Duration Calculus in COOZ: A Formal Specification Language for Real-Time Systems

Xiaodong Yuan
Information Technology Division
Nanjing Automation Research Institute
Nanjing, P.R.China, 210003
Email: yuanxd@hotmail.com

Ying Feng
Computer Science Department,
Indiana University,
Bloomington, IN 47405 USA
Email: yfeng@cs.indiana.edu

Abstract

This paper introduces the idea of continuous time into COOZ, an object-oriented extension of Z notation, through establishing absolute clock and object clock. Then we use Duration Calculus to specify the real-time and history constraints of objects. By these means, COOZ is adapted to formal specifications of real-time and concurrent systems. A case study of automatic fire alarm system and the comparison with related work are also given. We make in-depth discusses in the paper on problems about type conversion, integrability and isolated points of real-time state, etc.

Keyword

Real-time, concurrent, formal specification, Duration Calculus, COOZ

1. Introduction

COOZ (Complete Object-Oriented Z) is an object-oriented extension of Z[1]. This paper presents how to add time and temporal order descriptions into COOZ so that it can be used for the formal specifications of real-time and concurrent systems.

The real-time constraint is an important aspect for software performance. For general systems, the software specification does not include descriptions of time limit. In safety-critical real-time systems, however, the time limit becomes one of the most important requirements. If the time limits were not satisfied, it might cause serious unexpected consequences. Thus, a formal specification language for a real-time system should provide complete and convenient facilities for descriptions of time limit. Z just lacks this description power.

Duration Calculus (DC in short) is a formal design method for real-time systems based on interval logic[2]. DC models continuous time with real number and uses Boolean-valued functions with time as their parameter to describe the system states and events. DC has been used in some applications. It has powerful description abilities for real-time systems, and provides a set of inference rules. Therefore, we include DC in COOZ to make COOZ suitable for describing real-time and concurrent systems, and build up the time model of objects. In this way, we enhance the expressiveness and enlarge the application fields of COOZ. From the example given in this paper, we can see that it is convenient for describing the real-time requirements of systems.

2. DC in COOZ

2.1 Describing Continuous Time

Presently there are some researches on extending Z with time mechanisms. Most of them use integers to describe discrete time, such as the description of execution time of operations in document [3] and the communication protocols in document [4]. However, in real world time is continuous. Obviously, the formal specification that specifies only discrete time cannot satisfy the requirements of real-time systems. It has been a research field of formal methods to include description mechanisms for continuous time.

To describe continuous time, first there must be a data type corresponding to it. Naturally, the real type is suitable for representing continuous time. We can use clock variables of real type that change their values with time, to set up the system's absolute and relative clocks. The value of a clock variable with real type corresponds to a moment. The duration of time can be represented by interval of the real numbers.

The concept of time is independent to other aspects of a system. Clock variables are different from general variables, as they are autonomous variables. Once created, a clock variable changes only with time and can only be referenced (not modified) in the specification. To specify the special characteristics of clock variables, we declare them as type of Time, not type of real $\mathbb{R}$. For example, the declaration of clock variable t is given below:

$\mathrm{Time} == \mathbb{R}$

$| \; t : \mathrm{Time}$

We use Time as a predefined type of COOZ. Although it's equivalent to $\mathbb{R}$ in syntax, it's very different from $\mathbb{R}$ in semantics. That is, the variables of type Time change with time automatically and cannot be modified through operations. If we use second (the international standard unit of time) as the unit of clock variables and t is a variable of type Time, the following examples show a set of time-related formulas and their meanings:

$t = 3.5$ // the moment at 3.5 second.

$t \in [4,10.2]$ // the time duration between second 4 and second 10.2.

$\exists x : \mathbb{R} \bullet x \in [0,5] \wedge t = x \wedge a = b$ // There exists a moment between second 0 and second 5 such that a is equal to b.

2.2 Absolute Clock and Object Clock

Absolute clock is a system-wide unique and unambiguous time stipulation. For example, we use Beijing time as the absolute clock in China. Through declaring a clock variable of type Time, we can set up an absolute clock in a software system. It is enough to assign an initial value to the clock variable and fix the concrete time length corresponding to time unit 1. For example, if we set the clock variable to the initial value 0 and use one second as the time unit, any time in the system can be uniquely decided by the clock variable. The time before the execution of the system can be specified by negative value of Time.

It's still inconvenient to have only absolute time. In real-time systems To know the actual time corresponding to a moment of the system is usually not sufficient, we need to also know the time interval between two moments. If we use absolute time only, every moment concerned in the real-time requirement must be saved. On the other hand, the initial value of the clock variable does not affect the value of the time interval, which is only decided by the concrete length of unit time. Therefore, for the sake of convenience, we define an object clock when each object is created. This clock uses the moment when the object is created as time 0, and its length of unit time is the same as that of the absolute clock. The object clock would disappear when the object disappears. The *Real-Time and History Constraint*s in object schema of COOZ uses object clock as its time representation mechanism.

For simplicity, neither absolute clocks nor object clocks need to be defined in specifications explicitly. The clocks are automatically created.

2.3 Time-Related Object States

In almost all real-time systems there exist many time-related states, for example, the environment temperature in automatic fire alarm systems and the line states in computer communication systems. To conveniently build the description methods and inference rules of real-time systems, DC only considers those states with Boolean values. In COOZ, Boolean type $\mathbb{B}$ is a predefined type and the Boolean-valued real-time variables are functions from Time to $\mathbb{B}$:

$\mathbb{B} ::= \textbf{false} \mid \textbf{true}$

$State : \mathrm{Time} \rightarrow \mathbb{B}$

Except for Boolean-valued real-time state, COOZ also supports states of other types, such as multi-valued state represented by a finite set, continuous state represented by a real type, etc. By converting other types of real-time states to Boolean states, we can use DC to describe and process them. For example, in an automatic fire alarm system, variable *temperature* is a continuous state represented by the real type. However, among all the temperatures, we only care about several of them such as the warning temperature t_1 and the extinguishing temperature t_2. The condition *temperature*$>t_1$ and *temperature*$>t_2$ represent the converted Boolean states. In fact, when using COOZ to describe real-time systems, we do not explicitly give the definitions of most real-time Boolean states with the type `Time` $\rightarrow$ `B`, because all the Boolean-valued mathematical expressions with time parameter can be handled as real-time Boolean states.

To conveniently specify the real-time performance of a system, we provide two predefined real-time Boolean states for each object operation: *Methodname*.**executing** and *Methodname*.**waiting**. The former specifies that the operation is in execution, and the latter specifies that the operation is in the waiting state, e.g., waiting for results from other operations or data inputs. From the definitions of these two real-time Boolean states, we have the following inference:

Inference 2.1: Given method *m*, then *m*.**waiting** $\Rightarrow$ *m*.**executing**.

Ordinary software specifications usually fail to describe the execution time of methods and the waiting time during the execution. However, to safety-critical real-time systems, such descriptions are indispensable. For example, in communication systems processing rate of characters should not be lower than transfer rate, and in automatic fire alarm systems extinguishers should be able to be opened within a certain time period. The two predefined states in COOZ, *Methodname*.**executing** and *Methodname*.**waiting**, provide great convenience for direct descriptions of execution time and waiting time of operations.

2.4 Integrability of Boolean States

Duration Calculus extends interval logic to formulate real-time requirements of real-time systems. The main concern is to describe the total time when a Boolean state is true in an interval. If we let **false** of Boolean type correspond to the real number 0, and **true** correspond to the real number 1, all Boolean state functions with type `Time` $\rightarrow$ `B` can be changed into functions of `Time` $\rightarrow$ `R`. Then the total time when the real-time Boolean state P is true in interval $[b,e]$ can be formulated by the integration $\int_b^e f(P(t))dt$ (suppose the function is integrable), where f is a function of `B` $\rightarrow$ `R`, and f(**false**)=0, f(**true**)=1. We abbreviate it as $\lceil P$. $\lceil P$ is a real-valued function defined on time intervals. The interval type `Intv` and $\lceil P$ are given definition 2.1 and 2.2 as follows.

Definition 2.1: `Intv` $\triangleq \{ [b,e] \mid b,e \in$ `R` and $b \leq e \}$.

Definition 2.2: $\lceil P \triangleq \lambda\, [b,e]$: `Intv` $\bullet \int_b^e f(P(t))dt$, where $f = \{$ (**false**,0), (**true**,1) $\}$

Now we discuss the integrability of Boolean states. If Boolean state P is *finitely varied* (i.e., P has at most a finite number of discontinuous points in any bounded interval $[b,e]$), and $\lceil P$ is valid, then we say P is integrable. It can be proved that integrability of Boolean states is closed under first-order predicate calculus.

If Boolean state function P is neither left nor right continuous at time t, we call t an isolated point of P. In DC, isolated points can be ignored because they do not affect the value of $\lceil P$ in any interval. But here we have some different opinions on this. We think the isolated points of Boolean states should not be ignored, since the effects with or without isolated points may be different under the exist quantifiers. In fact, if we consider it in a practical system rather than from the viewpoint of pure mathematics, the problem is obvious. In a practical system, a state whose time duration is zero, i.e., an isolated state, is difficult to simulate unless we regard it as if not exsisting. Once a state does exist, its time duration would not be zero. Such being the case, why do isolated points exist? It is usually because of the limitation of measure precision. However precise it is in practice, it is impossible to reach the ideal minimum in mathematics. The time duration of a state that is less than the minimal measurable unit is regarded as zero. Considering the practical engineering, we should distinguish various situations when analyzing isolated points. In simple states that correspond to composite states consisting of more than one states through logic corporation, isolated points usually are the states lasting for an extremely short time. In composite states however, there are two situations: one is that the state does not take place indeed and the other is that the state lasts for an extremely short time. We need careful analysis to distinguish these two situations. When safety-critical system is concerned, some states must be absolutely forbidden, even if they exist as isolated points.

2.5 Real-time Requirement Specification of Integrable States

DC uses a set of Boolean states to model real-time systems. The real-time properties are formulated mainly by integral $\int P$ of states on time intervals. In the previous part of the text, we have established the absolute clock and the object clocks of a system, and defined real-time Boolean states and their integral $\int P$ on time intervals. By these means we can apply DC completely to the description of real-time properties of objects. All symbols and reasoning rules of DC are applicable to *Real-Time and History Constraint*s of COOZ. More details about DC can be found in document [2]. We will not repeat them one by one here, but will just give the semantics of chop modality $^\frown$ together with the definitions of interval length symbol ℓ and abbreviation symbol $\lceil\ \rceil$, which are not in first order logic of Z.

Definition 2.3: Given ϕ,ψ : Intv $\to$ B,

$[\phi^\frown\psi]^I([b,e]) = \exists m\in[b,e] \bullet [\phi]^I([b,m]) \wedge [\psi]^I([m,e])$.

$^\frown$ is a binary modality for "chopping" an interval into two consecutive subintervals. An interval satisfies $\phi^\frown\psi$ iff the interval can be chopped into two adjacent subintervals such that the first subinterval satisfies ϕ and the second one satisfies ψ. From theorems in interval logic we know that $^\frown$ satisfies the combination rule[5].

Definition 2.4: ℓ : Intv $\to$ R $\triangleq \lambda\ [b,e]$: Intv $\bullet$ *e-b*.

ℓ is a special symbol for an interval function denoting the interval length.

Definition 2.5: $\lceil\rceil \triangleq \ell = 0$.

$$\lceil S\rceil \triangleq \int S = \ell \wedge \ell>0.$$

The formula $\lceil\rceil$ holds iff the interval length is 0, and the formula $\lceil S\rceil$ holds in an interval $[b,e]$ iff $b<e$ and S is **true** everywhere in $[b,e]$ except at a finite number of time points.

The following are some examples of using the above notations to express real-time requirements (Suppose the time unit is one second).

1. Formula ϕ holds in one subinterval: **true** $^\frown\phi^\frown$ **true**

2. State *a* would last for more than 20 seconds if it took place: $\lceil\neg a\rceil^\frown\lceil a\rceil^\frown\lceil\neg a\rceil \Rightarrow \ell>20$

3. If *temperature≥40* lasted for more than 60 seconds, state *warning_bell* would be true:

$\lceil temperature\geq 40\rceil \wedge \ell>60 \Rightarrow \int warning\_bell>0$

Through two predefined states of operation *methodname*.**executing** and *Methodname*.**waiting** together with the above notations we can formulate constraints such as execution time of operations conveniently. See the following examples:

1. Execution time of method *m* is no more than 10 seconds: $\lceil m.\mathbf{executing}\rceil \Rightarrow \ell \leq 10$

2. If *temperature>50* lasts for more than 30 seconds, method *warn* will be executed:

$\lceil temperature\leq 50\rceil\ ^\frown\ (\lceil temperature>50\rceil \wedge \ell>30) \Rightarrow \ell>30\ ^\frown\lceil warn.\mathbf{executing}\rceil^\frown \mathbf{true}$

For more details about using DC to describe real-time constraints, please refer to the case study in Section 3.

2.6 Use DC to Describe History Constraints

History constraints are behavior constraints of objects that are related to the execution sequence of object methods. They describe history invariance satisfied by a class of objects. Object-Z uses temporal logic to specify history constraints. Z++ uses either linear temporal logic or real-time logic to formulate history predicates. As Duration Calculus, together with real-time state *Methodname*.**executing** and *Methodname*.**waiting,** is more powerful and understandable than temporal logic and real-time logic, we use DC to formulate history constraints.

The contents about history constraints of execution sequence of object methods have not been explicitly classified till now. We try to classify history constraints into four main categories as follows:

1. Preconditions of single method, which are described with only object state variables and must be satisfied when the method is to be executed.

2. Execution time of methods, describing when and how many times a method is executed.

3. Separated execution of methods: two executions of one method must be separated by the execution of another method.

4. Adjacent execution of methods: a method executed some time before (or after) one method must contain another method.

In the following we will show how to express the above four categories of history constraints using DC.

1. Suppose the precondition of method *m* is *P*,

it can be expressed as: $\lceil \neg m.\mathbf{executing} \rceil \frown \lceil m.\mathbf{executing} \rceil \Rightarrow \mathbf{true} \frown \lceil P \rceil \frown \lceil \neg m.\mathbf{executing} \rceil \frown \lceil m.\mathbf{executing} \rceil$.

2. For simplicity, we use **exec**(m) to represent the execution time of method m on an interval. It can be converted to formulas of DC by the following equivalence rules:

$\mathbf{exec}(m) = 0 \Leftrightarrow \lceil \neg m.\mathbf{executing} \rceil$

$\mathbf{exec}(m) > 0 \Leftrightarrow \neg \lceil \neg m.\mathbf{executing} \rceil$

$\mathbf{exec}(m) > k+1 \Leftrightarrow \mathbf{exec}(m) > k \frown \lceil \neg m.\mathbf{executing} \rceil \frown \lceil m.\mathbf{executing} \rceil \frown \mathbf{true}$

$\mathbf{exec}(m) = k+1 \Leftrightarrow \mathbf{exec}(m) > k \wedge \neg (\mathbf{exec}(m) > k+1)$

$\mathbf{exec}(m) < k \Leftrightarrow \neg (\mathbf{exec}(m) = k) \wedge \neg (\mathbf{exec}(m) > k)$

Method m executes n times can be expressed as: $\mathbf{exec}(m) = n$.

3. Suppose any two executions of method m should be separated by method p, it can be expressed as:

$$\lceil m.\mathbf{executing} \rceil \frown \lceil \neg m.\mathbf{executing} \rceil \frown \lceil m.\mathbf{executing} \rceil \Rightarrow \mathbf{true} \frown \lceil \neg m.\mathbf{executing} \wedge p.\mathbf{executing} \rceil \frown \mathbf{true}.$$

4. Suppose n methods following method m should contain method p, we can first define state **nonexecuting** which stands for that no method is executing:

$$\mathbf{nonexecuting} \triangleq \forall m{:}\mathit{Methodname} \bullet \lceil \neg m.\mathbf{executing} \rceil$$

Then it can be represented as:

$$\forall\ m_1, m_2, \ldots, m_n{:}\ \mathit{Methodname} \bullet \mathbf{true} \frown \lceil m.\mathbf{executing} \rceil \frown \mathbf{nonexecuting} \frown \lceil m_1.\mathbf{executing} \rceil \frown \mathbf{nonexecuting} \frown \lceil m_2.\mathbf{executing} \rceil \frown \ldots\ldots \frown \mathbf{nonexecuting} \frown \lceil m_n.\mathbf{executing} \rceil \Rightarrow p \in \{ m_1, m_2, \ldots, m_n \}$$

For convenience, we use $p \in \mathbf{after}(m,n)$ as its abbreviation. Similarly, we can state that n methods before method m must contain method p, and its abbreviation is $p \in \mathbf{before}(m,n)$.

Consequently, DC has the capability of representing all categories of history constraints. A special function of history constraints is to hide some methods of a supertype in a subtype by limiting them to execute 0 time during the objects' life-cycle:

$\mathbf{exec}(m) = 0$ or $\lceil \neg m.\mathbf{executing} \rceil$

3. A Case Study

In this section we give an example of an automatic fire alarm system. The automatic fire alarm system has two parts. One is the temperature measuring devices and the automatic extinguishers in the building floors. The other is the central control system in safe place to converge data from measuring devices and send orders to them. We use object type *FireWarning* and *SystemControl* to describe these two parts in the following partial specification written in COOZ (See below). The emphasis is on how to use DC to express real-time and concurrent constraints.

The method schema below whose name has prefix '*' is an active schema. For example, schema **measure_temperature* specifies the objects' measure temperature of environment continuously and update state variable *temperature* automatically. Notation '::' before method name indicates that only the method name is given here and the whole definition will be written outside the object schema.

There are many real-time and history constraints in object type *FireWarning*. The meaning of the first constraint is that the alarm would be fired automatically when the cumulative total time during which the temperature is more than the warning temperature, reaches 10 seconds within one minute. The second constraint means that the extinguisher would be switched on automatically if the cumulative total time during which the temperature is more than the extinguishing temperature, reaches 3 seconds within half a minute. The other constraints are easy to understand and will not be elaborated here.

Object *FireWarning*

$warn\_t, extinguish\_t : \mathbb{R}$

$warn\_t \leq extinguish\_t$

$temperature : \mathbb{R}$
$warn\_s, extinguish\_s : \mathbb{B}$

Init

$t?, t1?, t2? : \mathbb{R}$

$temperature = t?$
$warn\_t = t1?$
$extinguish\_t = t2?$
$\neg\, warn\_s$
$\neg\, extinguish\_s$

**measure_temperature*

$\Delta\, temperature$
$t? : \mathbb{R}$

$temperature' = t?$

*::*auto_warn*
*::*auto_extinguish*
::query
::warn
::extinguish
::stopwarn
::stopextinguish

$\lceil (temperature \geq warn\_t) > 10 \wedge \ell \leq 60$
$\quad \Rightarrow \lceil auto\_warn.\mathbf{executing} > 0$
$\lceil (temperature \geq extinguish\_t) > 3 \wedge \ell \leq 30$
$\quad \Rightarrow \lceil auto\_extinguish.\mathbf{executing} > 0$
$\lceil temperature < warn\_t \rceil \wedge \ell > 60$
$\quad \Rightarrow \mathbf{true} \frown \lceil \neg\, auto\_warn.\mathbf{executing} \rceil$
$\lceil temperature < warn\_t \rceil \wedge \ell > 60*20$
$\quad \Rightarrow \mathbf{true} \frown \lceil \neg\, auto\_extinguish.\mathbf{executing} \rceil$
$\lceil warn.\mathbf{executing} \rceil \frown \lceil \neg stopwarn.\mathbf{executing} \rceil$
$\quad \Rightarrow \lceil warn\_s \rceil$
$\lceil extinguish.\mathbf{executing} \rceil \frown$
$\quad \lceil \neg stopextinguish.\mathbf{executing} \rceil$
$\quad\quad \Rightarrow \lceil extinguish\_s \rceil$

The following is the specification of the object type *SystemControl*. The central control system is linked to not only fire alarm measure devices but also other devices such as TV watching. The real-time constraints in partial specification of object type *SystemControl* require that TV watching devices should be switched on in 10 seconds after the fire alarm is activated. Another constraint is that some orders (e.g., starting or closing some automatic extinguishers) should be sent within 5 minutes if the alarm has not been released.

Object *SystemControl*

$State \triangleq Normal \mid Warning \mid Extinguishing$

$antifire : \mathbf{seq}\ FireWarning$
$s : State$

Init

$antifire = \langle\rangle$
$s = Normal$

::AddFireWarning
::DeleteFireWarning
*::*FireQuery*
::SendOrder
::TVmonitor

$(\lceil s \neq Normal \rceil \frown \mathbf{true}) \wedge \ell > 10$
$\quad \Rightarrow \ell \leq 10 \frown \lceil TVmonitor.\mathbf{executing} \rceil \frown \mathbf{true}$
$\lceil s = Normal \rceil \frown (\lceil s \neq Normal \rceil \wedge \ell > 60*5)$
$\quad \Rightarrow \lceil s = Normal \rceil \frown \ell \leq 60*5 \frown$
$\quad\quad \lceil SendOrder.\mathbf{executing} \rceil \frown \mathbf{true}$

……

4 Related work

Z is a model-based formal specification language. With active schema, Z has some capabilities of formulating concurrent systems by itself. Document [6] puts forward an approach of using merely Z notations to specify and verify concurrent systems, but it can only specify the first category of history constraints given in section 2.6. Document [7] adds real-time requirements specified by discrete time in Z specification and gives their refinement rules, but they cannot be extended to continuous time. Document [8] extends Z notation in some aspects: (1) the units of quantities are given in specifying variables, so the abstract data may have concrete physical meanings such as time, space, (2) the continuous functions from *TIME* to the type of the quantity are used to model time-varying quantities.

Among current object-oriented extensions of Z notation, Object-Z[9] uses temporal logic to specify history constraints. Z++[10] uses either linear temporal logic or real-time logic to formulate history predicate. As a twin version of

Object-Z, oo-Z[11] gives history-based semantics of objects, then it can represent the time sequence of object' s behaviors and use some self-defining functions and first order logic to describe history predicates. The other extensions such as OOZE[12], ZEST[13], MooZ[14], etc., do not provide history description mechanisms, so it' s difficult to specify concurrent systems in them. The current extension languages generally only support discrete time but not continuous time, which limits the capability of language when used to specify real-time systems. Duration Calculus adopted in COOZ supports continuous time, and we use DC formulas to specify real-time and history constraints. The expressiveness of DC is more powerful than temporal logic when specifying real-time systems. This point can be enhanced by [2] in which it is proved that the basic temporal logic formulas can all be replaced by DC formulas.

5. Conclusion

This paper describes the notions of continuous time of COOZ. Object clocks supporting continuous time are established, which set up the basis for formal specifications of real-time systems. Consequently, we adopt Duration Calculus to specify real-time and history constraints. Compared to temporal logic, DC has more powerful capabilities in describing real-time systems, and can also express various history predicates. So COOZ is suitable for the descriptions of real-time and concurrent systems, and can be especially applied to safety-critical hybrid systems.

References

[1] Yuan Xiaodong, Hu Deqiang, Xu Hao, Li Yong and Zheng Guoliang. COOZ: complete object-oriented extension to Z. Software Journal. ACM SIGSOFT Software Engineering Notes, 1998,23(5).

[2] Zhou Chaochen, Hoare C A R and Ravn A P. A calculus of duration. *Information Processing Letters.* 40(5):269–276, 1991.

[3] Fidge C. Proof obligations for real-time refinement. In: *6th Refinement Workshop* (1994):279-305

[4] Yuan Xiaodong, Xu Dianxiang and Zheng Guoliang. The formal description and implementation of YMODEM protocol. The ninth proceeding of network and data communication sponsored by Chinese Computer Society, Xian, 1996, 10.

[5] Dutertre B. Complete proof systems for first order interval temporal logic. In Tenth Annual IEEE Symp. on logic in Computer Science. IEEE Press, 1995:36-43.

[6] Evans A S. Specifying & verifying concurrent systems using Z. In: Naftalin M, Denvir T and Bertran M(eds) *FME' 94: Industrial benefit of formal methods Vol 873 of Lecture Notes in Computer Science*: 366-400, Formal Method Europe, Springer-Verlag(1994).

[7] Fidge C. Adding real time to formal program development. In: Naftalin M, Denvir T and Bertran M(eds) *FME' 94*: *Industrial benefit of formal methods Vol 873 of Lecture Notes in Computer Science*:618-638, Formal Method Europe, Springer-Verlag(1994).

[8] Mahony B P and Hayes I J. A case-study in timed refinement: a mine pump. *IEEE Transactions on Software Engineering*, Vol. 18, No. 9, 1992:817-825

[9] Duke D, Duke R, King P, Rose G A and Smith G. Object-Z: an object-oriented extension to Z. A technical report: 91-1, Software Verification Research Centre, The University of Queensland.

[10] Lano K. Z++, an object-oriented extension to Z. In: Nicholls J(ed.): *Z User Workshop*, Oxford 1990, Springer-Verlag Workshops in Computer, 151-172, 1991.

[11] Duke D J. *Object-Oriented Formal Specification*. A thesis submitted to The Department of Computer Science University of Queensland. 1992.

[12] Alencar A J and Goguen J A. OOZE: an object oriented Z environment. In: *ECOOP'91*:181-199

[13] Cusack E and Zadeh H B. ZEST. In: Stepney S, Barden R and Cooper D(eds), *Object Orientation in Z.* Workshops in Computing, Springer-Verlag. 1992.

[14] Meira S R L and Cavalcanti A L C. Modular object-oriented Z specifications. In: *Z User Workshop*, 1990, Workshops in Computing, Springer-Verlag, 1991:173-192.