Formal Representation of Safety Verification Conditions

K. Wong

Department of Computer Science
University of British Columbia
Vancouver, BC, Canada V6T 1Z4
tel (604) 822-4912 fax (604) 822-5485
kwong@cs.ubc.ca

J.J. Joyce

Hughes Aircraft of Canada Limited
#200-13575 Commerce Parkway
Richmond, BC, Canada V6V 2L1
tel (604)279-5721 fax (604)279-5982
jjoyce@ccgate.hac.com

Abstract

This paper identifies the software information that must be represented in a formal specification of source code level "safety verification conditions" (SVCs) for an object-oriented software system. The formalization does not necessarily require a notation with object-oriented constructs. In particular, a semantically simpler notation based on typed predicate logic is adequate for representing these conditions. The formal source code level SVCs are used as input into software safety verification. However, formalizing the conditions in itself contributes to the safety analysis, by requiring a careful examination of the source code in order to identify the relevant software elements. The formalization is the final step of a process involving the refinement of system level SVCs into source code level SVCs. The process is outlined and demonstrated for a hypothetical chemical factory information system.

1. Introduction

An important step in the development of a software-intensive safety-critical system is the static analysis of the source code in order to raise assurance that the system is safe. Typically, safety is defined in terms of a set of system hazards, and static analysis will include review of the source code with respect to these hazards. The review must focus on the safety-critical source code and determine the risk of the software contributing to a hazard.

One approach to the static analysis is to derive system level "safety verification conditions" (SVCs) from the hazard and to use them as a guide in uncovering the hazard-related code and in performing the analysis [3]. However, hazards and the system level SVCs tend to be high-level statements of system behavior, far removed from the software implementation. It is necessary to identify the relevant software components and attributes, which must then be communicated to stakeholders, such as system developers and other safety engineers.

This paper outlines a process for refining system level SVCs into source code level SVCs. Furthermore, the source code level SVCs are expressed in a "codified form" [8], using clearly defined rules of expression. The goal is to obtain explicit and precise specification of the relevant software components and attributes in the system level SVCs, in order to have a more effective static analysis of the source code.

In creating a codified form of the source code level SVCs, it is necessary to determine which aspects of the software architecture must be captured. In the case of a system with an object-oriented architecture, this includes particular object-oriented features of the code. Since the source code level SVCs essentially specify safe software behavior, which for an object-oriented system is described by object interactions, the relevant object-oriented features are objects and their state and behavior. Though objects are typically instances of a class, which may belong to a class hierarchy, the source code level SVCs do not require specification of the underlying static class structure.

The notation used to codify the source code level SVCs must be able to represent the relevant object-oriented features. One possibility would be formal notation with object-oriented constructs, such as one of the object-oriented approaches to Z [1]. Such notations are particularly suited for formally specifying an object-oriented design, which includes the classes and their static relationships. However, as the source code level SVCs only make reference to the software behavior and not the underlying logical class design, a semantically simpler notation is adequate for representing the conditions.

In this paper, a notation based on typed predicate logic is used to specify the source code level SVCs. With this notation, the relevant object-oriented features, object state and behavior, are captured in a simple and natural way, by associating types with classes and functions with class attributes.

In Section 2, a process is outlined for the refinement of the source code level SVCs from system level SVCs. Section 3 describes the basic information that is to be captured in a codified form of the source code level SVCs. A safety-critical information system for a chemical factory is introduced in Section 4, along with a system hazard. A system level SVC, generated from analysis of the hazard, is presented in Section 5. The software architecture and the hazard-related modules are identified in Section 6, and then used to create functional block level SVCs. The source code is examined, in Section 7, to create source code level SVCs. S, which is a notation based on typed predicate logic, is introduced in Section 8. S is then used, in Section 9, in the formalization of the source code level SVCs. A discussion follows in Section 10, while Section 11 provides a summary of the paper.

2. Stepwise Refinement of Safety Verification Conditions

This paper is based on a process where source code level "safety verification conditions" (SVCs) are derived from system level SVCs. The source code level SVCs are then used during a static analysis of the source code in support of system safety verification.

The inputs for the stepwise refinement of the system level SVCs are:

- **A set of system level SVCs**.
- **Description of the software design and architecture**.
- **Source code for the relevant software components**.

The output of the process is:

- **A set of source code level SVCs**. The source code level SVCs are expressed in a codified form that refers directly to the software objects and their attributes, where the satisfaction of these source code level SVCs is sufficient to satisfy the input system level SVCs.

There are three distinct steps in this process:

1. **The system level SVCs are refined into functional block SVCs**. The software design and architecture are examined for the hazard-related functionality, which is then partitioned into "functional blocks", i.e., into blocks of code that execute in different processes. The functional blocks can be viewed as procedures with well-defined input and output parameters. The system level SVCs are then refined into conditions on the functional block's input and output, as well as a set of timing constraints.
2. **The functional block SVCs are refined into source code level SVCs**. The source code for each functional block is examined to determine the relevant source code objects and their attributes.
3. **The source code level SVCs are translated into a codified form**. The informal description of the source code level SVCs is re-written in a more precise form.

3. Formalization of Safety Verification Conditions

The final step of the process of deriving source code level SVCs involves translating the source code level SVCs into a codified form. For a system with an object-oriented software architecture, the choice of representation must be sufficiently expressive to represent the object-oriented features that are captured in the source code level SVCs.

3.1 Codified Form

A "codified form" is a term used by De Marco to motivate structured analysis [8]. A codified form involves a choice of representation with clearly defined rules of expression, along with validation checks that can be performed algorithmically. Though this could involve a formal syntax, the expressions could also be non-textual, such as data flow or object-scenario diagrams.

There are a number of motivations for having a codified form of the source code level SVCs. For example, they would then be less subject to personal style and more amenable to systematic review by peers. Furthermore, if a machine-readable notation were used, there would be the potential for tool-based support.

3.2 SVCs and Object-Oriented Software

In order to choose an appropriate notation for representing the source code level SVCs, it is first necessary to identify the software features that must be specified. The SVCs are essentially constraints on system behavior, which for an object-oriented system is described in terms of object interactions. Therefore, the relevant software features are objects and their behavior.

Objects may be described in terms of their identity, state and behavior [5]. An object state is determined by a set of properties and their values at a given instant of time. Object behavior is described in terms of the object's state changes and message passing. In general, the object behavior is a function of its state and the operations performed upon it.

The SVCs constrains the system behavior by defining a final "safe" state, given by a set of object states, in terms of earlier events, operations and object states. As such, a formal notation must be able to represent:

- Symbolic instants of time, i.e., through variables such as "T" and "T1";
- Symbolic data values, i.e., through variables such as "D" and "D1";
- Symbolic object references, i.e., through variables such as "Sensor" and "Message";
- Object states, where a state is defined in terms of a set of values corresponding to named attributes;
- An occurrence of events in an "uninterpreted" manner, i.e., naming the event without the operational details;
- Relationships between data values, as well as functions on the data values, in an "uninterpreted" manner;
- Standard propositional logic connectives, as well as quantification over variables.

Objects are typically an instance of a class, where the object properties correspond to class attributes, and the operations on objects correspond to class operations. The class will often be part of a class hierarchy, connected to the other classes through relationships such as inheritance and aggregation. Though a safety analysis of the source code will eventually involve tracing through the class hierarchy to identify the relevant object code, the SVCs themselves do not require specification of the underlying static class structure.

Furthermore, though the SVCs do specify object states at particular instances of time, they do not describe object dynamics. In other words, it will not be necessary to model the object behavior over time. Unlike software design and architecture artifacts, the SVCs do not provide a model of the system, but simply represent what constitutes safe system behavior.

One useful feature that a notation should provide is tracebility between the objects referenced in the source code level SVCs and the source code. This is important during the static analysis, which involves examination of the relevant objects in the source code. For example, this traceability could be achieved by lexical similarity between the ways objects are referenced in the source code, to the ways they are referenced in the conditions.

3.3 Choice of Notation

The notation used to represent the source code level SVCs must be able to capture the object-oriented features discussed in the previous section. An obvious choice would be an object-oriented formal notation, such as one of the object-oriented extensions to Z [1]. These are particularly useful when specifying the system's object-oriented design.

However, specification of the source code level SVCs requires capturing only certain object attributes and behavior, and not the logical design of the system. Therefore, the choice of notation should not be based on how readily it can duplicate the details of the object-oriented design. In fact, there would be advantages in using a semantically simpler notation, without the object-oriented constructs.

In particular, a notation based on a typed predicate logic would be adequate for specifying the conditions. For example, the semantics of one formulation of typed predicate logic, namely, the "term language" of the HOL system, can be given by just five axioms and eight inference rules [6].

The notation used in this paper is S [3], which is a machine-readable specification notation based on typed higher-order logic, and is closely related to HOL's term language. As well, S is supported by the type-checker Fuss, and S specifications may also be used as input into the HOL theorem prover.

4. Example - Chemical Factory Information System

For illustration purposes, we consider a hypothetical real-time safety-critical information system for a chemical factory, with Ada-based software architecture. In particular, the code is written in Ada 83, which is an object-based language with support for encapsulation, but not for inheritance, and has been used in many safety-critical systems. This particular system is safety-critical in the sense that its purpose is to provide information to human operators who make critical decisions.

4.1 System Description

The physical layout of the factory consists of a set of reactor vessels each with a pair of redundant but prioritized sensors that record vessel data such as the temperature. The sensors are connected over a LAN to a central server and a set of operator consoles. The chemical factory control system runs on the central server and the operator consoles, maintaining and processing the vessel information as it is received over the LAN, which it displays on the operator console monitors.

4.2 Safety-Related Hazard

Following a process such as the system safety engineering process outlined in [2], we assume that the display of an "invalid" value as the temperature of a vessel is identified as a system hazard for the chemical factory control and monitoring system.

> ***Hazard:*** *An invalid temperature, D, is displayed for vessel V at time T.*

It may be assumed that the identification of this hazard resulted from some earlier analysis which shows that the display of an invalid value as the temperature of a vessel, in combination with other conditions, could lead to a mishap such as a fire or explosion. Further analysis of the hazard reveals that a displayed value may be invalid because it has been corrupted or because it has become stale.

5. System Level Safety Verification Conditions

The system level SVCs are used as input into the process, described in Section 2, for deriving the codified form of the source code level SVCs. The system level SVCs are initially generated in support of a rigorous argument about the system safety, i.e., an argument that a system hazard does not occur. A brief sketch of the argument for the chemical factory hazard is given below, while full details can be found in reference [3].

The argument begins with the assumption that the hazardous condition identified in the previous section is possible. The analysis then proceeds in a stepwise manner by attempting to show that this assumption leads to a logical contradiction. The analysis of this particular hazard will branch as a result of reasoning by cases. When the analysis branches into one or more cases, each branch must be "closed" by showing that each branch leads to a logical contradiction. In the course of generating contradictions, system level SVCs are introduced. Each system level SVC is intended to be the minimum condition required to close a particular branch of the analysis.

The result of the analysis of the hazard is the discovery of six distinct system level SVCs. The following is an example of one the system level SVCs:

> ***Safety Verification Condition:***
> *For all vessels, v, displayed temperatures, d, and times, t, if the displayed temperature of vessel v is set to d at time t, then at some time no earlier than MAX_SYSTEM_PROPAGATION_TIME milliseconds before t the system received a report from the external sensor monitoring system that the temperature of vessel v is d.*

MAX_SYSTEM_PROPAGATION_TIME is a constant that specifies the maximum time the system should take to display a vessel temperature on the screen after receiving a sensor update for that vessel.

6. Functional Block Level Safety Verification Condition

The first step in the process of creating source code level SVCs takes as input the system level SVCs, along with the software design and architecture, and creates as output a set of functional block level SVCs. This step involves identifying the hazard-related functionality and modules, and then partitioning the functionality into blocks, which execute in separate processes. Each of these “functional blocks” can be treated as a procedure, with well-defined input and output parameters.

The system level SVCs integrates system functionality and timing issues, which will be dealt with separately in the functional block level SVCs. This separation of functionality and timing issues then leads to the possibility of deriving functional block pre- and post-conditions from the non-temporal functional block SVCs [3].

6.1 The Hazard-Related Code

The hazard concerns the display of invalid vessel temperatures, where an update in the display of vessel temperature is typically the result of system reception of vessel sensor data. The functionality involved in the reception, processing and display of a vessel temperature are depicted in Figure 1, using the Booch notation [5]. This includes the interface to the sensors, which is provided by the Sensor Interface module, the processing of the sensor information, which is performed by the SensorServer module, and the user interface, which is maintained, by UIManager module.

In general, a search for the hazard-related code will involve more than just the “forward” functionality, beginning with the standard sensor inputs, described above. Any system functionality that could contribute to a hazard must be identified, which is best performed by a “backward” trace of the system, beginning with a hazardous output and then following the code backwards to all potential system inputs [3].

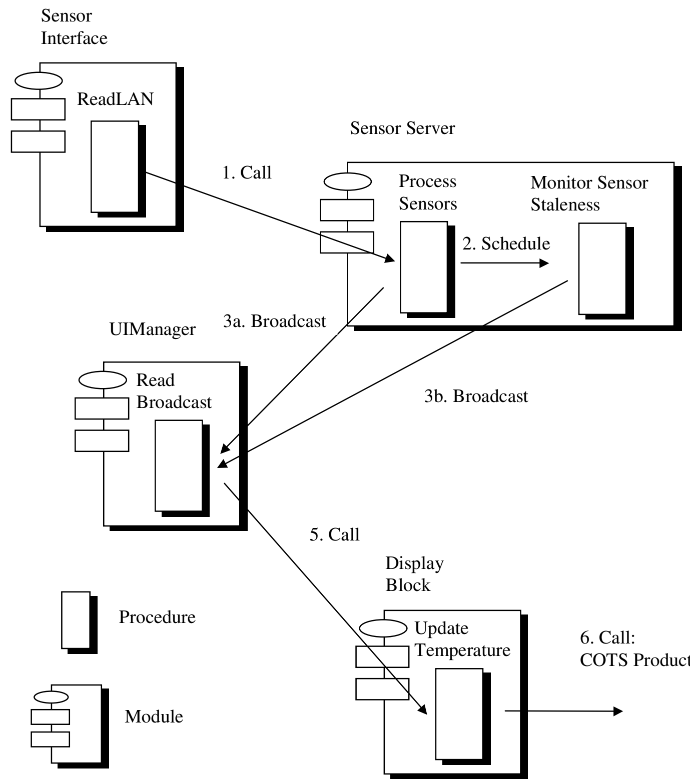

Figure 1 Temperature data flow highlighting the key modules and procedures.

6.2 The Functional Blocks

It is convenient to partition the hazard-related code into smaller blocks of functionality. The partition is performed along dynamic lines, i.e., according to functionality that execute in different processes. For example, this could involve a process on a different computer, or a different "thread", i.e., a lightweight process, on the same processor. Such a partition will allow for reasoning about the effect of properties such as concurrency and the order of execution of each block, on safety.

The blocks are uncovered by following the relevant execution sequences until a procedure call results in the execution of code in a new process. The input parameters of the procedure call are then the input parameters of this functional block. The execution sequences are then followed until a procedure call results in system output or the execution of code in different process. The system output, or the input into the new process, is the functional block output.

The hazard-related functionality for the chemical factory information system can be partitioned into three different functional blocks, as shown in Figure 2.

When the external sensor monitoring system sends a LAN message to the information system, the LAN message is read by the procedure `ReadLAN` and then processed by the procedure `ProcessSensors`. We regard the reading and processing of a LAN message in this manner as a single functional block, LANToBroadcast.

Processing of a LAN message, which contains an updated sensor reading for a vessel temperature, causes a separate process, `MonitorSensorStaleness`, to be initiated. This process is intended to mark the display of the temperature for a vessel as "unavailable" if the most recently displayed value becomes stale. We regard this as a second functional block, MonitorSensorStaleness, of that portion of the implementation related to this hazard.

A third functional block, BroadcastToDisplay, involving the procedures ReadBroadcast and UpdateTemperature, handles changes to the display broadcast by either ProcessSensors or MonitorSensorStaleness.

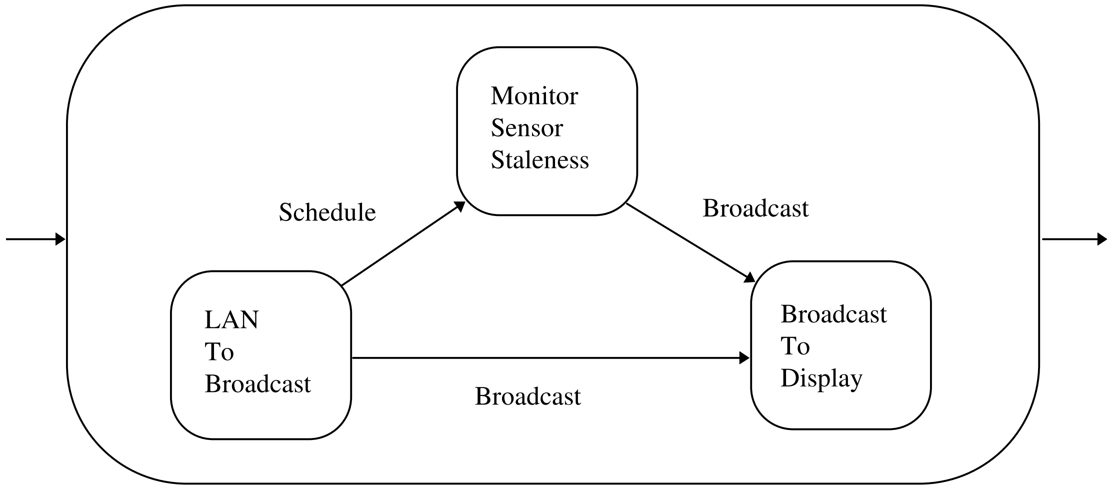

Figure 2 The hazard-related functional blocks.

6.3 Refinement of the System Level Safety Verification Conditions

The SVC described in Section 5, applies to the propagation of the temperature through the system. The two main functional blocks, LANToBroadcast and BroadcastToDisplay, carry out this functionality with a Broadcast mechanism providing the block communication, as shown in Figure 2. As a result, the system level SVC can be refined into functional block level SVCs involving the relevant functional blocks and the Broadcast mechanism. The resulting functional block level SVCs are such that satisfying them imply that the system level SVCs are satisfied.

The system level SVCs are described in terms of system inputs and outputs, i.e., data values and events, at particular instants of time. To map the system level SVC onto the functional blocks, which can be viewed as a set of procedures, the events are associated with functional block invocations and output.

As well, the time constraints and the data conditions are dealt with in separate functional block SVCs. This will simplify the subsequent safety analysis of the source code. The association of input with output is then made through the functional block, i.e., the functional block takes the input and creates the output. Such a view of the functional block level SVCs lends itself to the creation of functional block pre- and post-conditions.

For example, the LANToBroadcast block takes as input a LANMessage and creates as output broadcast messages. A functional block level SVC involving the non-temporal relationship between the input and output is:

LANToBroadcast Functional Block Level SVC:

(1) For all broadcast messages, m, vessels, v, and temperatures, d, if the LANToBroadcast block broadcasts message, m, with the information that the temperature of v is d,

(2) then there exists a LANMessage, L, which the LANToBroadcast block received from the external sensor monitoring system that the temperature of vessel v is d, which the LANToBroadcast block processed into the broadcast message.

7. Source Code Level Safety Verification Conditions

The second step in the process of creating a codified form of the source code level SVCs takes as input the functional block level SVCs, along with the source code, and creates as output the informal source code level SVCs. This step involves identifying the objects in the source code that carry out the system behavior described by the functional block level SVCs.

The chemical factory information system is written in Ada 83, which does not support a class construct. Instead, a class can be represented by an Ada package, which exports a private or limited private type, named `Object`, which is sometimes made "public" for convenience [7]. The `Object` type is typically an Ada record type, where each component represents a class attribute. Instances of the class are then instances of the `Object` type.

The source code level SVC for the LANToBroadcast block is given by:

***LANToBroadcast Block Source Code Level SVC (informal)**:*

(1) For all BroadcastMessages, m, Sensor Objects, r, if the LANToBroadcast block broadcasts message, m, with Sensor Object, r, with the information that the vessel with SensorID, S, has PresentTemperature, D,

(2) then there exists a LANMessage, L, with SensorUpdate, U, which the LANToBroadcast block received from the external sensor monitoring system from a sensor with a SensorCodeEstab, S1, which is correlated to the SensorID, S, and a TemperatureEstab, D1, which is converted from D, which the LANToBroadcast block processed into the broadcast message.

The construction of this source code level SVC from the functional block level SVC is now examined. Each part of the functional block level SVC will be discussed in turn.

(1) For all broadcast messages, m, vessels, v, and temperatures, d, if the LANToBroadcast block broadcasts message, m, with the information that the temperature of v is d,

The outgoing broadcast message is an array of sensor objects,

```
type BroadcastMessage is array (SensorRange) of Sensor.Object;
Message : BroadcastMessage;
```

The sensor object attributes include `SID`, the sensor ID, and `SensorTemperature`, the vessel temperature reading. A sensor update is received from a sensor with a unique sensor ID. There is no direct reference to external vessels in this functional block, which is only concerned with the sensors and sensor readings. The correlation of the sensor ID to the corresponding vessel is performed in the BroadcastToDisplay functional block.

The sensor object attributes are represented in the Ada code as,

```
package Sensor is
   type Object is
      record
      SID : SensorID;
      SensorOperation : Operation;
      SensorQuality : Quality;
      SensorTemperature : Temperature;
      end record;
```

The first part of the functional block level SVC can be re-written as:

(1) For all BroadcastMessages, m, Sensor Objects, r, if the LANToBroadcast block broadcasts message, m, with Sensor Object, r, with the information that the vessel with SensorID, S, has PresentTemperature, D,

(2) then there exists a LANMessage, L, which the LANToBroadcast block received from the external sensor monitoring system that the temperature of vessel v is d, which the LANToBroadcast block processed into the broadcast message.

The incoming LANMessage is a LANMessage object, whose attributes are represented in the source code as,

```
package LANMessage is
   type Object is
      record
         LMID : LANMessageID;
         NumUpdates : SensorUpdateRange;
         Updates : SensorUpdateArray;
      end record;
```

The sensor readings are stored in an array of `SensorUpdates`,

```
type SensorUpdateArray is array (SensorUpdateRange) of SensorUpdate;
```

where a `SensorUpdate` is a Ada record type,

```
type SensorUpdate is
   record
   InterpolatedState : InterpolationRange;
   TemperatureEstab : Temperature_S;
   SensorCodeEstab : SensorCode;
   end record;
```

The `TemperatureEstab` attribute is the raw sensor temperature reading, which is transformed into the `SensorTemperature`. The `SensorCodeEstab` attribute is the the raw sensor ID, which is converted into the `SID`.

The second part of the functional block level SVC can be re-written as,

(2) then there exists a LANMessage, L, with SensorUpdate, U, which the LANToBroadcast block received from the external sensor monitoring system from a sensor with a SensorCodeEstab, S1, which is correlated to the SensorID, S, and a TemperatureEstab, D1, which is converted from D, which the LANToBroadcast block processed into the broadcast message.

8. Typed Predicate Logic

The source code level SVCs may be codified using a formal specification notation based on typed predicate logic. This notation, called "S", was developed at the University of British Columbia to serve as a foundation for a variety of different approaches to formal specification [4,9,10,11]. This section provides a brief description of S and how it may be used to specify source code level SVCs in a manner which achieves a high degree of lexical correspondence with object-oriented code.

An S specification is a sequence of "paragraphs". Each paragraph is a fragment of ASCII text terminated by a semi-colon, which serves one of the following purposes:

- declares or defines a new type;
- introduces an abbreviation for a type expression;

- declares or defines a new constant;
- declares or defines a new function;
- declares or defines a new predicate;
- expresses an assertion.

The formalization described in Section 9 uses S in a particular style where objects are represented as "uninterpreted" types. For instance, a type named sensorObject is introduced by a type declaration,

```
: sensorObject;
```

to serve as the formal representation of sensor objects. In this style of S specification, attributes of an object are selectively introduced as selector functions. For example, the `SID` and `SensorTemperature` attributes of a sensor object, may be formally represented by a pair of S function declarations,

```
SID : sensorObject -> sensorID;
SensorTemperature : sensorObject -> temperature;
```

which declare SID and SensorTemperature to be functions which map values of type sensorObject to values of type sensorID and temperature respectively.

As means of achieving a degree of lexical correspondence with the Ada 83 source code, references to attributes of an object may be given using "dot notation". For example, the S expression

```
Sensor1.SensorTemperature
```

denotes the value of the SensorTemperature of the sensor object denoted by Sensor1. In S, dot notation is merely postfix application of a function to a value. For instance, Sensor1.SensorTemperature denotes the application of the function SensorTemperature to Sensor1. It is semantically equivalent to SensorTemperature (Sensor1), i.e., prefix application of a function to a value.

In addition to the formal representation of objects, uninterpreted types are used to represented data values whose composition is not relevant to the formalization task. For instance, the above example includes references to sensorID and temperature, both of which are introduced as uninterpreted types by the following S type declarations:

```
: sensorID;
: temperature;
```

Several other uninterpreted types are used in Section 9 to support the formalization of the source code level SVCs including sensorRange, sensorUpdateRange, sensorCode and temperature_S. This use of uninterpreted types is analogous to the typical use of "basic types" in Z specifications.

In the chemical factory example, the formalization of the source code SVCs includes the formal representation of elements of the source code, which are realized as arrays. For instance, the broadcast message created by the LANToBroadcast functional block is implemented as an array of sensor objects in the source code. For this purpose, an S type abbreviation,

```
: broadcastMessage == sensorRange --> sensorObject;
```

can be used to introduce broadcastMessage as a name for a function type which maps an array index to a sensor object. As indicated by the above type abbreviation, the value of the array index is constrained to be a value of type sensorRange. The formal representation of an array in this manner allows array references to be expressed in a manner which corresponds lexically to the array references in the source code. For example, the S expression,

```
Message (I).SensorTemperature
```

denotes the value of the SensorTemperature attribute of the I*th* element of the array denoted by Message. In this example, Message is either a variable or constant of type broadcastMessage and I is a variable or constant of type sensorRange.

The formalization of an SVC also involves the expression of relationships between denotable values. For instance, Section 9 introduces an uninterpreted infix predicate named is_converted_temperature_of,

```
(_ is_converted_temperature_of _) : temperature -> temperature_S -> bool;
```

which expresses a relationship between a value of type temperature and a value of type temperature_S. As presented here, this predicate is left "uninterpreted" on the basis of a decision that the relationship expressed by this predicate does not need to be formalized. Alternatively, this predicate could be defined, rather than declared, as a means of formalizing this relationship.

9. Formalization of the Source Code Level Safety Verification Conditions

The final step of the process of constructing codified source code level SVCs is the "codification" of the informal source code level SVCs. As discussed earlier, the notation used in this paper is S, which is based on a typed predicate logic.

The formalized source code level SVC is given in its entirety as,

```
forall (Message:broadcastMessage, I:sensorRange).
    (let D := Message(I).SensorTemperature in
  (let S := Message(I).SID in
    if(Message.isBroadcast)
    then (exists (LanMessage:lanMessage, J:sensorUpdateRange).
        (LanMessage.isReceivedLanMessage)
         AND (D is_converted_temperature_of
             ( (LanMessage.Updates)(J).TemperatureEstab))
         AND (S is_converted_sensor_code_of
             ( (LanMessage.Updates)(J).SensorCodeEstab) ))));
```

The formalization of each part of the source code level SVC is now presented.

(1) For all BroadcastMessages, m, Sensor Objects, r, if the LANToBroadcast block broadcasts message, m, with Sensor Object, r, with the information that the vessel with SensorID, S, has PresentTemperature, D,

As discussed in Section 8, the S type abbreviation, broadcastMessage, is used to represent broadcast messages, which are arrays of sensor objects. The S type sensorObject is used to represent sensor objects, and the S functions, SID and SensorTemperature, represent the corresponding sensor object attributes.

The first part of the source code level SVC can be represented by the following S fragment:

```
forall (Message:broadcastMessage, I:sensorRange).
    (let D := Message(I).SensorTemperature in
  (let S := Message(I).SID in
    if (Message.isBroadcast)
```

(2) then there exists a LANMessage, L, with SensorUpdate, U, which the LANToBroadcast block received from the external sensor monitoring system from a sensor with a SensorCodeEstab, S1, which is correlated to the SensorID, S, and a TemperatureEstab, D1, which is converted from D, which the LANToBroadcast block processed into the broadcast message.

The `Update` attribute of a LANMessage object is a `SensorUpdateArray` in the Ada Code. This array can again be represented by a type abbreviation, i.e.,

```
: sensorUpdateArray == sensorUpdateRange -> sensorUpdate;
```

Sensor updates are represented by the S type,

```
: sensorUpdate;
```

and the S functions,

```
TemperatureEstab : sensorUpdate -> temperature_S;
SensorCodeEstab : sensorUpdate -> sensorCode;
```

represent the corresponding sensor update attributes.

The relationship between the `TemperatureEstab` and `SensorTemperature` attributes is denoted by the uninterpreted, infix predicate, is_converted_temperature_of, as defined in section 8. Similarly, the relationship between SID and SensorCodeEstab can be represented by the uninterpreted, infix predicate, is_converted_sensor_code_from,

```
(_ is_converted_sensor_code_from _) : sensorID -> sensorCode -> bool;
```

The formalized segment of the SVC is then represented by the S fragment:

```
then (exists (LanMessage:lanMessage, J:SensorUpdateRange).
        (LanMessage.isReceivedLanMessage)
        AND (D is_converted_temperature_of
              ( (LanMessage.Updates)(J).TemperatureEstab))
         AND (S is_converted_sensor_code_of
              ( (LanMessage.Updates)(J).SensorCodeEstab) ))));
```

10. Discussion

Though the construction and formalization of the source code level SVC was presented as two separate steps, in practice, they overlapped. Using a formal notation to represent the relevant object states resulted in a careful and disciplined examination of the source code. Writing down the states in a precise manner was a useful aid in identifying the relevant object attributes and relationships. In particular, identifying the initial and final object states led to questions regarding the relationship between them, and helped to uncover intermediate operations.

In this regard, S was found to be particularly useful. The ability to represent the relevant objects and attribute types as simple uninterpreted S types and predicates was an invaluable book-keeping device. The simple semantics and the ability to achieve lexical similarity between the code and the source code level SVC meant that using the formal notation did not complicate the capture of the relevant object information. For example, the use of the post-fix form of function application to represent object attributes, closely mirrored the source code representation of object attributes as record fields. As a result, S did not require a large cognitive shift when used to record the relevant object attributes and operations.

11. Summary

During system safety verification, it is important to have a precise definition of safety that captures the critical aspects of the code. In this paper, safety was defined in terms of system level SVCs, which were refined into source code level SVCs. The relevant source code elements were identified as object states, as well as the relationships between them. A simple typed-predicate logic was seen to be an effective notation for representing the objects. In particular, it was not necessary to use an object-oriented formal notation. The result was a clear and precise specification of the relevant objects and their states, which could be used during a safety analysis of the source code. In fact, the formalization of the source code level SVCs itself could be considered an effective preliminary source code safety analysis.

12. Acknowledgments

This research was partially supported by B.C. Advanced Systems Institute, Hughes Aircraft of Canada Limited, and Macdonald Dettwiler. This work is a component of the formalWARE university-industrial collaborative project.

13. References

1. Susan Stepney, Rosalind Barden, and David Cooper (Eds), “Object Orientation in Z”, Workshops in Computing, Springer-Verlag, 1992.

2. Nancy G. Leveson, “Safeware: System Safety and Computers”, Addison-Wesley, 1995.

3. Ken Wong, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1998.

4. Jeff J. Joyce, Nancy Day, and Mike Donat, “S: A Machine Readable Specification Notation Based on Higher Order Logic”, in *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pp. 285-299, 1994.

5. Grady Booch, “Object-Oriented Analysis and Design with Applications (Second Edition)”, Benjamin/Cummings Pub. Co., Redwood City, California, 1994.

6. Mike J. Gordon and Tom F. Melham, “Introduction to HOL: A Theorem Proving Environment for Higher Order Logic”, Cambridge University Press, Cambridge, UK, 1993.

7. Grady Booch, “Software Components with Ada”, Benjamin/Cummings Pub. Co., Redwood City, California, 1987.

8. Tom DeMarco, “Structured Analysis and System Specification”, Prentice-Hall, Engelwood Cliffs, N.J, 1979.

9. Nancy Day, Jeff Joyce and Gerry Pelletier, “Formalization and Analysis of the Separation Minima for Aircraft in the North Atlantic Region”, in *4th NASA Langley Formal Method Workshop*, Hampton Virgina, USA, September 10-12 1997.

10. J.H. Andrews, N. A. Day and J. J. Joyce, “Using a Formal Description Technique to Model Aspects of a Global Air Traffic Telecommunications Network”, in *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocal Specification, Testing, and Verification*, Osaka, Japan, November 18-21, 1997.

11. Nancy Day, “A Model Checker for Statecharts”, Technical Report 93-35, UBC-CS, October, 1993.