1 Introduction

" 'Where shall I begin, please your Majesty?' he asked.
'Begin at the beginning,' the King said, gravely,
'and go on till you come to the end: then stop.' "

[Lewis Carroll, Alice's Adventures in Wonderland, 1865]

1.1 Three Paradigms and a View of the Field

1.1.1 The First Paradigm or Objects as Conquerors

Few paradigms have had such a significant impact on the field of software development as the object-oriented approach. One can argue that actually there is nothing really new under the sun of technology, and that the object-oriented paradigm simply built upon the results of many honest structured methods exercised intensively on various domains of application over a significant number of years. The time of object-orientation just had to come, one may say, and this is probably true considering the constant progress within the computers' world, but we still cannot stop admiring its fundamental naturalness and the benefits it made possible.

The object-oriented paradigm has shifted the developers' focus from the solution domain (computer implementation) to the problem domain (the real-world that we relentlessly try to model and control) and brought with it a much greater modelling power –resulting primarily from the natural correspondence between objects and real-world entities. The object-oriented approach has also come with solutions for improved control of complexity –mainly

through abstraction, information hiding, and localisation, and provided effective answers for code reusability and extensibility via encapsulation and inheritance. The object concept proves to be remarkably powerful while essentially simple –the key characteristics of any true successful solution. And of any true conqueror.

1.1.2 The Resilient Field of Real-Time Applications

While a large variety of general-purpose object-oriented development methods have been proposed, among the most notable Shlaer and Mellor [Shlaer88, Shlaer91], Coad and Yourdon [Coad90, Coad91], OMT (Object Modeling Technique) [Rumbaugh91], Booch [Booch94], OOSE (Object-Oriented Software Engineering) [Jacobson94], Fusion [Coleman94], and more recently the Unified Software Development Process [Jacobson99], there has been comparatively a smaller production of object-oriented methods dedicated to real-time systems. This type of applications seemed to be more resilient to potential conquerors, including the objects. The explanation resides mostly in the efficiency concerns developers of real-time systems may have. As Bran Selic points out, even though the object paradigm is suitable for real-time applications (due to its equal emphasis on both structure and behaviour, which appropriately answers the needs of real-time systems development methodologies) it nevertheless extended over the real-time domain more slowly than over other areas of software development [Selic98]. The cause, the author indicates, lies in the rather scarce attention paid to important aspects of real-time execution, such as concurrency and efficient allocation of memory. Indeed, the constraints on execution speed and memory space are much stricter for real-time systems which, among other things, must meet deadlines and operate in typically unfriendly environments. Consequently, the traditional solution for ensuring both high execution speed and low memory utilization was to write lower level code using assembly language or languages such as C, Ada or Occam. These languages in turn provided relatively little support for the implementation of object-oriented designs. On the other hand, where support was provided (e.g., C++, Smalltalk) the overhead for manipulating objects at run time seemed to be costly, precluding the implementation of real-time systems in all but the more relaxed (softer) cases.

Nevertheless, some newer object-oriented approaches for real-time development such as ROOM [Selic94], Octopus [Awad96], and Comet [Gomaa00] have been successfully developed over the last years. This is certainly related to the constant improvements in hardware –faster, more powerful, and more compact processors being able to alleviate a number of issues related to the development of real-time systems in the "object-oriented way" and extend the application range of the OO paradigm in areas never tackled before. Commonly, the object-oriented analysis and design techniques that focus on real-time systems extend the traditional capabilities of general-purpose object-oriented methodologies with support for modelling aspects such as concurrency, distribution, timing constraints, synchronisation, communication, interrupts, and exceptions. At the implementation level, newer languages such as Ada95 [Barnes96] and Java [Gosling96] offer good support for writing real-time applications in an object-oriented manner (Java's capability for real-time programming is amply illustrated in [Bollella00]). These realities provide solid grounds for us to predict, for the near future, an increased interest in applying the object-oriented technology to the field of real-time applications. In other words, the field's resilience has been eroded to the point of the complete acceptance of the conqueror objects.

1.1.3 The Second Paradigm or Formalisation as a Controlling Factor

Software developers need to be resourceful, imaginative, alert, and quick to react to new challenges. This is due to the dynamics of their profession, in which daily novelties represent the only constant characteristic of the work environment. The need for fast and efficient solutions for new problems exercises tremendously the creativity of developers. But in the rush for delivering the expected solutions errors happen and bugs sneak in the software produced. Sometimes, the entire architecture of a program turns out to be erroneous. The craft of software developers needs reality checks, more so if the application domain is safety-critical or security-critical. Formalisms are needed as controlling factors of a developer's work; creativity must be channeled properly, and some moderation in art is necessary. It is well known that the best masterpieces brightly combine inspiration with rigor. In software development, formal methods are precisely employed to bring in the latter.

As shown by Gerhart et al., “formal methods are mathematical synthesis and analysis techniques used to develop computer-controlled systems” [Gerhart94, pp.5]. While it is observed that the technological transfer of formal development approaches from the academia to the industry is rather slow, an increased interest in the application of formal methods to software construction has been signaled over the last years [Fraser94, Clarke96, Hall98, Abernethy00]. Typically, what prompts the usage of formal techniques are safety concerns, regulatory standards, or the need to demonstrate that the implementation of a system corresponds to the system’s requirements. However, we believe that the most important reason for applying formal methods in industrial applications lies in the improved understanding of the system under construction and, generally speaking, in increased intellectual control over the software being developed.

Numerous formalisms or formal development frameworks have been proposed, among the most notable being Temporal Logic (TL) [Rescher71, Pnueli77], the Vienna Development Methodology (VDM) [Bjørner78, Jones90], Communicating Sequential Processes (CSP) [Hoare78, Hoare85], Calculus of Communicating Systems (CCS) [Milner80], Larch [Gutag85, Guttag93], Statecharts [Harel87, Harel96], and the Language of Temporal Logic Specification (LOTOS) [ISO89], but we will focus our attention on the formalism that emerged as one of the most popular over the last decade: the specification language Z, originated from the Oxford University Computing Laboratory, U.K., and currently used by many organisations all over the world. Very good classifications of formal approaches can be found in [Fraser94], [Gaudel94], and [Liu97], while authoritative references on Z are [Spivey92] and [Wordsworth92].

While successfully employed for formally describing and analysing numerous data-intensive, non real-time applications, the specification language Z has been only occasionally utilised for the development of time-constrained systems. Although mathematically sound, mature, expressive, and elegant, Z has been traditionally deemed of limited applicability in describing systems essentially characterised by strict demands on their meeting of prescribed deadlines, systems that most often are also concurrent in nature and complex, and possibly even safety

critical. This limitation is due mainly to Z's intrinsic lack of support for capturing temporal properties of systems and to its reduced capability for simulation, which makes difficult the construction of executable prototypes that could allow developers to interactively refine and validate the specifications. In addition, due precisely to its generality and expressiveness, Z does not typically allow for automated translation of specification into implementation code. However, newer studies have been focused on finding modalities of using Z for specifying real-time systems [Fidge97, Periyasamy97, Mahony98] and it has also been shown that by employing additional conventions and structuring mechanisms it is possible to animate a large subset of Z descriptions [Utting95, Jia98b]. Both these studies and the well-known, solid mathematical foundation offered by Z for formally capturing various properties of systems have encouraged us to investigate the possibility of using Z (more precisely, an object-oriented extension of Z) in the development of real-time systems (which, for reasons explained later in this chapter, we will refer to as time-constrained systems). In short, to approach successfully the field of real-time systems, we believe that objects alone are not sufficient: mathematical rigor is needed, and should be provided as early as possible in the software development process.

1.1.4 The Third Paradigm or The Power of Pictures

Descriptions of computer applications, at least in what regards the software components, used to be mostly if not entirely textual. There were hardly any other forms of representation but text and perhaps formulae and tables (both of them in essence some other forms of organised text). Driven by the technological engine that has produced increasingly faster processors and constantly larger-capacity devices, the world of software itself has changed in the last decade or so. The words of David Harel, in a 1988 seminal article, proved to be prophetic: "We are entirely convinced the future is visual. We believe that in the next few years many more of our daily technical and scientifical chores will be carried out visually, and graphical facilities will be far better and cheaper than today's" [Harel88, pp. 528]. While after more than a decade we can extend this prediction to incorporate multimedia facilities, today we feel fortunate to witness the truthfulness of Harel's prediction and admire the

accuracy of his vision. The graphical symbols (for practical purposes we exclude from them the classical letters of the alphabet), the icons, the visual metaphors, the animation, are now common parts of our daily interaction with the computers. Actually, it is hard to imagine today any significant software development environment based exclusively on text. Even the more conservative Unix systems have included graphical interfaces into their environments. As Harel predicted, the present is and indeed the future will continue to be visual. We have complied with this reality by incorporating a graphical notation in our modelling approach and by providing a graphical user interface to the tool that supports this approach.

1.2 Motivations

The motivations for our research approach can be summarised as follows:

1.2.1 Effectiveness and Simplicity

First and foremost, we have the fundamental belief that any new, practical approach should necessarily be both effective and simple –or, to be more precise and use one of Einstein's well known quotes, "as simple as possible, but no simpler" (this quote is cited, among others, by Stroustrup in his landmark book on C++ [Stroustrup97, pp. 723]). Obviously, any academic research should have a motivation that ultimately relates to practical needs. Overcomplicated software development approaches have difficulties gaining widespread acceptance in industrial environments, and as such they take the risk of remaining mere exercises in abstraction. The point here is not to underestimate the need for complex, sound, thoroughly refined theoretical foundations for new software development techniques, but rather to emphasise the necessity of hiding such foundations under apparently unsophisticated facades. In other words, we are driven in our approach by the desire to "engineer the illusion of simplicity" [Booch94, pp. 6].

1.2.2 Capability of Tackling Complex Tasks

We see the real-time systems as a complex, challenging field of investigation that is open to new research and offer the promise of rewarding methodological improvements. Benefits of

effective application development in this area are potentially enormous [Kopetz97, McUmber99, Douglass99].

1.2.3 Early Detection of Errors

Cost-benefit considerations also provide for us the compelling reason to focus on the early stages of the software development process, where detecting and correcting an error is usually between tens and hundreds of times less expensive than later, during implementation and maintenance [Boehm84, Schach99].

1.2.4 Powerful Combination of Paradigms

We consider that the accurate combination of several major paradigms that emerged vigorously within the software development world can provide the basis for a technologically sound, useful, and efficient methodological solution.

1.2.5 Understandability and Practicality

Effectiveness requires excellent communication and minimal departure from the problem domain in terms of description of functionality. As such, we see use cases and scenarios as the most appropriate means of interactivity, as key elements for bridging the gap between the users' understanding of the system under development and the developers' view of the same thing (the system). In software specification, capturing the behaviour of a system is probably more important than describing the system's structure, because the latter can generally be subjected to some approximations and refined in later stages.

1.2.6 Ease of Communication

Speed of communication and shared understanding depend on the way the information is organised and on the quality of the information's conveyor. Visual representations and graphical symbols are very powerful means of transmitting information. One cannot rely exclusively on unadorned text for capturing the intricacies of real-time systems. We are

compelled by today's technology, in which visual descriptions play a very important role in conveying information, to incorporate in our approach forms of graphical representation.

1.2.7 Expressiveness and Modernity

Because we deal with the specification of software systems we are compelled, for reasons outlined in Subsection 1.1.1, to proceed in an object-oriented manner. We use object-orientation as the wrapper paradigm of our approach that also incorporates formality and focuses on real-time issues. The widespread success of this paradigm accounts for our choice, there is no real competition for objects at this point in time.

1.2.8 Rigor and Precision

Formality or, in other words, mathematical rigor is a condition for dependability and assurance when dealing with real-time systems. Not only are we convinced that the key parts of the more complex software specifications should be treated formally, but we make out of formalisation an important component of our approach.

1.2.9 Refinement

Finally, to supply our approach with the necessary characteristic of "naturalness" (synonymous to "developer-friendly") we have included the classical technique of refinement in the modelling approach proposed (the term is used here in the sense of iterative revision of the model for gradual improvement, not in the sense of successive detailing of the model up to executable code). It should be point out that refinement is not used simply as a universal remedy, but as an important constituent of our approach, as shown in Chapter 7 of the thesis.

1.3 Challenges

Based on the above-mentioned considerations, our essential goal, stated briefly, is to propose a new, theoretically sound, yet user-friendly and pragmatic methodological approach for

specifying time-constrained systems. The approach aims at incorporating both object-oriented principles and formal techniques for describing the software under construction. We have identified a number of major challenges for our endeavor, as outlined below.

1.3.1 Efficient Combination of Techniques and Notations

There is an apparent dichotomy between graphical (specifically, semi-formal or informal) and formal techniques for software specifications, but there is also a growing number of approaches that attempt to integrate them and reap the benefits of both, as shown in Chapter 4. (To be precise, graphical notations can be formal, as discussed more in Section 2.5 of this thesis, but unless specified otherwise we refer in our dissertation to the larger category of semi-formal and informal graphical notations –see also the notes on terminology in Section 1.5). Typically, specification approaches based on semi-formal or informal graphical representations are designed to provide a user-friendly apparatus for software development, and focus primarily on suitable methodological steps and on the inclusion of a an easy to manipulate set of modelling symbols. The concern for rapid development plays an important role in the definition of such approaches. Conversely, formal techniques are employed rather as sophisticated tools for demonstrating properties of the systems, and are generally used only in situations that require special attention, such as safety analysis or security enforcement. Formal methods can provide greater intellectual control even though, as pointed out by Gerhart et al., no single method is general enough to completely cover an application domain, and it is rather unclear how to combine formal methods with other methods [Gerhart94]. However, as indicated by Perry Alexander, the two types of models, formal and informal, are not competitive, but complementary [Alexander95]. On the one hand, graphical models are natural and easy to understand and on the other hand formal models ensure precise specification and proof capability. The integration of the two models would combine, as well said by Alexander, "the best of both worlds," thus offering a solution for reliable, efficient software development. The challenge remains, obviously, to seamlessly integrate them in an efficient, unified approach, balanced between formal and informal, flexible enough to be used for a large class of applications, and able to adapt to various degrees of rigorousness demands.

1.3.2 Approaching Time-Constrained Systems from an Object-Oriented Perspective

The application of the object-oriented paradigm has been extended relatively recently to the area of real-time systems (more details are presented in Chapters 2, 3, and 4 of the thesis). However, there are numerous aspects of such systems that need particular attention when dealt with from an object-oriented point of view. As pointed out by numerous authors, the specification of real-time systems using the object paradigm remains an area of ongoing research [Yang96, Evans99, McUmber99]. The modelling challenges in the case of time-constrained systems embrace both structural aspects (such as identification and structuring of classes, establishing relationships, and deciding on object responsibilities) and behavioural aspects, including message passing, synchronisation, communication, parallel execution and, of course, capturing of time properties in the form of precise temporal constraints imposed on the run-time execution of the system.

1.3.3 Developing Mechanisms for Formalisation of Graphical Representations

The translation of models described using a semi-formal graphical representation into their formal, mathematically sound counterparts has been the object of previous research work, as presented in more detail in Chapter 4 of the thesis. However, rules for formalisation have been designed primarily in the context of structured methods, such as SSADM (Structured Systems Analysis and Design Method) [Pollack92] or the RRT (Rigorous Review Technique) [Aujla94], while relatively few attempts have targeted the object-oriented models, and even fewer have been dedicated to the specification of real-time systems. With the emergence of the modelling standard UML (Unified Modelling Language) [Booch98] some recent approaches have focused on translating UML notations into formal equivalents or on employing UML in conjunction with formal notations, as discussed in more detail in Chapters 3 and 4. Yet, there is still a need for continued work in this new direction, especially if we take into consideration real-time aspects of the systems.

1.3.4 Rigorous Treatment of Temporal Constraints

In a frequently cited paper, Dasarathy stresses the importance of specialised constructs in requirements languages for capturing timing constraints [Dasarathy85]. He points out that temporal restrictions typically considered are performance constraints (placed on the system's response) although the same importance should be given to behavioural constraints, which impose limits on the rate of stimuli on a system. Dealing with time in a rigorous fashion is in itself a complex problem. Accurately capturing timing properties of systems has been for some time the subject of considerable research work [Hoare78, Dasarathy85, Ostroff89, Shaw92, Mathai96, and many others]. However, including temporal aspects in object-oriented models is an even greater issue, a subject that over the last few years has increasingly attracted the attention of researchers (examples of research in this direction include [Vishnuvajjala96], [Selic99a], [Alagar00], and [Kim00b]). We strongly agree with Leung and Chan that "being such an important notion, time deserves a proper treatment" [Leung96, pp. 246]. Consequently, we attempt to include in our notation a set of modelling constructs capable to provide the necessary support for expressing our expectations of punctuality and collaboration regarding the components of the system being developed.

1.3.5 Provisions for User Acceptance

As previously mentioned, one of our goals is that of understandability and practicality. We advocate the application of formal techniques in software development, particularly in software specification, but we are aware that the acceptance of such techniques by the software development community can be achieved only by proposing a well-defined, relatively small, yet expressive set of notations, incorporated into a straightforward and easy-to-follow modelling technique. Therefore, the challenge is to reach the equilibrium between the true expressiveness of the approach and its apparent complexity, which must not be perceived as too complicated to its intended users. Of course, it will not be possible to completely hide the mathematical foundation of the approach behind graphical symbols but, as pointed out by Gerhart et al., the main challenge for applying formal methods consists not of teaching the developers the mathematics involved, but of training the users how to model

the systems properly [Gerhart94]. Hence, we need work on the notation, but must not forget the method.

1.3.6 Tool Support

A recognised issue with the formal techniques in general is the lack of tool support [Gerhart94, Dill96]. Software tools are necessary for enhanced interaction with the user, including navigation and visualisation, for type checking, and for reasoning about the consistency of specifications across larger projects. Also, improved mechanisms of version control, as well as facilities for maintaining conformance between formal specifications and their corresponding design rationales are needed [Johnson96]. Consequently, our intention is to supply the theoretical results of our work with suitable tool support, in the form of an environment for object-oriented, visual and formal modelling of systems. Even though some desirable capabilities of this environment, such as formal proof and animation, would not be included in our tool at this stage (such features would require separate, complex research investigations) our intention is to include sufficient functionality in the tool to illustrate the practicality of our approach.

1.3.7 Capability of Extension

Even though potentially very rewarding, dealing with formal aspects at the specification level must be seen only as a starting point towards the application of the proposed dual approach, formal and semi-formal, to the entire software development process. We would like to see beyond the present dissertation and leave the door open for potential extensions beyond the modelling phase, for instance for prototyping and simulation, refinement to executable code, specification based testing, and formally-conducted maintenance. In practical terms, the challenge is that both the notation and the deliverables of our specification approach should be ready for use in subsequent software development phases as well as in association with alternative software construction techniques and tools.

1.4 Notes on Terminology

Before outlining the approach proposed in this thesis several notes on terminology are necessary.

First of all, we rely on Fraser et al. to distinguish between formal, semi-formal, and informal specification techniques [Fraser94]. Specifically, informal techniques, represented by natural language and unstructured pictures, "do not have complete sets of rules to constrain the models that can be created," semi-formal techniques have well-defined syntax and their "typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities," while formal techniques, such as specification languages based on predicate logic, have precise syntax and semantics and "there is an underlying model against which a description expressed in a mathematical notation can be verified." [Fraser94, pp. 79].

Secondly, as many other authors, for instance [Spivey92] and [France97], we use the term notation as a substitute for language, although rigorously speaking notation refers only to the set of symbols belonging to the language. This commonly used promotion of the term helps avoiding tedious repetitions and simplifies the discourse of the thesis.

Thirdly, we use the word specification in the sense defined by Alan Davis, that of a document containing a description (in our case, of the software under construction). According to this definition, one can use terms such as requirements specification, design specification, or test specification [Davis93, pp. 372].

Fourthly, the word modelling, which also appears in the title of our thesis, is used to denote the activity of creating a model; that is, of developing a representation of the real thing (which, again, in our case is the software system being built). We see the two concepts, specification and modelling, closely connected and the difference between them of rather

fine nuance. Specification, in our view, is a description that may propose a model, while a model, in its analytical form, is recorded in a specification (for the sake of completeness, generally speaking a specification may not contain a model and a model may not have a specification). In our approach the distinction between specification and modelling is especially difficult to highlight; using well-established terminology, we employ the modelling notation UML and a variant of the specification language Z to create object-oriented models of the system, described in documents (specifications) that encompass both analysis and design aspects.

Finally, we use the term time-constrained systems (TCS) as an alternative to real-time or reactive systems in order to emphasise the temporal restrictions imposed on such systems and to shift the focus from specialised, less approachable products confined to rather restricted domains (military, nuclear energy generation, or medical devices), to more accessible products such as operating systems, transaction processing systems, cellular phones, and microwave oven controllers. In our view, one can consider the term time-constrained systems a substitute for both hard and soft real-time systems –a substitute that stresses the importance of timing properties that characterise these systems. Nevertheless, in order to avoid repetitions and employ recognised terminology when necessary, the terms time-constrained systems and real-time systems are used interchangeably in this thesis.

1.5 The Proposed Approach

This thesis is about the integration of semi-formal, graphical representations with formal notations within a modelling approach aimed at the construction of time-constrained systems. We believe that the two types of notation, graphical (semi-formal) and, respectively, formal, can efficiently complement each other and provide the basis for a software specification approach that can be both rigorous and practical. The former notations, relying on graphical symbols and diagrams, bring the “power of pictures,” which manifests through better representation of abstractions and higher expressiveness. The latter notations, precise, based on mathematics, increase the developer’s assurance and intellectual control and make possible automated synthesis and verification. Although many authors have envisaged the

advantages of combining informality with formality in software construction, there are very few reports that address the problem within the context of object-orientation and project its solution over the canvas of TCS modelling.

The pillars of our approach are the following: the combination of formal and semi-formal notations for specification purposes, the integration into an object-oriented approach of modelling capabilities that target properties of TCS, the elaboration of detailed translation algorithms from diagrammatic representations to formal specifications, and the proposal of a procedural frame for effective and reliable development of TCS. Principles and an outline for the reverse translation, from formal specifications to graphical representations, an auxiliary process intended to support the understanding of the system's model by developers and users not trained in formal methods, are also included in the approach.

While the graphical notation employed is a subset of the UML, the formal notations used are Lano and Haughton's Z++ object-oriented variant of Z [Lano91, Lano94a, Lano95] and Jahanian and Mok's Real Time Logic [Jahanian86, Jahanian94]. Both structural and dynamic aspects of the system are considered and a new modelling element, denoted class compound and consisting of a simple yet practical aggregation of the UML class and state diagram constructs, is proposed in order to facilitate the specification process.

From a methodological point of view, after several UML-based modelling steps are completed the formalisation process can take place, the result being a formal specification derived from the graphical representations obtained in the earlier steps. The integrated, semi-formal and formal model of the system can be subsequently enhanced while the designed translation mechanisms allow changes in the graphical representations to be reflected into the formal specifications as well as modifications of the formal specifications to be fed back into the diagrammatic descriptions of the system.

A case study, an Elevator System, is included in the thesis to illustrate the application of the proposed approach and the GUI-centred design of Harmony, an integrated specification environment intended to support the approach, is also presented.

Although we believe the proposed approach offers a viable solution for modelling software systems, it has nevertheless a number of limitations that need be pointed out. Firstly, the translation of UML constructs is restricted to a subset of the notation, and the treatment of state diagrams is confined to sequential, non-composite executions (composite states and aspects related to concurrency are not covered), which reduces the applicability of the translation algorithms to modelling TCS. Secondly, although temporal constraints can be attached to structural UML constructs in the regular way (using UML time marks, time expressions, and timing constraints –see Table 3.VI for details), we have not tackled their mechanised translation to Z++, and there is a limited incorporation of such constraints in the state diagrams employed. More precisely, the timing information pertaining to state diagrams considered in the formalisation process is only in the form of bounds [lower, upper] included in the label of transitions and in the form of transition triggers of the kind passage of time events (all other sorts of temporal constraints need be added manually by the Z++ specifier). Thirdly, the formal language employed, Z++, is currently lacking in supporting tools, which can be an impediment to the use of the proposed approach in industrial applications (our Harmony tool is not yet implemented, and we have not intended to deal with tool-supported formal analysis and formal refinement in the present thesis). In fact, we are aware of tools for Z++ only via [Lano94d], in which it is mentioned that such tools have been written in Quintus Prolog and ProWindows, but we have not investigated the possible connection of our approach to these tools. Fourthly, for the formalisation algorithms a set of rules for well-formedness and a set of principles for translation are given without using meta-models for UML and Z++/RTL, yet the use of these meta-models would have probably allowed a more concise and precise description of the algorithms. Also, there are a number of issues related to the application of the formalisation and deformalisation algorithms, indicated in Section 6.6, that deserve further investigation and require additional work.

However, our belief is that, through future work, the above limitations can be overcome and our proposal can thus become a stronger contender in the landscape of object-oriented approaches for modelling TCS.

1.6 Overview of the Thesis

The present thesis, in its remaining chapters, is organised as follows. Chapter 2, Background: Context and Concepts, defines the space of our research, localises in this space the topic of our dissertation, and presents the most significant aspects of the "domains" that belong to the space of our investigation. The distinguishing characteristics of real-time systems are examined, essential object-oriented principles and concepts are surveyed, observations on the value of graphical notations are presented, and the utilisation of formal methods in software development is discussed. In Chapter 3, Background: Notations, the focus is shifted from general concepts to the two particular specification languages employed in our integrated approach. A description of the specification language Z is given, together with a short presentation of some of Z's variants. In particular, Z++, the object-oriented variant of Z used in the proposed approach is briefly introduced. Also, an overview of UML, including its capability for modelling real-time systems, as well as a look on UML's perspectives are included. A survey of reported research that is similar to ours is taken in Chapter 4, Related Work. In this chapter, the major ways of integrating informality with formality in the specification phase are identified, related approaches focused on real-time systems are examined, and existing ways of dealing with time in Z-based approaches are discussed. Details on the formal resources employed for dealing with time in a rigorous manner are presented in Chapter 5, Formal Specification of Temporal Constraints. This chapter includes a section on the major types of timing constraints that are considered when modelling time-constrained systems and gives details on the specific RTL constructs employed for capturing time-related properties of the systems. Details on the translation processes from UML class diagrams to Z specifications, including the automated formalisation of classes, relationships, and state diagrams are given in Chapter 6, Translations between UML and Z++: Formalisation and Deformalisation. Guidelines for completing the reverse translation, from Z++ to UML, are also suggested in this chapter. Chapter 7, A Procedural Frame, brings the translation mechanisms proposed in the previous chapter under the methodological umbrella of a complete modelling approach. The proposed dual (semi-formal and formal) modelling process is detailed through a series of steps organised in stages,

in each step a set of artefacts being produced, making up the combined diagrammatic and formal model of the system. An illustration of applying the proposed dual, integrated approach to modelling time-constrained systems is provided in Chapter 8, An Application: The Case of the Elevator System. Since any new methodological approach for software development is best served by an accompanying tool, Chapter 9, Towards an Integrated Environment: A Prototype for Harmony, presents the GUI-centred design of the software specification environment that we have envisaged as supporting tool for the proposed modelling approach. Finally, Chapter 10, Conclusions, analyses the merits and limitations of our approach, presents a summary of our contributions, and opens a window to the future by pointing to a series of connected research directions that we believe deserve further investigation.

1.7 Chapter Summary

In this chapter we have taken a view on the big picture, that of today's computer-related technologies, and introduced the larger scene of our research. We have explained the motivations of our endeavor, pointed out the major challenges related to our work, and outlined the proposed dual, integrated formal/semi-formal software specification approach. This approach, aimed at the development of time-constrained systems, has the main goal of harmoniously integrating graphical (semi-formal) and mathematical notations in a theoretically sound, yet friendly, flexible, and easy-to-use software specification methodology. An overview of the chapters that follow has been presented as well. In this initial chapter a brief analysis of three major paradigms that pervade today's software development world was also included. We believe that the foundation for sound, effective improvements of software development methodologies resides in the right combination of the three paradigms, object-orientation, formal specification, and visual representation. At the convergence of these powerful paradigms we place the topic of our thesis.