system: The home heating system problem and the elevator problem were used to test the combination of specification extensions and to illustrate proposed extensions to the Descartes language.

- Validate Descartes' satisfaction of needed specification or propose additional extensions: Additional extensions to Descartes were proposed to satisfy the object oriented design of the real-time home heating system and the elevator problem. These systems were studied in detail to ensure each design was correct and efficient.
- Compare the Descartes specification to a UML design for further verification of system design: UML design and notation was studied in detail. Then a UML diagram was created for the home heating system and the elevator problem to validate the design and compare the Descartes specification language to a commonly used modeling technique.
- Analyze the security requirements for the real time object oriented systems and the UML design of security aspects.

II. Overview of Descartes

The Descartes specification language is a formal specification language, which creates an abstract interpretation of a software system. Formal software specifications can relieve incompleteness and uncertainty found in a natural language requirements document. Not only can formal specification relieve ambiguities, but also can provide consistency checking, automatic design creation, and static analysis validation [1, 2, 3]. Thus, formal languages provide a strong base for the implementation step in software development.

Descartes uses a tree structure notation called a "Hoare tree" to characterize and define data. This hierarchical structure is key to the languages syntax and is used to refine input and output data of a specification. Descartes is based on the functional model; thus, specifications must be expressed by defining the input and output data, then relating the output data as a function of the input data. Descartes also uses three data structuring methods based on the "Hoare tree" notation. These data structures include direct product, discriminated union, and sequence. A direct product defines a node that concatenates a set of elements or that has an exact set of subnodes. This data structure is used by default in Descartes [1]. A discriminated union is used to denote exactly one choice out of a set of elements. In Descartes, discriminated union is denoted by the plus sign (+) after a node name [1]. A sequence allows for specification of zero or more occurrences of elements. In Descartes, a sequence is denoted by an asterisk, "*", after a node name [1]. In addition, a range notation may be used after a sequence node name. The language also can use string literals to refine data, which are denoted by a string enclosed in single quotes.

Nodes in the Descartes language can be segregated into two categories: match nodes or reference nodes. A match node is represented by lower case letters, while a reference node is represented by upper case letters. A match node is a node that can obtain a value as a consequence of matching. The root node of a synthesis tree, intermediate nodes, and terminal nodes for abstract execution are denoted as match nodes. Intermediate nodes can be used to shape a tree [1, 2, 3]. A reference node is a node that can obtain a value of a match node with the same name. Module titles, parameters in title, terminal literal nodes, and module call nodes are denoted as reference nodes. Both matching and reference nodes acquire values during analysis or synthesis. Descartes pattern matching rules follow the matching rules of the SNOBOL programming language [1, 4].

To increase effectiveness of the language, Descartes supports top-down development using modules [1, 2, 3, 4]. Module calls are, in essence, unique reference nodes. Module nodes acquire values from what is returned from the called module. This approval can be thought of as calling a function or method in most programming languages. Descartes also has predefined modules for convenience. Below in Figure 1 is a sample specification.

```
INTEGER_OF_(FIXED_POINT_NUMBER)
        FIXED_POINT_NUMBER
                before_decimal(1.. )
                        DIGIT
                        '.'
                after_decimal(1.. )
                        DIGIT
return
                BEFORE_DECIMAL
```

Figure 1: Sample Specification

This small example illustrates the “Hoare tree” structure and top-down modularity in Descartes. The example also illustrates syntax for range specific sequences, string literals, and direct products.

III. Related Work

A. Software Requirements Specification (SRS)

Software requirements engineering is the process of determining, recording, negotiating, validating, and managing a set of requirements for a software system. More specifically, this process is referred to as an SRS. Requirements engineering can be decomposed into sub-phases of its own. These activities of the requirements engineering process can fall into the following categories: requirements elicitation, requirements analysis, requirements documentation, and requirements validation [5]. Specification entails identifying what the software should do, but not how the software should be implemented. The requirements phase of an SRS simply formulates ideas and concepts, while specification refines those ideas into consistent and complete descriptions. This refinement is done through understanding the external and internal behaviors of the software. This understanding is best

accomplished through a formal specification language that can be checked for ambiguity and consistency.

B. UML and Other Specification Languages

The Unified Modeling Language (UML) is a standard graphical language for visually modeling and constructing a software system. The Object Management Group created UML in 1997, exclusively for the design of object oriented programming languages [6]. Today, UML is one of the most widely used design methods. UML diagrams are not only understandable to developers, but to business users and end-users as well. This simplicity provides all members of a company insight on a software project. The overall goal of UML is to be a simple modeling tool to develop practical software systems [6]. UML was used for comparison and validation of the Descartes specification language in this research effort due to its wide use by software developers and its ability to specify real-time object oriented software. Other widely used specification languages include: Z notation, Object-Z (an object oriented extension to Z notation), Vienna Development Method (VDM), LePUS3 (a visual object oriented language), and several other extensions to UML [2].

C. Real-Time Systems

Real-time systems are such that the accuracy of a system does not only depend on the outcome of the computation, but also on the time in which the desired outcomes are produced [4, 7] Consequently, a correct result that is produced in the incorrect timing bounds will place a system in an incorrect state. Both the results and the timing must be correct in order for the system to be correctly functioning. Systems that have strict timing bounds are often called "hard real-time systems", while those systems that are more lenient are called "soft real-time systems." Slow response and missed deadlines can be tolerated on occasion in a soft real-time system; however, in a hard real-time system, there can be no missed timing constrains. There is absolutely no usefulness in producing a real-time task after its deadline has passed. In fact, producing an estimated result before the deadline is more desirable than a precise result after the deadline [4]. The timing of a real-time task should be completely predictable.

A real-time system can also be viewed as a stimulus/response system [7]. In other words, the system must have a timely response to a particular input. This input can be divided into the categories of periodic stimuli and aperiodic stimuli. A periodic stimuli is input that occurs at predicable time intervals. For instance, a system may inspect a sensor every 30 milliseconds and respond based on the sensor value. An aperiodic stimuli is input that occurs sporadically. This irregular input uses some type of interruption method to indicate some type of input in order for the system to respond accordingly [7].

Traditionally, the strict timing constraints of real-time systems cause the systems to be highly underutilized by using low-level programming techniques [4]. Low-level programming allows for development of quick and efficient programs. The importance of a system to respond quickly is more important than using higher-level techniques in creation. The disadvantage of using low-level programming languages is the lack of concurrency and shared resource constructs. In the past few years, high level languages, such as Java, have been extended for real-time systems [4]. Thus, as of late, real-time systems are becoming utilized at a higher level. Not only are traditional real-time systems being utilized, but also software systems in general are gaining more real-time constrains. One of these higher-level techniques being used is object oriented design.

D. Object Oriented Design

Object oriented (OO) design is a software design method that uses interacting objects to encapsulate attributes, states, and operations. Each object maintains their own local condition and provides actions based on that condition. Hence, the state of the object is only known by the object itself and cannot be directly accessed from external objects. Sommerville [4] defines an object as:

> *"An object is an entity that has a state and a defined set of operations that operates on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) that request these services when some computation is required."*

The object class is then merely a template for creating objects. Accordingly, to achieve this encapsulation of data, classes and the relationships between those classes are designed to define objects and object attributes [4]. Thus, objects can be dynamically created from these classes. This design process allows for easy modification due to the independency of the objects. Modifying one object should not have an effect on any other object. Objects are often associated with real-world entities; therefore, improving understandability and maintainability of the system design.

E. Specifying Intelligent Agents using AUML and the comparison with the Descartes Specification Language

Agent UML, an extended version of UML, which was developed primarily to capture the features of multi-agent systems, will be used to design the intelligent agents involved, along with their interactions. Syntax, semantics, and interaction protocol documents of AUML were analyzed effectively for designing an intelligent software agent system. Subburaj and Urban [11] described an approach to map the specifications written in extended Descartes into a design representation in AUML. The specifications written in the extended Descartes specification language for specifying intelligent software agents were represented in a high-level design. The attributes, behaviors, and relationships that describe the intelligent agent characteristics were transformed into their corresponding design representation in AUML. A direct mapping procedure ensures correct representation of design elements from the extended Descartes specifications.

IV. DESCARTES-REAL-TIME OBJECT ORIENTED (RTOO)

A. Overview of Descartes-RT

This section represents a brief overview of the real-time extensions made to Descartes in 1992 by Sung [4]. Six concepts were added to Descartes to specify real-time systems. Those concepts include: process concept, parallelism, message passing, traceability, time constraints, and validation tools [4].

The process concept entails state changes within a process. A process in Descartes-RT is described by a specification module, while a process state in Descartes-RT is described by match nodes called state variables in an analysis tree [4]. The state changes are then achieved by modifying the values of the state variables. This concept can specify a sequence of updating cycles in a real-time system; hence, describing the on-going behavior of the state changes.

The next real-time extension to Descartes is a primitive called "parallel." Parallelism is an inevitable concept of real-time systems. This primitive will convey concurrent execution of multiple processes. These processes are self-contained to avoid unnecessary information sharing.

Message passing between processes is an important concept in real-time systems; thus, a mechanism to fulfill this concept is the third real-time extension to Descartes. In Descartes-RT, process communication is asymmetrical. This type of communication means the calling process must know the name of the called process; however, a called process does not know the name of the calling process [4, 8]. A message in the called process can be communicated to the calling process by setting up a "communication path." This communication path can be set up in Descartes-RT by matching a name tag of the called module followed by a period and the name of the variable in the called module (refer to Figure 2). To synchronize these processes, Descartes-RT uses timing constraints. These constraints can delay the next execution cycle of a called process until the calling process is ready to receive the value of a variable [4]. Figure 2 illustrates message passing between parallel processes specified in Descartes-RT.

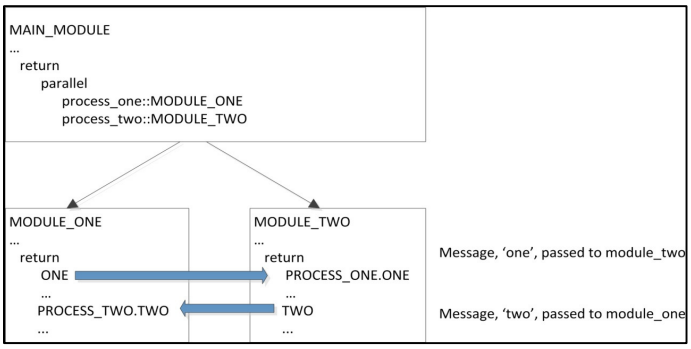

Figure 2: Message passing between parallel processes

The fourth extension in Descartes-RT is the traceability concept. Traceability refers to the ability to link the product requirements to the product design. Descartes-RT expresses traceability by two constructs: requirement and design. The requirement construct allows for stakeholders to trace back to the natural language document from a particular specification. The design construct allows for stakeholders to trace a part of the design from a particular specification [4]. Both constructs are optionally used in Descartes-RT.

Descartes-RT can specify maximum, minimum, and durational timing constraints. To express maximum timing constraints, a timer can be used to denote the maximum time span during which the event should take place [4]. Minimum, durational, and relative timing constraints can be specified in a similar manner. Descartes-RT also provides a validation procedure; however, this feature of Descartes-RT was not used in this research.

B. Overview of Descartes-OO

This section represents a brief overview of the object oriented extensions made to Descartes in 1994 by Wu [2]. Five primary concepts were added to Descartes to specify object oriented systems. Those concepts include: building classes, declaring objects, defining services, access control, and identifying hierarchies [2].

Building classes is an important concept in object oriented design. This concept was incorporated in Descartes-OO through class modules. A class module describes the object attributes and services that influence those attributes. A class module is initiated by the keyword "class" followed by the module title, which uniquely names the class [2, 3]. For example:

class (CLASS_NAME)_CLASS

After this title, an analysis tree is used to specify the class name, object attributes and services. A synthesis tree ('return' root node) contains the set of rules for each service. The class building extension also supports a generic mechanism. Generic classes are template classes that can represent collections of objects [2, 3]. The keyword "with" is used to denote a generic class instantiation relationship. For example:

class (CUSTOMER_LINE)_CLASS with QUEUE

In this case, QUEUE is the generic class that serves as a template for the CUSTOMER_LINE class. To ensure the organization of many classes, the concept of class categories was added to Descartes-OO. Class categories simply organize classes into meaningful pieces. There are four types of class category visibilities in Descartes-OO: exported, private, imported, and global

The next extension in Descartes-OO is declaring objects. Classes are templates for objects. In Descartes-OO, variables serve as unique names for objects. Two primitive modules were added with Descartes-OO to support object identities [2].

(OBJ_A)_IDENTICAL_COPY_(OBJ_B)
(OBJ_A)_IDENTICAL_EQUAL_(OBJ_B)

The primitive module ()_IDENTICAL_COPY_() copies OBJ_A to OBJ_B. The primitive module

()_IDENTICAL_EQUAL_() returns a Boolean value of true if the objects are equal, otherwise returns a Boolean value of false. Descartes-OO uses analysis and synthesis trees to specify object attributes, operations on the object, and object states [2].

Descartes-OO also supports the concept of defining services. Analysis trees specify service behavior. However, modules in synthesis trees define the implementation behavior of services. If these services become complex, they can be decomposed into sub-services using modules (similar to modules in original Descartes) [2]. The communication between these objects is attained by messaging. The sender and receiver of the message are denoted in Descartes-OO as the following:

sender
LIBRARY::ADD_(A_BOOK)
...
***BOOK.NEW**_(BOOK_REC)*

receiver
class (BOOK)_CLASS
...
NEW_BOOK_OP
***NEW**_(A_BOOK_REC)*

Wu uses this example in the Descartes-OO thesis work [2]. Descartes-OO also defines service types. The five kinds of services are modifier, selector, iterator, constructor, and destructor. A modifier alters the state of an object, while a selector accesses the state of an object. An iterator allows all parts of an object to be accessed. A constructor creates an object, while a destructor destroys an object. These service types can be specified through commenting above each service specification using "//" notation [2, 3].

Descartes-OO supports the concept of polymorphism by using several selection rules. These protocols resolve which precise service is being referenced. First, Descartes-OO bases the selection on the class name. Second, Descartes-OO bases the selection on the use of scope. If the service is not referenced by the class module that it is declared in, the closest class module will be referenced. Third, Descartes-OO bases the selection of a service on the argument signature (similar to the Java programming language). Descartes-OO also supports access control similar to the C++ and Java programming languages using the keywords private, public, and protected.

Lastly, Descartes-OO supports identifying hierarchies. This concept is crucial to object oriented design. In object oriented design, hierarchies essentially identify the ordering of abstractions. Descartes-OO supports two types of hierarchies: inheritance and aggregation. Inheritance is shown in Descartes-OO by an "access_modifier." This access modifier illustrates the accessibility of inherited members. Aggregation hierarchies are shown in Descartes-OO by connecting the object attribute with a class name [2, 3]. The object oriented extensions of Descartes were used significantly in this research effort.

C. The Home Heating System Problem (RTOO Approach)

The Home Heating System problem is a classic problem used to exemplify a real-time system domain. The problem has been adapted from several authors [9, 10] and was originally used to only demonstrate real-time and reactive system behavior; however, the problem was used here to demonstrate an object oriented system with real-time behavior. This problem was chosen due to the inevitable real-time actions found in the problem and its frequent use in previous real-time system design research. The informal system requirements specification document used in this study is shown in Table 1. The variables were adapted for use in this study.

"The controller of an oil hot water home heating system regulates in-flow of heat, by turning the furnace on and off, and monitors the status of combustion and fuel flow of the furnace system, provided the master switch is set to HEAT position. The controller activates the furnace whenever the home temperature t falls below t_r - 2 degrees, where t_r is the desired temperature set by the user. The activation procedure is as follows: 1. the controller signals the motor to be activated. 2. the controller monitors the motor speed and once the speed is adequate it signals the ignition and oil valve to be activated. 3. the controller monitors the water temperature and once the temperature is reached a predefined value it signals the circulation valve to be opened. The heater water than starts to circulate through the house. 4. a fuel flow indicator and an optional combustion sensor signal the controller if abnormalities occur. In this case the controller signals the system to be shut off. 5. once the home temperature reaches t_r + 2 degrees, the controller deactivates the furnace by first closing the oil valve and then, after 5 seconds, stopping the motor. Further the system is subject to the following constraints: 1. minimum time for furnace restart after prior operation is 5 minutes. 2. furnace turn-off shall be indicated within 5 seconds of master switch shut off or fuel flow shut off."

Table 1: Requirements of the Home Heating System [9]

D. UML and the Development of the Home Heating System Problem

The home heating system problem was analyzed and redeveloped to fit an object oriented structure. This redesign allowed for all timing constraints to be reconsidered while in an object oriented environment. The primary tool for this redevelopment effort was UML. Figure 3 shows the UML static class diagram for the home heating system.

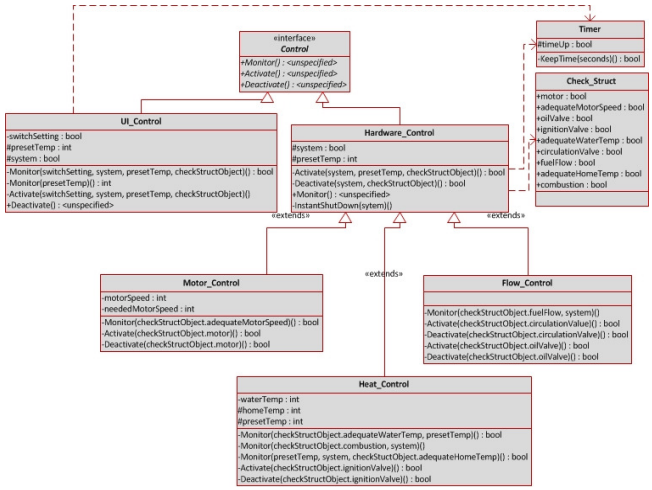

Figure 3: Static Class Diagram of HHS

During the analysis and design process of the Home Heating System, both static and dynamic UML diagrams were used. The design choice follows a high level of abstraction approach to object oriented design. The system uses an interface to fulfill an implementation-independent level, while the abstraction is refined throughout the hierarchy to create implementation-dependent levels. This design not only takes advantage of abstraction, but also heavily uses overloaded methods as used in the Java programming language. Every class (or object) will either monitor hardware, activate hardware or objects, or deactivate hardware or objects. Thus, the interface provides abstract methods to fulfill these services. Each class can overload the method implementation by using different argument signatures. This technique allows for both centralization and encapsulation throughout the system.

The system hierarchy is dependent on a "Timer" object. This timer class will keep time for every process in the system; hence, enforcing the timing constraints of the real-time system. The system also is dependent on a class called "Check_Struct." This class was made to act like a struct would act in the C programming language. The “Check_Struct” class essentially holds the states of the physical elements in the system. A class named “Hardware_Control” uses the attributes of the “Check_Struct” class to appropriately use services from the other control classes. This technique also provides centralization to the system. The design was chosen overall to illustrate a completely object oriented real-time system. A partial Descartes specification for the home heating system can be found in the Appendix A.

E. The Elevator Problem (RTOO Approach)

The Elevator problem is another classic specification problem used to exemplify a real-time system domain. This problem has also been adapted by many different authors for use in real-time system behavior research [13]. In this research, the problem was analyzed in its original context, and then redesigned in an object-oriented context.

This problem was not only chosen due to the inevitable real-time behavior found in the system, but due to its capability to utilize other features within the Descartes-RT and Descartes-OO extensions. The informal system requirements document used in this problem is shown in Table 2 [13].

“Rider (Actor)	**Elevator (System)**
	The elevator shall monitor rider requests from each floor.
The rider requests an elevator from the floor.	After a rider request from a floor, the elevator shall move to that floor.
	After arrival at a requested floor, the elevator shall open its doors.
	The elevator shall monitor rider entry.
The rider enters the elevator.	After a rider enters the elevator, the elevator shall close its doors.
	The elevator shall monitor rider destination floor requests.
The rider requests a destination floor .	The elevator shall move the elevator to the rider’s destination floor.
	After arrival at a requested floor, the elevator shall open its doors.
	The elevator shall monitor rider exit.”
The rider exits the elevator.	

Table 2: Requirements of the Elevator System [13]

F. UML and Design of the Elevator System Problem

This problem followed the same design process as the Home Heating System problem; however, several design choices were made differently in order to better fit the elevator problem. In the Elevator problem, the design still follows a high level of abstraction and takes advantage of overloading methods; yet, the system does not use an overarching "struct-like" class for control access. Instead, some protected and public attributes are used while the encapsulation of each object and class is still maintained. The Elevator system also does not use a Timer object as the Home Heating system did, but takes advantage of the timer feature in the Descartes-RT extension. The design of the elevator system

will also take advantage of other Descartes-RT primitives that were not used in the previous problem. This portion of the research and redesign relied on the comparison of static and dynamic UML diagrams to the Descartes specification language as in the previous problem. Figure 2 shows the UML static class diagram for the elevator problem.

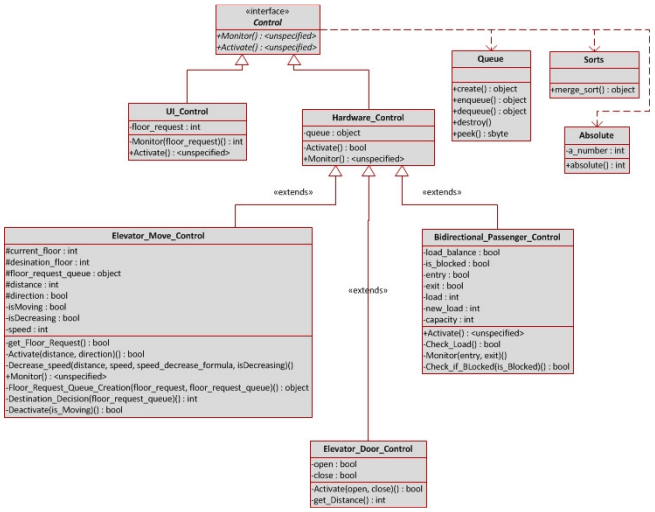

Figure 2. Static Class Diagram of Elevator Problem

This system has a Control interface that contains an abstract monitor class and an abstract activate class. This interface is used similar to the interface used in the Home Heating System. From the control interface, both a user interface control class and a hardware control class are extended. In the user interface control class, the floor requests for the elevator system are constantly monitored by an overloaded monitor method that is inherited from the control interface. The hardware control class is the centralized control of all hardware based classes throughout the structure. The use of the hardware control class is similar to the use of the hardware control class in the Home Heating System problem. Inherited from the hardware control class is the move control class, door control class, and the passenger control class. Each object will control only data and actuators encapsulated in each associated class. Each classes' operations will be called by the hardware control class.

A partial Descartes specification for the elevator problem can be found in Appendix A.

G. Extensions to Descartes

After the system was designed to be object oriented, a Descartes specification using both real-time and object oriented extensions was written. This combination of extensions in Descartes has never been tested prior to this research effort; thus, some trial and error was required.

For the most part, Descartes can be used to specify the Home Heating System and the Elevator Problem well; however, two extensions were proposed to better specify the system. The first extension is the keyword "abstract", which represents a class or method that is not implemented or instantiated. This keyword is similar to the keyword "abstract" that is used in the Java programming language. The concept of specifying abstraction is needed in Descartes to better refine the object oriented design of systems. In the Home Heating System and the Elevator problem, this keyword is used to illustrate abstract methods passed through the hierarchy.

The second extension is the keyword "interface", which represents a generic template class that can only contain method signatures and constant variables. This keyword is similar to the keyword "interface" used in the Java programming language. The concept of specifying interfaces is needed if a high abstraction approach is used in developing the system. The object oriented extension already included a generic inheritance clause using the keyword "with"; in spite of this aspect, the clause only illustrates that a template class is being used, not what the template class actually specifies. Thus, the keyword interface can be used to specify what is inside of a template class. In the Home Heating System problem and Elevator problem, this keyword was used for just that.

V. Security Aspects of Real Time Object Oriented Systems

Software systems, in recent years, face a number of security issues. Securing software systems can be an afterthought and often neglected due to the lack of proper modeling notations and practices in software engineering. The modeling of the actual system with its functional attributes is given much importance over the security issues. One other problem that is commonly found in software development is the improper training and awareness of software engineers about the security aspects and concerns of a software system. The security concerns must be identified and addressed at the beginning of the software life cycle to prevent major problems and high cost. Security aspects of software systems should not be addressed separately, but have to be developed in accordance with the functional requirements of the software system. In order to ensure secure software systems, security requirements should be addressed during the requirements elicitation and design stages rather that in implementation stages.

Software security requirements in real time object oriented systems have an important role when considering secure software systems. Security requirements will vary among different software systems and need to be identified in accordance with the security goals of the system. Security requirements must also include assumptions to predict the security attacks that might occur in the system after development. A predefined set of security requirements identified from security goals were considered [14, 15]. These requirements were as follows:

- authentication requirements;
- authorization requirements;

- integrity requirements;
- intrusion detection requirements; and
- non-repudiation requirements.

Additional security goals and requirements can be added as future work.

Software systems in general are modeled using a number of methods. Developing new methods for eliciting and modeling security requirements along with the functional requirements is a challenging task. One of the most popular standards used for modeling are the UML modeling notations. Research efforts focus around UML modeling notations to weave security concepts due to the fact that these modeling notations have been in practice and are easy to be interpreted by the developers. In this research effort the Descartes specification language has been extended to specify real time object-oriented systems. Also, the Descartes specifications are compared to the UML design for validation of the system design. In this research effort, UML modeling notations were used to incorporate security requirements. Use cases, one of the most popular methods to elicit and analyze the functional requirements, were used for specifying security requirements.

The research methodology defines a framework to help the designers focus on the security aspects of the software system using extensions to UML modeling notations with use case diagrams.

Steps to follow:

- identify functional requirements and goals for the system to be developed;
- identify use cases from the functional requirements;
- identify security goals for the system;
- define security requirements for achieving the security goals;
- redefine the functional requirements by mapping the security requirements with the functional requirements;
- identify use cases for the new set of requirements along with the security requirements; and
- use the Use Case Security Requirements Diagram to draw the use case diagram for the functional and security requirements of the system.

The methodology uses extended UML Use case diagrams to specify security requirements. The UML 2.0 profile was used for the extensions. The methodology commenced with identifying security goals, security requirements, and the new modeling notations to specify security requirements along with functional requirements. Security goals were identified and were mapped to functional requirements. This approach also addressed the challenging issue of unifying security requirements along with the application requirements. The future directions will include the registering of the proposed UML profiles, using Eclipse to facilitate the use of the defined profile in the diagrams. Also, the future work will analyze how the proposed extensions will be used with UML activity and sequence diagrams. Additional security goals can be addressed as future extensions. Incorporating OCL (Object Constraint Language) to the proposed methodology can also be performed in the future.

VI. Validation of The Proposed Methodology

In this paper, the validation of the extended Descartes specification for the object oriented real time systems has been done using two methods 1) assessment of the Descartes extensions to capture the properties of the real time object oriented system and 2) evaluation based on the guidelines proposed by the IEEE 830-1998 standard for good software requirement specification.

A. Real-Time Object Oriented Properties Satisfaction

The methodology was study of the Descartes specification language, including the proficient writing of Descartes specifications for a variety of systems. These system specifications were then compared to UML specifications to discover the strengths and weakness of both languages. Once the study of the Descartes specification language and UML was complete, comprehensive understanding of real-time systems, object oriented systems, and the combination of the two was undertaken. This comprehensive understanding was primarily built through widespread reading and test design of real-time object oriented systems. After both specification and system design were entirely understood, the larger problem of the home heating system and the elevator problems were taken into account. This specific problem was studied in its entirety. Then the problems were redesigned in an object oriented matter based upon the original requirements document of the problem. This redesign allowed for all timing constraints to be reconsidered while in an object oriented environment rather than the structure design environment it was originally intended for. This new design of the system was first illustrated using static and dynamic UML diagrams. After these diagrams were finalized, the writing of the Descartes specification began. The Descartes language required the specification to be much more precise, which uncovered flaws in the original UML design. This experience alone provided a much needed assessment of the effectiveness of formal specification languages. When the Descartes specification was complete, additional extensions were proposed to improve the language definition for real-time object oriented systems. These proposed extensions were validated using the home heating and elevator example problems described in the paper.

B. IEEE 830-1998 standard

The IEEE 830-1998 standard lists the following characteristics for good software requirement specification [12]. The software requirement specification should be: correct; unambiguous; complete; consistent; ranked for importance and/or stability; verifiable; modifiable; and traceable. The extended Descartes specification for the real time objects

oriented system was identified to be satisfying all of the IEEE 830-1998 criteria's.

VII. Summary and Results

Software development has become one of the most demanding industries in the world. End-users are expecting better performance and faster deployments throughout software development. Whether that area is business, entertainment, travel or somewhere in-between, the demand for better software is increasing. With these high demands, real-time constraints are important in software systems. Much of technology has some sort of real-time embedded software system that must meet demands. These systems are being developed with high-level, object oriented techniques in order to take advantage of reusability, portability, and maintainability. If software developers are not efficient in the development cycle, these demands will be difficult to meet. To ensure this efficiency, the specification and design phases of the software development life cycle must be effective. One way to guarantee the effectiveness of a software specification is to use a formal specification language, such as Descartes.

This research project extended Descartes to specify real-time object oriented systems. The extensions covered the needed object oriented concepts and were useful in the Home Heating System and the Elevator problem specification. This extensions and assessment provides software developers with an improved formal method of specifying real-time object oriented systems. Some syntactical issues between the two Descartes extensions were noticed; however, the issue did not appear in both the HHS specification and the Elevator specification. In Descartes-RT, '::' is used for creating process name-tags, while in Descartes-OO '::' is used for showing where a service is declared. Since the conflict was not found in both the HHS problem and the Elevator problem, this conflict will have to be resolved in future work with another RTOO case.

Descartes was found to increase the precision of system development as compared to UML. This enhanced precision was found when converting UML to Descartes. Descartes revealed the flaws, therefore, fulfilling the languages intended purpose. Overall, the objectives were completed; however, future work is needed for verification of Descartes' satisfaction and extensions for RTOO systems.

One full RTOO specification was written and evaluated in Descartes. The future research for this project includes:

- specify more RTOO systems in Descartes to validate proposed extensions and language satisfaction;
- use different object oriented approaches for the Home Heating System and the Elevator Problem in order to fully utilize the features of Descartes;
- use data from other RTOO specifications to propose additionally needed extensions or approve the satisfaction of the language; and
- research the executable portion of Descartes and use to execute RTOO specifications.

This research will be validated by the use of more RTOO specification and design cases.

Acknowledgement

This material is partially based upon work supported by the National Science Foundation under Grant No. CNS-1005212. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Urban, J. E., *A Specification Language and Its Processor*, Ph.D. Dissertation, University of Louisiana at Lafayette, 1977.

[2] Wu, Y., *A Methodology for Deriving a Booch Object Oriented Design from Extensions to The Descartes Specification Language*, Masters Thesis, Arizona State University, Dec. 1994.

[3] Pichai, R. V. and Urban, J. E., "A Technique for Validating Booch Object-Oriented Designs from Extensions to the Descartes Specification Language," *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, 1996, pp. 40-47.

[4] Sung, K-Y., *Real-Time Extensions to the Descartes Software Specification Language*, Ph.D. Dissertation, Arizona State University, Dec. 1992.

[5] Sommerville, I. and Kotonya G., *Requirements Engineering: Processes and Techniques*. Chichester: John, 1998, pp. 1-135.

[6] *Tutorials Point: UML*, Tutorials Point, 2012. [Online]. Available: http://www.tutorialspoint.com/uml/uml_interaction_diagram.htm. [Accessed 2012].

[7] Sommerville, I., *Software Engineering*. London: Pearson, 2004 Seventh Edition.

[8] Nielsen K. and Shumate K., *Designing Large Real-Time Systems with Ada*, McGraw-Hill Book CO., 1988.

[9] Toetenel, H. and Katwijik, J.V., "Stepwise Development of Model-Oriented Real-Time Specifications From Action/Event Models," *Lecture Notes in Computer Science*, Vol 571 Springer-Verlag, 1991, pp. 547-570.

[10] Problem set for the Fourth International Workshop on Specification and Design, *Proceedings of the Fourth International Workshop on Specification and Design*, 1987. nr.2: Heating System.

[11] Subburaj, V. H. and Urban J. E., "Intelligent Agent Software Development Using AUML and the Descartes Specification Language," *Proceedings of the 2nd IEEE International Workshop on Object / component/service-oriented Real-time Networked Ultra-dependable Systems (WORNUS 2011)*, Newport Beach, California, March 28, 2011, pp. 297-305.

[12] ANSI/IEEE, *IEEE std. 830-1998 Recommended Practice for Software Requirements Specification*, (ANSI/IEEE), 1998.

[13] Robinson, W., and Elofson, N., "Goal Directed Analysis with Use Cases," *Journal of Object Technology*, vol 3, no. 5, pp. 125-142, 2004.

[14] Haley, C. B., Laney, R. C., Moffett, J.D., Nuseibeh, B., "Security Requirements Engineering: A Framework for Representation and Analysis," *IEEE Transactions on Software Engineering*, vol 34, no. 1, pp. 133-153, 2008.

[15] Hadavi, M. A., Hamishagi, V. S., and Sangchi, H. M., "Security Requirements Engineering; State of the Art and Research Challenges," *Proceedings of the International Multi Conference of Engineers and Computer Scientists, IMECS*, Hong Kong, vol. I, pp. 19–21, 2008.

Appendix A

The Home Heating System Descartes Specification (Partial Specification)

The following partial specification of the Home Heating System illustrates both real-time and object oriented features of the system. The title of the module shows that this section of the specification is for the UI_CONTROL class in the home heating system. The attributes of the class are specified under the private and protected subnodes. The operations of the UI_CONTOL class are also specified in the Descartes specification class module.

```
class (UI_CONTROL)_CLASS with CONTROL
  UI_CONTROL
    user_interface
      CLASS_NAME
  private:
    switch_setting
      boolean_type+
        true
          'physical system switch is on'
        false
          'physical system switch is off'
  protected:
    system
      boolean_type+
        true
          'previous system state is on'
        false
          'previous system state is off'
    preset_temp
      integer_type
        'physical temperature set by home owner on user interface'
  private:
    operations*
      selection+
        monitor
          'monitor'
            preset_temp
        monitor
          'monitor'
            switch_setting
            system
            preset_temp
            check_struct_object
        activate
          'activate'
            switch_setting
            system
            preset_temp
  abstract:
    operation
      deactivate
```

```
return
    USER_INTERFACE
      OPERATIONS*
        SELECTION+
          MONITOR
            MONITOR_(PRESET_TEMP)
          MONITOR
            MONITOR_(SWITCH_SETTING)_(SYSTEM)
            _(PRESET_TEMP)_(CHECK_STRUCT_OBJECT)
          ACTIVATE
            ACTIVATE_(SWITCH_SETTING)_(SYSTEM)_(PRESET_TEMP)
```

The Elevator System Descartes Specification (Partial Specification)

The following partial specification of the Elevator System illustrates both real-time and object oriented features of the system. The title of the module shows that this section of the specification is for the ELEVATOR_MOVE_CONTROL class in the elevator system. The attributes of the class are specified under the private and global subnodes. The operations of the ELEVATOR_MOVE_CONTROL class are also specified in the Descartes specification class module.

```
class (ELEVATOR_MOVE_CONTROL)_CLASS:private HARDWARE_CONTROL
  ELEVATOR_MOVE_CONTROL
    elevator_movement
      CLASS_NAME
  global:
    current_floor
      integer_type
        'read from floor level sensor'
    destination_floor
      integer_type
        'returned by get_floor_request method'
    distance
      integer_type
        ABSOLUTE.ABSOLUTE((CURRENT_FLOOR)_MINUS_(DESTINATION_FLOOR))
    direction
      boolean_type+
        true
          ((CURRENT_FLOOR)_MINUS_(DESTINATION_FLOOR))_LESS_THAN_('0')
        false
          ((CURRENT_FLOOR)_MINUS_(DESTINATION_FLOOR))_GREATER_THAN_('0')

  private:
    is_moving
      boolean_type+
        true
          'elevator is moving'
        false
          'elevator is not moving'
    index_c
      integer_type
        'index of current floor in queue array'
    index_d
      integer_type
        'index of destination floor in queue array'
    speed
      integer_type
```

```
      'read from sensor'
  speed_descrease_formula
    integer_type
      'formula of distance needed for elevator to begin decreasing speed'
  is_decreasing
    boolean_type+
      true
        'elevator is slowing down'
      false
        'elevator is speeding up'
  floor_request_queue
    queue_object
      QUEUE.CREATE_()
        'floor request added to queue, then sorted by closest floor request'
private:
  operations*
    selection+
      get_floor_request
        'get value of floor_request variable from UI_CONTROL class'
      floor_reqeust_queue_creation
        FLOOR_REQUEST
        FLOOR_REQUEST_QUEUE
      activate
        IS_MOVING
        DISTANCE
        DIRECTION
      decrease_speed
        DISTANCE
        SPEED
        SPEED_DECREASE_FORMULA
        IS_DECREASING
abstract:
  operation
    monitor

return
  ELEVATOR_MOVEMENT
    OPERATIONS*
      SELECTION+
        GET_FLOOR_REQUEST
          UI_CONTROL.FLOOR_REQUEST
        FLOOR_REQUEST_QUEUE_CREATION
          FLOOR_REQUEST_QUEUE_CREATION_(FLOOR_REQUEST)_(FLOOR_REQUEST_QUEUE)
        ACTIVATE
          ACTIVATE_(IS_MOVING)_(DISTANCE)_(DIRECTION)
        DECREASE_SPEED
          DECREASE_SPEED_(DISTANCE)_(SPEED)_(SPEED_DECREASE_FORMULA)_(IS_DECREASING)
```