EXTENSIONS TO THE DESCARTES SPECIFICATION
LANGUAGE FOR THE DEVELOPMENT OF REAL-TIME OBJECT
ORIENTED SYSTEMS

Benjamin D. Horne
Department of Computer
Science
Union University
Jackson, TN USA
ben.horne@my.uu.edu

Vinitha Hannah Subburaj
Department of Computer
Science
Texas Tech University
Lubbock, TX USA
vinitha.subburaj@ttu.edu

Joseph E. Urban
Department of Industrial
Engineering
Texas Tech University
Lubbock, TX USA
joseph.urban@ttu.edu

ABSTRACT
Traditionally, real-time system software focuses on low-level programming
techniques to increase timeliness and operate within constraints; however, more
recently higher-level, object-oriented methodologies are being used to create real-
time system software. This increase in object-oriented design for real-time systems
is due to the improved use of active objects and concurrency in object-oriented
languages. Real-time aspects have not only improved in many programming languages,
but these real-time constraints are becoming more essential in general object-oriented
software development. Consequently, the specification of real-time object-oriented
systems is becoming vastly important in today's software development processes.
One of the many tools used to specify software is an executable specification
language called Descartes. Descartes relates output data to input data as a function of the
input data through a tree structure notation called "Hoare trees." This useful
specification structure has been extended and supported for several types of systems
since the language's creation in 1977. In

peculiar, Descartes has been extended for the specification of real-time systems and
for object-oriented designed systems. In spite of this, the combination of the two
extensions for specification of today's widespread real-time object oriented
(RTOO) systems has not been tested. Thus, the main objective of this research effort
was to verify that the combination of the two
Descartes extensions either satisfied the
needed specification for RTOO systems or
that the language needed to be additionally
extended. To gain further verification of
these united Descartes extensions, the
Unified Modeling Language (UML) has
been used for comparison in aspects of
design and effectiveness.

KEYWORDS
specification, real-time object-oriented,
Descartes.

1 INTRODUCTION
Software requirements specification is a
description of the behavior of a system to
communicate the requirements to anyone
from end-users to developers. This
communication sounds simple; then
again, end-users, managers, and
developers can interpret this document
differently. In fact, the gap between the
natural language requirements and the
software implementation can be
enormous. With any natural language
comes ambiguity. Yet, software
developers cannot afford ambiguity.
Different interpretations of a
requirements document will increase
cost, decrease quality, and even compromise the safety of the software. Thus, the gap between specification and implementation needs to be closed. Expressing requirements with a consistent formal specification method can complete this closure. The Descartes specification language is one such formal method.

The Descartes specification language and other specification methods have been in use for several decades; in spite of this, specifying today’s software can be even more challenging due to new constraints and enhanced design methodologies. Consumers are constantly demanding elegant and timely solutions to software problems. If these needs are not met; then life in the software market will be short. Thus, the software development process must also become more elegant and timely. One such challenging area of software development is found in real-time object oriented systems. Traditionally, these categories of software are mutually exclusive; however, more recently, real-time objectives in object oriented software are becoming vastly important. If developers are going to create quality software in today’s market, the specification of real-time object oriented software is vital.

1.1 Objectives

This research was conducted in accordance with the following objectives:

- Study of Descartes and Descartes extensions: The Descartes specification language and extensions were studied in detail.
- Study of real-time object oriented systems and design: The design and nature of real-time systems was studied in detail. Focus was especially on designing a real-time system with object oriented methodologies.
- Combine the Descartes object oriented extension with the Descartes real-time extension to create a specification language for a real-time object oriented system: The home heating system problem was used to test the combination of specification extensions and to illustrate proposed extensions to the Descartes language.
- Verify Descartes' satisfaction of needed specification or propose additional extensions: Additional extensions to Descartes were proposed to satisfy the object oriented specification of the real-time home heating system problem. The system was studied in detail to ensure specification was correct and efficient.
- Compare the Descartes specification to a Unified Modeling Language (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 problem to validate the design and compare the Descartes specification language to a commonly used modeling technique.
2 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 specifications 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 the 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, "\(^\ast\)". 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 [1].

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 approach 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.
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.

3 RELATED WORK

3.1 Software Requirements and 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 software requirements and specification (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 can be accomplished through a formal specification language that can be checked for ambiguity and consistency.

3.2 Unified Modeling Language (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 an organization 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].
3.3 Real-Time Systems

Real-time systems are systems in which 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 predictable 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 constraints. One of these higher-level techniques being used is object oriented design.

3.4 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 [7] 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 [7]. 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.

3.5 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.

4 DESCARTES-REAL-TIME OBJECT ORIENTED (RTOO)

4.1 Overview of Descartes-RT

This section represents a brief overview of the real-time extensions made to Descartes in 1992 by Keum-Young 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 ongoing 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 was inspired by work in [4].

![Diagram of message passing between processes](image)

**Figure 2: Message passing between parallel processes**

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 project.

### 4.2 Overview of Descartes-OO

This section represents a brief overview of the object oriented extensions made to Descartes in 1994 by Yin-Yu 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 an object oriented method. 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 are 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 divided into sub-services using modules (similar to modules in the 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 a "//" 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 an object oriented method. 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.

### 4.3 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, in this research effort, the problem has been used to demonstrate an object oriented system with real-time behavior. This problem was chosen for this research effort 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>
<thead>
<tr>
<th>Table 1: Requirements of the Home Heating System [9]</th>
</tr>
</thead>
</table>

### 4.4 UML in the Home Heating System Problem

In this research effort, the home heating system problem was analyzed and redesigned 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 redesign was the Unified Modeling Language (UML). 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.

4.5 Proposed 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 was never 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 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 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 designing 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, this keyword was used for just that.

With the proposed extensions and original Descartes-OO extensions, the aspects of object oriented specification have been covered. These aspects include concepts such as polymorphism, inheritance, classes, interfacing, abstraction, and encapsulation. Thus, the gain from this extension includes the ability to specify a completely object oriented system. These proposed extensions plus the new combination of previous extensions for Descartes allows
specification for real-time object oriented systems. These gains provide the Descartes language with a new domain to specify software in.

With the combined extensions and proposed extensions, Descartes gains little in complexity. The original goal of Descartes was to be a simple to understand; yet, effective specification language. This goal was taken into consideration when proposing extensions.

5 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 in every area of 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 becoming vastly important in software systems. Most 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 not be met. To ensure this efficiency, the specification and design phases of the software development life cycle must be effective. One way to improve the effectiveness of a software specification is to use a formal specification language, such as Descartes.

This research effort extended Descartes to specify real-time object oriented systems. The proposed extensions covered the needed object oriented concepts and were proven useful in the Home Heating System specification. This extension 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 the Home Heating System 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 the Home Heating System problem, this conflict will be resolved in future work with other RTOO case studies.

During this research effort, 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 some flaws that UML did not; therefore, fulfilling the languages intended purpose. Overall, the objectives set for this research effort 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 verify proposed extensions and language satisfaction.
- use different object oriented approaches for the Home Heating System in order to fully utilize the features of Descartes-RTOO.
- 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.

6 REFERENCES