2 BACKGROUND: CONTEXT And CoNCEPTS

"You must pin down the butterfly of time."

[Michael Jackson, Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices, Addison-Wesley, 1995, pp. 78]

2.1 Introduction

In this chapter the research space and the coordinates of the thesis' topic are defined using a classification based on 'domains and 'sub-domains' of exploration and the major aspects of the larger framework in which we have undertaken our research are overviewed. By analysing the distinctive features of real-time (or, in our vocabulary, time-constrained) systems, object-oriented modelling, and formality in software development the larger contour of our work is drawn. The main characteristics of real-time systems are analysed with the dual intent of establishing the context of the present research and of identifying specific challenges of capturing temporal properties of systems. The impact of the object-oriented paradigm on the software development process is also dscussed and the value of graphical notations is emphasised. Some of the most significant aspects of employing formal notations in various phases of the software life-cycle are examined, and arguments pro and contra this employment are reviewed. As part of the examination of formality and formalisms, the newer category of light formal methods, which circumscribes our approach, is briefly discussed. Thus, Chapter 2 sets the scene for a closer look (in Chapter 3, "Background: Notations") at the two specification languages used in our approach, one formal (Z) and the other graphical, semi-formal, and object-oriented (UML). The same scene is then used in Chapter 4, "Related Work," to identify existing research approaches that are situated in the vicinity of our topic's location.

2.2 Research Space and Topic Location

The research space that encompasses the topic of the present thesis can be described by considering three domains of exploration (Table 2.I and Fig. 2.1). For simplification, in the case of the first domain, which characterises the formality of a modelling approach, only its 'formal' sub-domain is considered. The second domain describes the methodological paradigm used for software development, which can be either object-oriented or non object-oriented. The third domain classifies approaches as having or not having RT modelling capabilities. Within each of the three domains of exploration a number of sub-domains (areas) of interest can be further delimited according to various criteria. We have been interested in specifying the Z, Z++, and UML "dimensions" of a given approach, hence the classification in Table 2.I, which distinguishes areas denoted A, B, and C in the first domain, 1, 2, and 3 in the second, and • and + in the third.

Table 2.I Classification of Research Approaches Based on Domains of Exploration

Formality Domain	Methodology Domain	Real-Time Domain
Area A [non-Z]: Formalism involved, but not Z-centred	Area 1 [non-OO]: Not an object-oriented methodology	Area · [non-RT]: No RT modelling capabilities
Area B [Z but non-OOZ]: Formalism involved, Z-centred, but not OOZ	Area 2 [OO, non-UML]: An OO methodology, but UML not involved	Area + [RT]: RT modelling capabilities provided
Area C [OOZ]: Formalism involved, and an OO version of Z used	Area 3 [UML]: An OO methodology that uses UML	N/A

The 18 possible combinations of areas from the three domains provide a classification scheme in which a given approach has a class between A1• and C3+ (with the exception of classes C1• and C1+, which do not make sense because an OOZ notation can be used only in conjunction with an OO modelling strategy). This sharp delimitation of domains involves a certain simplification, since things are almost never purely "black or white" (formal or categorically non-formal, for instance), but it nevertheless serves well our localisation purpose. Based on the classification presented in Table 2.I, a graphical representation of domains can be drawn, as presented in Fig. 2.1.

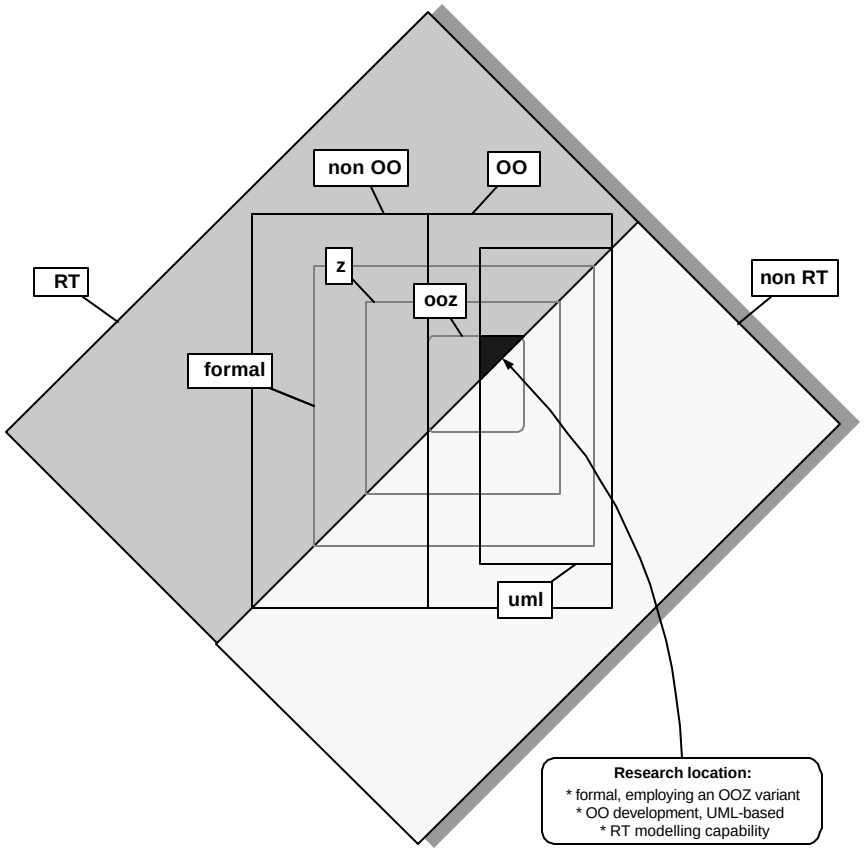

Fig. 2.1 Domains of Research Space and Topic Location

The figure indicates that our class C3+ approach is placed at the intersection of the three major domains described above, and also enjoys the special characteristics provided by its

"OOZ-centred" and "UML-based" dimensions. Before moving to investigate further the three domains defining the research space and discuss the details of the Z and UML notations used to demarcate areas in these domains, several comments are necessary. Firstly, to keep the figure simple, the areas in Fig. 2.1 have not been textually labelled as indicated in Table 2.I, but the identification of specific classes, e.g., B2•, should be straightforward. Secondly, the graphical representation of the abstract topology presented in Fig. 2.1 is in not intended to reflect the proportionality of existing approaches (for instance, there is no intention on our part to claim a 50-50 distribution between OO and non-OO approaches, as the figure might suggest). Thirdly, the classification presented in Table 2.I and its depiction shown in Fig. 2.1 will be used again in Chapter 4, where a survey of related approaches is presented.

2.3 On Specifying Real-Time Systems

2.3.1 Characteristics of Real-Time Systems

In today's fast evolving world of computing, real-time systems are taking an increasingly important role and are extending their reign over a growing number of application domains. Real-time systems prove to be useful in many areas of human activity: numerous commercial, industrial, medical, and military products that must pay careful attention to the precious resource which is the time are used on daily basis. As pointed out by Stankovic, "a real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced" ([Stankovic96b], pp. 751). In a similar way, Everett and Honiden indicate that "a real-time system must respond to externally generated stimuli within a finite, specifiable time delay" [Everett95, pp.13]. Severe consequences may result if timing as well as logical correctness properties are not satisfied. Based on the severity of consequences, the real-time systems can be classified as hard real-time systems, where the failure of meeting the deadline can result in an important loss (including loss of human life, injury, and/or major equipment damage), or soft real-time

systems, in which the deadline can be occasionally missed, but the utility of the result decreases after the deadline [Burns97, Kopetz97]. Some authors take into consideration an intermediate category, firm real-time systems, which in essence can be described as having a shorter soft deadline and a longer hard deadline [Douglass98] (Fig. 2.2 presents a summary characterisation of the three types of real-time systems based on a generic utility-time function.) Hard real-time systems encompass aircraft controllers, process control systems, factory robots, traffic lights controllers, and medical devices such as heart pacemakers, while examples of soft real-time systems include automatic banking machines, ticket reservation systems, general-purpose communication systems, and embedded commercial products such as television sets and videocassette recorders. An example of firm real-time system is that of a patient ventilator system, in which an occasional late breath in the range of few seconds is tolerated, while a several minute delay is catastrophic [Douglass98].

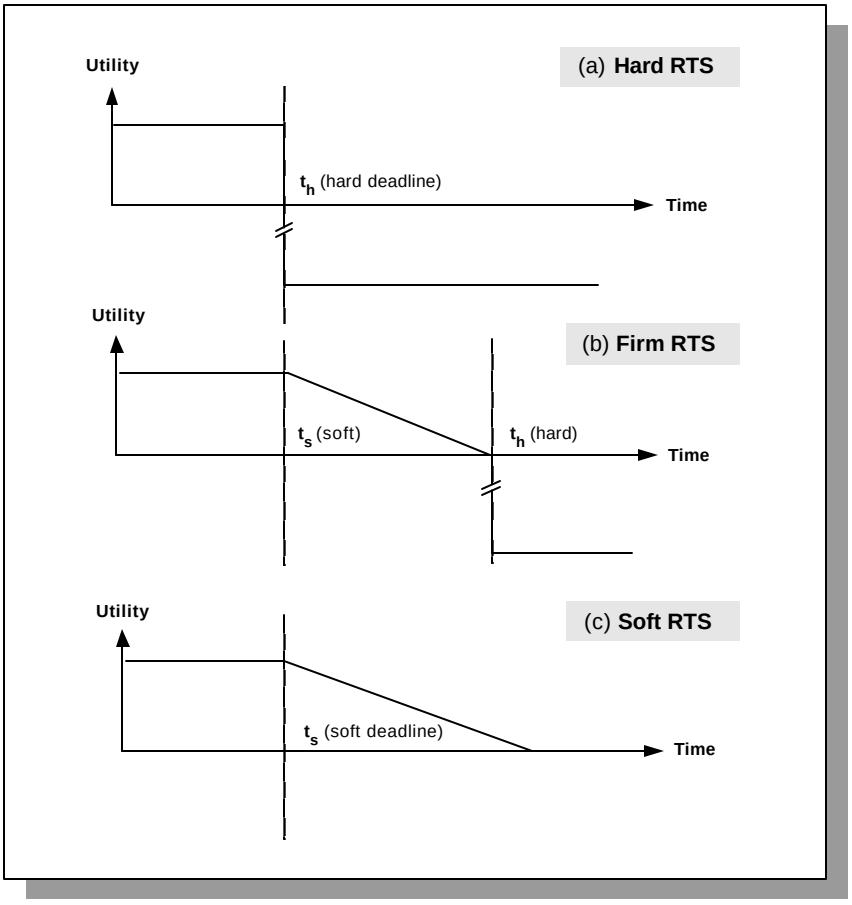

Fig. 2.2 Hard, Firm, and Soft Real-Time Systems

As pointed out by Stankovic, most of the activities of real-time systems have to occur in a timely fashion, but some non time-critical activities also coexist. The former activities are referred to as real-time tasks (or time-critical tasks) while the later can be simply called tasks [Stankovic88]. Timing constraints on tasks can be periodic, if activated every T units of time, aperiodic if activated at unpredictable times, or sporadic, if they aperiodic behaviour is further restrained by a minimum interval of time between activations [Stankovic96a]. The complexity of designing RTS also arises from additional types of constraints and requirements such as resource constraints, concurrence constraints, precedence relationships, placement constraints, communication requirements, criticalness. A real-time system differs from a traditional system (non real-time) in at least the following aspects: deadlines are attached to some or all of the system's tasks, faults in the system –including timing faults– may lead to catastrophic consequences, the system should have the ability to deal with exceptions, the system must be fast, predictable, reliable, and adaptive [Stankovic88]. Lin and Burke show that RTS are very difficult to debug and modify, and –since there are always demands for new functions and configurations– they must be easy to change and reconfigure [Lin92]. Other authors also point out that the design of real-time software is resource-constrained, the software itself is intricate and contains highly complex time critical parts, and the real-time software should be able to detect the occurrence of failures [Natarajan92, Everett95]. Everett and Honiden show that "development of most software focuses on how to handle a normal situation, but real-time, critical-application development also focuses on how to handle the abnormal situation" (Everett95, pp.15). And, unfortunately, as Gibbs points out, "errors in real-time systems ... are devilishly difficult to spot because, like that suspicious sound in your car engine, they only occur only when conditions are just so" [Gibbs94, pp.88]. In short, as noted by Douglass, RTS "must operate under more-severe constraints than 'normal' software systems yet perform reliably for long periods of time" [Douglass99, pp. 57].

In what follows, we give a more detailed account, albeit not exhaustive, of characteristics pertaining to RTS and analyse their implications on the design of a dedicated specification approach. Of course, it is rather difficult to find an example that exhibits all the properties

listed below, and it will be a massive task, if not impossible, to develop a specification method capable of rigorously handling all these properties. In fact, in Subsection 2.3.2 we focus on a reduced number of capabilities we have aimed to include in our modelling approach, but at this point it is useful to have a closer look at the impressive complexity of the RT domain. The starting point of our selection has been the list of requirements for specification languages presented by Narayan and Gajski in [Narayan93] and further analysed by Narayan in [Narayan96]. Their list of requirements is concerned, however, with the more restricted category of embedded systems so we have resorted to additional references in order to describe the larger class of real-time systems.

- Timeliness. The essential characteristic of RTS is that deadlines are imposed to some or all the tasks of the system. Timeliness is part of the definition of a real-time system [Douglass99], such system being required to work under predefined temporal constraints and correctly react to stimuli from its environment "on time" [Selic94]. In specification terms, the modelling notation should incorporate a time metric, as well as facilities for expressing both relative and absolute timing constraints;
- Reliability. One of the most imperative requirements placed on RTS, particularly on hard RTS, is that of reliability. Due to the gravity of the potential damages that can result as a consequence of a real-time system failing to function correctly, additional measures must be taken into consideration. As pointed out by Nancy Leveson, the vast majority of software faults have roots in incorrect specification [Leveson86], therefore the specification languages and techniques employed in the development of RTS must provide adequate support for incorporating reliability measures and for assessing the system's safeness. Essentially, means to deliver specifications that are complete, consistent, comprehensible, and unambiguous are necessary [Burns97]. The use of formal techniques is required, at least for the security and safety-critical parts of the system;
- Intensive dynamics. Due to the typically intensive dynamics of RTS, modelling the states of such systems is an essential requirement for a dedicated specification language. Diagrammatic notations with solid mathematic foundation have been proposed (among

the most notable Petri Nets [Petri62, Reisig85] and Statecharts [Harel87]) and proved to be extremely valuable for specifying the dynamic behaviour of RTS. In particular, finite-state machines have been used successfully in various phases of the software development process [Avnur90, Ding93, Harel96];

- Input/Output. Obviously, due to the continuous interaction with the environment in which a real-time system typically operates, an adequate set of symbols and operators for describing input/output operations should be included in the specification language;
- Exceptions. Real-time systems must react promptly to stimuli from their environment or to internal events that necessitate immediate attention. Some events, external or internal, are more important than others, and taking appropriate measures in response to critical situations is a strict requirement for such systems. Exception and interrupt handling are inherent in the implementation of RTS and it is desirable to have them described at the specification level;
- Concurrency. Even though concurrency is not part of the definition of RTS, many such systems exhibit concurrent behaviour. Moreover, as pointed out in [Shaw92], due to the very nature of RTS it is not sufficient to model only the system, it is also necessary to capture the environment in which it operates. And, this environment is inherently concurrent, with multiple sources of stimuli that influence the behaviour of the system. Consequently, a specification language for RTS should provide appropriate support for expressing concurrency;
- Distribution. As in the case of concurrency, distribution is not necessarily a characteristic of RTS, but it is nevertheless impossible to ignore it, at least in the case of large-scale systems. Capturing the distributed nature of a complex real-time system is becoming a necessary feature in these days when the Internet and the World-Wide Web have obtained the status of common nouns. However, the task of expressing both concurrency and distribution is very demanding [Douglass99];
- Communication and synchronisation. One cannot possibly imagine a useful system in which various software components do not communicate and synchronise, more so in a real-time system that is required to be both concurrent and distributed. Clear description

of communication and synchronisation is necessary and the specification notation must provide constructs and mechanisms to support it;

- Resource allocation. Because many RTS are also distributed, it is desirable that facilities for describing allocation of resources should be included in a specification language that aims at modelling such systems;
- Size. RTS are not only special in their dealing with time but in many cases they are also large and complex, involving numerous processes and threads, as well as a significant number of input/output variables. Size alone is obviously an element that affects the development of a system, but in the case of RTS a complicating factor is that largeness is inherently associated with continuous change, so provisions for extensibility should be built in the design of such systems [Burns97]. Both structured and object-oriented methods provide means of dealing with increasingly more demanding requirements on size; modules, classes, components, and patterns are typical solutions for dealing with large-size software products. Operators for expressing composition and decomposition, as well as mechanisms for modelling hierarchical structures are necessary;
- Non time-constrained activities. Although it would appear that non time-critical activities should not be deemed an issue, it has been shown that incorporating such activities in the development of RTS may prove to be a complicating factor. The most common problem raised by non time-constrained activities is that a worst-case execution time for them (e.g., the answer from a human user) cannot be easily evaluated [Audsley96];
- Computations. Typically, RTS must continuously interact with their environment and provide appropriate response under conditions imposed by the environment. The computation of the system's response can be complex, for instance in the case of process control systems, which involve solving systems of possibly complicated differential equations. Consequently, the implementation of RTS requires the ability of manipulating real, fixed or floating-point numbers [Burns97]. This translates into a requirement for the specification language, which must be able to handle both quantitative and qualitative intricacies of RTS;
- Data modelling. RTS are the most complex type of systems –as put by Alderson et al., they "have proved troublesome to produce, with all the difficulties of the other kinds of

software-based systems together with a number of specific additional problems" [Alderson98, pp. 442]. Among the traditional difficulties, data modelling is a challenging issue if not for all, but for an increasing number of time-constrained systems. In fact, a branch of time-constrained systems is that of real-time database systems (RTDBS), in which both timely response and the ability to manipulate data that has temporal validity are required [Lin94]. In this respect, solutions to unambiguously specify aspects such as relationships between consistency constraints and timing constraints, the validity of external data consistency, abstractions for data, and also data transformations are needed [Sahraoui97];

- Reuse. As pointed out by Mrva, the real-time systems, particularly the embedded systems, appear to be poor candidates for reuse [Mrva97]. This can be explained by the fact that most of the RTS are specialised, typically required to resolve needs of a rather particular nature. Reuse seems hard to achieve with RTS for the simple reason that rarely two applications exhibit more than limited similarity. However, as indicated by Mrva, reuse is not only desirable but also possible within the realm of such systems and the major factors on which the reusability value of a real-time system depends on are the frequency and the utility of reuse, which are related to comprehensibility, habitability (measures how "at home" a potential user feels with the reusable components), and independence of components with respect to their environment [Mrva97]. The object-oriented paradigm offers an avenue of investigation for the designers of RTS, together with the newer pattern-based techniques;
- Animation/execution. The need for animation is advocated by many authors who stress the importance of rapid-prototyping and early client feedback in the development of RTS. Animation of specifications is generally desired, because it can provide a rapid feedback to the designer and facilitate a better understanding of the system's requirements. Although the more ambitious goal of an animation system is to generate a full-scale prototype or even a complete implementation of the system being developed, animation can be used interactively for immediate exploration purposes: the consequences of a specification can be evaluated dynamically, during the composition of

the system's specifications, thus allowing the refining and optimisation of specifications [Utting95].

2.3.2 Focus On Time

Obviously, the real-time domain is very complex and very demanding. The approach we take is to tackle some of its complexity and deal with several of the aspects mentioned above. Since the defining property of a real-time system is timeliness, we decided to focus on expressing temporal properties of the systems at the specification level. Because of this, and for reasons outlined in Subsection 1.4, we use time-constrained systems (TCS) as the preferred term in denoting the systems our approach is focused on, although when needed (primarily, for referencing purposes) the traditional RTS denomination is also used in this dissertation.

Our "selection of emphasis" has also been based on the observation that while timeliness is a characteristic of both hard and soft real-time systems if we speak about TCS (as opposed to RTS) more stringent ("harder") requirements placed on these systems, such as reliability and safety, are gently pushed towards the background. The intention, of course, has not been to ignore such demanding requirements, but to come up with a "more popular," more pragmatic specification approach that would appeal to both software developers and users and would not scare them away by suggesting an emphasis on the more difficult (and less "popular") subclass of complex safety-critical applications. And, while our method can address the modelling of hard RTS (e.g., traffic lights controllers), it is only fair to say that "really hard" RTS –if we may introduce this distinction– such as aircraft autopilot controllers or nuclear process control systems would need supplementary treatment, provided by some additional techniques and tools. On the other hand, while we acknowledge our approach's focus to the "softer side" of RTS (in fact, not necessarily soft, since it could be either "lighter hard", "firm," or indeed "soft"!) we note that the term TCS has an additional advantage: it covers both reactive (or event-driven) systems and time-based (or time-driven) systems

because timeliness is part of both of them. (If it were to speak simply about time-based systems it would have meant that we address only systems whose behaviour is driven by the passage of time or the arrival of time epochs [Douglass99], and this would have been somewhat too restrictive).

Our emphasis on timing properties is illustrated by the fact that the starting point in the design of our approach has been provided by the archetypical classes of temporal constraints identified in [Dasarathy85] and that Real-Time Logic (RTL) [Jahanian86, Jahanian94], which offers very good support for expressing both absolute and relative timing properties, has been included in the proposed integrated specification method (more details are provided in Chapters 5 and 6). In terms of the characteristics of RTS discussed in the preceding subsection, the approach presented in this thesis can be summarily described as follows: it places primary emphasis on timeliness, provides a good modelling coverage of intensive dynamics, input/output, exceptions, non time-constrained activities, computations, data-modelling and reuse, offers a fair support for dealing with concurrency, communication, synchronisation, and size, and does not address distribution, resource allocation, and animation/execution. In what regards reliability, the formal basis is here, with Z++ and RTL its pillars, but the particular specification approach we propose here need be complemented by analysis techniques that have been left outside the scope of the present dissertation.

2.4 Brief Immersion in Object-Orientation

2.4.1 On Objects and Their Modelling Power

Over the years, the structured paradigm proved to be less effective than initially thought. By mid-eighties, the practitioners in the field became aware that it did not live up to earlier expectations, particularly in two major respects: it did not cope well with the increasing size of modern software products and did not support adequately the maintenance of such products [Schach99]. As indicated by Schach, the essential limitation of the structured paradigm is that its approaches for software development are either action-oriented or data-

oriented, but not both. In response to this situation, a new alternative, soon to be known as object-oriented, emerged with remarkable power. Although an important breakthrough in software development, the apparition of the new approach was not spontaneous, but the cumulated result of the work of many scientists and developers [Booch94]. The origins of some concepts that helped shape the new approach can be traced back to as early as the 1960s, most notably to Dahl and Nygaard (the class construct in Simula67), and to Alan Kay, Adele Goldberg and their team at the Xerox Palo Alto Research Center, California (messages and inheritance in Smalltalk) [Page-Jones99]. Other major contributors, according to the same author, include Larry Constantine (coupling and cohesion), Dijkstra (layers of abstraction), Barbara Liskov (abstract data types), David Parnas (information hiding), Jean Ichbiah (packages and genericity in Ada83), Bjarne Stroustrup (C++), Bertrand Meyer (Eiffel), Grady Booch, Ivar Jacobson, and James Rumbaugh (OOA, OOD, and UML). To these, we have to add Peter Chen, whose ERD (Entity-Relationship Diagrams) contribution [Chen76] is a recognised source of inspiration for object-oriented approaches. And, interestingly, if we follow Kouichi Kishida's observations and look carefully we can find precursors to OO even in ancient times (Confucius) as well as in the 19th century (the German philosopher Max Weber) [Kishida96]! In fact, this should not be so surprising, since in his survey of the foundations of the object model, Booch also makes references to ancient philosophy, Greek in his case, as well as to Descartes [Booch94, pp. 36-37].

The newer approach, the object-oriented paradigm, is founded on the concept of object, which can be defined concisely as "a unified software component that incorporates both the data and the actions that operate on that data" [Schach99, pp. 17] or as "a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand" [Rumbaugh91, pp. 21]. More completely, an object is "an entity that: has state; is characterized by the actions that it suffers and that it requires of other objects; is an instance of some (possibly anonymous) class; is denoted by a name; has restricted visibility of and by other objects; may be viewed either by its specification or by its implementation" [Booch86, pp. 215]. The internal structure of an object is described by attributes, and messages can be sent to an object to invoke one of its methods (or operations) –that is, to invoke actions that

generally operate on the internal structure of the object. Typically, we ignore many details of objects, and are concerned mostly with ways of manipulating them through operations. In software, the notion of object covers tangible things (such as book, floor, door, or thermometer), persons (e.g., student, teacher, employee), roles (e.g., dispatcher, supervisor, controller), events (e.g., take-off, interrupt, shutdown) and an infinite variety of other things (e.g., proposals, meetings, poetic ideas, eulogies, referrals, rebuttals, etc.). In OO terminology a class is a template for objects that have similar features, more precisely the objects belonging to the same class have the same structure and the same behaviour (e.g., raymondsAlarmClock is an object of the class AlarmClock). A class can be seen as an abstract data type that supports inheritance.

Three major principles are promoted by the OO paradigm:

- Encapsulation, the defining principle of object-orientation, which signifies putting together in a single unit of both data and operations pertaining to some entity that can be qualified as an object (or, to be more precise, as a class, the "blueprint for creating objects" [Mughal00, pp. 2]). Encapsulation supports abstraction and information hiding, key ingredients for developing high-quality software products;
- Inheritance, the mechanism of creating a new class from existing ones and the provider of the strongest foundation for reuse;
- Polymorphism, essentially an instrument for abstraction and an enhancer of flexibility, with its meaning taken from the Greek equivalent of "having multiple forms," and used in the OO world with the significance "same name for different behaviours."

The major breakthrough brought by the OO approach comes from the fact that the conceptual and physical independence of components reduces the level of complexity of software. Thus, both development and maintenance are simplified [Schach99]. Among the most important benefits of the OO approach we would nominate:

- Greater modelling power, since objects correspond more naturally to real-world entities and as such the problem domain is better described;

- Increased code reusability and extensibility, due to encapsulation and inheritance, which offer strong support for code reuse and product extension;
- Improved control of complexity, mainly through abstraction, information hiding, and localisation –the management of complexity is helped since the emphasis is on interfaces and interactions among independent, collaborating entities (objects).

These benefits, together with a series of other advantages of the OO approach, such as production of software more resilient to change, greater level of confidence in the correctness of software through separation of its state space [Booch94], greater stability of designs over time, more flexible and adaptable development, and easier transition between the development phases [Johnson00], have lead to a proliferation of OO techniques and tools for software construction. Of course, there are less beneficial aspects of OO development that the software professionals are aware of, most significantly longer initial development time, decreased run-time performance, and unavailability of adequate OO DBMS, but overall the newer approach has gained the confidence of the software development world [Johnson00]. And, while there are some isolated opinions that the OO paradigm is only a "hype," possibly less effective than the structured one [Niemann99], and some scientists have even proclaimed its impending demise [Davis98], we share Bertrand Meyer's position that "OO solutions are our best bet" and, in fact, "it's the only game in town" [Meyer99, pp. 144], the newly emerged component-based development actually assuming and making use of the OO technology.

2.4.2 Object-Orientation in the Real-Time Domain

For reasons mentioned in Subsection 1.1.2, the "conqueror objects" have only relatively recently expanded over the real-time domain. However, as the OO technology has matured, the focus of numerous scientists has shifted towards tackling the complexity of real-time applications via the OO avenue. Currently, there is a significant amount of work in this direction, and a number of important methods and methodologies have been proposed, among the most notable ROOM [Selic94, Selic96], TRIO [Bucci94, Ciapessoni99],

Octopus [Awad96], and Comet [Gomaa00]. The considerable attention currently paid by researchers and developers to the application of the OO techniques to the development of RT software is both indicative of the economical importance of RTS and illustrative for the general recognition of the OO paradigm's modelling prowess. And, there is probably no better illustration for the current concerted effort in this direction than the development of powerful dedicated commercial tools such as I-Logix Inc.'s Rhapsody [Rhapsody01] and Rational Software Corporation's Rational Rose Real-Time [RationalRoseRT01]. In addition, the hottest general OO programming language of the moment, Java, has recently enhanced its support for RT applications through the definition of the preliminary version of the Real-Time Specification for Java (RTSJ), expected by E. Douglas Jensen "to become the first real-time programming language to be both commercially and technologically successful" [Bollella00, pp. xxi]. With strong research directions and important programs such as OMG's Real-Time Analysis and Design Initiative [Selic99a], major advances in the development of industrial-use IDEs, and considerable RT support from an OO language such as Java that is used by a large number of programmers, the trend is obvious. We can safely assume that it will continue strongly in the foreseeable future.

2.5 On The Importance of Graphical Notations

It has been mentioned in Chapter 1 that today is almost impossible to create a viable software development tool without an adequate GUI interface. The provision for an easy-to-use, friendly and functionally complete graphical interface is not simply a trend of the moment but a stringent requirement for any development tool intended for practical use. Although it might seem like a futile argumentation, it is nevertheless useful to stress the importance of visual interfaces in such tools. And perhaps there is no better way to emphasise this idea than by paraphrasing David Taylor who, in a recent article, recalls the following prediction he made more than 15 years ago about the OO paradigm: "by the year 2000 no one would talk about objects any more because the technology would be so thoroughly absorbed into the mainstream that no one would think to mention it" [Taylor99, pp. 50]. While we share Meyer's position and question the accuracy of Taylor's affirmation in the

OO context (Meyer considers that several more years are still needed before Taylor's affirmation can be fully supported [Meyer99]), we believe that this is truly the case for graphical interfaces in the context of software tools. Thus, we can state that they are here for quite a while and practically taken for granted, so "nobody would think to mention them." In fact, animation and multimedia capabilities are an important part of our interaction with the computers and they are expected to have an increasingly larger presence in modern professional tools, so perhaps discussing GUI advantages runs the risk of obsoleteness.

But it is not only the graphical user interface we are referring to; the use in our approach of UML, defined as a "visual modelling language" [Quatrani98], corresponds to another reality, that of the need for visual notations in analysis and design. In Chapter 1 the motivations for a combination graphical notation (semi-formal in our case) with a formal language for software specification have been presented and while we do not intend to discuss here visual languages and environments in general, we refer nevertheless to [Green96] for a complete list of cognitive dimensions that can be used to evaluate the benefits of visual notations, including closeness of mapping, abstraction gradient, role-expressiveness, consistency, progressive evaluation, and visibility. Also, for a thorough rebuttal of some common objections to the use of visual representations in the computing process we refer to [Cox93]. However, the task is simpler in our case, since OO methodologies have been traditionally supported by graphical notations, and it is quite hard today to imagine such a methodology without an accompanying set of graphical symbols for classes, relationships, collaboration diagrams, etc.

In our opinion, the use of visual representations, as opposed to simply employing text, is strongly justified by enhanced support for abstraction, better representation of information in terms of structures (components and their relationships), increased expressiveness (richness of information content), simpler syntax, capability for direct manipulation, and increased naturalness (which facilitates communication).

Many scientists have acknowledged the advantages of visual notations in software development by adding a "visual dimension" to their specification approaches, for instance

Buhr's diagrams for the design of Ada applications [Buhr90], Dillon et al.'s Graphical Interval Logic (GIL) aimed at representing the temporal evolution of concurrent systems' properties [Dillon94], Roman et al.'s custom built Pavane visualisations for capturing formally expressed specifications and designs [Roman96], and Taentzer's visual rules for declarative specification of behaviour in OO modelling techniques [Taentzer99]. In the "Z area" an interesting approach is the one taken by Kim and Carrington who, based on Kent's Constraint diagrams [Kent97] and Kent and Gil's Contract Box notation [Kent98] propose 3D visualisations of Z expressions to facilitate the understanding of specifications [Kim99b].

It is also important to note that visual notations are not necessary semi-formal (or informal) because when accompanied by precise semantics they fit in the class of formal notations (this is the case, for example, of Petri Nets and Statecharts, two powerful techniques used for modelling specific aspects of RTS). But even in the case of more general semi-formal notations such as DFD (Data Flow Diagrams), ERD, or UML the expressive power provided by their graphical representation is of considerable help during the development process.

And, to conclude the case for graphical notations, perhaps apparently a minor aspect, but nevertheless solidly backed by its acceptance in practice is Together Soft Corporation's inclusion of colours in the modelling process [TogetherSoft00b]. Colours and other elements of visualisation are, in our opinion, great enhancers of productivity in developing software products.

2.6 Formal Notations in Software Development

2.6.1 Alexander's Definition of a Formal System

A clear and concise definition of a formal system can be found in [Alexander95]. The author uses the following terminology (key terms are highlighted by us using italics):

- A formal system consists of a formal language and a deductive apparatus;
- A formal language has two essential components: an alphabet of symbols and a set of grammar rules;
- The grammar rules are used to construct well-formed formulas;
- A deductive apparatus is a set of axioms (basic truths) plus a set of inference rules (e.g., substitution, simplification, expansion rules);
- The inference rules produce a well-formed formula from other well-formed formulas; the deductive apparatus also provides means to establish whether a well-formed formula is a direct consequence of another;
- To apply a formal system to a problem, the formal system must be given semantics, which in essence provide a mapping between objects in the problem domain and well-formed formulas in the formal language;
- With the semantic mapping established, the formal system can be used to create a formal model of "known characteristics" of the problem domain.

As pointed out by Alexander, software systems requirements describe the desired behaviour of a system within its operational environment. In essence, the execution of a software artefact can be described formally by a precondition I(x) and a post-condition O(x,z), where x is the input of the execution and z is its output. In short, when I(x) is true, the execution of the software artefact generates z, which satisfies O(x,z). The key issue in software development is to find some program P(x) that produces z under the conditions stipulated by the pre-condition I and the post-condition O. This process of determining an appropriate P(x) is complex, and requires successive refinements, each producing a more concrete model of the system (the starting point being a high-level model of the requirements). Each refinement involves two fundamental processes, synthesis and analysis. Alexander points out that in general both synthesis, the creation of a new model of the system, and analysis, the verification of the model with respect to the original model, can be reliable only if formal models are employed. Semi-formal models are unable to predict or verify most of the system's characteristics.

According to the same author, "a software specification is a model of a developing software system" and "formal specification is representing software specification using a formal model" [Alexander95, pp. 30]. The "foundations picture" drawn by Alexander can be extended with a couple of definitions proposed earlier by Jeannette Wing:

- "A formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification" [Wing90, pp. 10];
- Formal specification languages supply the mathematical basis for formal methods, which are "mathematically based techniques for describing system properties" [Wing90, pp. 8].

2.6.2 Classifications and Examples of Formal Methods

The usual way of classifying formal methods is based on the traditional model-oriented versus property-oriented criterion. The distinction between these two categories of methods stems from the way the behaviour of the system is defined, directly or indirectly. A model-oriented method directly describes the behaviour of a system in terms of sequences of states (each state being characterised by a set of instance variables) and operations that can cause state transitions. The property-oriented methods can be further classified as axiomatic or algebraic, depending on their underlying mathematical foundation (first order predicate logic or many sorted algebras). In both cases, property-oriented methods define the behaviour of the system indirectly, via a set of properties usually expressed as axioms that the system must satisfy [Wing90]. In their comprehensive survey of formalisms Liu and Zedan propose a more refined taxonomy by identifying five classes of formal methods, specifically model-based, logic-based (logics are employed to express the desired properties of the systems, including temporal and probabilistic behaviours), algebraic, process algebra-based (differ from algebraic by supporting explicit representation of concurrency), and net-based (graphical notations with precise formal semantics) [Liu97]. On the same topic, Gaudel points out that a finer distinction between formal methods can be made by using additional criteria, specifically [Gaudel94]:

- Their level of formality –the methods can be classified according to three key terms, namely 'formalised,' 'conceptual,' and 'deductive'. Formal methods are obviously formalised but the degree of the notation's formalisation and the potential of performing various types of checks are different from method to method. Similarly, different techniques emphasise in various degrees their capability of modelling conceptual aspects of systems and exhibit deduction systems of various degrees of complexity;
- The life-cycle stages where the techniques are applied. The classification encompasses activities such as domain specification, requirements engineering, design by refinement, proof of correctness, software re-engineering, and reuse;
- The specific aspects of computing they address. Algebraic methods are focused on describing abstract data types in an implementation-independent manner, model-oriented techniques aim at explicitly dealing with the dynamics of state-based systems, while other approaches address aspects specific to reactive and distributed systems, such as communication and concurrency;
- The mathematical foundation on which they are based, in terms of conceptual framework and deduction system. The conceptual foundations include process algebras, automata, set theory, and partial functions, while the deduction systems can be based on first-order predicate logic, higher-order logic, temporal logic, etc.;
- The methodological apparatus accompanying the method. Typically, this may consist of data tool kits in the case of model-oriented techniques, or may be provided as a kernel for property specification in the case of algebraic or axiomatic methods.

Some of the most representative formal methods are, in alphabetical order, Abrial's B-Method [Abrial96], Hoare's CSP [Hoare78, Hoare85], Milner's CCS [Milner80], ITL (Interval Temporal Logic) [Moszkowski86], Larch [Guttag85, Guttag93], LOTOS [ISO89], Petri Nets [Petri62, Reisig85], RTL (Real-Time Logic) [Jahanian86], RTTL (Real-Time Temporal Logic) [Ostroff89], Statecharts [Harel87], Temporal Logic [Rescher71, Pnueli77, Manna81], VDM [Jones90], and Z [Spivey92]. A large variety of environments and tools have been developed to accompany the existing formal methods and a significant number of

extensions and variations have been proposed. Several notable variants and tools pertaining to the "Z sub-domain" are discussed in Subsection 3.2.2 of this thesis.

2.6.3 Advantages and Disadvantages of Formal Methods

There has been a fair amount of debate over the applicability of formal methods in practice and especially over their potential of becoming working instruments for the large community of software developers. The attitudes vary from strong skepticism [Lawrence96, Glass96] to resolute conviction [Hall90, Meyer97, Kapur00], with many views within the range delimited by the above positions. We note however that the tendency is to recognise the benefits of formality in software development, but to caution also about its perceived disadvantages.

In what follows we present a summary of both benefits and disadvantages of applying formal techniques but not before mentioning that precisely the intricacies of these techniques prompted us to decide on the fundamental theme of our thesis, that of integrating formality with semi-formality in software specification.

The main reasons for employing formal methods are related to achieving the following goals:

- Better understanding of the system through formal specification and increased intellectual control [Gerhart94, Sommerville95, Clarke96, Hall96]. Daniel Jackson, in particular, remarkably refutes Brian Lawrence's opinion that, due to the difficulties associated to their application, formal methods may not be actually needed. Jackson considers that documents written in a natural language cannot be adequate repositories of an analyst's insights and that the greatest benefits of formalising requirements reside in clarifying ideas, revealing unexpected issues, and providing relevant feedback for the discussion with the client (Jackson's counterpoint to Lawrence's opinion [Lawrence96], in [Jackson96a]). Also, Jeannette Wing remarkably notes that "the greatest benefit in

applying a formal method often comes from the process of formalizing rather than from the end result" [Wing90, pp. 13];

- Higher degree of confidence through rigorous verification and property proving, particularly needed for the development of safety or security-critical systems [Sommerville95, Liu97, Schach99, Kapur00];
- Increased customer satisfaction and higher quality of products, including earlier detection and minimisation of errors, as well as enhanced functionality and performance [Gerhart94, Larsen96];
- Improved communication via supplemental notations [Gerhart94, Jackson96a];
- Power of abstraction or, as expressively stated by D. Jackson, "simplicity by omission" [Jackson96a, pp. 21];
- Competitive advantage resulting from applying the best practice [Gerhart94, Kelley-Sobel00];
- Compliance with standards or certification requirements [Hinchey96, Kapur00];
- Possibility of automatic transformation from specification to implementation [Sommerville95];
- Potential for reuse by enhanced identification of commonality [Bowen95a, Jackson96a, Meyer97];
- Educational benefits, including better understanding of research-and-design issues [Gerhart94] and improvement of complex problem solving skills [Kelley-Sobel00].

On the negative side, the following are considered the main disadvantages of formal methods:

- Difficult to use in practice due to their underlying mathematics, perceived by many developers as being hard to master [Gaudel94, Sommerville95, Lawrence96]. Representatively, Stephen Schach lists as weaknesses of formal specification methods "hard for team to learn, hard to use, almost impossible for most clients to understand" [Schach99, pp. 364];

- Lack of supporting tools [Gerhart94, Morgan94, Dill96, Holloway96];
- Not general enough, and not yet sufficiently employed in combination with other methods, formal or informal [Gerhart94, Clarke96, Lawrence96];
- Insufficient formal education and training of developers [Jones96, Hinchey96, Clarke96, Zimmerman00] and lack of educational support, including suitable textbooks [Kelley-Sobel00];
- Slow technology transfer from research to industry [Gaudel94, Glass96, Clarke96];
- Unwillingness of customers to invest effort in acquiring the necessary skills for dealing with formal representations of the systems [Sommerville95];
- Insufficient management support [Sommerville95];
- Inadequate notation, difficult to understand and use [Parnas96];
- Lack of application on significant, complex real-world problems [Holloway96, Dill96, Zimmerman00] and lack of truly impactive, convincing results [Parnas96].

Based on the analysis of a number of negative opinions about formal methods Hall [Hall90] and Bowen and Hinchey [Bowen95b] point out that many of the perceived disadvantages are actually "myths" and aptly dispel these myths with counterexamples and solid justification. Among the typical "myths" (or misconceptions) about formal methods the most common are: they increase development costs, can be applied only to safety critical systems, delay the delivery of the product, require a high level of mathematical skill, and are not actually necessary.

Overall, we share the view of those who advocate the application of formal methods, and believe that the difficulties of learning them are well paid off by the benefits they can bring. On the other hand, we agree with Anthony Hall that they are not a universal panacea [Hall90] and believe that integrating them into a software development approach that combines formality with informality can increase their chances of success in practice. Also, we need not forget that like most other things in life, formal methods should not be overused, otherwise they may turn out to be actual obstacles in the completion path of a

software product. Or, in Bowen and Hinchey's words, "thou shalt formalize, but not overformalize" [Bowen95a, pp. 57].

2.6.4 Formal Techniques within the Software Development Process

As indicated in Subsection 2.6.1, a formal system can essentially perform two kinds of activity, analysis and synthesis. On practical terms, formal techniques can be applied during all stages of formal development. Specific activities include rigorous specification of requirements, specification verification and validation, program refinement from specifications, specification-based testing, re-engineering, and reuse [Wing90, Gaudel94]. As pointed out by many authors, the greatest benefits can be obtained by applying formal techniques in the initial stages of development, when the early detection of errors saves a considerable amount of time and money [Leveson86, Morgan94, Larsen96, Schach99].

2.6.5 A New Trend: Lighter Use of Formal Methods

Recognising the need for a larger acceptance of formal methods, a new direction of investigation has emerged within the last few years, focused on a more pragmatic application of formalisms in software development. In order to increase the use of formal methods in industrial applications, including large-scale projects and applications outside the safety-critical area, cost-effective ways of improving the quality of software have been proposed. In this direction, Jones suggests the use of formal methods light, an approach focused on sketching the abstract model of the system, seen as crucial for understanding the architecture of the system, with minimum emphasis on notational details [Jones96]. In the same line of research, Jackson and Wing consider that lightweight formal methods, characterised by partiality in language, modelling, analysis, and composition, can bring greater benefits at reduced cost by allowing economically feasible automatic analysis of selected parts of the system [Jackson96b]. The authors' opinion is that the generality of an expressive language such as Z is an impediment for tool-supported analysis while simpler, less expressive, but more "focused" formal methods, can have greater effect in practical applications. An

exponent of the new direction, the lightweight modelling notation Alloy, based on a subset of Z and incorporating a limited number of extra features necessary for object modelling, has been recently developed at the Massachussetts Institute of Technology, together with a supporting tool entitled Alloy Constraint Analyzer [Jackson00a, Jackson00b]. Under the same umbrella of lightweight formal methods, Easterbrook et al. report very promising results of applying, in three NASA projects, "partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specification" [Easterbrook98, pp. 5]. Also, based on two other NASA case studies, Feather concludes that lightweight formal methods are useful for rapid analysis of specifications, yield results in a cost-effective and timely manner, and can be successfully used as complements to other forms of quality assurance [Feather98]. In a similar direction, Cau et al. propose the use of lean formal methods, envisaged as methods adequately accompanied by suites of affordable and practicable tools capable of supporting rapid prototyping, testing, and verification [Cau98], and Rushby suggests "invisible" formal methods, unobtrusively integrated in familiar software engineering tools [Rushby00]. The approach presented in this thesis also proposes a lighter application of formal methods.

2.7 Chapter Summary

In this chapter the larger space of our research has been surveyed and the topic of the dissertation has been localised on precise coordinates by using a "zoom-in" technique of exploration. Since the location of the thesis' topic lays at the intersection of three major domains of software development and investigation, namely real-time systems, formality, and object-orientation, an overview of these domains has been presented and specific challenges, advantages, and disadvantages have been pointed out. This overview has provided the groundwork for next focusing the "investigation lense" on the two specification notations used in our approach (in Chapter 3) and, respectively, on the existing research studies that share similarities with our work (in Chapter 4).