satisfactorily at the syntactical level. The second technique is to assure the *correctness* of the data flow reachability via *specification testing*, where by correctness we mean that the data flow reachability is both syntactically correct and semantically sound, see section 4.2 for details. However, due to the intrinsic limitation of testing technique in general, such a checking cannot guarantee the correctness of the data flow reachability. To improve the quality of the verification, we suggest another technique called *rigorous proofs* which serve the same purpose as specification testing, but take more rigorous approach.

The major contributions of this paper include:

- Extending SOFL to specify timing requirements.
- Applying the extended SOFL to specify a realistic *railway crossing controller*.
- Providing the following three techniques for checking data flow reachability and its correctness to assure safety properties: *data flow reachability checking*, *specification testing*, and *rigorous proofs*.
- Evaluating the effectiveness of SOFL and the proposed verification techniques by giving our experience and comparing with the existing formal methods applications.

The remainder of this paper is organized as follows. Section 2 briefly describes SOFL and its extension to allow the modeling of timing behaviors. Section 3 takes a railway crossing controller as an example to discuss how to derive a formal specification from informal requirements for systems. Section 4 describes techniques for checking data flow reachability and its correctness. Section 5 evaluates the SOFL approach by giving our experience and comparisons with related works. Finally, section 6 gives conclusions and outlines future research.

2 SOFL and Its Extension

We first describe briefly the original SOFL and then discuss its extension to allow the modeling of timing behaviors.

2.1 Brief Introduction to SOFL

A requirements specification or design for a software using SOFL is a hierarchical condition data flow diagrams (CDFD), each level CDFD is associated with a *specification module*. The diagram remains at a high level of abstraction, describing interactions between condition processes in terms of data flows and data stores, while the specification module defines pre and postconditions for each condition process, and types for each data flow or data store, as illustrated by Figure 1.

A condition process models a transformation from its inputs to outputs; a data flow indicates data in motion between condition processes; and a data store represents data at rest (like a file or database). The specification module functions like process specification and data dictionary for traditional data flow diagrams, but we use formal notation (e.g. abstract data types, set theory, first order logic) for the definition. For example, we build a specification module, indicated by the keyword **s-module**, for the top level CDFD in Figure 1(a) in which all the condition processes, data flows, and data stores are defined precisely. For some reason, condition process 1 and 3 are decomposed into the lower level CDFDs, respectively.

According to SOFL semantics [8], when inputs of a condition process are available, it will enable the condition process to fire. As a result of the firing, the inputs of the condition process are consumed and its outputs are made available. For example, Figure 2 shows a CDFD that describes one condition process to produce y from x. It means that when data x is available, it will enable the condition process A to fire. After A is fired, data bound to x is consumed and data bound to y will be produced as a result.

A good feature of SOFL is that it provides a com-

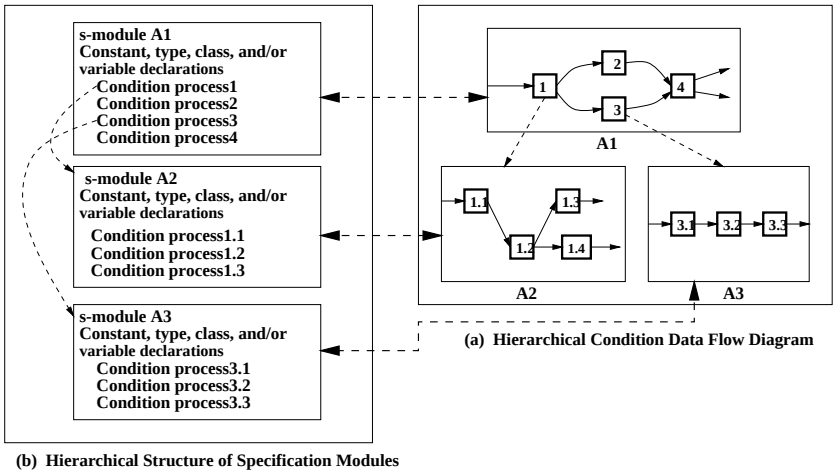

Figure 1: The outline of a SOFL specification

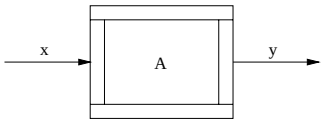

Figure 2: A simple CDFD

plementary graphical and textual formal notation for constructing systems hierarchically and comprehensibly. Due to condition data flow diagrams and their close link with corresponding specification modules, SOFL specifications possess good traceability for specification modifications and transformation. The first author's experience in applying SOFL to other software systems [6, 9] suggests that these advantages are much more effective for enhancing software quality and productivity than other commonly used formal methods such as Z and VDM. A detailed description of SOFL is given in [6].

2.2 Extension of SOFL

SOFL semantics has no notion of time. Firing a CDFD is like the execution of a C program. In a CDFD firing condition processes is a non-determinism. At any given time there may be several condition processes (like transitions in Petri Nets), which are enabled to fire. One of these is chosen non-deterministically. However, when we apply SOFL to the railway crossing controller, a real-time system, we do need to specify the precise time before a firing of the condition process occurs and the duration the firing lasts. Therefore, we take the similar approach for timing Petri Nets [10] to extend SOFL to deal with timing requirements as follows.

Two time intervals are added to the specification of a condition process with different meanings. The first time interval, say [a, b], means that when the condition process is enabled, it must be fired between time a and b (inclusive), where time unit can be *second, minute, hour* or an abstract unit depending on applications. The second time interval, say [c, d], means that the firing must take time between time c and d (inclusive). In fact, [a, b] is part of the precondition to specify when the condition process can be fired, while [c, d] is part of the postcondition to specify how long the firing takes. a and c must be greater or equal to zero while b and d must be greater than or equal to a and c respectively and can be # representing unlimited time.

The two time intervals are given sequentially, separated by a comma, following the keyword **inter** in the specification of the condition process, for example, **inter** [a, b], [c, d]. It is optional, but useful to write them in the precondition and postcondition boxies respectively in the graphical notation of the corresponding condition process. Consider condition process A in Figure 2 as an example, let its specification be:

c-process A(x: **int**) y: **int**
  inter [5, 10], [20, 25]
  pre x > 0
  post y = x + 1
end-process.

The first time interval [5, 10] specifies that when the in-

put x is available, condition process A must be fired at any time point between 5 and 10 units (inclusive). The time interval [20, 25] indicates that the firing of the condition process must take at least 20 time units and at most 25 units. In addition, this condition process requires x > 0 in its precondition, specifying that if the input x of the condition process satisfies x > 0, then after the firing of this condition process, the output y must satisfies its postcondition y = x + 1. Note that both the first time interval (for precondition) and the second interval (for postcondition) are optional for the condition process. If they are not given, it means that a firing of the condition process occurs as soon as possible after its inputs become available and takes as little time as possible, and the meaning of the condition process remains the same as in the original SOFL. From now on in this paper we always refer to the extended SOFL whenever we mention SOFL.

3 Formal Specification of a Railway Crossing Controller

We first describe an automatic railway crossing system informally and then formalize the relevant part of the system using SOFL, which indicates the requirements for the software serving as the controller for this system.

3.1 System Description

An automatic railway crossing is a system which has two primary functions. One is to warn automobiles and human beings of coming trains beforehand by ringing bells and flashing signals, the other is to close a pair of crossing gates while trains are passing through the crossing. It is a safety-critical system because if it did not work as expected, a catastrophic accident like a collision between a train and automobiles would occur. Therefore, it must be designed carefully to meet the required regulations and specifications for safety before deployed in practice.

Obviously, the safety requirement is that gates must be closed whenever a train is passing through the crossing. As for the regulations, the Ministry of Transport of Japan defines the primary behavior as follows: there should be around fifteen seconds between the start of activating signals and the finish of closing gates, which gives enough time to clear up the crossing; and there should be another twenty seconds before a train reaches the crossing.

In addition, there are other requirements. For example, the signals should be stopped and the gates should be opened for normal operation as soon as a train has passed the crossing; time for lifting up gates should be taken into account, which usually takes around five seconds. There may also be other functional requirements railway companies require.

To meet these requirements, various kinds of crossing controllers have been designed and deployed. Most of them are based on conventional wired logic by relays, but there is a trend recently to deploy computerized controllers in order to enhance serviceability and to reduce maintenance costs. In this paper, we focus on the computerized controllers, which adopt the algorithm called "sequence control method" given in [11]. It is briefly described as follows.

In general, a crossing controller uses detectors (electrical devices) at certain points on a track to detect trains running over and at least two detectors are used before and after the railway crossing respectively. The detector, called ADC, before the crossing detects trains coming to the crossing for activating signals and closing gates. Another detector, called BDC, after the crossing detects trains passing by for stopping signals and opening gates. The section between the two detectors is called a *warning section*.

Based on results of the detection, the controller handles two kinds of files, a SeQuence (SQ) file and a Train Counter (TC) file, to control the equipment appropriately. The SQ file records trains' status. Each train in the warning section has three status. The first status is stored when a train is detected by ADC. The status is shifted to the second when the train has passed this detector (during this status, the train cannot be detected, but its existence can be known by the sequence) and then shifted to the last status when the train is detected by BDC. Finally, when the train has passed this detector, the status is cleared up. In other words, the number of recorded status in SQ is equal to the number of trains running in the warning section.

The TC file records the number of trains existing in the warning section which is derived by counting the status in the SQ file. When the number is changed from zero to one, the controller starts activating signals, and after ten seconds, it starts closing gates. As long as the number is not zero, signals continues warning and gates cannot be opened. Three seconds after the number is changed to zero, the controller stops signals and starts opening gates.

As described above, the procedure is relatively simple, but even this, an unexpected situation against the regulations has been found by Sasaki and Yoneda [11] when

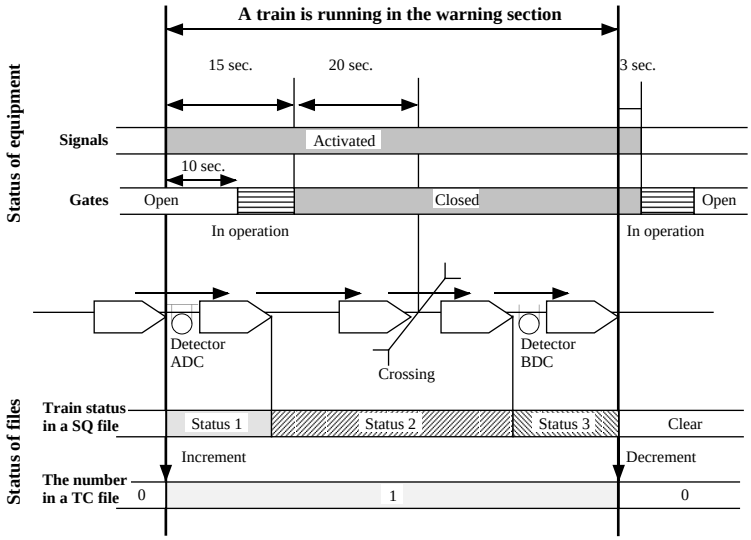

Figure 3: Railway crossing system

multiple trains and control timings are concerned. Controllers deployed in practice for a multiple track line are required to operate gates which have two barriers, one of which should be operated some seconds behind the other, and to have other basic service functions. Such a system becomes more complex than one track system. Therefore, an effective method is needed to enable us to develop software meeting safety requirements. It is important that such a method provides an effective approach to producing the final code from requirements so that human errors can be significantly prevented or reduced.

3.2 Formal Specification

Our primary concern is to formalize the user requirements for the crossing controller. Since the controller interacts with the surrounding operational environment (e.g. taking the value of ADC and producing signals), we need to model the connections of the controller and the related hardware devices in order to obtain an accurate specification. As we mentioned before, SOFL supports the use of structured methods for specifications. This process is carried out as follows. First, a top level CDFD is built to model the most abstract behaviors of the system which are primary requirements of interest to the user. Then some condition processes in this CDFD may be decomposed into lower level CDFDs, describing in details how their inputs are transformed to outputs. There are three reasons for decomposing a condition process.

- The functionality is too difficult to be described in terms of pre and postconditions.
- The relation between the inputs and outputs cannot be established in the postcondition.
- How the condition process transforms its inputs to outputs needs to be described.

In the formal specification information from hardware devices (e.g. detectors, signals) must be represented as logical variables which are either inputs or outputs for some condition processes. Thus, requirements for software can be accurately and precisely specified to meet the functional and safety requirements.

By following this approach, we formalize the railway crossing control system as six CDFDs on three levels. For reasons of space, we only give the first and second CDFDs in Figure 5 and 6 below, while the rest figures are omitted for brevity. The outline of this CDFD hierarchy is illustrated by Figure 4.

This CDFD describes the railway crossing control process at the top level. When train comes over the electrical detector ADC, modeled as a data store, ADC is immediately set to OCCUPY. After the entire train passes the ADC, the ADC is set to CLEAR and the train proceeds to the crossing area until reaching the detector BDC which is modeled as another data store. BDC is set to OCCUPY once the train comes over it. After the entire train passes the BDC, the BDC is set to CLEAR.

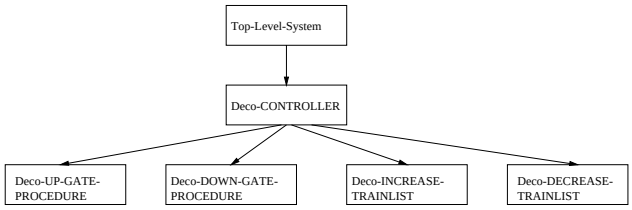

Figure 4: The outline of CDFD hierarchy for the railway crossing control system

This process interacts with the software CONTROLLER via the two detectors ADC and BDC. Based on the values of these two detectors and the current gate status, modeled by the store GATE-STATUS, the CONTROLLER activates signals for ringing the bell, setting the traffic light, and operating crossing gates. These signals are modeled by the data stores BELL-OUTPUT and GATE-OUTPUT. The real actions to ring the bell, set the traffic light, and operate the gates based on these signals are modeled by the condition processes UP-LOCKED, DOWN-OPERATION, DOWN-LOCKED, and UP-OPERATION, respectively. The marked data flows with a black dot at their start points (e.g. WHILE-NOT-DOWN, WHILE-NOT-UP) indicate that they are available initially.

In this CDFD, we also specify the timing requirements for condition processes as necessary. For instance, when train is available, ON-ADC is fired immediately (because of timing requirements given by the first time interval [0, 0]) and any firing of this condition process takes no time, indicated by the second time interval [0, 0]. Condition processes without explicit time intervals are not constrained by time (but expect to be fired as soon as possible once they are enabled).

To define precisely the components of this top level CDFD (e.g. data flows, data stores, condition processes), we build a specification module Top-Level-System as follows.

s-module Top-Level-System : Figure 5;

type

Detector = {CLEAR, OCCUPY};

GateStatus = {UP, DOWN, IN-OPERATION};

BellOutput = {ON, OFF};

GateOutput = {UP, DOWN, LOCK};

var

train: **void**; /* train is only used to indicate the availablity of trains in the real world*/

next-status: **void**;

ADC, BDC: Detector = CLEAR; /* Both ADC and BDC variables of type Detector which are initialized with the value CLEAR. */

GATE-STATES: GateStatus;

BELL-OUTPUT: BellOutput;

GATE-OUTPUT: GateOutput;

WHILE-NOT-DOWN: **void**;

inv

BDC = OCCUPY **implies** GATE-STATUS = DOWN **and** BELL-OUTPUT = ON; /* This invariant describes a safety-critical property which must be held throughout the entire system. */

c-process CONTROLLER()

ext	**rd** ADC, BDC: Detector
	rd GATE-STATUS: GateStatus
	wr BELL-OUTPUT: BellOutput
	wr GATE-OUTPUT: GateOutput
pre	**true**
post	**true**
decomposition	Deco-CONTROLLER

comment
This condition process uses and/or changes the data stores (external variables) ADC, BDC, GATE-STATUS, BELL-OUTPUT, GATE-OUTPUT. As the description of its pre and postconditions needs some lower level data stores, the detailed specification of this condition process is not given at this level. To express such an abstraction, the pre and postcondition are specified as **true** and **true**, respectively. The detailed description of the functionality of this condition process is given through its decomposition given in Figure 6 and the specification module Deco-CONTROLLER.
end-process;

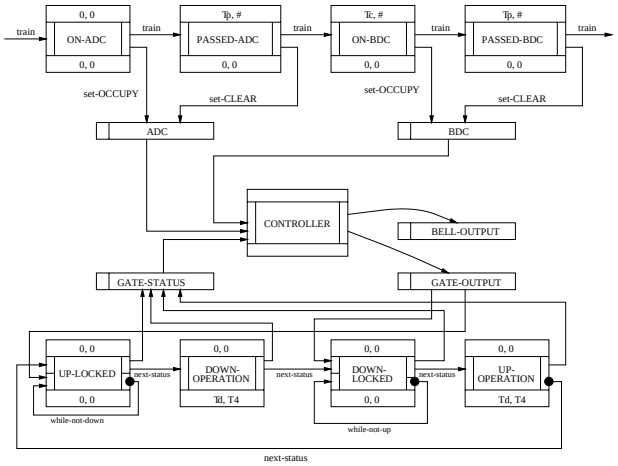

Figure 5: The top level CDFD of the railway crossing control system

c-process ON-ADC(train: **void**) train1: **void**
inter [0, 0], [0, 0]
ext **wr** ADC : Detector
pre **true**
post ADC = OCCUPY **and** **bound**(train1)

comment
This specification requires that once the input train is available, a firing of this condition process must immediately occurs and the firing takes no time at all.
The precondition imposes no specific constraint on input train.
The postcondition states that ADC is set to OCCUPY and train1 is made available (denoted by **bound**(train1)).
This condition process is a lowest level condition process (indicated by the lack of **decomposition** part).
end-process;

/* In SOFL specifications the predicate **bound**(x) is **true** when x is bound to any value in its type and **false** otherwise. The only thing that can be said about a variable of **void** type is whether or not it is bound. */

c-process PASSED-ADC(train: **void**) train1: **void**
inter [Tp, #], [0, 0]
ext **wr** ADC : Detector
pre **true**
post ADC = CLEAR and **bound**(trian1)

comment
The time interval [Tp, #] indicates that once the input train is available, it will takes at least Tp time before firing this condition process, where Tp is computed based on the length and speed of the train and the distance between the two detectors ADC and BDC.
This condition process sets ADC to CLEAR and transfers the train to next condition process.
end-process;

/* For the sake of space, the specifications of the rest condition processes in this module are omitted. */

...

end-module;

Since the condition process CONTROLLER is a complicated condition process and difficult to be specified at one level, we decompose it to the lower level CDFD in Figure 6.

To keep the context of this paper concise, the associated specification module for this CDFD is omitted.

Specification modules and their corresponding CDFDs are referred each other through their figure numbers, for example, Top-Level-System : Figure 5. In each specification module all of the data flows and data stores occurring in the corresponding CDFD are declared as state variables. The parameters of each condition process in its specification can be freely chosen as long as they have consistent positions and types with the corresponding state variables used as data flows in the CDFD for activating this condition process. Modules are referred each other by using *linking* information. If a condition process

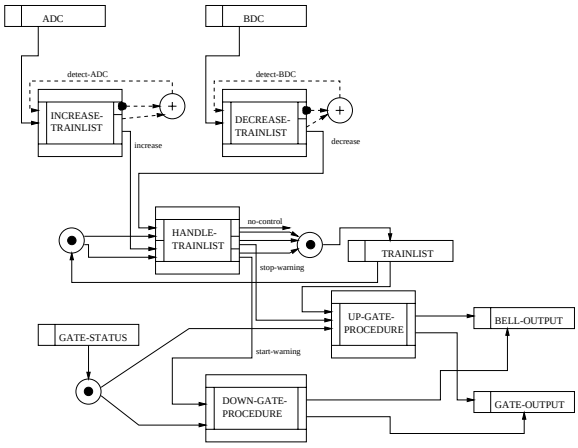

Figure 6: The decomposition of CONTROLLER

is decomposed into a lower level CDFD, such as CONTROLLER, the name of the corresponding specification module for this CDFD is given following the keyword **decomposition**, like Deco-CONTROLLER. If a condition process has no decomposition, the **decomposition** part is omitted.

4 Verification of Safety Properties

We need to make sure that the above formal requirements specification does satisfy the required safety properties. For this purpose, two respects need to be verified. One is that the required safety-related functional requirements are correctly specified. That is, the functional behaviors whose failure may cause safety problem to the system must be specified correctly with respect to the user requirements. Another is that the exceptional behaviors, which deal with hazardous situations, must also specified properly in the specification. As the verification of each of these two aspects is a big issue to address, in this paper we focus on the former and will report our work on the latter elsewhere.

As mentioned in section 3.1, the safety property concerned with functional behaviors for the railway crossing system includes the following:

- Gates must be closed whenever a train is passing through the crossing.

Such a safety property is usually required by safety engineers through safety analysis of the entire system. To check whether this property is met by the given specification, we first need to decide what it means in the context of the specification. It means the following two properties:

- There exists a correct path from the data flow train going into the condition process ON-ADC to the stores BELL-OUTPUT and GATE-OUTPUT. By path we mean a sequence of data flows and/or stores, and a correct path means that the path is semantically sound.
- When TRAINLIST is not zero, indicating the existence of trains within the crossing area, the data store GATE-OUTPUT must be kept to DOWN.

Three techniques can be used to verify these properties with different trustabilities. They are *data flow reachability checking*, *specification testing*, and *rigorous proofs*, respectively.

4.1 Data Flow Reachability Checking

The essential idea of data flow reachability checking is to check whether the specification provides functions to possiblely realize transitions from one data flow to another through condition processes. Of course, even if a path exists between two data flows, there is still no guarantee that this path is really realized because there might be bugs within some condition processes associated with this path. However, this technique provides two benefits. One is that it can automatically verify if a safety property is taken into account in the functional

specification by finding a path. Another benefit is that such a path can serve as a trace to guide the further verification in depth using either specification testing or rigorous proofs to be discussed latter.

To find a path, we first build a data flow *adjacent matrix* M_d and then computes its *transitive closure* M_d^+. If M_d^+[a, b] = 1, then it confirms that there is a path (at least one) from the data flow a to b. Obviously, the important issue here is how to build the data flow adjacent matrix for a given CDFD. We give an algorithm as follows:

Algorithm 4.1

1. Let each data flow or store corresponds to a row and a column in the matrix respectively. Initialize every entry in the matrix M_d with zero (0).
2. If data flows or stores a and b are an input and an output data flows or stores of a condition process respectively, then let M_d[a, b] = 1.
3. If data flows or stores a and b are an input and an output data flows or stores of a node (including condition-node, exclusive-or-node, and and-node [6]) respectively, then let M_d[a, b] = 1.

Consider the top level CDFD in Figure 5 for the railway crossing control system as an example. We build the data flow adjacent matrix as shown in Figure 7.

where

x_1 = train going into ON-ADC
x_2 = train going into PASSED-ADC
x_3 = train going into ON-BDC
x_4 = train going into PASSED-BDC
x_5 = train coming out of PASSED-BDC
x_6 = ADC
x_7 = BDC
x_8 = GATE-STATUS

x_9 = BELL-OUTPUT
x_{10} = GATE-OUTPUT
x_{11} = next-status going into DOWN-OPERATION
x_{12} = next-status going into DOWN-LOCKED
x_{13} = next-status going into UP-OPERATION
x_{14} = next-status coming out of UP-OPERATION
x_{15} = WHILE-NOT-DOWN
x_{16} = WHILE-NOT-UP

The transitive closure of this matrix is given in Figure 8.

Since both M_d^+[x_1, BELL-OUTPUT] = 1 and M_d^+[x_1, GATE-OUTPUT] = 1, it proofs that there exists a path between the train going into ON-ADC and both BELL-OUTPUT and GATE-OUTPUT. These two pathes are expressed as: [x_1, ADC, BELL-OUTPUT] and [x_1, ADC, GATE-OUTPUT], respectively.

We can also use this method to check other functional requirements. For example, one functional requirement is when a train passes the crossing area and no other trains exist in the crossing area, the GATE-OUTPUT must be changed to UP. If we can find a path from the data flow train going into condition process PASSED-BDC to data flow GATE-OUTPUT, it proofs that this functional requirement is specified in terms of data flow reachability. In fact, this is true because M_d^+[x_4, GATE-OUTPUT] = 1.

The above verification only shows that the data flow train going into ON-ADC can reaches both BELL-OUTPUT and GATE-OUTPUT within this top level CDFD. If some condition process like CONTROLLER has its decomposition, then this reachability will depend on whether the inputs of CONTROLLER can reach its outputs BELL-OUTPUT and GATE-OUTPUT within its decomposition (i.e. CDFD in Figure 6). In order to make sure this reachability is true in the context of the CDFD hierarchy, we need to use the same method to check the reachability from the input ADC of CONTROLLER to both BELL-OUTPUT and GATE-OUTPUT in the CDFD given in Figure 6 (the checking process is omitted for brevity). Thus, data flow reachability in a hierarchical CDFDs can be checked separately from level to level throughout the entire CDFD hierarchy. We believe that this checking method would be effective and practical for large and complex systems.

Data flow reachability, however, has a limitation. It may unable to accurately determine whether a path under checking really satisfies the data reachability semantically if data flows in the path are dependent on the detailed value and/or certain conditions. An example of such a case is when train is available to the condition process ON-ADC, whether it can reach data store BELL-OUTPUT depends on if the values of ADC, BDC and GATE-STATUS satisfy the precondition of CONTROLLER. Furthermore, even if we can ensure that the above condition is satisfied, we cannot guarantee that BELL-OUTPUT will be bound to value ON, because it will depend on whether the CONTROLLER is specified correctly or not. To address this problem, software testing and/or rigorous proofs can be applied.

$$M_d = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{array} \begin{array}{c} \begin{array}{cccccccccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \end{array} \\ \left[\begin{array}{cccccccccccccccc} & 1 & & & & 1 & & & & & & & & & & \\ & & 1 & & & 1 & & & & & & & & & & \\ & & & 1 & & & 1 & & & & & & & & & \\ & & & & 1 & & 1 & & & & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & & 1 & 1 & & & & & & \\ & & & & & & & & 1 & 1 & & & & & & \\ & & & & & & & & 1 & 1 & & & & & & \\ & & & & & & & & & & & & & & & \\ & & & & & & & 1 & & & 1 & & & 1 & & \\ & & & & & & & 1 & & & & 1 & & & & \\ & & & & & & & 1 & & & & & 1 & & 1 & \\ & & & & & & & 1 & & & & & & & & 1 \\ & & & & & & & 1 & & & 1 & & & 1 & & \\ & & & & & & & 1 & & & & & 1 & & 1 & \\ & & & & & & & 1 & & & 1 & & & 1 & & 1 \end{array} \right] \end{array}$$

Figure 7: Adjacent matrix

$$M_d^+ = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{array} \begin{array}{c} \begin{array}{cccccccccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \end{array} \\ \left[\begin{array}{cccccccccccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & 1 & 1 & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & 1 & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & & & & & & & & & \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & & & & & & & & & \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right] \end{array}$$

Figure 8: Transitive closure

4.2 Specification Testing

Specification testing consists of three steps. First, test data are generated based on the specification of condition processes and the structure of CDFD using certain criteria (e.g. branch testing), as described by Offutt and Liu [12]. Second, the specification is evaluated with the test data. Finally, testing results are analized to decide whether the test is successful (i.e. bugs are found) or not by examining if they satisfy the proof obligation.

To demonstrate that the path for ensuring the corresponding safety property is semantically sound using testing, we need to test two things. The first is to test each condition process associated with this path to assure the consistency of its specification with respect to its semantics. Another thing is to test the two adjacent condition processes associated with this path to ensure that the relevant part of the postcondition of predecessor condition process implies the precondition of its successive condition process.

For example, to verify the correctness of the path [x_1, ADC, BELL-OUTPUT], we need to test the following three things:

1. Condition processes ON-ADC. Especially, when train is available, ADC is set to OCCUPY.
2. The postcondition of ON-ADC, regardless of the values of BDC and GATE-STATUS, implies the precondition of CONTROLLER.
3. Condition process CONTROLLER. Especially, when ADC is equal to OCCUPY, BELL-OUTPUT should be set to ON and GATE-OUTPUT to DOWN.

The most important thing in testing is to generate test data. For the sake of space, we use the first and second cases as an example to show the principle of this approach. To test this condition process, we need to generate test data for train, train1, and ADC based on its specification in the specification module to demonstrate that when the precondition is **true**, there exists a **true** postcondition. As long as we can find one group test data which satisfies this semantics of the condition process, we can assert that ADC is semantically reachable from data flow train. We generate a test consisting of several group test data as follows:

where **bound** means that data on a variable is available while **nil** means either data on a variable is not available or the value of an expression is undefined.

As the last group test data in this table satisfies the condition that when the precondition of the condition process is true, its postcondition is also true, this test shows that the specification of this condition process is consistent with its semantics [8]. The same approach can be applied to test CONTROLLER.

Now let us test whether the postcondition of ON-ADC, together with any given values for BDC and GATE-STATUS, implies the precondition of CONTROLLER. A test is given in the following table:

where $Post_D$ and Pre_C denote the postcondition and the precondition of condition processes ON-ADC and CONTROLLER, respectively.

This test demonstrates that the postcondition of ON-ADC implies the precondition of CONTROLLER, according to the SOFL logic, the same three logic as for VDM [3]. Although this testing is not as convincing as a formal proof, it does give us confidence of the correctness of the data flow reachability of the path concerned. In general, the more test data are used, the more trustable the test result is.

4.3 Rigorous Proofs

Rigorous proofs can be used to verify the same properties as testing does, but they are supposed to be more convincing for the assurance of the correctness of data flow reachability semantically than testing because they tend to cover all the possible cases. To verify the correctness of a path, we need to verify that each condition process associated with the path is correctly specified and the relevant part of the postconditions of its predecessor condition processes imply its precondition.

Let us take the same path [x_1, ADC, BELL-OUTPUT] as an example. We first need to prove that both ON-ADC and CONTROLLER are specified consistently with its semantics. Then we need to prove that the postcondition of ON-ADC, regardless of the values for BDC and GATE-STATUS, implies the precondition of CONTROLLER. If the postcondition of ON-ADC alone cannot imply the precondition of CONTROLLER, that means the reachability of this path from x_1 to BELL-OUTPUT is under the constraint of the conditions of BDC and GATE-STATUS, which may cause a failure of sending a signal to close the gates of the railway crossing.

To prove the consistency of ON-ADC, for example, we need to show that the following condition is not a *contradiction*.

Pre_D **and** $Post_D$

where Pre_D and $Post_D$ denote the precondition and the postcondition of condition process ON-ADC.

train	train1	ADC	precondition	postcondition
bound	**nil**	OCCUPY	**true**	**nil**
nil	bound	CLEAR	**nil**	**nil**
bound	**bound**	OCCUPY	**true**	**true**

train	train1	ADC	BDC	GATE-STATUS	$Post_D$	Pre_C	$Post_D \Rightarrow Pre_C$
bound	**nil**	OCCUPY	CLEAR	UP	**nil**	**true**	nil
bound	**bound**	OCCUPY	CLEAR	UP	**true**	**true**	**true**
bound	**bound**	OCCUPY	CLEAR	DOWN	true	**true**	**true**

In fact, this is true because

true and ADC = OCCUPY **and bound**(train1)

$\vdash$

ADC = OCCUPY **and bound**(train1).

This predicate formula is not a contradiction because ADC and train1 are independent, and when ADC = OCCUPY and **bound**(train1), this overall proposition yields **true**. Therefore, it proves that the specification of ON-ADC is consistent.

To prove that the postcondition of ON-ADC implies the precondition of CONTROLLER, we need to establish the following theorem:

$Post_D \vdash Pre_C$.

The proof of this theorem is as follows:

ADC = OCCUPY **and bound**(train1).

$\vdash$

ADC = OCCUPY

$\vdash$

true

end-of-proof.

Since condition process CONTROLLER has a decomposition given in Figure 6, it is also necessary to prove that, for example, both BELL-OUTPUT and GATE-OUTPUT are reachable from ADC by using the same approach.

It is worth mentioning that rigorous proofs are usually complicated for large and complex systems and usually cannot be automatically conducted, while the reachability checking and testing can be highly automated if a tool is provided.

5 Evaluation of SOFL Approach

We evaluate the SOFL approach in two respects. One is to present our experience of applying the SOFL approach to the railway crossing controller, and the other is to compare SOFL approach with related works from an analytical view.

5.1 Experience

In the beginning of this project the first author was familiar with SOFL language and method, but had little knowledge about railway crossing controller and other railway control systems. He also had little experience in applying SOFL to safety-critical and real-time systems, although with some experience in applying SOFL to information systems. In contrast, the other three authors knew little about SOFL and formal methods in general, but were very experienced in designing railway control systems. Our cooperation started with the construction of the requirements specification of the railway crossing controller based on the description in [11]. Through this project, we find several features of SOFL which are supportive to the application of formal methods in a practical manner.

SOFL is easy to learn and effective for communications. This advantage is mainly due to the fact that SOFL integrates Petri Nets, data flow diagrams, and VDM-SL where the first two are familiar to all the authors. The formal notation used to write specifications for condition processes in the form of pre and postconditions was unfamiliar to the other three authors, but it did not take much time for them to learn the notation because the notation uses the first order logic and basic set theory which they know well. Furthermore, SOFL specifications are organized hierarchically where lower level condition data flow diagrams describe a refinement of upper level

condition processes. Thus, the complexity of each level condition data flow diagram of an entire specification can be limited to a level which can be easily managed.

Formal specifications help to clarify effectively ambiguity and misunderstanding. After studying the informal description of a railway crossing controller given in [11], the first author built the first version of the specification using SOFL to model the object system and the controller, but with many misunderstanding and insufficient information because he lacks knowledge in detail about how the railway crossing operation is controlled in practice. However, the specification offered us a firm basis for discussion and improvement. In particular the first author has gained much understanding through three times revisions of the first specification, while the other three authors have gained the skills of reading and writing SOFL specifications. In fact, the specification of the railway crossing controller given in this paper is finalized by the second author based on our discussions and previous versions.

Three techniques for SOFL specifications offer a flexibility for verifying the safety properties of systems based on the developer's engineering judgement. Data flow reachability can be automatically done (if a tool is available), but it can only check on syntactical level. Specification testing can be used to check the consistency of the data flow reachability with respect to the semantics of SOFL, but with some limitation due to the intrinsic limitation of testing approach. Rigorous proofs are more convincing than testing, but usually require high skills to conduct and may be very expensive. The developer can choose the rigorous proofs for verifying the most critical and less confident parts and choose the other two for less critical and more confident parts of the system.

Despite the above advantages of SOFL, we find that lack of effective tool support for SOFL may be a barrier for its application to large scale projects. During specification and verification, it becomes critically important for assurance of system quality and productivity that developers are able to efficiently draw and modify CDFDs, write specification modules, and verify safety properties.

5.2 Comparison with Related Works

Several different formal methods have been used to develop railway control systems, such as railway signaling controllers and railway crossing controllers. In this section we summarize and compare the related works with our work on the application of SOFL to the railway crossing controller presented in this paper.

5.2.1 Textual Formal Notations

One of the most well-known applications of formal methods is to specify the automatic train protection system SACEM (Systèm d'Aide à la Conduite Et à la Maintenance) using B method by GEC-ALSTHOM-Transport ordered by RATP (the Paris Subway Authority) [13]. The basic purpose of SACEM was to permit the safe reduction of the interval between two consecutive trains from 2 minutes 30 down to 2 minutes. SACEM was originally produced using Modula2 based on a standard V life cycle and the code was passed through a Hoare verification process from its informal specification. However, as the verification process was based on the informal specification, the customer (RATP) demanded an extra check that consisted in writing a formal specification of procedures, using the B formal language [14]. Despite some encouraging results, many problems, such as linking an informal specification to a formal design specification in B and providing a programming guide, were still unsolved.

Another application of formal methods is to formalize British Rail's Signaling Rules using Z by a small team from Praxis for British Rail's Network SouthEast (now Railtrack) [15]. The result of this work was a general Requirements Specification document for railway interlocking systems and the formal specification accounted for 135 pages of this document. The Requirements Specification will support Railtrack's evolving strategy of procurement based on performance-based scheme specification. In this document, Z is used to provide three major specifications. One is the specification of the *track network topology*, including segments and nodes, locations, paths, and interference between paths. The second one is the *conceptual foundation* that specify an abstract railway system which builds on the track topology. Another specification is the Signaling System specification that defines items such as signals and track circuits in order to express the British Rail signaling rules. This application shows that using Z language is effective in understanding the signaling practices. It also shows that the model-based approaches such as Z is helpful in clarifying and identifying requirements.

Despite the reported benefits of using B and Z, there is also a critical difficulty. As both B and Z only uses textual mathematical notation, communications between the developers and the domain experts via the formal specifications are extremely difficult. In contrast, SOFL offers data-oriented, comprehensible, graphical notation for describing the overall structure of the specification that helps communications between developers and domain experts. The condition data flow diagrams also

provide a guideline for formalizing the components of the system.

5.2.2 Graphical Formal Notations

Two graphical formal notations have been used to develop railway crossing controllers. One is *Timing Petri Nets* which is applied to model a railway crossing system and to prove that the system is safe [11]. In the model *token* is used to represent the arrival of the train and *transition* is used to represent an action to transfer the status of the train. The waiting time for firing a transition is used to model the timing behaviors of the train at each status. This work is useful, but also indicates several problems in using Timing Petri Nets. Petri nets do not provide an expressive mechanism to specify the detailed functionality of transitions, but only use graphical symbols to show the transitions of tokens. Token is an abstract object and it does not represent explicit information.

Another application of graphical, formal notation is described by Dierks and Dietz [16]. This work uses a *generalized railroad crossing* to illustrate a fully graphical approach to the formal development of correct reactive real-time systems. The graphical language *Constraint Diagrams* is used to formally capture requirements and to support the formal refinement steps towards implementable requirements. Since the formal semantics of the graphical language is given based on Duration Calculus [17], correctness of refinement steps can be formally proven. It is claimed that visuality eases application of formal reasoning and helps understanding timing behaviors of the system.

However, both applications are on a quite small example of the railway railroad crossing. Fully graphical language approach may work well in this circumstance, but it will meet a strong challenge when it is applied to a large scale system. The system would be more complicated, more difficult to write and understand, and would take more time to document and manage. An appropriate integration of graphical notation and textual notation is usually effective to deal with complexity of systems. This is a primary advantage of SOFL. The graphical notation CDFDs in SOFL is suitable for describing the overall structure of the entire system, while the formal, textual notation is convenient to be used in specification modules to define data flows, stores, and condition processes occurring in the corresponding CDFDs. This structure also facilitates rigorous proofs of the internal consistency. Furthermore, SOFL offers a flexibility for the developer to choose the graphical notation or formal notation as the primary tool to describe systems for different applications.

6 Conclusions and Future Research

We describe a formal, practical approach to specifying and verifying software for safety-critical systems using SOFL and demonstrate its usability by presenting its application to a railway crossing controller. Specifying requirements focuses on the functionality, with the consideration of safety constraints and its interaction with the surrounding operational environment. We provide three techniques to verify whether the specification satisfies safety properties of the enitre system. The techniques are data flow reachability checking, specification testing, and rigorous proofs, respectively. They provide assurance both syntactically and semantically. These techniques can also be applied to verify normal functional requirements.

Although we have extended SOFL to model timing behaviors, we have not worked out any appropriate way to verify the timing property of specifications yet. This issue will be one of our future research. Our another plan is to apply the proposed approach to other larger and more complex railway systems, and to develop a software environment to support the application of SOFL and the three verification techniques. Furthermore, we are also interested in exploring techniques to verify safety properties of SOFL specifications by using fault tree techniques.

7 Acknowledgements

We would like to thank Hiroshima City University and Mitsubishi Electric Corporation for their research grants to support this work.

References

[1] Anthony Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13(2):66–76, March 1996.

[2] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1994.

[3] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International(UK) Ltd., 1990.

[4] The RAISE language group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall International, 1992.

[5] Hossein Saiedian. An Invitation to Formal Methods. *Computer*, pages 16–30, Aril 1996.

[6] Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.

[7] Shaoying Liu and Yong Sun. Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems*, pages 137–144, Ft. Landerdale, Florida, U.S.A., November 6 - 10, 1995. IEEE Computer Society Press.

[8] Chris Ho-Stuart and Shaoying Liu. A Formal Operational Semantics for SOFL. In *Proceedings of The 1997 Asia-Pacific Software Engineering Conference*, pages 52–61, Hong Kong, December 1997. IEEE Computer Society Press.

[9] Shaoying Liu. A Formal Requirements Specification Method Based on Data Flow Analysis. *The Journal of Systems and Software*, 21:141–149, 1993.

[10] B. Berthomieu and M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[11] Ryouetsu Sasaki and Tomohiro Yoneda. Formal Verification of a Railway Crossing Controller Using Petri Nets. *Transactions of IEE Japan (in Japanese)*, 115-C(1):157–164, 1995.

[12] A Jeff Offutt and Shaoying Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, (to appear).

[13] Babak Dehbonei and Fernando Mejia. Formal Methods in the Railways Signalling Industry. In *FME'94: Industrial Benefit of Formal Methods*, Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994. Springer-Verlag.

[14] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag London Limited, 1996.

[15] Trevor King. Formalising British Rail's Signalling Rules. In *FME'94: Industrial Benefit of Formal Methods*, Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994. Springer-Verlag.

[16] Henning Dierks and Cheryl Dietz. Graphical Specification and Reasoning: Case Study Generalised Railroad Crossing. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, Proceedings of 4th International Symposium of Formal Methods Europe, Graz, Austria, September 1997. Springer-Verlag.

[17] Chaochen Zhou, C.A.R. Hoar, and A.P. Ravn. A calculus of durations. *Information Process Letters*, 40(5):269–276, 1991.