TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Design and Conception of an Environment for Rapid GUI Prototyping of Smart Devices

MASTER THESIS

Submitted to
Vienna University of Technology
in partial fulfilment of the requirements for the degree
Diplomingenieur (Dipl.-Ing.)

Written at the
Institute of Design & Assessment of Technology
Research Group Human Computer Interaction

and
Siemens AG Austria
Program System Engineering KBB Department

Supervised by
Prof. Dipl-Ing. Dr.techn. Peter Purgathofer

and
Dipl-Ing. Zsolt Nagy

by
Taoufik Naceur Gharbi
MatNr: 9426868

Vienna, at ----------------

Abstract

In recent years, electronic devices have become ubiquitous in daily life, quite sophisticated and smart enough to respond appropriately to human hankering. In addition, devices have become mobile and act like conventional computers in the sense of hardware and software.

Indeed, devices, e.g., cell phones, PDA's, smart phones, can now perform essential services, like those achieved with computers, due to the embedded software in them. This software has become complex and large enough insomuch that software engineering methodologies are needed to be practiced during design and development phases.

Further, time to market has become a crucial factor for the success of any new industrial product. Reducing this factor represents not only an aim but also a challenge for manufacturers and developers, who are acting in a competitive market, where products are incessantly changing to satisfy customer needs and demands.

As traditional software development methodologies can not really produce applications as faster as needed to keep up with customer demands, new methodologies and tools, therefore, should be used to speed up the time-critical design phase and the construction of prototypes of those applications to capture and validate the requirements early without errors and to reduce the development overall cost.

Rapid prototyping approaches have been identified to be a response to the question of the need of new development methodologies. Rapid prototyping is a software methodology that can be applied early in the system development cycle and that allows designers and developers to iteratively design and evaluate software applications with the intended users through creating prototypes.

In this thesis, a development environment, the **G**UI **Ra**pid **P**rototyping **E**nvironment (GRAPE), for rapid prototyping of smart devices will be presented. Its conception, design and how it can be used to create virtual prototypes of smart devices will be then described.

This development environment consists of an integrated set of appropriate tools and techniques, i.e. behavioral modeling using formal specification languages. This set provides opportunities for validation of the requirements early in the life cycle.

Zusammenfassung

In den letzten Jahren sind elektronische Geräte unverzichtbar für unser tägliches Leben geworden. Sie sind weit genug entwickelt, um für Menschen ohne viel Spezialwissen einfach bedienbar zu sein. Viele dieser Geräte sind mobile Geräte, die im Sinne von Hardware und Software wie Computer funktionieren.

Tatsächlich stellen Geräte wie Handys, PDA oder Smart Phones dank der inkludierten Software Funktionalitäten zur Verfügung, die wir von Computern kennen.
Diese Software ist mittlerweile so komplex, dass für die Design-, Entwicklungs- und Deployment-Phase eigene Software Engineering Methoden erforderlich sind.

Darüber hinaus ist die Produkteinführungszeit ein kritischer Erfolgsfaktor geworden. Die Hersteller agieren am freien Markt, der bedingt durch sich rasch ändernde Kundenanforderungen hoch dynamisch ist und laufend nach neuen oder geänderten Produkten verlangt. Das Verringern der Produkteinführungszeit ist dadurch sowohl Ziel als auch Herausforderung für Hersteller und Entwickler geworden.

Mit traditionellen Software-Entwicklungs-Methoden können Anwendungen nicht ausreichend rasch produziert werden, um den dynamischen Kundenanforderungen zu genügen. Es sind neue Methoden und Tools erforderlich, um die zeitkritische Design- und Prototypphase der Applikationen zu beschleunigen. Dadurch werden die Kunden-Anforderungen frühzeitig validiert, Fehler frühzeitig erkannt und damit auch Entwicklungskosten gesenkt.

„Rapid Prototyping” – Ansätze haben sich als adäquate Lösung dafür erwiesen. „Rapid Prototyping” ist eine Software-Entwicklungsmethode, die sehr früh im Entwicklungszyklus eingesetzt werden kann. Sie erlaubt den Designern und Entwicklern ein iteratives Vorgehen. Diese wiederum ermöglicht es, dass die Software schon während der Erstellung durch Prototypen, die User ausprobieren können, laufend überprüft und adaptiert werden kann.

In der vorliegenden Arbeit wird eine Entwicklungsumgebung, nämlich **G**UI **Ra**pid **P**rototyping **E**nvironment (GRAPE), für das „rapid prototyping“ von smart phones vorgestellt. Sowohl Konzept und Design, als auch die Verwendung von GRAPE werden anhand der Erstellung eines virtuellen Prototyps beschrieben.

Die Entwicklungsumgebung selbst besteht aus mehreren integrierten Tools und Techniken, wie etwa die Beschreibung des Systemverhaltens mittels einer formalen Sprache. Diese Tools und Techniken bieten Möglichkeiten, die Anforderungen sehr früh im Entstehungsprozess zu validieren.

Acknowledgements

I'd like to thank

Prof. Dipl-Ing. Dr.techn. Peter Purgathofer

Dipl-Ing. Zsolt Nagy

and all the people that I have met in my life for the simple reason that they enriched my knowledge through learning something, whatever small or big, from them.

Contents

List of Figures

1 Introduction

Today, time to market has become a crucial factor for the success of any new industrial product. Reducing this factor represents not only an aim but also a challenge for manufacturers and developers, who are acting in a competitive market, where products are incessantly changing to satisfy customer needs and demands [25].

In recent years, electronic devices had conquered almost every field of daily life. They are present everywhere and they are not to be thought away in human everyday behavior, policy and use. As customer needs and demands will never cease, devices have become smarter than before and quite sophisticated to respond appropriately to human hankering.

To be smarter in the way, devices have the ability to make decision and to communicate with its environment. To be sophisticated in the way, devices provide substantial services to make life easier and cushier to its user. In other words, devices act like conventional computers in the sense of hardware and software.

Indeed, devices, e.g., cell phones, PDA's, smart phones, can now perform some services, like those done with computers, due to the embedded software [28] in them. This software has become complex and large enough [27] insomuch that software engineering methodologies are needed to be practiced during design and development phases.

In spite of the fact that software engineering methodologies and tools have proven substantial improvements during the last two decades, requirements engineering still remains a key issue in software development [26].

"*The hardest part of the software task is the setting of the exact requirements*" [29]

The deficiency of early requirements validation constitutes one of the primary sources of intricacy in the software development process. Requirements are mostly misunderstood and change frequently while development. Therefore validation of requirements is still a difficult, cumbersome and costly task [26].

Evidently speeding up this task, it means rapid capturing and validation of requirements early in the life cycle, can be a response to the question of shortening the time-to-market factor, accelerating the development process and reducing costs, if and only if the applied methodologies are appropriate enough [25].

Therefore, new methodologies and tools should be developed to speed up the time-critical design phase and the construction of a prototype of any new device to capture and validate the requirements as early as possible without errors. This can cause the reduction of the development overall cost.

Within the scope of this thesis, an important approach to early requirements validation named "Rapid Prototyping" will be introduced. A software prototype can be developed from a specification outline. End users will then have the possibility to experiment with the prototype to determine functionalities and behavior of the final system, and to make improvements and refinements.

Based on prototyping, a development environment, the **G**UI **Ra**pid **P**rototyping **E**nvironment (GRAPE), for rapid prototyping of smart devices has been designed. Its conception and design constitute the main objective of the thesis that has been achieved within the scope of this work.

This development environment consists of an integrated set of appropriate tools and techniques. This set provides opportunities for capturing and validation of the requirements early in the life cycle.

The main components of GRAPE are Macromedia Flash and SICAT toolset. Macromedia Flash provides full design control and strong visual metaphors for developing graphic objects and animations. SICAT toolset supports system behaviour modelling using a formal specification language, i.e. the **S**pecification **D**escription **L**anguage (SDL), and provides code und documentation generation from the models automatically.

Using GRAPE virtual prototypes of smart devices can be created. Through these prototypes the end user can be involved during development phases by expressing their opinion and wishes about the behavior of the system and how it should look like. This user feedback occurs while experimenting with the prototype and it helps designers and developers enormously to modify already available requirements as well as developing new ones accordingly.

As mentioned previously, GRAPE is a software development environment intended for rapid prototyping of smart devices. With “Smart Devices” is meant, roughly speaking, every device that has at least user interfaces, like buttons and displays, i.e. screen, that enables interaction with the environment, i.e. human.

Finally, a “Smart Device” can be defined as follows:
A “Smart Device” is a physical object with embedded processor, memory and optionally a network connection. Smart devices have user interfaces and information’s displays in order to enable interaction with their physical environment.

2 Rapid Prototyping in Software Development

The misunderstanding of system requirements and the lack of adequate validation of the correctness of requirements have been identified to be a main cause of system failure and customer dissatisfaction [26].

Validation of requirements early in the life cycle constitutes one of the key issues in software development because while requirements validation failure can occur and result in frequent and expensive changes in later life cycle phases [26].

In conformity with a study realized by the University of West Virginia in Cooperation with the US Air Force [30] the origin of the most failures have been found in the early phases of the software design and development process (see Figure 2.1) [25].

As shown in the figure below the faulty translation of requirements represents the quote of 36% of the entire errors. Only 5% of errors are caused by incomplete requirements, whereas, 28% of the whole errors are traced back to logical design failures [25].

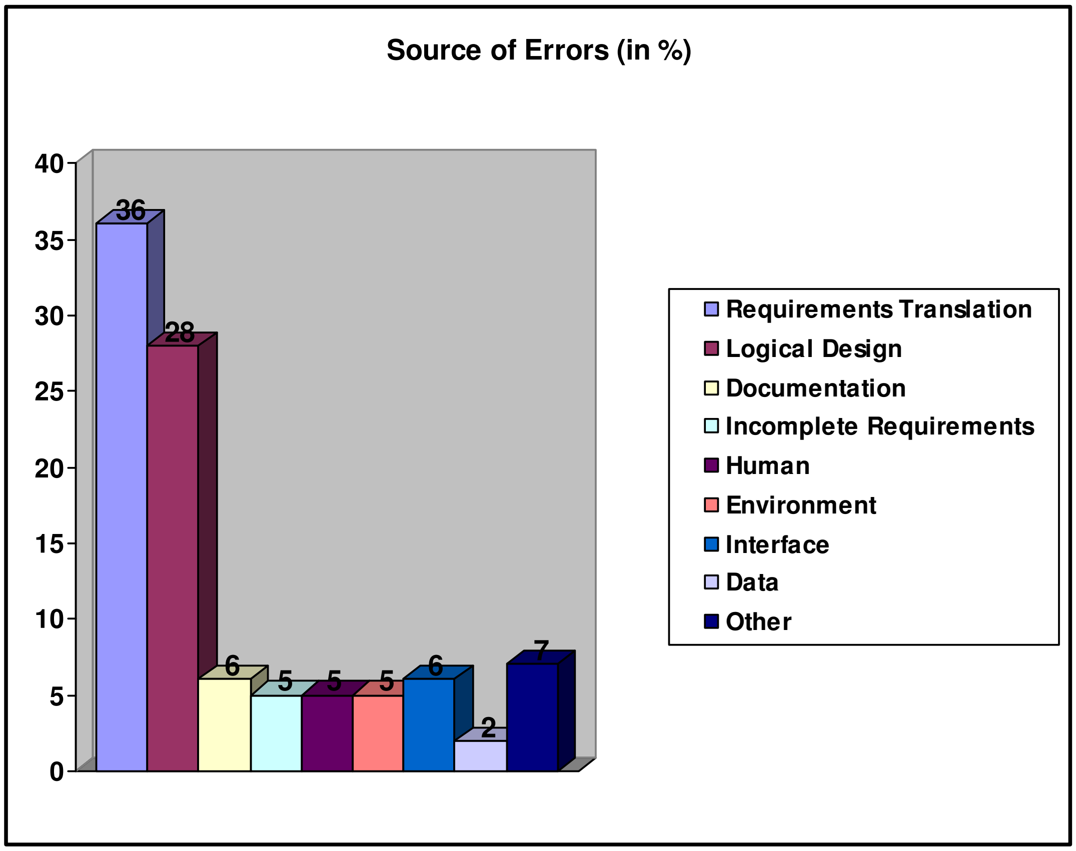
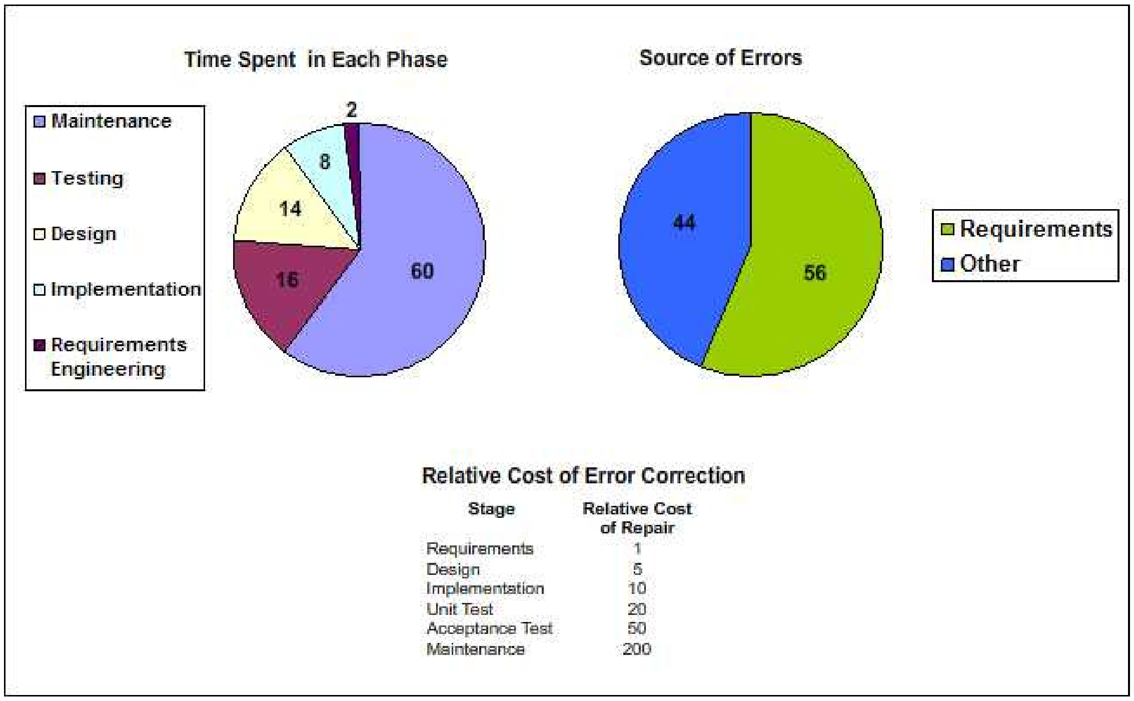


Figure 2.1: Causes of Errors



Relative Cost of Error Correction

Stage	Relative Cost of Repair
Requirements	1
Design	5
Implementation	10
Unit Test	20
Acceptance Test	50
Maintenance	200

Figure 2.2: Significance of Early Requirements Validation

The significance of early validation was testified and illustrated in Figure 2.2. According to “Boehm”, 54% of all errors detected in a certain software projects were detected after the implementation and testing phase. 83% of these errors could be traced back to requirements and design phase [26] [31]. Also “DeMarco” related that about 56% of all detected defects could be traced also to requirements [26] [32].

Further several errors in requirement phase are transmitted undetected to later phases of the life cycle. The patch of these errors, while or after coding phase, has been identified as one of the most costly task [26][33].

“*… early defect fixes are typically two orders of magnitude cheaper than late defect fixes and the early requirements and design defects typically leave more serious operational consequences.”* [26] [34]

The type of requirement errors has been found to be technical ones (see Figure 2.3).
77% of these errors were non-clerical errors, 49% were incorrect facts and 31% were omissions [26] [35]. Inconsistency and ambiguity were about 18% of all non-clerical errors.

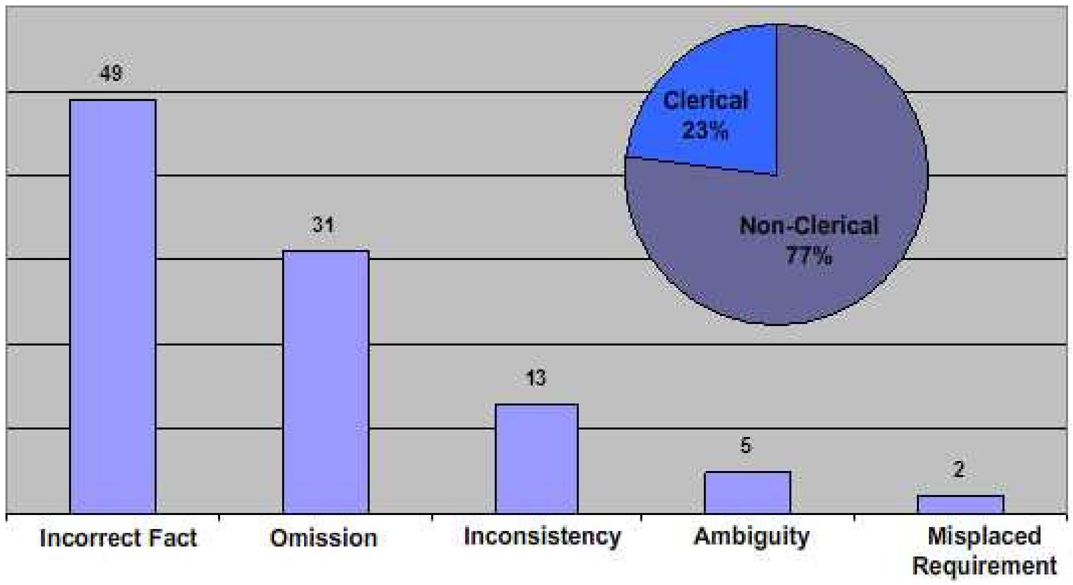


Figure 2.3: Types of Requirements Errors

By waterfall life cycle model a complete requirement specification is required before development. A complete requirements specification means, in this case, a contribution to the problematic mentioned above, concerning early validation of requirements, because of the frequent changes of the specification, and seems to be unsolvable without errors for complex and large system[26] [36].

Therefore, new methodologies based on prototyping have been used. In order to speed up the time-critical design phase and the construction of a prototype and to capture and validate the requirements, such methodologies have been applied.

Within the scope of this chapter, an important approach to early requirements validation named “Rapid Prototyping” will be introduced. A software prototype can be developed from a specification outline. End users will have the possibility to experiment with the prototype to determine functionalities and behavior of the final system, and to make improvements and refinements. This chapter will discuss the prototyping concept in the software process, its advantages and its benefits. A classification of these approaches will be provided. Some prototype techniques will be briefly described and listed. Then the user interface design based on prototyping will be presented as an effective method to design and analysis the graphical user interface early in the development lifecycle.

2.1 What is Prototyping?

By classical waterfall model and all its variations, software development phases must be clearly defined and detailed before execution. To come through the waterfall model lack, several new approaches and methodologies in the software development, such as incremental development, rapid prototyping and evolutionary prototyping, have been proposed [37].

Software development approaches based on prototyping have gained importance and significance. They had proven to have the ability to respond dynamically to frequent requirements changes, to reduce the amount of revision, and to assist the controlling of incomplete requirement and its risk. In addition prototyping is cost effective, improves communication between all persons involved while development process, helps determining technical feasibility, and is an appropriate method for risk management. End-user involvement and participation, while development based on prototyping, has been found as essential [37].

"*Prototyping is the process of building a model of a system. In terms of an information system, prototypes are employed to help system designers build an information system. That is intuitive and easy to manipulate for end users. Prototyping is an iterative process that is part of the analysis phase of the systems development life cycle*" [38]

In the analysis phase and while capturing and determination the requirements of a system, the analysts collect information about the business processes. Furthermore, they study the current system, lead interviews with users and pick up documentation. This information, together combined, could support the analysts to develop an initial set of the required system. This process could be increased by prototyping, which converts intangible specification to tangible, indeed with limited functionality but working, model of the required system [38].

After developing a physical prototype of the system, the end users can touch, perceive and experiment with the prototype. As a consequence users could change his wishes and express their opinion about how the system should be like and so on. This user feedback helps the analysts enormously to modify already available requirements as well as developing new ones accordingly [38].

The use of prototypes can be found in many disciplines. In the manufacturing and manufacturing engineering, for example, prototypes of products were created to explore and control uncertainly during product design, or to investigate difficulties in the production process before the real mass production started [37].

"*The software industry has adopted this industrial technique to construct prototypes as models, simulations, or as partial implementations of systems and to use them for a variety of different purposes, e.g., to test the feasibility of certain technical aspects of a system, or as specification tools to determine user requirements.*" [37]

Further by breaking up a complex system into smaller and simpler parts, the prototyping process could elate the development phase to be efficient. A prototyping development approach can, in the other hand, help build and refine a product to satisfy customer needs and market expectations. [37]

2.1.1 Definition

Rapid prototyping has become a typical term, which is used to describe many prototyping processes (see Figure 2.4) [8].

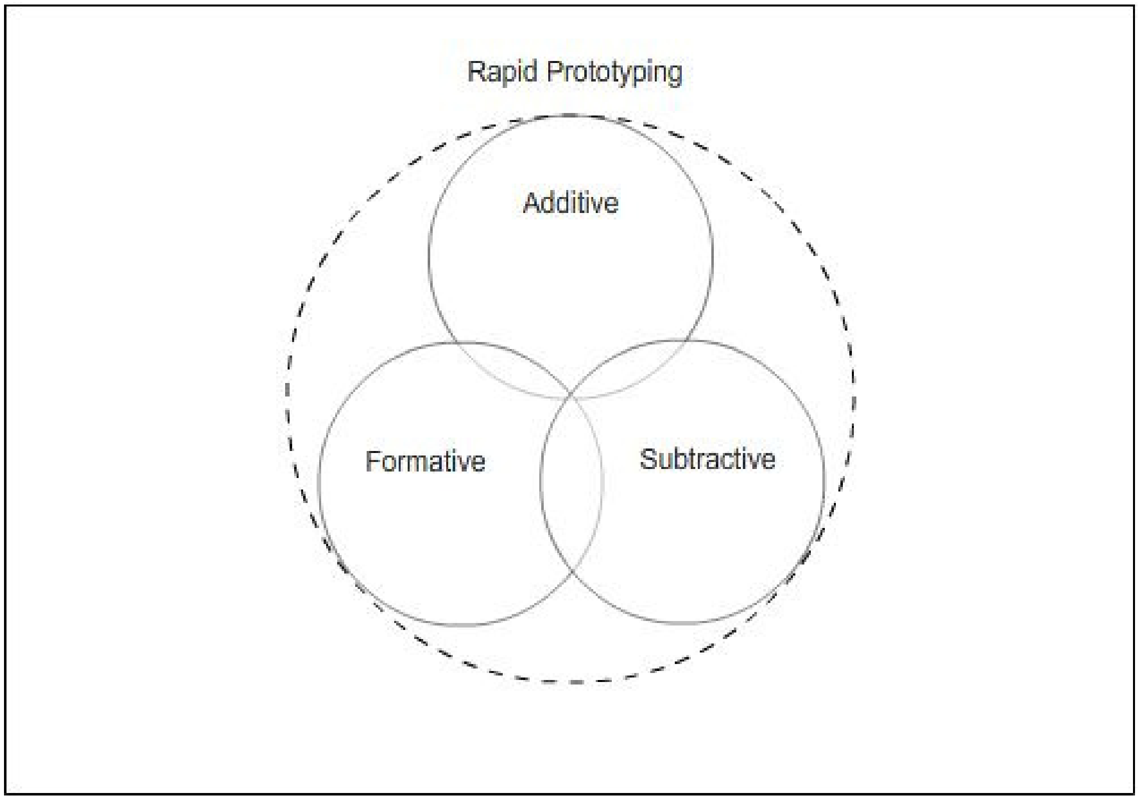


Figure 2.4: Rapid prototyping is a process that generates prototypes quickly

"*Today, everything is about speed, efficiency, and productivity, so the words are commonly applied to any process that generates prototypes rapidly*" [8].

In the literature there are a numerous terms and definitions for prototype and rapid prototyping [6][7][8][26][37].

Within the scope of this thesis and with respect to what this thesis is about, the following definition of "prototype" respectively "rapid prototyping" has been chosen:

"*A prototype is an enactable mock-up or model of a software system that enables evaluation of features or functions through user and developer interaction with operational scenarios. Prototyping exposes functional and behavioral aspects of the system as well as implementation considerations, thereby increasing the accuracy of requirements and helping to control their volatility during development.*" [26]

2.2 Prototyping in the Software Process

Software development and validation approaches are multitudinously described in the literature. These approaches referred in general to prototyping. A little agreement seems nevertheless to exist concerning the exact process of creating and using the prototypes [26].

About eighteen prototyping approaches described by six authors have been identified by Sage and Palmer's "Software Systems Engineering", who have showed the high degree of commonality among classes of approaches by mapping the different approaches into three classes [39].

The planed use and the intended users of the prototype can be a convenient way to classify prototyping approaches. In this sense, prototyping approaches can be grouped into two distinct categories. The first one contains prototyping approaches, which enfold the creation of a series of fielded prototypes. These approaches are referred to the so called "Evolutionary Prototyping". The second category comprise those approaches that intent on exploring ideas without resorting to field deployment. These approaches are named as "Throw-away Prototyping" [26].

The evolutionary prototyping starts with a relative simple system, which considers the most important user requirements. This system will be then extended and changed depending on requirements, until the final needed system comes out. There is none detailed system specification and in some cases there is not even a formal document for requirements [24].

In contrast to evolutionary prototyping approach, the throw-away one tends to clarify and to refine system specification. Therefore the prototype will be developed, evaluated and modified. The prototype evaluation represents a feedback and provides information for the development of the detailed specification. This latter is an essential part of the written fixed system requirements. As soon as the detailed specification is composed, the prototype has achieved its aim and will not be needed anymore [24]. Figure 2.5 below illustrates the both approaches of prototyping.

Between the objective of evolutionary prototyping and throw-away prototyping, an important difference could be noticed, which is as follows [24]:

1. Developing a functional system for end user represents an aim for development based on evolutionary prototyping approach. This means the developer should start with the best understood requirements. Requirements, which are not well defined, will be then implemented, when they are requisitioned from users

2. Determining and valuation of system requirements constitute, in this case, the aim of throw-away prototyping development approach. Developer should begin with the requirements, which are not yet definitely clarified, because they must find out them before. Simple and clarified requirements may not need to be tested through a prototype

Another important difference between the both prototyping approaches is for the quality management significant. Prototypes created while development based on throw-away approach, have a short life cycle. It is possible to change these prototypes during development rapidly. A long-term maintenance is not required. As far as the prototype has fulfilled its main functionality, a little efficiency and reliability could be accepted [24].

In contrast to that, prototypes are further developed with the same standard quality of the whole remaining software. The prototypes, which are here meant, are those created during development based on evolutionary prototyping and which constitute the basis of the final system. These prototypes must have a stable structure to sustain for many years. They must also be reliable and efficient, and correspond to the organization standards [24].

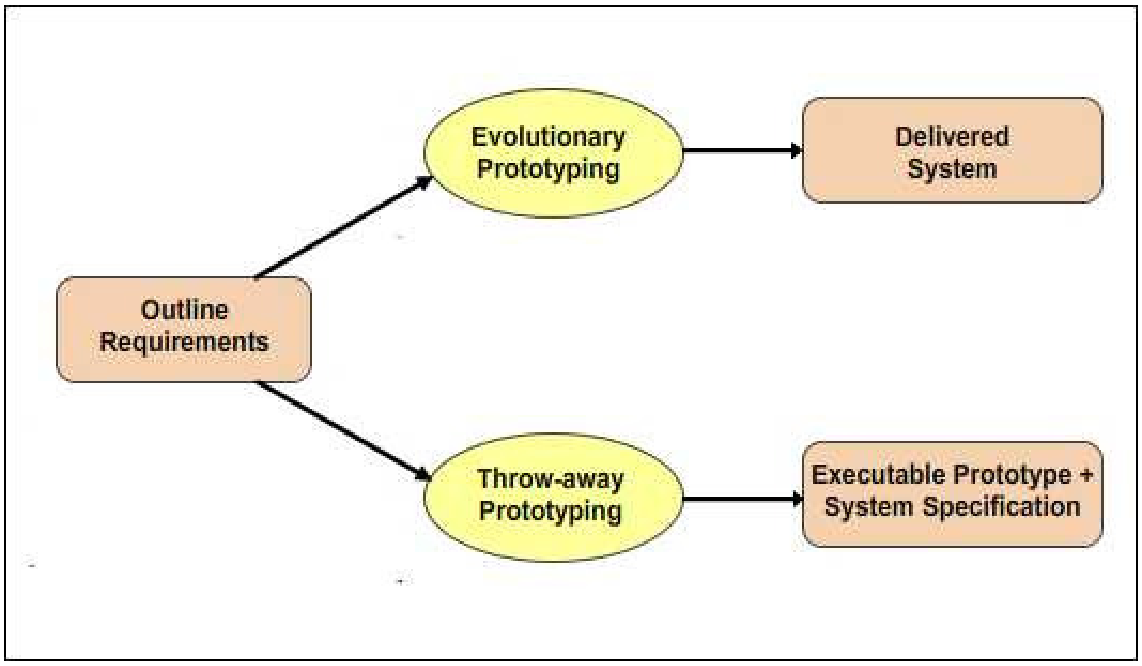


Figure 2.5: Prototyping Approaches

2.2.1 Evolutionary Prototyping

Evolutionary Prototyping is an approach to develop systems, whereat an initial prototype is produced. This prototype will be presented for comments to customer, and then the prototype will be iteratively refined until an adequate system will be reached (see Figure 2.6 below).The development process starts from an outlined not detailed specification, which contains the most understood requirements [24].

This development approach should be used for systems, e.g. AI systems, which are difficult to be specified, or where the specification could not be developed previously. This approach could be used also for systems, where development is based on techniques that allow system iterations rapidly. Because of the absence of the specification, verification seems to be impossible, but validation, in this case, means the demonstration of the system suitableness [24].

There are two main advantages for the use of the evolutionary prototyping approach in the software development process [24]:

1. *Accelerated delivery of the system*: In some cases, rapid delivery and deployment are important than functionality or long-term software maintainability

2. *User engagement with the system*: The system is not only probable to meet user requirements, but the users are more probable to commit at the use of the system

Beside its advantages, the evolutionary approach has also problems. There are three main problems, which are particularly important, in the case of development of large or for a long life span system [24]:

1. *Management problems*: To evaluate the progress, existing management processes assume a waterfall model of development. As prototypes could be created quickly, high documentation effort seems to be not effective with respect to costs. Further development based on rapid prototyping requires sometimes the use of unusual technologies, which cause the need of specialist skills. This latter could not be available in all development teams, which means, on the other hand, costs increasing

2. *Maintenance problems*: Continual change could effectuate system structure corruption. This implies that the system will be difficult to understand for external development teams. Moreover the special technologies, which were used during rapid prototyping development, could be deprecated quickly. In this sense, long-term maintenance could be expensive.

3. *Contractual problems*: Normally a contract between customer and software developer is based on system specification. As this latter does not exist, so it is easy to imagine how difficult it can be, to work out such a contract

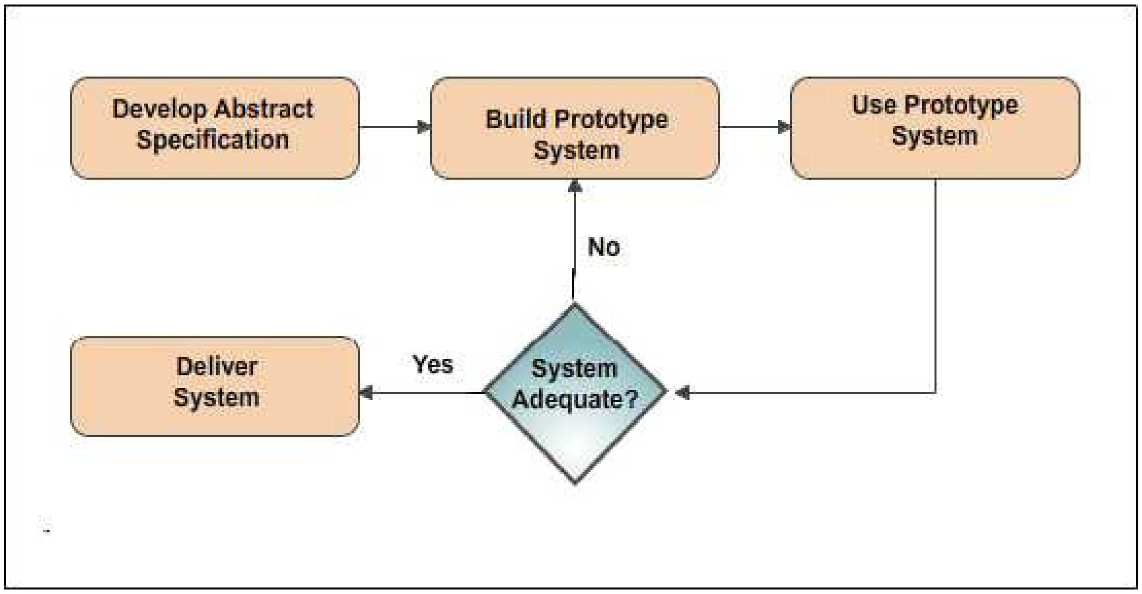


Figure 2.6: Evolutionary Prototyping

2.2.2 Throw-away Prototyping

The throw-away prototyping approach aims with extending the requirements analysis phase to reduce costs for the whole life cycle. Requirements determination and providing additional information, to help the management estimating risks, constitute the substantial function of the prototype. This latter will be not needed anymore after valuation and not used as basis for further system development [24].

On the other hand a prototype, created during development based on throw-away prototyping approach, will be used to fix the system requirements but not to validate a design. The prototype should be developed as fast as possible, to give to users chance to experiment with the prototype and then share their experiences with the developers. The user feedback helps then developers to clarify the system specification. In other words, a prototype, which is commonly a practical implementation of the system, is produced to help discover requirements problems and then discarding them. The system is then developed using some other development process [24].

As shown in Figure 2.7, the prototype is developed from an initial specification and delivered for experiment. This prototype will be modified until the user will be satisfied with its functionality. Then from the prototype the system specification will be derived. Reusable components could be used while system production process to reduce in some way costs. In some cases, there is no need to derive system specification from the prototype, because the prototype itself is the specification [24].

Like the evolutionary prototyping approach, the throw-away one has also its problems, which are as follows [24]:

1. Some system characteristics may have been left out to facilitate a rapid implementation. Some important parts of the system, e.g. security functions, could be not simulated with a prototype

2. For customer and developer, an implementation is not legally binding

3. Non-functional requirements concerning reliability, stability and security could be not tested with prototypes adequately

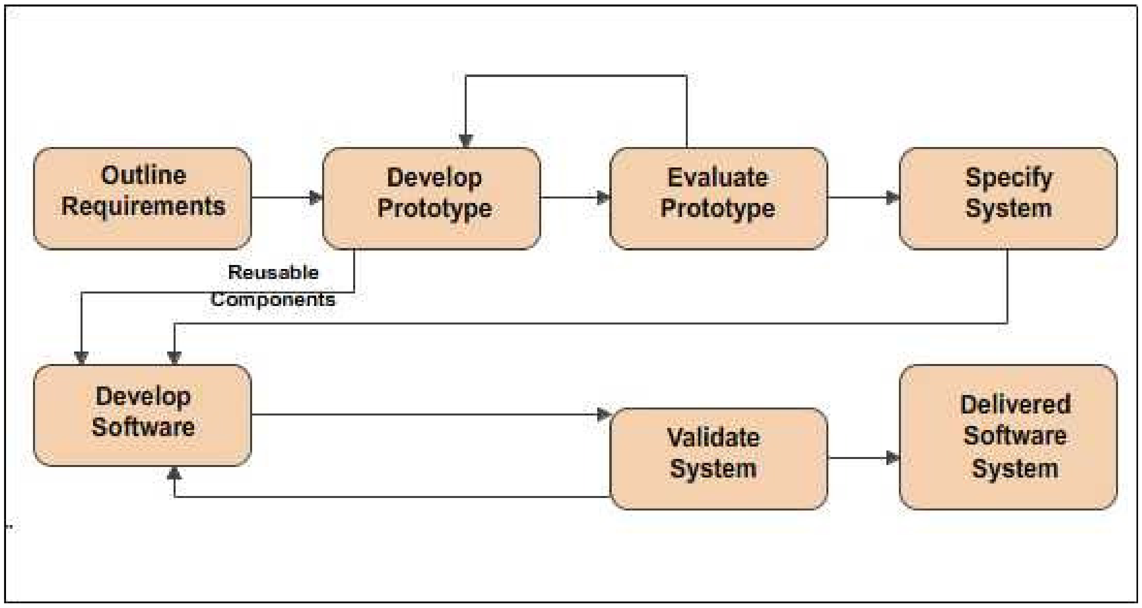


Figure 2.7: Throw-away Prototyping

Another general problem by this prototyping approach is that the prototype should not be considered as a final system. The tester of the prototype will be interested in the system as it is but not to be a typical system user. The time to know the prototype during evaluation could be insufficient. Further if the prototype is slow, the functioning will be corresponding attuned while evaluation and some system functions with long execution time could be avoided [24].

For utility purposes of requirements development, the prototype must not be executable.
In some cases, on paper and mostly scenario-based models of the system user interface have been identified as an effective way to support user to refine the user interface of the system. Such a prototype could be developed in few days and with a little cost. An extension of this technique could be for example a simple prototype, on which the user interface was developed merely [24].

2.3 Rapid Prototyping Techniques

During rapid prototyping development, diverse techniques could be used.
There are three techniques, which have been found to be practicable in industrial prototype development [24]:

1. Dynamic high-level language development
2. Database programming
3. Component and application assembly

In practice these techniques are often used together. For example a database programming language could be used to process data, while reusable components will be executed for detailed processing. Further the system user interface could be defined through visual programming [24].

Today the most systems of creating prototype support the visual programming rudiment, by which parts or the whole of the prototype can be developed interactively. Instead of writing a sequential program, the prototype developer works with graphical symbols or objects, which represent the functions, data or components of the user interface. Further, the developer assigns these symbols to the workflow. Then from the visual model of the system an executable system can be generated. This method simplifies the program development and decreases the prototype costs [24].

2.3.1 Dynamic High-Level Language Development

Dynamic High-Level languages are programming languages, which include powerful data management at runtime. These languages facilitate program development, because a lot of problems like those with memory management could be reduced. Examples for High-Level languages are Lisp, Prolog, Smalltalk and Java. This latter united the advantages of High-Level languages and has many reusable components. Java has been identified as very suitable language for evolutionary prototyping [24].

The choice of prototyping language can be facilitated by answering the following questions [24]:

1. *What is the application domain of the problem?* Certain languages are suitable for certain application domain. For example Smalltalk or Java is more convenient for interactive systems than other languages. Lisp or Prolog is suitable for symbolic processing like the natural language processing.

2. *What user interaction is required?* Usability is differently supported from languages to languages. For example, Java or Smalltalk are suitable for Web-applications, because they provide user interface development facilities, while others, like Prolog is more convenient for text-based user interface than the others.

3. *What support environment comes with the language?* Matured support environment with many tools and easy access to reusable components facilitate the software development process.

In some cases different parts of the system can be programmed in different languages. For that a communication framework between languages must be established. The advantage of this approach with different languages is that, for every logical application part the most suitable language should be chosen. However, problems with language communications may occur because of the difference between data objects of the languages. Thus a translation of an object from a language to another must be done. This means long code sections will be needed to be implemented [24].

2.3.2 Database Programming

In the meantime, the most of commercial applications occupied with data processing from a database, produces then results, which contain data organization and formatting. To support the development of such applications, domain specific languages based around a database management system are used, which provide embedded knowledge about the database and functions for data manipulation [24].

The term “Fourth Generations Language” (4GL) is used for both the database language and its environment. 4GLs are used to create interactive applications, which extract information from database and provide them to end user on their monitors, and then to update the database with the information from the user interface [24].

4GL environment may be integrated with a CASE toolset, as illustrated in Figure 2.8, which are [24]:

1. *Database query language*: the user can query the database with a query language (today Structured Query Language SQL) directly or through the completion of a form, and then the query will be automatically generated

2. *Screen generator*: to create forms for data input and output

3. *Report generator*: to define and create reports from the database

4. *Spreadsheet*: to analyze and process numeric information

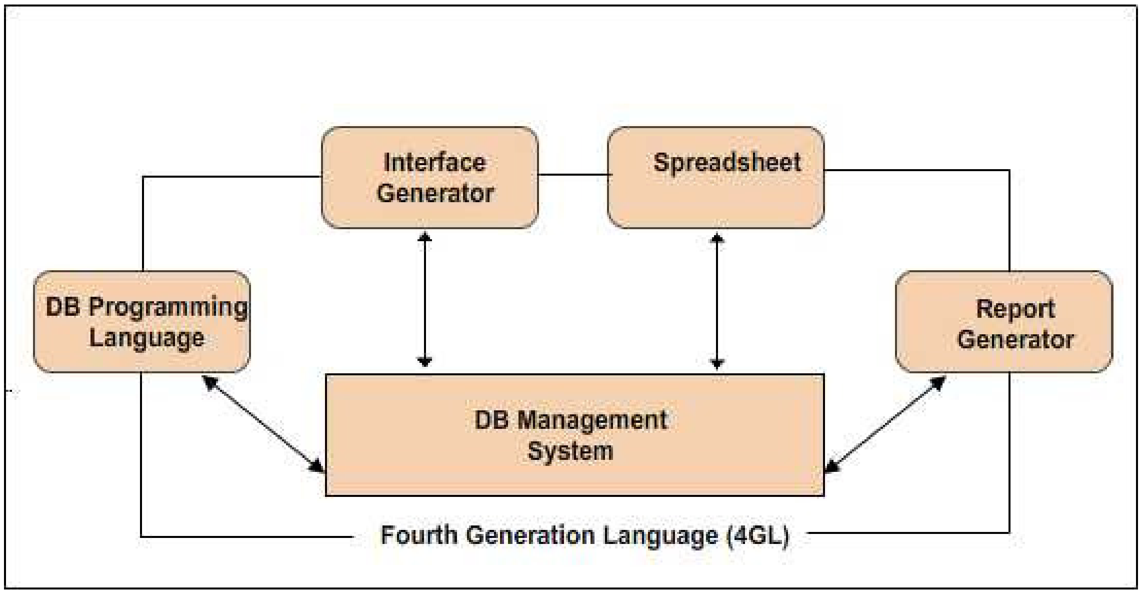


Figure 2.8: Database Programming

Development based on 4GLs may be applied for evolutionary prototyping, whereat system models are used to create prototypes, whose maintenance has been found to be better than those systems, which are created manually. Even though 4GLs languages are suitable for rapid prototyping development, they evince also disadvantages. Programs written with 4GLs are slower than others written in conventional languages and need much more memory. In addition costs for the whole life cycle of large systems can not be clarified previously. Programs written with 4GL tend to be difficult to maintain. Moreover 4GLs are not standardized. This means user programs should be rewritten, because the language may become deprecated in the meantime [24].

2.3.3 Component and Application Assembly

Reusing system parts can shorten development time. Hence prototypes can be created quickly from a set of reusable components and thereto some mechanisms to integrate these components together are provided. As shown in Figure 2.9 below, the composition mechanism must include control facilities and a mechanism for communication between components [24].

The system specification must take into account the availability and functionality of existing components. This means that some compromises while requirements capturing are needed to be taken. It may be possible, that the functionality of the already existing components does not cover the user requirements, which are in the most cases quite flexible, in the way that this procedure has been identified to be suitable for prototyping development approaches [24].

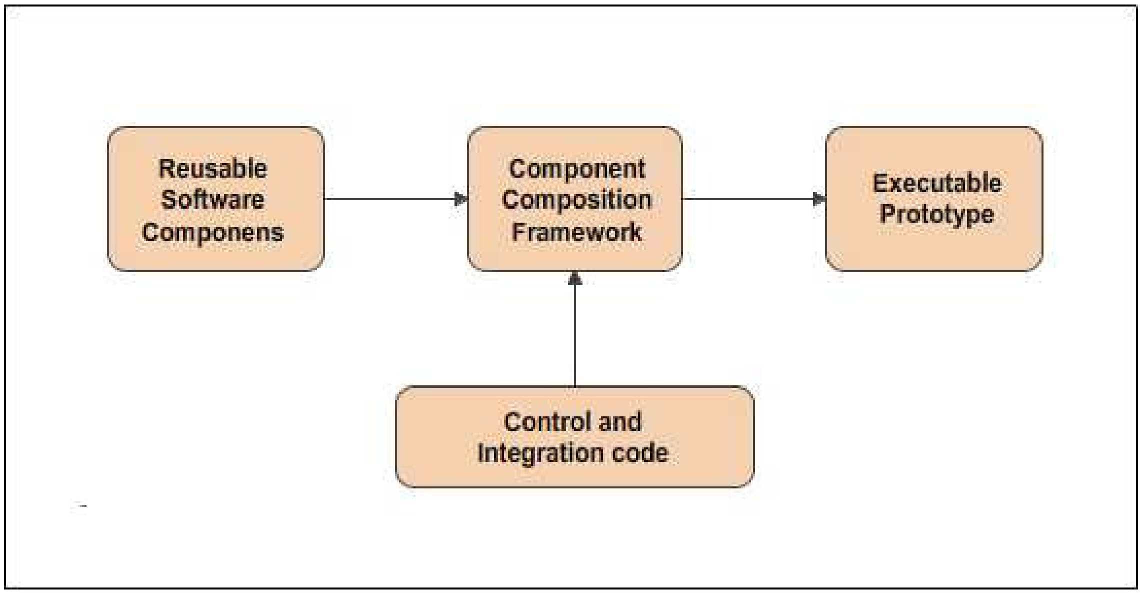

Reusable Software Componens
Component Composition Framework
Executable Prototype
Control and Integration code

Figure 2.9: Reusable Component Composition

Prototyping development based on component reuse may be used on two levels [24]:

1. *On the application level*, where the entire application systems are integrated with the prototype so that their functionality can be shared. For example, when a text processing is needed, a standard text processor can be used

2. *On the component level*, where individual components are integrated within a standard framework to implement the system. This standard framework can be a scripting language, like Visual Basic, Python or Perl. It can be alternatively an integration framework such as DCOM, CORBA or JavaBeans

The main advantage of prototyping with component reuse is that the major part of prototype and its functionality can be implemented rapidly and with little cost. But problems with the prototype performance may occur while switching between the application functions. Further it is not always possible to reuse entire application. Development with reuse is based on subtle structured reusable components. This latter may be functions or objects, which execute certain operations, e.g. sorting, searching or displaying something [24].

Visual programming can support this approach based on components reuse, where the system can be built interactively. The prototype, for example, can be developed by creating a user interface from standard items, like text fields, buttons or menus, and then by associating components with these items [24].

2.4 Analysis and Prototyping of GUI

In recent years, user interface has gained importance and received more attention in the software development process. User interface (UI) design has become a field, which has evinced a need of new techniques to ameliorate development methodology [40].

Further the UI is not only important, because from the point of view of the user, the interface is the system, but also it constitutes one of the main criteria of acceptance of the software from user. For many applications, the UI development represents an expensive task, which could be particularly seen in the development of embedded systems, where the cost of display development is typically between 10 and 40 percent of the overall cost [40].

Several reasons have been found to denote UI design as a critical aspect of the software development. Firstly, the UI depicts the visible, and at the same time the most important, factor for end users, who are interested in high-level objectives but not in the underlying implementation. Contrary to developers, who understand the implementation but not high-level objectives, the UI for them is less important. Secondly, the UI can be difficult to change, since changes can affect documentation, training, and other activities [41].

Rapid prototyping approach has been identified as a response to the question of the need of new techniques for development methodology improvement for UI design.

"*User interface prototyping is the process of developing and refining the user interface of a product, in order to learn more about the interface before full development begins. Prototyping can be done for software (e.g. web sites), hardware (e.g. cell phones and VCRs), and even services*" [41]

User satisfaction can be significantly improved by an effective UI prototyping. This latter can additionally clarify user requirements and product scope, reduce time, resources, and rework needed for product development [41].

An evaluation of UI design early in the design cycle can be accomplished by creating a prototype for the real system. This can give the possibility to the user to experiment with the prototype with the objective to make completion and respectively refinements to the system in terms of feedback to the developers [42].

Moreover, rapid prototyping facilitates the iterative design and evaluation of the effectiveness of the UI with the intended users. This iterative approach to evaluation and modification of the UI can be performed without costly software modifications, and represents an approach for testing the usability of the UI [43].

Rapid-Prototyping of a UI provides a way to the developers of a certain application to test the UI with the intended users of the system. The users experiment and interact directly with the prototype. This latter helps the following crucial purposes to be achieved [43]:

- Testing and modification of the UI by the developers to optimize the interaction between the user and the system

- Comparison between alternative interfaces can be facilitated

- Evaluation of the interface by the system users, who can suggest changes early in the system design cycle

- Determination of the correct specification of the original system concept early in the development cycle

Applying analysis and prototyping methodology with respect to the graphical user interface early in the development life cycle aims to produce the most convenient interface to the end user. Another critical result of early analysis and prototyping is the information that early analysis provides. This information helps the analyst, for example, to improve the usability concerning the user interface [44].

The benefits of early analysis and prototyping of graphical user interface can be summarized as follows [44]:

- Distinguishing between critical features and those features that can be deleted or added incrementally after product release

- Discussion interface behavior with users can avoid misunderstanding and misinterpretation

- Clarifying the user interface specification before implementation

- Testing respectively improving usability by user involvement while experiencing interactively with the prototype and providing feedback to the developers

2.5 Summary

Misunderstanding of system requirements and the deficiency of an adequate requirements validation has been identified as a major cause of system failure and customer dissatisfaction. To patch these errors while or after coding phase has been found as one of most costly task overall development costs [26].

Due to the fact that requirements evolve over time, software lifecycle models that exploit the evolutionary nature of requirements could be a solution to the requirements problems. As prototyping paradigms possess the ability to bypass the evolutionary property of requirements, they have been identified as important approaches to early requirements validation [26].

There are two major prototyping approaches, which are: evolutionary prototyping and throw-away prototyping.

By throw-away prototyping, the prototype is developed from an initial specification and delivered for experiment to the end user. This prototype will be modified and refined until the user will be satisfied with its functionality. Then from the prototype the system specification will be derived or the prototype, in some cases, can be the specification itself.

The evolutionary Prototyping is an approach to develop systems, whereat an initial prototype is produced. This prototype will be presented for comments to user, and then the prototype will be iteratively refined until an adequate system will be reached.

The principal reason for using prototyping approaches is to help customers and developers understand the requirements for the system. On the one hand, users can experiment with the prototype to know the system. In this way the requirement elicitation can be accomplished. On the other hand the prototype can reveal errors and omissions in the requirements. This means the requirement validation can be easily performed [24].

The benefits of prototyping can be summarized as fallows:

- Development time and overall effort can be reduced
- Development costs can be reduced
- User involvement while system development
- Quantifiable user feedback to help developers
- System implementation facilities
- Improvement of system usability and design quality

3 Modeling Techniques of Device Behavior

Applying software development techniques rigorously constitute one of the main factors to detect software requirements and design lack in early stage during development process. As mentioned in the previous chapter, this early detection and respectively correction of these defects means considerable time and cost reducing.

"*A number of rigorous techniques that target specific development phases exist. Industrial use of these techniques requires the development of appropriate integration strategies that result in cohesive sets of techniques that effectively cover the software development process*" [45]

Human and machine interaction is omnipresent in everyday life. For certain tasks or users, some interaction techniques and methods have been found to be more efficient or effective than the others [46].

By waterfall life cycle model a complete requirements specification is required before development. This means, for complex and large system, a complete requirements specification seems to be unsolvable without errors because of the frequent changes of user conceptions and specification.

As a consequence an iterative software development model, such as the prototyping approaches, by which a prototype can be created in early development phases, is needed to overcome the waterfall model lack. Such a user and task oriented development cycle is described by ISO 13407 [54] and illustrated in Figure 3.1 below [46].

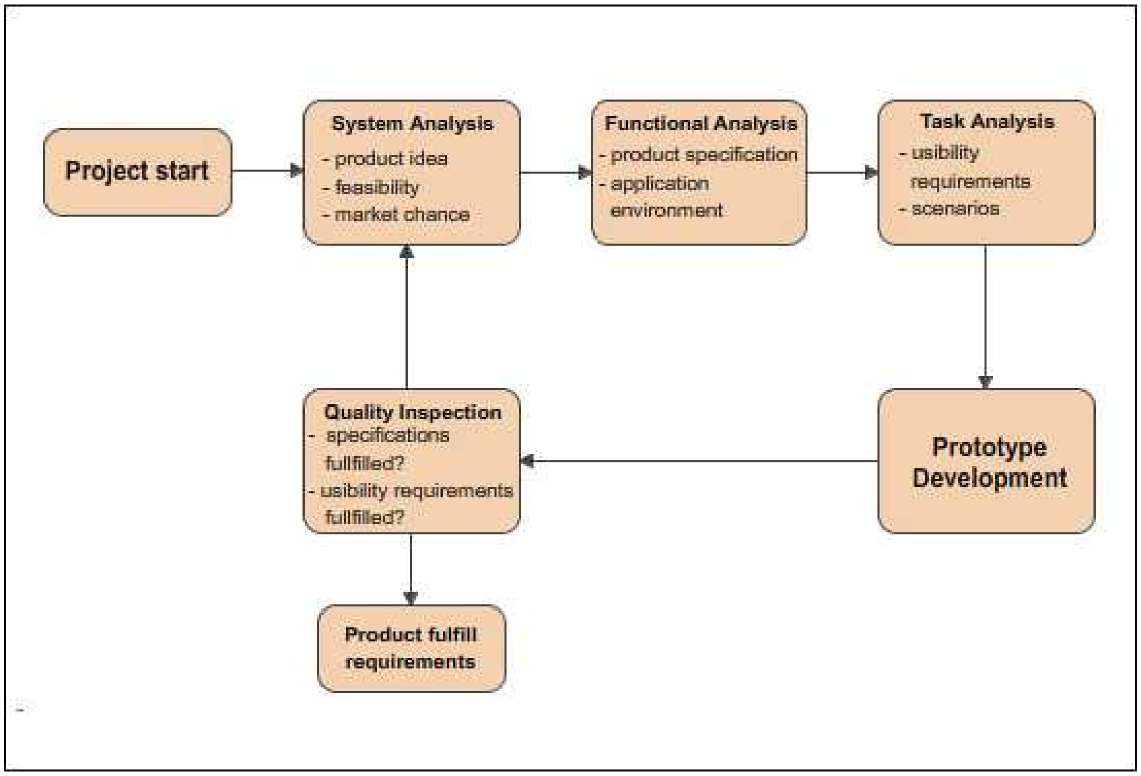


Figure 3.1: Development cycle for interactive systems

As shown in Figure 3.1, the requirements and their functionality, as well as the environment of the application should be determined and analyzed firstly. Then the usability requirements can be derived, thereby a prototype of the system can be developed. This prototype will be refined and improved until all requirements will be fulfilled [46].

Nowadays software systems become large and sophisticated. Its complexity does not cease to increase. That's why it seems to be very cumbersome and in some cases even impossible to capture the entirely requirements in a written form. Even if a document, describing such large and complex systems, has been somehow achieved, it must be certainly voluminous and huge in the way that human could not understand the entire document content. In addition reviewing such documents constitutes a durable and costly task. Thereto verifying seems to be unfeasible and sometimes impossible to accomplish if the requirements have been completely and consistently documented [46].

"*Because of the increasing complexity of real-time embedded systems and their constant need of correctness, it seems obvious to call for well-structured and formal methods dealing with software quality in order to efficiently develop such applications.*" [47]

In software engineering discipline modeling has become an activity, which can be used in every software system during the development process. Localizing relevant issues about the system at every level of abstraction needed, from all points of view, constitutes the purpose of this activity [53].

This modeling concept provides the possibility to create models of tasks, users and devices by the developers. Then connecting these models together facilitates the visualization of the relations between the various models [46].

"A model is a representation of structural or behavioral aspects of a system in a language that has a well-defined syntax, semantics, and possibly rules for analysis, inference, or proof" [49]

Models can be used in various manners during development process. On the one hand, a model can be used to enjoin properties of a system or system part to be built. This kind of models is named "prescriptive" models. On the other hand, there is another kind of models called "descriptive" models, by which a model is used to describe an existing system or system part [49].

In the case of prescriptive models, designers produce models of a system. These models introduce information, which specify the intended characteristics of the system. The information required for modeling can be obtained along the development trajectory, and will be documented in several manners [49].

Mastering with a large amount of interrelated aspects may let the complex system to be understandable. But capturing all the design aspects in a single model has a consequence that this model will be too complicated and in some cases useless [55].

"Therefore, models should be derived using specific sets of abstraction criteria, which allow one to focus on particular aspects of the system at a time" [49].

In addition, models provide the possibility to design with taking into consideration the structure and behavior of the system. Different points of view can have as a result different models, which capture different aspects of the system [53].

Further, due to modeling concept, an analytical evaluation of the system can be performed based on the resulting models. Moreover, in contrast to waterfall model, using this concept, systems can be developed faster without neglecting the quality and user satisfaction. [46].

Device behavior can be described with models, which contains details about the inner functionality of the device. These device models can be then specified with the corresponding specifications languages [46].

In software development, specification languages have become an integral part. They provide the possibility to specify or model the structure and semantics of software systems on various abstraction levels. Further they can be used in several ways, for example to analyze requirements in a descriptive way or to verify certain properties of complex systems. In addition, if appropriate tools to analyze specifications produced by these languages are available, performance estimation, automatic code and test cases generation can be performed easily [48].

Specification languages can be classified into formal and informal languages. Formal specifications not only define the semantics of a program, for example using a procedural language but also include enough information for testing and simulating the specification on a certain level. Informal specifications, on the other hand, may contain information, which supports the code generation of object oriented classes with taking into consideration the structure and properties of the specified objects. In contrast to formal specification, the execution and simulation of informal specification cannot be performed [48].

Execution can be also a criterion to classify specification languages. A specification can describe system behavior, but without defining the implementation of the behavior. Executable specifications, on the other hand, can provide mechanisms for defining states and rules to define at the end the semantics of the system, e.g. program logic. Nevertheless, they still represent abstract behavior and not the actual implementation of the system [48].

"There are many specification languages that have differing semantics and power of description at various levels of abstraction, for example: UML, LOTOS, SDL, and Promela." [48]

Device model can be specified with state charts [56]. State charts extend the finite state machine and enable hierarchical and concurrent structure organization [57]. The model of a device represents a set of states and state transitions. This model should be created by the systems designer. In some cases it represents an early form of the user manual document of the device [46].

In this chapter state charts will be introduced. It will be explained through examples how to use state charts to describe device behavior while interaction with the user. For this purpose, UML and SDL state charts have been chosen and will be presented.

3.1 What is a State chart?

A state chart is a diagram (see Figure 3.2) representing an extended state machine, which is a graph that consists of states and state transitions. A state can contain a list of internal transitions. Internal transitions, themselves, consist of internal actions or activities, which should be performed by the state machine while the machine is located in that corresponding state. An event is some “occurrence”, which can trigger a state transition. Events can be either the change of some boolean value, the expiration of a timeout, an operation call or a signal. Transitions are triggered by events. A transition is a relation between two states (let’s say state A and state B). This transition defines that, whenever the state machine is in “State A” and the event “Event 1” that triggers the transition is processed, the state machine moves to “State B”, as shown in Figure 3.2. Just one event is evaluated at a time. This event is either discarded in the case that, it does not trigger any transition, or it triggers only one transition [49].

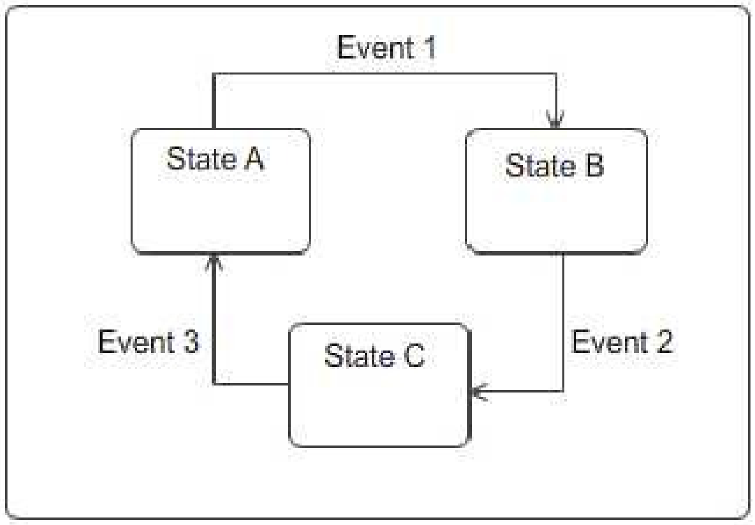


Figure 3.2: A State chart diagram

State charts can be used to describe the behavior of a device while user interaction. It is a graphical characterization of the device behavior. Each state represents a different context for device behaviors, such as an OFF and ON state (see Figure 3.6). For example, in case of a CD-Player, pushing the button “STOP”, when the device is switched on, has another effect than when the device is switched off.

State charts are suitable to manage the design complexity of devices and to specify the behavior of complex reactive system by decomposing the whole system into smaller subsystems. Each subsystem can be then modeled with a single state chart. To retrieve the specification of the whole system, the models of the subsystems can be combined and integrated together. For example, an airplane can be seen as a set of integrated subsystems. One state chart can be used to specify the pneumatics system, whereas another state chart can describe the hydraulics system. In addition, state charts provide an excellent documentation tool for the device and can also serve as powerful communication tools between the members of the developer team to discuss the relationship between the subsystems [61].

3.2 UML State charts

The Unified Modeling Language (UML) state chart diagram constitutes an important part of the standard UML language, which is highly expressive hierarchical modeling language with well defined syntax [60].

Originally, state chart diagrams were developed by David Harel [57] in the 1980's. Recently, the notation of state charts has been adopted from Harel's original version into the UML with the addition of the object oriented features [60].

UML state charts, which extend the finite state machine with terms of hierarchy and concurrency in the structure organization, are a highly expressive visual language and are typically used to model and specify the dynamic behavior of complex systems [58] [59].

The UML state chart is a diagram, which consists of discrete states and transitions between them. Transitions may contain internal actions or activities, which should be performed by the state machines. In the following subsections some symbols of UML state chart will be described briefly.

3.2.1 States

States represent various contexts in which system behavior happens, in our case those of a device. .
On a state chart diagram, states are denoted by a rounded square symbol.
States have status. This means that they can be either in an active or inactive state. For example, when a state is active, the system or device is said to be in that state.

UML states can be simple, final or composite. Simple states, as shown in Figure 3.3, are the basic constructs. The meaning of "final" is that the state does not contain other state constructs within it. In contrast to final states, composite states contain a set of states inside, as illustrated in Figure 3.4. Further the notation is extended by the use of the so called pseudo-states, such as initial state denoted by a black dot symbol, and history states denoted by "H" and "H*" inside a circle. An initial state represents the source of a transition, which points to the default sub-state of the composite state containing the initial state, while a history state records the most recent active state information of its containing state.

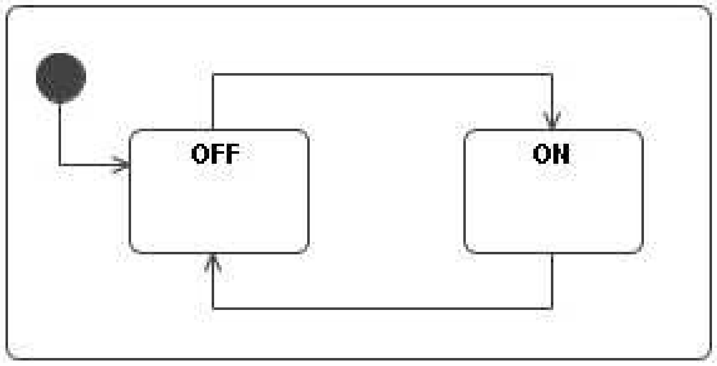


Figure 3.3: Simple States

One of the powerful and useful features of state charts is that they allow designers to group related contexts in hierarchical state. Graphically, the hierarchy of a UML state chart is depicted by the embankment relationship between states. For example, as shown in Figure 3.4, the “Active” state represents at a time a hierarchical state and a composite state, which contains the sub-states called “Stop”, “Play”.

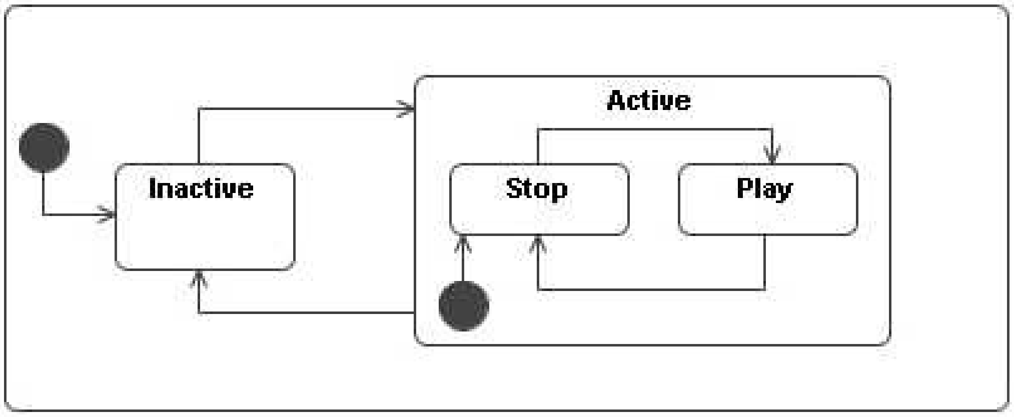


Figure 3.4: Hierarchical States

Hierarchical states can be simple or concurrent. A simple (exclusive) hierarchical state means that the state chart consists of sub-state contexts that are mutually exclusive or sequential. This means that the system can be in only one of these sub-states at a time.

Alternatively, a hierarchical state can be also concurrent (orthogonal). This means that the state contains at least two sub-machines, each of which has a set of mutually exclusive states and is separated from others by a dashed line. This type of states allows designers to model independent parts of a system. For instance, to describe the font characteristic buttons, such as boldface and italics, while modeling a word processor interface, the state charts, as shown in Figure 3.5, can be used. These buttons, independent from each other, can be in the “ON” or “OFF” position (state).

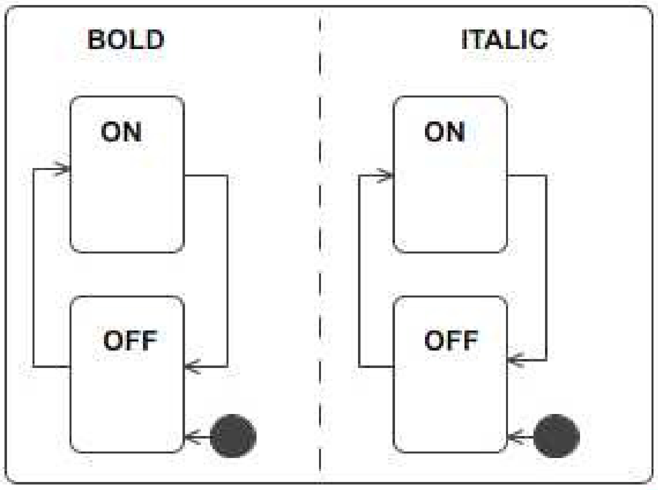


Figure 3.5: Concurrent States

The composite state that contains other states is named as “parent” of the contained sub-states, which called “children” states.

Every state chart begins with a root state that encloses the entire state machine. Some large or complex systems may require the designer to model subsystems separately. After the subsystems have been modeled, their models can be joined by creating an outermost root state around the state chart of the subsystem. In addition each sub-machine must designate a start state, which represents the first context that should be activated when the state is activated.

3.2.2 Transitions

On the state chart, transitions are denoted by directed edges between states. They can be fired when they are trigged by events. Transitions can be guarded by conditions and can specify actions or activities to be executed or events to be sent when the transition is fired.

Simple transition, as shown in Figure 3.6, is a transition between a state and a sibling state. “Sibling” means that both source state and target state are located at the same hierarchy level of the state machine.

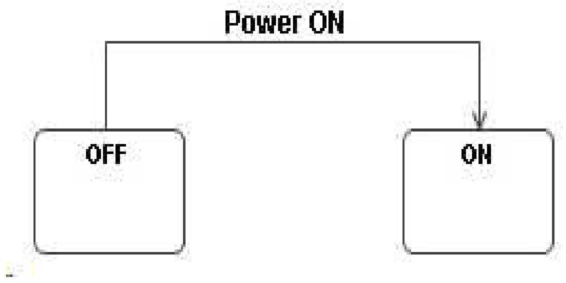


Figure 3.6: Simple Transition

In certain design scenarios, there can be a need to transition from a source state to a target state, which is not located at the same hierarchy level as the source state. For instance, as illustrated in Figure 3.7, the transition labeled with “T” from the state “Stop” to the state “Inactive” is a multilevel transition, because the source state “Stop” is deeper than the target state “Inactive” in the hierarchy of the state chart. The “Inactive” state and “Active” state are sibling states and they exist at the same level of the hierarchy.

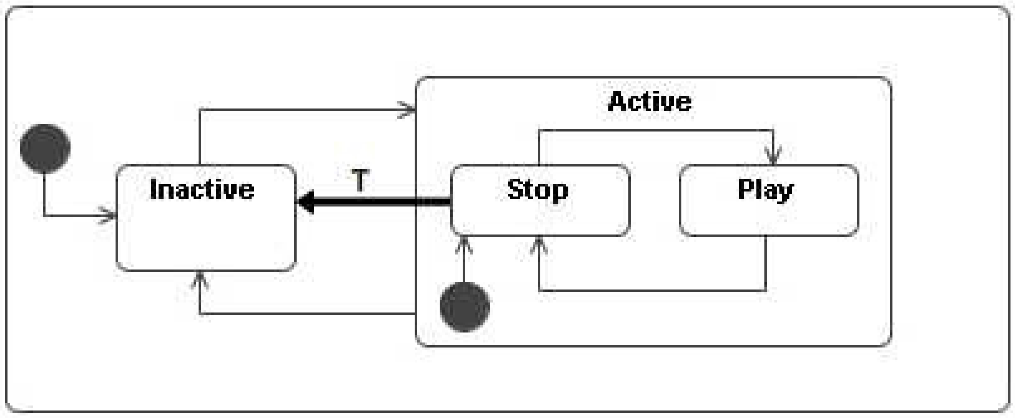


Figure 3.7: Multi-Level Transition

The UML state chart notation provides a mechanism that memorizes the sub-state last visited in the composite state. This mechanism is named the history mechanism and denoted by H or H* within a circle.

The history mechanism is useful when the device has left a composite state, done something else for a while and then needs to get back to the composite state and context. For example, as shown in Figure 3.8, a state machine represents a part of the behavior of a dishwasher. If a power cut happens while in state “Running”, it triggers the transition to the history state. This means that the execution will resume from the state, from which it has been left. For instance, if the power cut occurred while in state “Rinsing” then the state will return to “Rinsing” state.

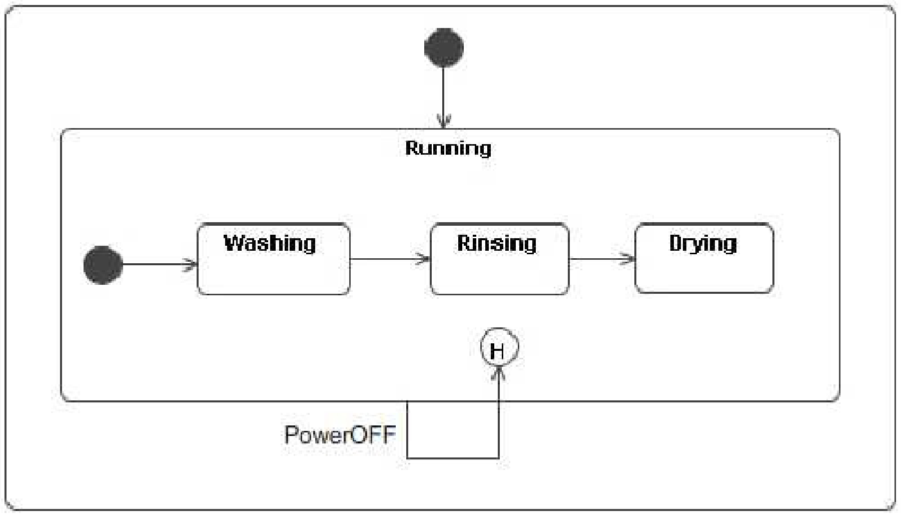


Figure 3.8: The History Mechanism

The history state in the figure above is called “Shallow history” pseudo-state. If the sub-state “Washing”, for example, is a composite state that itself contains sub-states, which should be remembered; in this case the so called “Deep history” pseudo-state must be used and is denoted by H* within a circle.

A transition to self, as shown in Figure 3.9 below, is a transition that leaves a state and return to the same state. This transition denoted by a looping transition arrow from the source state to itself.

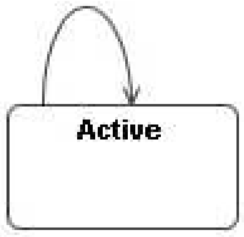


Figure 3.9: Transition to Self

Transition to self can be used, for example, to trigger actions to occur without leaving the current active state of the state machine.

3.2.3 Actions and Activities

Actions are behaviors that can be executed at certain points in a state machine. The execution of actions has been assumed to take a very short time and there is no way to interrupt them.

The syntax of the action expressions has no exact definition from the UML standard. That's why there has been found many notations of actions on the state chart in the literature. Structured English text, predicates in the way of programming languages, or metaphors can be used to denote actions.

There are three places for actions on the state chart. They can be placed on a state entry, on a transition or on a state exit. The notation of actions is separated from the event name, guard and the key words (entry, exit and do) with a slash "/".

As shown in Figure 3.10 below, the text "init()" and "print()" placed on the "IDLE" state, and "sendmsg()" placed on the transition represent the so called actions. For example, the action "init()" means any initialization mechanisms of the system when entering in the "IDLE" state.

The keyword "entry" indicates the entry action list, whereas the keyword "exit" indicates the exit action list. These actions will be then executed when entering and respectively exiting the corresponding state.

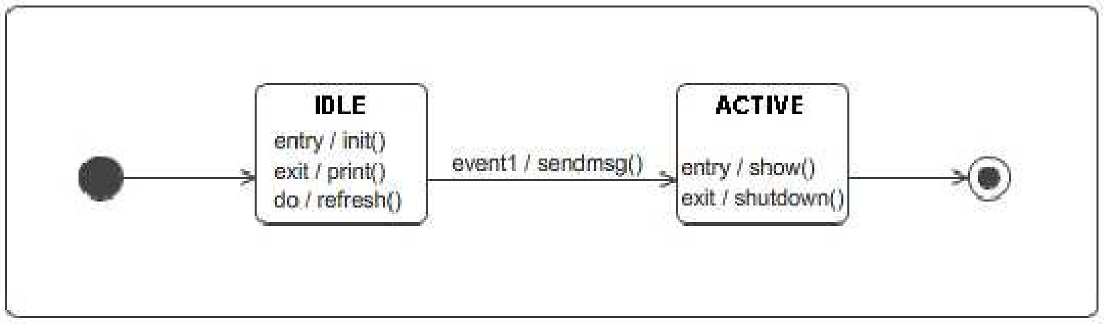


Figure 3.10: Actions and Activities

Generally, actions are always executed in the following order:

1. Exit actions of the source state
2. Transition actions
3. Entry actions of the target state

For example, as illustrated in Figure 3.10 above, if the event named “event1” occurs while the system resides in the “IDLE” state, the exit action “print()” will be first executed, the transition action “sendmsg()” will be then performed and, finally, the entry action of the target state “ACTIVE” called “show()” will be achieved.

As mentioned before, actions are non-durable and non-interruptible behaviors. In contrast to them, activities are durable and interruptible behaviors that can be performed as long as a system or object resides in a certain state. They can be, for example, long calculation or continuous control mechanisms or whatever that needs time to be executed and can be interruptible.

Activities are denoted like actions but after the “do” keyword followed by a slash within the states using the do keyword, such as in “IDLE” state on Figure 3.10. Activities begin once a state becomes active, it means when all entry actions have completed first, and terminate once a state becomes inactive but prior to the execution of the exit actions.

3.2.4 Summary

Behavioral modeling aims to describe the behavior of a system or a device with state machines. For this purpose, UML state charts have been identified to be applicative enough to describe such a behavior through graphical models. These models can be then used as documentation for the system, or, if a graph-interpreter tool is already available, to generate code automatically.

UML state chart, which extends the finite state machine, is a graph consisting of states and transitions. Its notation has been adopted from Harel’s original version into UML and has been extended by object oriented features. UML state chart notation provides, beside the basic constructs introduced in this chapter, other powerful symbols. Describing the entire syntax and semantic of UML state chart will surely go beyond the scope of this thesis, that’s why the references concerning UML in bibliography subsection will be highly recommended to interested reader.

UML has now reached the version 2.0 ([63], [5]). State chart syntax and semantic are copiously extended with many constructs and symbols, some of them come from SDL state chart. This latter is the subject of the next subsection and will be then introduced.

3.3 SDL State charts

Specification and Description Language (SDL) is a graphical object oriented, formal language defined by the International Telecommunications Union-Telecommunications Standardization Sector (ITU-T) [66] as recommendation Z.100 [62] [64]. This language is intended for the specification of complex event-driven and interactive applications that have many activities, which communicate concurrently using asynchronous discrete signals [48] [64] [65].

The start of the development of SDL was in 1972. At that time, a study group, representing many countries and notable large companies (i.e. Bellcore, Ericsson, and Motorola), began within the "**C**omité **C**onsultatif **I**nternational **T**elegraphique et **T**elephonique" (CCITT), to research on a standard specification language for the telecommunications industry. As a consequence the first version of the language was born in 1976. Then many other versions had followed in 1980, 1984, 1988, 1992, and 1996. The latest versions improved and expanded the language considerably [64].

"*Today SDL is a complete language in all senses.*" [64]

The language has been identified to be able to describe not only the structure and data but also the behavior of real-time and distributed communicating systems. It has an intuitive graphic syntax that makes it easy to understand specifications and designs written with SDL. SDL is intuitive in the way, that not involved people during design or development can obtain an overview of a system's structure and behavior quickly and easily, even to those people who have little knowledge of the language [64][65].

SDL provides a practical way to specify systems with a set of extended finite state machines (EFSM) which communicate with each other and run concurrently [64][65].

An SDL system consists of the following components [64]:

- System-, block-, process- and procedure structure hierarchy
- Communication channels, called also signal routes, and communication signal with optional parameters
- Behavior described in processes
- Data (Abstract data types ADT)
- Inheritance (OO concepts)

System, blocks, processes and procedures represent the main hierarchical levels of SDL. Static descriptions of the system structure can be done in the system and block hierarchy. The dynamic behavior of the SDL system can be described and specified in the processes hierarchy [64].

The SDL specification of a system is a set of diagrams. Each diagram has one or more presentation "pages". Each page must have a name and number as identifier, and should contain a frame and a diagram. This diagram should have a header, which is placed in the top left corner and indicates the kind and identity of the item that has been described by the diagram itself [65].

The process diagram represents the graphical description of the system behavior. This diagram consists of discrete states and states transitions. Transitions may contain actions or activities, which should be executed by the process. The basic constructs and symbols of this diagram will be introduced in the next following subsections.

3.3.1 States, Start and Stop Symbols

As mentioned before, states represent various contexts in which system behaviors occur.
In a process diagram, states are denoted by a rounded square symbol.
States have status. They can be either in an active or inactive status and when a state is active, the system is said to be in that state. States can be simple or composite. In both cases, states must be denoted by a state symbols. The composite state represents a state machine that should be drawn in a separate diagram. In this way, it means drawing state machines in separate diagrams, a hierarchy can be warranted. For example, as illustrated in Figure 3.23 and Figure 3.24, "ACTIVE_STATE" state is a composite one.

State symbol must be labeled with a unique name "<STATE NAME>", as shown in Figure 3.12 (a). When a state is labeled with "-", as illustrated in Figure 3.12 (b), this means that the source state of a transition and its target state are the same. The notation "-" is only an abbreviation that has a simplifying purpose for designers while drawing the diagram.

The label "_*", as shown in Figure 3.12 (c), denotes the so called "history" state.
The history mechanism in SDL is the same as in UML, and means that the last visited sub-state in a composite state will be memorized. The asterisk label "*" means all states.

Every process diagram must begin with start symbol (see Figure 3.11 (a)). This symbol represents the first execution point of a process. The transition between the start symbol and the next first state is the first thing, which must be done after starting the process.

The symbol shown in Figure 3.11 (b) is a start symbol that is specific to a procedure diagram and indicates, in a manner, the entry point of a procedure.

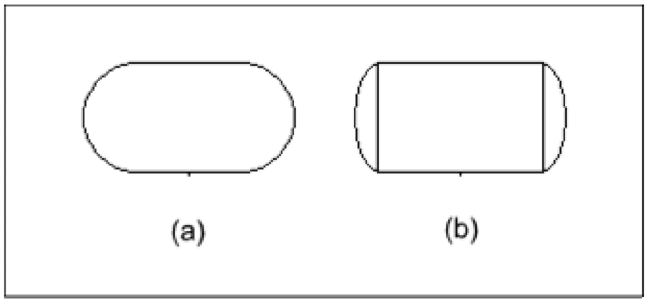


Figure 3.11: Start Symbols

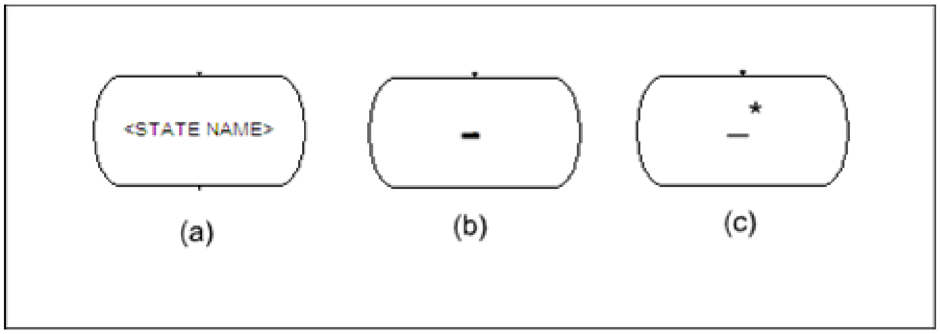


Figure 3.12: State Symbols

As processes start, they must stop sometime. That's why in a process diagram a stop symbol can be drawn. This symbol is shown in Figure 3.13 (a) and means that the process has terminated. Analogously when a procedure starts, its execution must terminate and may return a value. This is denoted by the symbol shown in Figure 3.13 (b) below, which is called "Procedure return". This symbol is normally specific to a procedure diagram and indicates that the procedure has terminated.

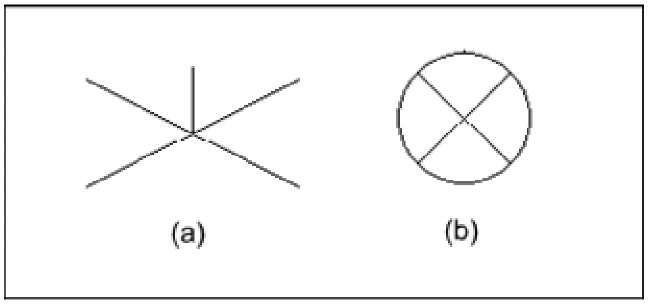


Figure 3.13: Stop Symbols

3.3.2 Transitions

As mentioned before, an SDL system is a set of communicating EFSM by signals. A process diagram represents such an EFSM that consists of states and transitions.

Unlike UML state charts, transitions in a process diagram are not denoted by a single symbol. Transitions are structures containing a set of consecutively symbols that are placed vertical. In other words, transitions must be read from top to down and not from left to right.
Only one transition must be executed completely at a time before another transition can start.

A transition in a process or procedure diagram begins always with a state symbol, immediately followed by an input symbols and some other syntactically predetermined symbols[1]. Except these symbols, no other symbol is allowed to be placed after a state symbol. The symbols, after the allowed symbols, are called transition body. The end of a transition is denoted by either a next state, or a stop symbol or a return symbol in case of procedure diagram. Some other also predetermined symbols[2] can denote the end of a transition.

[1] These predetermined symbols are: a spontaneous transition, priority input, continuous signal and save [4]
[2] For example: a join symbol.

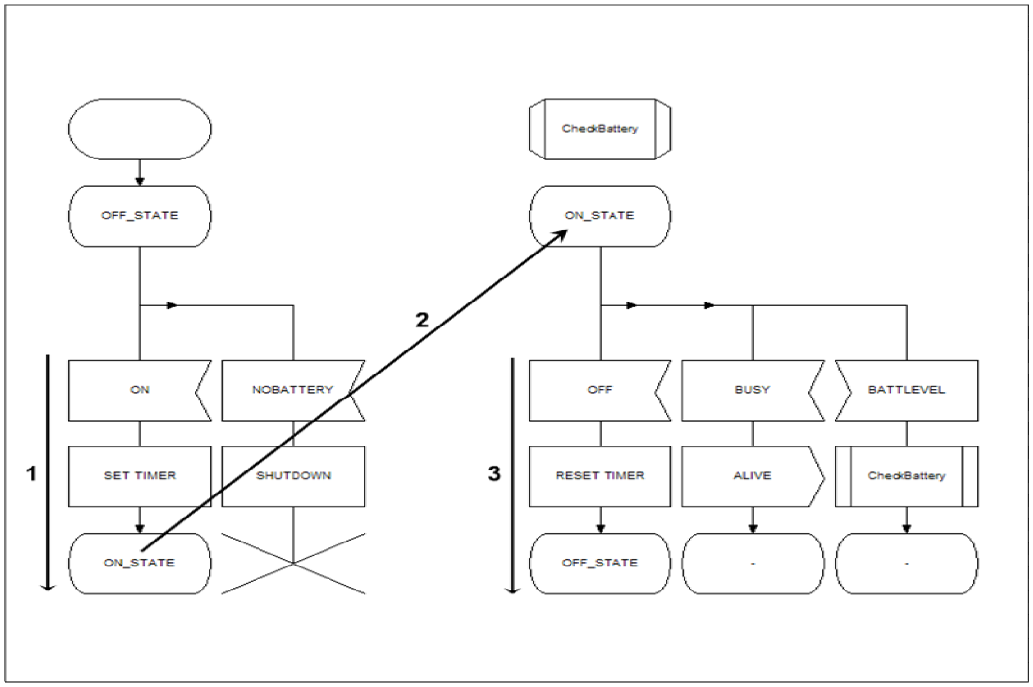


Figure 3.14: Example of a Transition Order

The Figure 3.14 above illustrates how a transition order occurs. First, the transition between the start symbol and the next first state named “OFF_STATE” will be executed after the process starts. The process is now in “OFF_STATE” and waits for incoming inputs or signals. Only the message inputs called “ON” and “NOBATTERY” can affect the process to transition. Let’s suppose the input named “ON” is coming before than “NOBATTERY” as the process is in “OFF_STATE” state. Then the transition will occur (along arrow 1) as follows:

- The action denoted by a task symbol will be executed. The task symbol, in this case, represents a timer that has been set
- The process switched the state to “ON_STATE”

As the process is in state “ON_STATE”, the diagram should be read now from the “ON_STATE” symbol as arrow 2[3] indicates. When the input “OFF” comes first, the transition indicated by arrow 3 will occur. First the action “RESET TIMER” will be executed then the process switched to the state “OFF_STATE”. The diagram should be read now from “OFF_STATE” symbol anew.

In case that the process in state “ON_STATE” and the input “BATTLEVEL” comes first then the procedure named “CheckBattery” will be called and the process remains in the same state (The label of the state symbol is “-”). But if the “BUSY” input comes first while the process is in state “ON_STATE”, a signal will be sent and the process remains also in the same state. In the diagram, sending signal is denoted by the message output symbol.

[3] Arrow 2 in Figure 3.14 is not a transition. It indicates only the position from where the diagram should be read. The arrows are not part of SDL syntax. They serve only to show for SDL newbie how to read the process diagram.

In case that the process in “OFF_STATE” and the “NOBATTERY” input comes first the process will be terminated.

Message input symbols are illustrated in Figure 3.15 below. They have unique name that represents the label of the symbol. They can be right (a) or left (b). There is no semantic difference between left or right but it can indicate from where the message input is coming. The asterisk label “*”, as shown in Figure 3.15 (c) and (d), denotes all inputs.

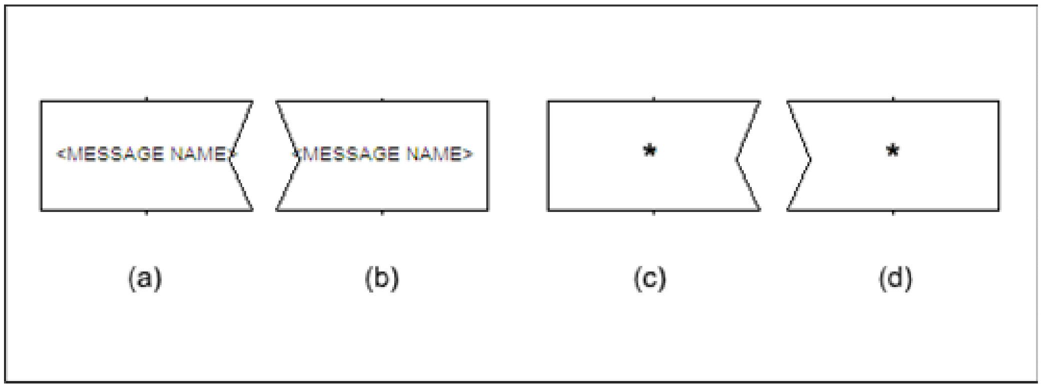


Figure 3.15: Message Input Right (Left) Symbols

3.3.3 Actions and Activities

Actions are behaviors that can be executed while a transition occurs or a state becomes active or inactive. Actions are denoted by many symbols. These symbols are placed in the body of a transition. The task symbol, shown in Figure 3.16 (b), is an action symbol and can denote many action types. For example the task symbol can denote a set or reset of a timer. It can contain also simply instructions of a certain programming language (i.e. C statements), which will be taken into consideration during code generation.

The procedure call symbol, as shown in Figure 3.16 (a), represents an action and can be used to call an SDL procedure, which must be first declared with the procedure declaration symbol (see Figure 3.16 (c)). This declaration must be placed in the process diagram and not in the transition body, and must be labeled with same label as by the procedure call symbol. In other words, when a procedure call symbol labeled with “Pro1” exists in a body of a transition, a procedure declaration symbol with the same label “Pro1” must exist in the process diagram.

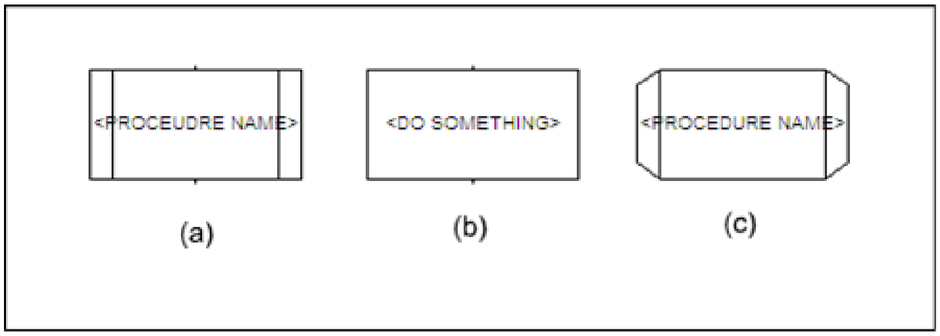


Figure 3.16: Actions Symbols

Message output symbols, right (b) or left (a) as shown on the figure below, represent also an action. They are used to exchange information between processes.

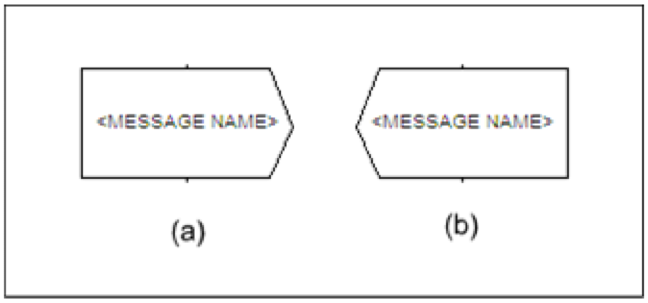


Figure 3.17: Message Output Right (Left) Symbols

All symbols placed in a transition body represent more or less actions. Some constructs are really powerful for designing purposes. For instance, process creation, decision and exceptions handling can be mentioned.

3.3.4 Summary

Specification and description language (SDL) is a graphical object oriented and formal language. SDL specifications have been identified to be unambiguous, and can be used either for documentation or for automatic test cases and code generation.

The ability to specify not only the structure and data of a system, but also its behavior constitutes the strength of SDL. In addition, the OO concepts of SDL give the user powerful tools for structuring and reuse.

The SDL specification of a system is a set of diagrams. Process diagrams have undertaken the description of the system behavior. This diagram, which represents a communicating EFSM by signals, consists of states and transitions. Process diagram notation provides, beside the basic constructs introduced in this chapter, other powerful symbols. Describing the entire syntax and semantic of SDL, as by UML, will surely go beyond the scope of this thesis, that's why the references concerning SDL in bibliography subsection will be also highly recommended to interested reader.

The language has been evolving since 1980 until now. SDL has now reached the version SDL-2000 ([4], [65]), which enhances the object modeling and code generation. SDL Diagram syntax and semantic have widely been extended with many constructs and symbols, for instance exception handling.

3.4 Example of Modeling a Simple Device Behavior with State Charts

Let's suppose that, while designing a mobile phone device, a small part of the menu navigation should be specified. The customer had expressed his wishes in a written form specification, which implies roughly the following:

Specification:
"… When the device is switched off and the user had pressed the button "ON", the device will become switched on. Then the entry display will appear. If she (user) pressed the "OK" button, she will attain the menu items. Then she can navigate up or down between them and with pressing the "OK" button the submenu, if any, of the selected item will be displayed, but when pressing the "ESC" button the device should turn back and show the entry display. Finally if the user does not operate the device for longer than 10 seconds the recent shown display should be darkened."

This small piece of a fictive specification corresponds to a typical human machine interaction. The user stimulates the device by pressing buttons, in other words, she sends signals, and the device reacts correspondingly. This specification is nothing else than a behavior description.

As mentioned before that behavior can be described with state charts, I will try now to translate the above written specification into state charts as follows:

- *" … When the device is switched off and the user had pressed the button "ON", the device will become switched on. Then the entry display will appear. …"*:

This behavior is illustrated in Figure 3.18 below. In this context two states called "Off" and "On" are necessary to describe the wished behavior. To transition between them two events or signals named "ON" and "OFF" are needed.

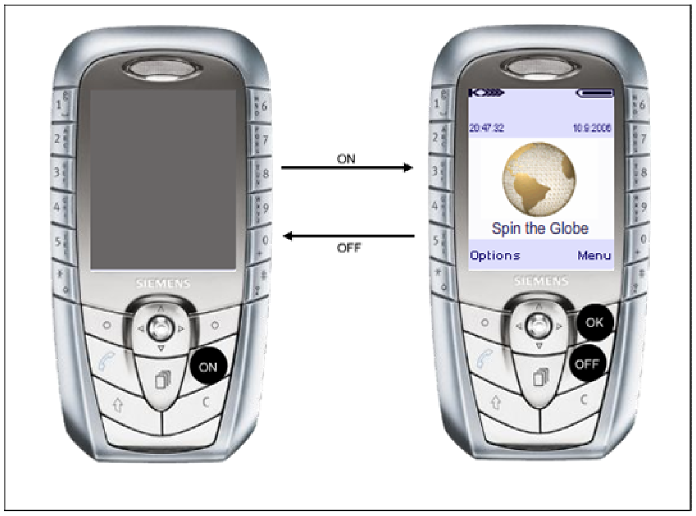


Figure 3.18: Transition from Off to On State

- *"If she (user) pressed the "OK" button, she will attain the menu items. …"* and *"… but when pressing the "ESC" button the device should turn back and show the entry display. … "*

Figure 3.19 below illustrates the above behavior. To realize these contexts, two states called "Logo" and "Navigation", and two events named "OK" and "ESC", have been chosen.

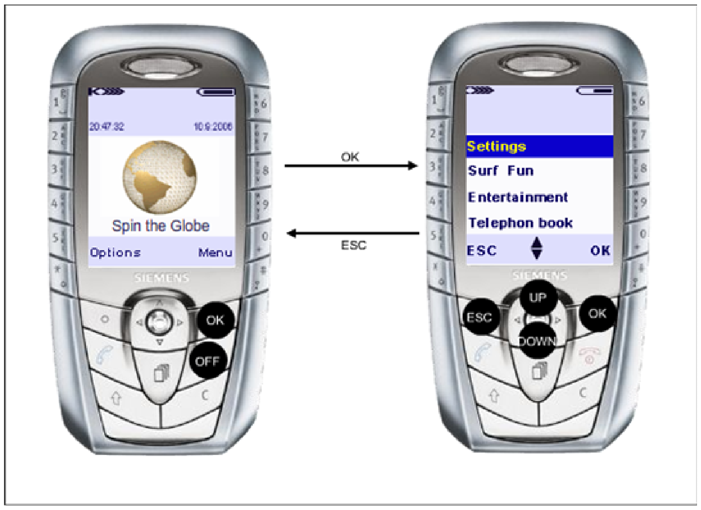


Figure 3.19: Transition from Logo to Navigation State

- *"Then she can navigate up or down between them and with pressing the "OK" button the submenu, if any, of the selected item will be displayed …"*

This behavior is illustrated in Figure 3.20. In this case two states called "Menu" and "Submenu", and two additional events named "UP" and "DOWN" for the navigation purposes, are needed to describe the behavior.

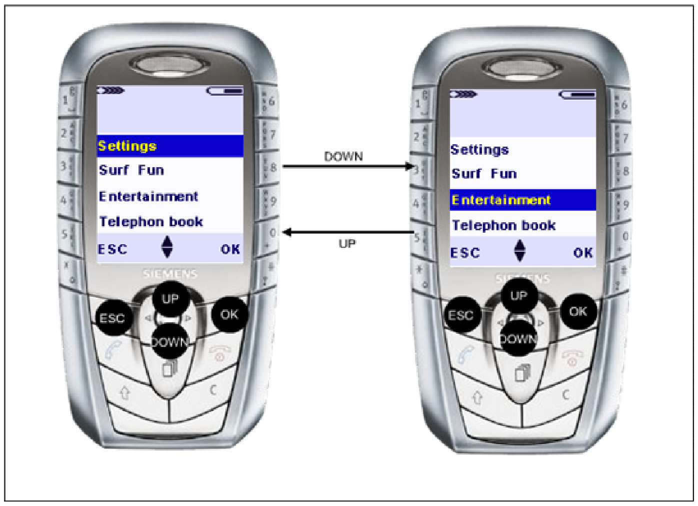


Figure 3.20: Transition from Menu State to itself

- *"Finally if the user does not operate the device for longer than 10 seconds the recent shown display should be darkened."*

This behavior has been explained in Figure 3.21 and Figure 3.22. The behavior is a typical case for history mechanism. When a device shows any lightened display, e.g. "Logo" or "Menu" display, and the user does not operate for longer than 10 seconds, then the recent display will be darkened. The word "recent" gives us a hint that the actually shown display must be memorized. To realize the whole behavior two states named "Active" and "Inactive" are needed. The behavior, which indicates that the user does not operate as mentioned in the specification, can be coded with a transition triggered by an event called "NOEVENT", which can be generated by a timer. Finally, when the device is inactive and shows a darkened display, every event can transition to active state. In other words, when the recent shown display is darkened and the user presses any button, the display will become lightened.

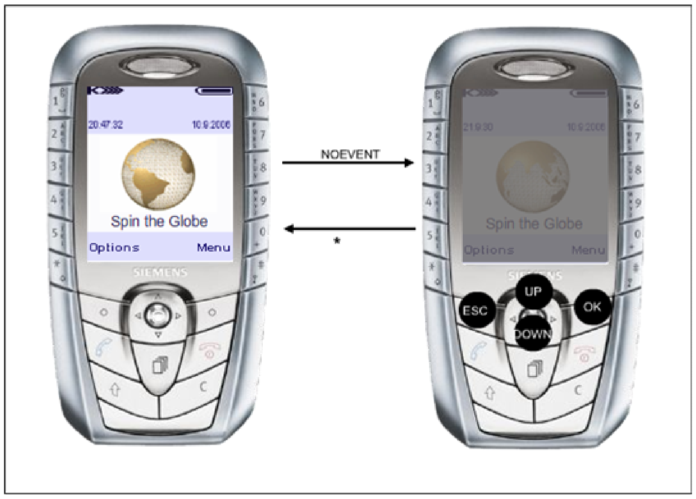


Figure 3.21: Transition from Active to Inactive State (Case 1)

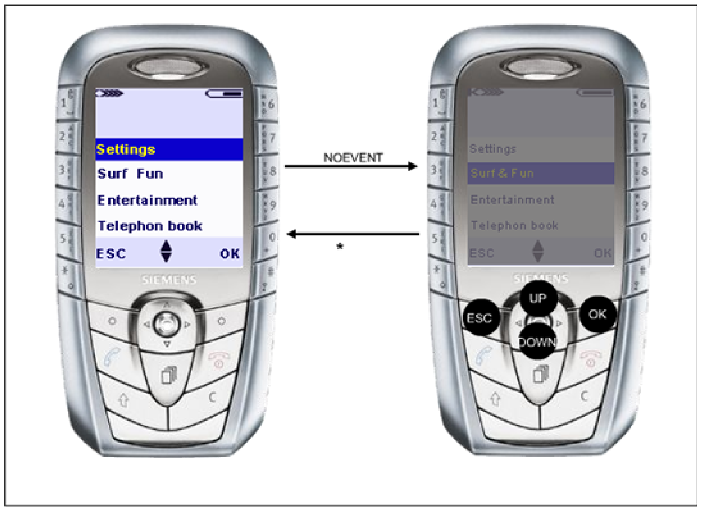


Figure 3.22: Transition from Active to Inactive State (Case 2)

Let’s summarize the states and events that are necessary to describe the desired device behavior. By now the following states respectively events have been enumerated:

- **States:** “Off’, “On", "Logo”, “Navigation”, “Menu”, “Submenu”, “Active” and “Inactive”
- **Events:** "ON", "OFF", “OK”, “ESC”, “UP”, “DOWN” and "NOEVENT"

How to wrap all these states and events in a state chart or diagram? This question can be answered as follows:

The first idea that can be got is that the “On” state should be a composite one, for the simple reason that the “logo” or “menu” or any other display can not be seen from user if the device is not switched on.

As a result of this idea, the simple state named “Off ” and the composite state named “On” can be drawn. The transitions between them can be triggered by “ON” und “OFF” events, which execute the “SetTimer()” respectively “StopTimer()” actions. The “StopTimer()” action stops a started timer. While the “SetTimer()” action starts a timer for 10 seconds.

When the timer elapses, the event “NOEVENT” will be generated and sent to the device. In this way we could cover the specification requirement that said:

“Finally if the user does not operate the device for longer than 10 seconds the recent shown display should be darkened.”

As the “On” state is a composite state, it must contain a set of states and one of them must be a default start state. Therefore within it two states can be drawn, namely “Active” and “Inactive” states. These states should describe the contexts of lightened respectively darkened displays. The transition from “Active” to “Inactive” state is triggered by the event “NOEVENT” generated by the timer. Inversely, it means from “Inactive” to “Active”, the transition is triggered by any event and execute “SetTimer()” action anew. It means when the user presses any button while the device shows a darkened display, the display will be lightened.

The “Active” state describes the context of lightened displays. For the reason that the device can have many lightened displays, one of them can be shown where appropriate, a history mechanism is in such situation necessary to guarantee the transition between “Inactive” and “Active” state. That’s why the “Active” state must be a composite one and contains, besides to the history state, the “Logo” and “Navigation” state.

The “Logo” state describes the context of showing the first entry display after the device has been switched on. While the “Navigation” describes the context when the user had attained the main menu display and navigates between the menu items. The “Navigation” state should be a composite state containing the “Menu” and “Submenu” state.

The transitions between “Logo” and “Navigation” states can be triggered by “OK” and “ESC” events and execute the “ResetTimer()” action. This action stops the already started timer and starts it anew.

The transitions from “Menu” states to self, which can be triggered by “UP” and “DOWN” events, describes the context of navigation between the main menu items and executes the action “ResetTimer()”. While the transitions between “Menu” and “Submenu” state, which can be triggered by “OK” and “ESC” events, can describe the context when the user has selected a menu item and pressed the “OK” button.

I have to mention that the small piece of the specification has not clarified what should be displayed when a menu item has been selected. It means, what is under the menu item can be several things like another menu rows, or SMS editor or an address book or whatever. Therefore, for simplicity purposes, the state submenu remains a simple state which can be extended with the desired behavior by being composite state, which may contain a state machine drawing in a separate diagram.

3.4.1 With UML State Charts

The result of the thoughts and preliminary considerations trying to translate the written specification into state charts can be seen in Figure 3.23 below. This figure represents an UML state chart of the small part of the menu navigation specification in page 47.

I have to mention that this suggestion to draw the state chart is not the only one.
Of course, the state chart can be drawn in other way, but for our purposes this state chart seems to cover the specification requirements.

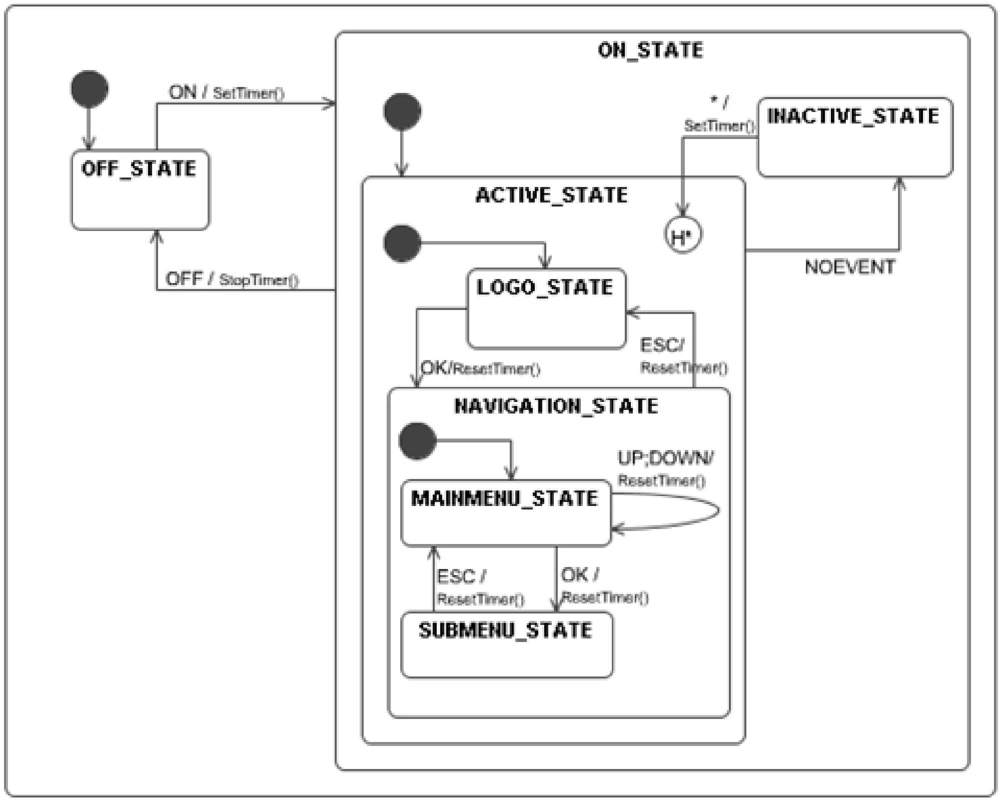


Figure 3.23: Device Behavior Model with UML State Chart

3.4.2 With SDL State Charts

Analogously to UML, the result of the same thoughts mentioned previously that try to translate the written specification into state charts can be seen in Figure 3.24, Figure 3.25, Figure 3.26 and Figure 3.27 below.

These figures represent a set of SDL state charts of the small part of the menu navigation specification in page 47.

As mentioned before, this suggestion to draw the state charts is not the only one.
Of course, the state charts can be drawn in other way, but for our purposes this state chart seems to cover the specification requirements.

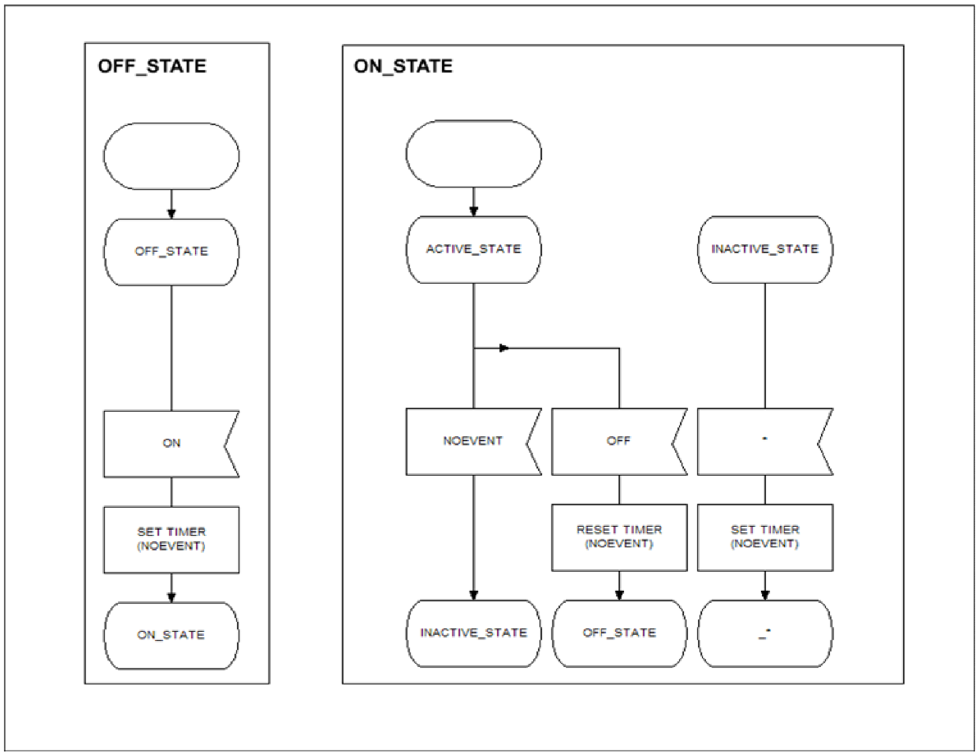


Figure 3.24: Device Behavior Model with SDL state charts (Part 1)

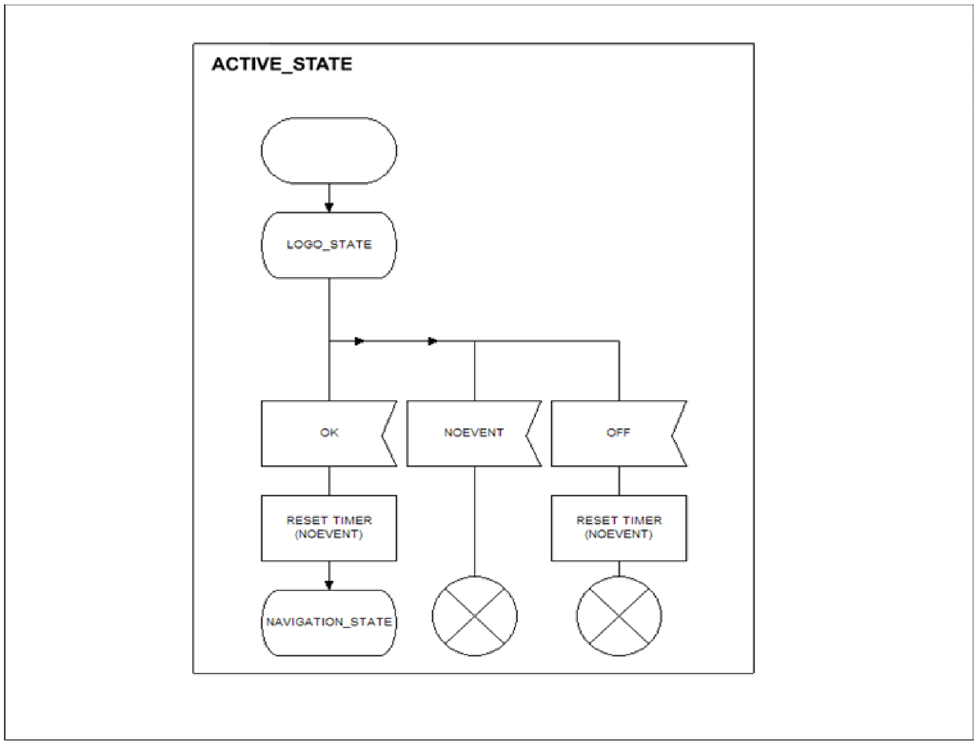


Figure 3.25: Device Behavior Model with SDL state charts (Part 2)

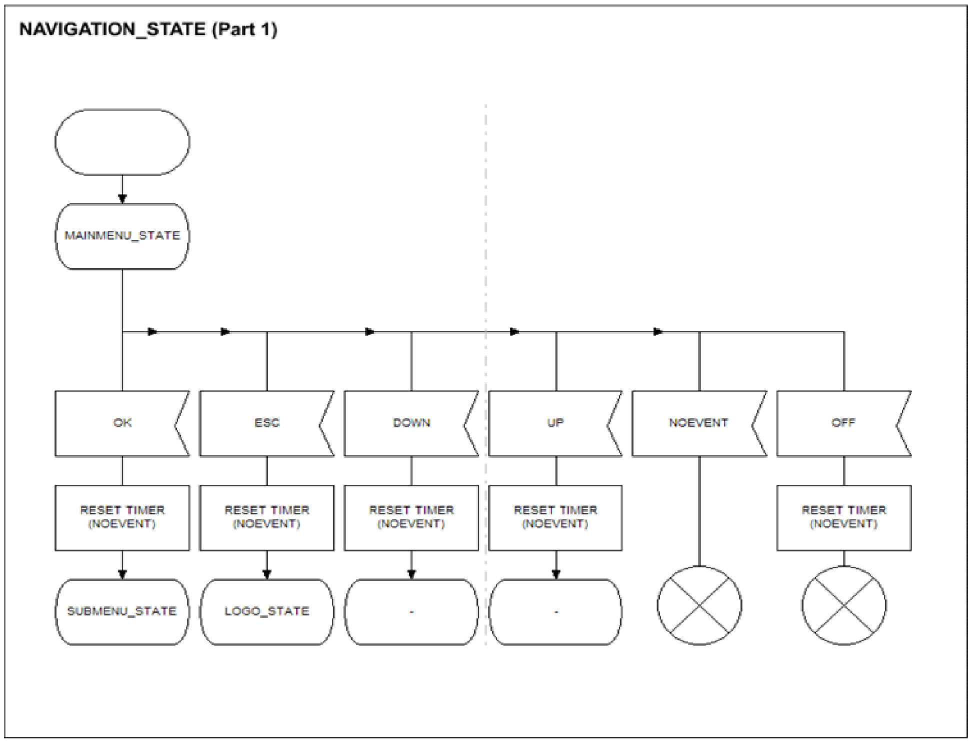


Figure 3.26: Device Behavior Model with SDL state charts (Part 3)

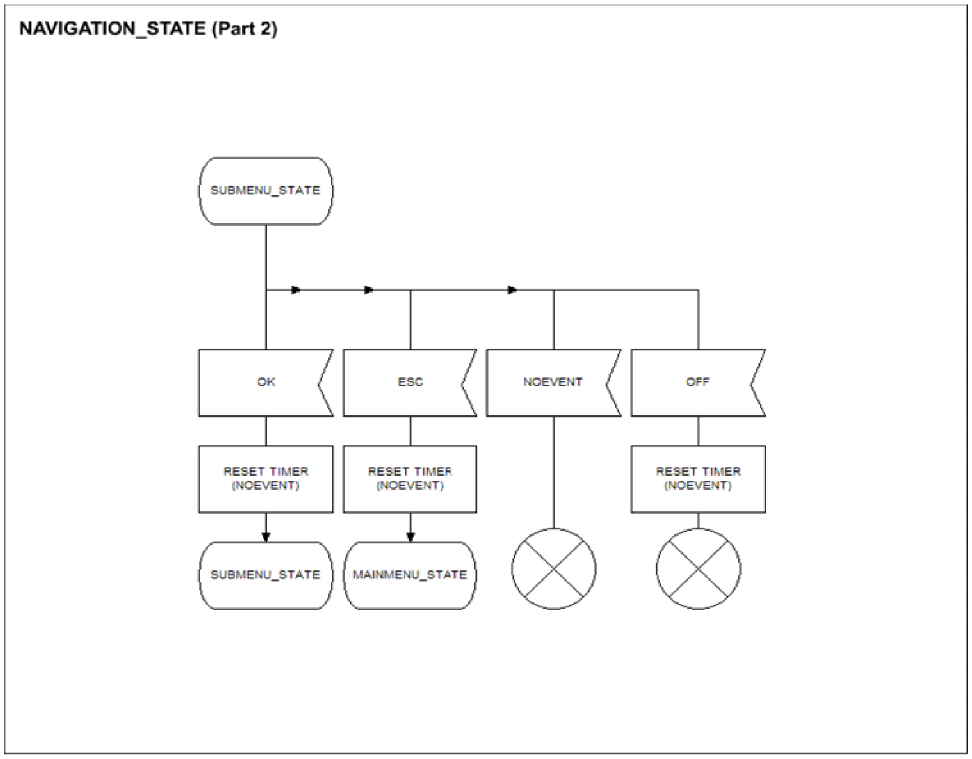


Figure 3.27: Device Behavior Model with SDL state charts (Part 4)

3.5 Summary

Behavioral modeling aims to describe the behavior of a device. This behavior can be described with models, which contain details about the inner functionality of the device, and can be specified with the corresponding specifications languages.

UML or SDL state chart can be used to realize these models and describe the behavior of a device while user interaction. It extends the finite state machine and represents a graphical characterization of the device behavior. Each state represents a different context for device behaviors.

These models can then be used as documentation for the device, or, if a graph-interpreter tool is already available, to generate code or test cases automatically.

There is no doubt that the key to a successful system development is the production of a complete exact system specification and design. This task requires absolutely a suitable specification language that is unambiguous, precise and provides a mechanism for analyzing the specification and provides not only a basis to verify the specification consistency and conformity to the implementation but also a methodology for supporting application generation.

"SDL has been defined to meet these demands." [64]

4 Design and Conception of GRAPE

As mentioned previously, rapid prototyping of smart devices aims to create device prototypes to capture customer requirements in order to achieve the design phase and before starting the mass production phase in the industry field.

These virtual prototypes should be fully interactive to enable users to operate with, should act, as far as possible, like the real devices and should have a realistic look. In this sense, prototypes are in terms of an animation or a simulation running as a stand alone application or web application on the desktop (PC).

It is not an obligation that the entire features of the intended device must be simulated or animated. Some approaches have been found to be able to limit prototype functionality. They can be depicted by cutting down the number of features, so that the resultant system becomes narrower and includes a thorough functionality for a few selected features, or by reducing the level of functionality, in the way that the result can be a surface layer that contains the entire user interface to a full-featured system but without underlying functionality.

Smart devices are usually composed of displays and interfaces, like buttons or touch screens, which enable interaction with the user. As mentioned previously, interaction is nothing else than a device behavior controlled by the user and caused by events. Some of the resultant actions are visual effects shown on the display or in the device periphery, such a LED[4] or an indicator lamp.

In this sense, device prototyping can be divided into two distinct parts. The first part must take care for the behaviour description and the second one should take care for the visual appearance of the device. To achieve each part, an adequate or appropriate software technique or tool should be found and chosen. For instance to describe behaviour, state charts can be used, or to design the graphical part of the device a flexible tool to create, edit, and modifying graphic objects in an easy manner should be chosen. While choosing these software tools and techniques, the main purpose of prototyping, which is reducing time and costs, must be taken into consideration and complied.

Indeed a set of techniques or tools that target specific development phases exists. Industrial use of these techniques requires the development of appropriate integration strategies that result in cohesive sets of techniques that effectively cover the prototype development process.

This set of integrated tools should support the visual construction, analyses, and transformation of device models, and the linking of these models with graphic objects across the development phases.

In this chapter, a development environment for rapid prototyping of smart devices will be introduced. This environment is the result of the use of a set of adequate software techniques and integrated tools and components, which will be then presented.

[4] LED stands for Light-Emitting Diode

4.1 Motivation

The trigger idea of designing “GRAPE” is modelling user interface behaviour with state charts.

Due to the fact that commercial rapid prototyping tools are not only expensive, because of licensing agreement and other copy right procedures, but also inflexible, the following question arises:

Is there a way to provide rapid prototyping with in-house[5] tools and minimum costs, which should be adaptable to already available in-house development process?

SICAT tools were already available as an in-house product. These tools provide not only describing system behaviours with SDL state charts, but also the possibility to generate code in many programming languages such as Assembler, CHILL, C, C++, Java and others from the SDL model. In this way we have a guarantee to model device behaviours and generate code without great effort and high costs.

But the question, which was not answered at that point of time, is how to create and design the graphical parts or objects of a device. It means which development environment should realise the graphical design part, because no tool from SICAT toolset could perform this task.

In summer 2003 the first version of GRAPE was born. This version was based on Windows Metafiles (WMF) templates [74], which are stored in a database. The Microsoft Visual Studio .NET C++ environment [75] had been integrated as an external tool. It had been chosen to achieve the graphical design part and to manage and maintain the whole device prototype project.

The bridge, connecting SICAT components with Visual Studio .NET, was a self-implemented wizard [77]. This later one creates automatically a device’s project with basic functionalities and links all the needed sources and resources of the device prototype together in a transparent way for the developer. It means with mouse clicks the developer could reach the other components from the Microsoft Visual Studio .NET solutions panel.

The developer should travel between the endpoints of the bridge to complete the implementation of the desired device as a windows standalone executable and as an ActiveX Control for web deployment purposes [76].

With travelling is meant, a developer models the device behaviours with SICAT and generates C++ source code. Then using the templates stored in a certain database, he designs the graphical parts of the device, makes refinements and implements the selected functionalities of the device with Microsoft Visual Studio .NET (see Figure 4.1).

Indeed with these integrated components mentioned above a prototype device has been created with success in a short period of time with minimum costs.

The following figure shows an overview of the first version of GRAPE and its components.

[5] With in-house is meant “Siemens Austria Company PSE KBB” department.

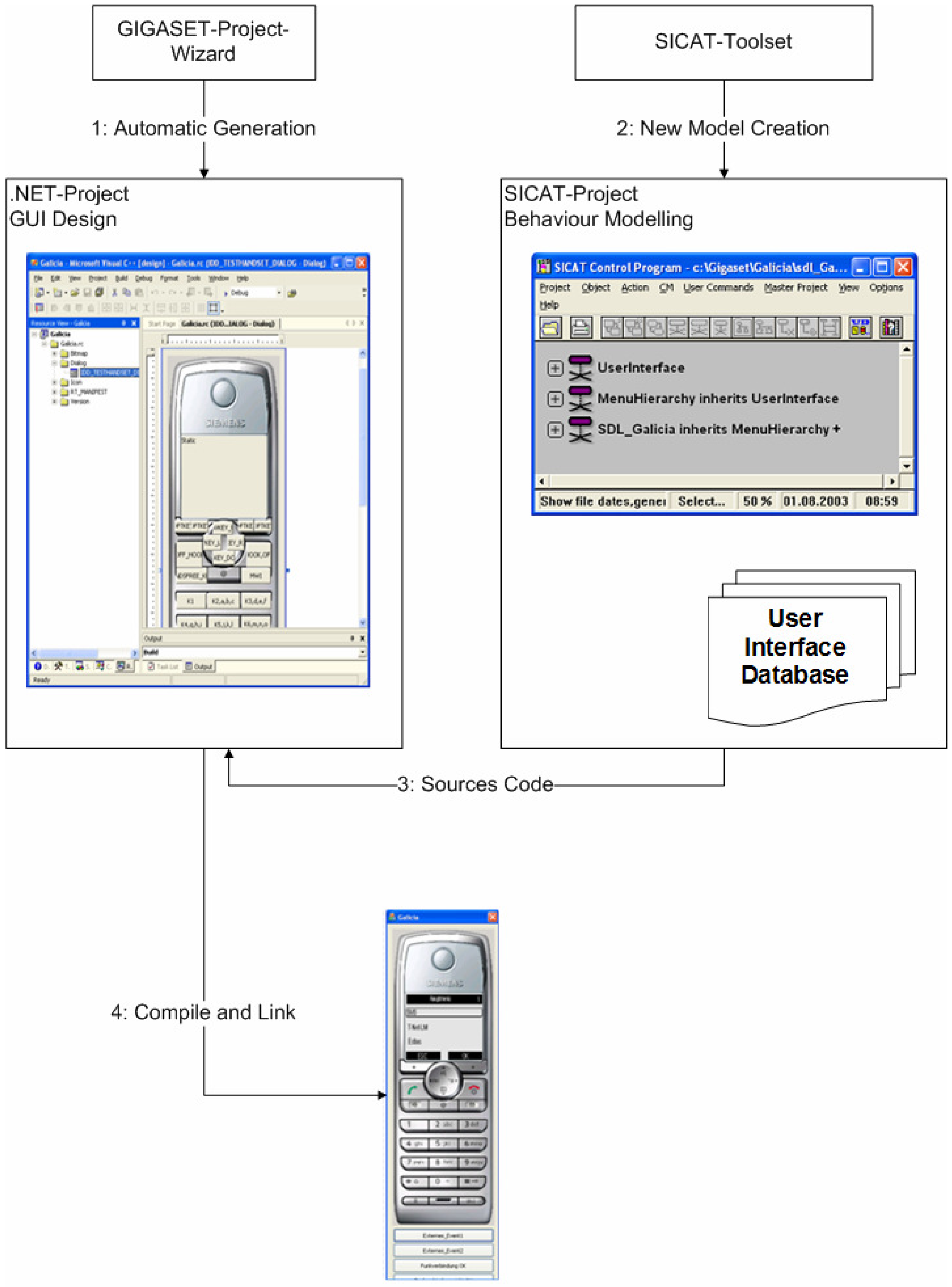

GIGASET-Project-Wizard
SICAT-Toolset
1: Automatic Generation
2: New Model Creation
.NET-Project
GUI Design
SICAT-Project
Behaviour Modelling
SICAT Control Program - c:\Gigaset\Galicia\sdl_Ga...
Project Object Action CM User Commands Master Project View Options Help
UserInterface
MenuHierarchy inherits UserInterface
SDL_Galicia inherits MenuHierarchy +
Show file dates,gener Select... 50 % 01.08.2003 08:59
User Interface Database
3: Sources Code
4: Compile and Link

Figure 4.1: GRAPE First Version Overview

The first version of “GRAPE” has beside its advantages also its disadvantages. The major disadvantage of GRAPE, at that time, is that GRAPE was tailored and specified for Siemens GIGASET phone devices [78]. In addition, graphical objects, which are based on WMF template files, were not easy to manage, to handle and to edit. They could not be reused for another prototype device family. Further ActiveX control was also impractical because of the need of a digital signature and certificate from a trusted third party to deploy it through the web.

In other words, the first version of GRAPE was not a general solution for all devices, because the resources created to produce a prototype device from a given family could not be reused for another prototype device family.

To overcome this lack and to warrant generality, it means the possibility to prototype all kinds of devices should be available, another graphical development environment should be found and should replace the WMF methodology.

For this purpose Macromedia Flash Professional environment has been chosen.
Flash provides strong visual metaphors for developing graphic objects and animations, and can import work from a variety of media development tools. These graphic objects are easy to manage, to edit and to modify.

The most fundamental element in Flash is the movie clip. Movie clips [79] are reusable pieces of animations. In the object oriented sense, they are objects that have a physical presence on the stage [80] and can be accessed programmatically through their methods and properties.

In addition, Flash provides full design control to maximize creativity and interactivity. Due to its player, Flash can run on the Web browser and on a variety of platforms like Windows, Macintosh, Unix, PDAs, and even cell phones.

ActionScript [72] is the scripting language used by Macromedia Flash. This language not only makes Flash content interactive but also provides an efficient mechanism to do things in Flash, from creating simple animations through designing complex, data-rich, interactive application interfaces.

ActionScript 2.0 [73] is formal and an object oriented programming language that supports full class inheritance and all the features that developers demand from a mature language. It offers a more flexible programming environment and superior debugging abilities that reduce coding and maintenance time.

But the question now is: How to integrate SICAT toolset and Macromedia Flash together to achieve rapid prototyping? The answer is simple, which is to provide a code generator that can transform the SDL models into ActionScript classes.

Indeed, that’s what happened and the result was a new environment of integrated tools named “GRAPE”.

4.2 What is GRAPE?

GRAPE, (**G**UI **Ra**pid **P**rototyping **E**nvironment) is a software development environment intended for rapid GUI prototyping of smart devices. With GRAPE many device prototypes can be created in easy and quick manner for many kinds of devices like:

- Cell, cordless or smart phones and PDA's
- Devices for entertainment electronics
- Medical equipments
- Devices and consoles for automotive electronics
- Instruments and control panels in automation and drives domain

In other words, GRAPE can be practically used to prototype any smart device having user interfaces consisting of displays, buttons, indicators etc. Rapid prototyping means in this case exploring ideas before investing in them.

The device prototype created by GRAPE is a photo-realistic graphical model, which looks like the final product and acts like it. The prototype is available at an early stage of the product lifecycle. A first simple-featured basic version can be enhanced and refined gradually.

4.2.1 GRAPE Features

The main features and concepts of the new version of GRAPE are:

- Photo-realistic graphics of the planned device for the skin design
- Model driven development approach for device behaviour. It means the logic is modelled with SDL state charts. Also a domain specific widget library and automatic code generation is provided
- Prototypes are generated as Flash movie clip. This later one could run as standalone application with Flash player or embedded in the web browser.
- Documentation generation of the prototype model. As mentioned the SDL model of the prototype, which is an extended state charts, can be converted into WinWord format
- Support of distributed projects and development
- Animated state chart test monitoring
- Inter-device communication can be modelled

4.2.2 Benefits using GRAPE

The most significant benefit, which GRAPE can provide for rapid prototyping, is saving time, costs and resources.

GRAPE holds many others important and tangible benefits, which will be listed as follows:

- Quick time to market
- Lower device development costs
- High quality stakeholder feedback at an early stage (requirement review)
- Enabling usability and design review (look and feel)
- Automatically generated documentation
- Prototype deployment via Internet (Flash movie clip)
- Fully customizable features (in-house tooling)

4.2.3 GRAPE References

GRAPE was successfully used for prototyping the following devices:

- GIGASET handset from Siemens (optiPoint WL2 professional)
- Chinese version of the control panel SINAMICS-AOP for SIMEA
- Digital SAT-receiver Thortsen2
- smart@phone from Siemens
- GIGASET handset (function modelling for France Telecom)

4.3 GRAPE Components

GRAPE is based on two major components. The first one called “SICAT”, which is responsible for describing the behaviour of the device, for generating code and supports the documentation of the device. The second one is “Macromedia Flash Professional 8” [13], which achieves the graphical design and creates the virtual interactive device prototype.

4.3.1 SICAT

SICAT (**S**DL **I**ntegrated **C**omputer **A**ided **T**oolset) is a development environment for SDL 92 [10]. Using SICAT, the architecture of event oriented systems and the behaviour of these systems while intercommunication can be specified and described easily.

SICAT (see Figure 4.2) is a Siemens in-house tool and was used in several IT-Technology fields like telecommunication, real-time systems and distributed systems.

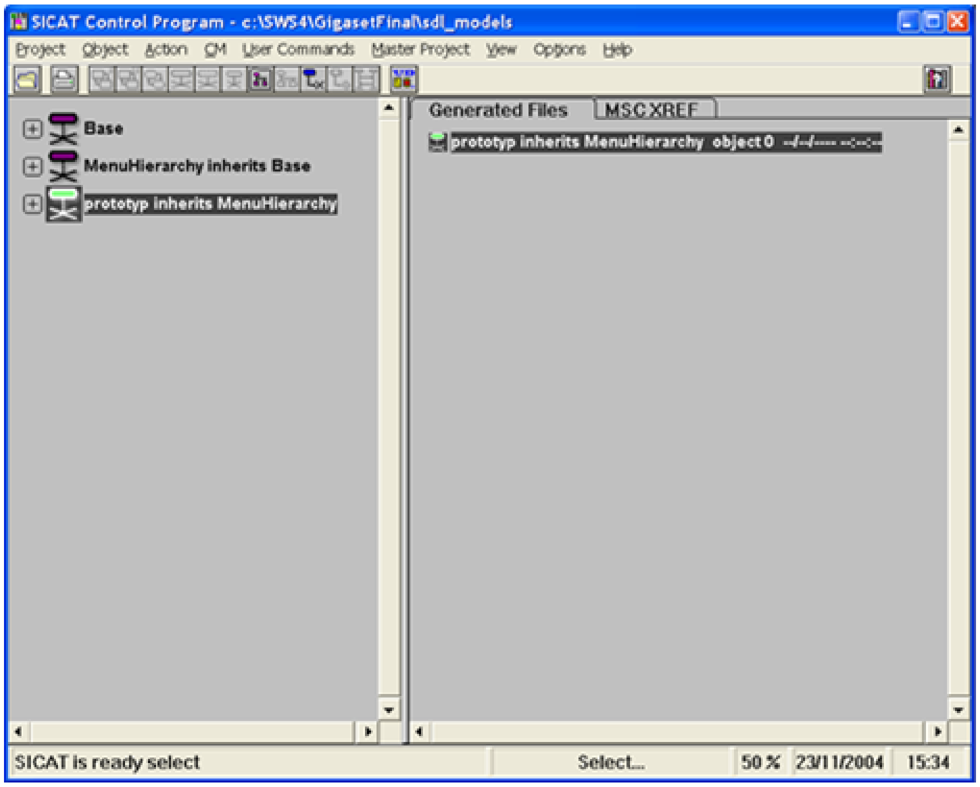

SICAT Control Program - c:\SWS4\GigasetFinal\sdl_models
Project Object Action CM User Commands Master Project View Options Help
Base
MenuHierarchy inherits Base
prototyp inherits MenuHierarchy
Generated Files
MSC XREF
prototyp inherits MenuHierarchy object 0 --/--/---- --:--:--
SICAT is ready select
Select...
50 %
23/11/2004
15:34

Figure 4.2: SICAT Development Environment (Control Program main Panel)

SICAT is a toolset. It comprises many tools. The most important ones are Process-Diagram-Editor (PD Editor), Message-Sequence-Charts-Editor (MSC Editor) and Code-Generators.

4.3.1.1 PD Editor

The PD editor (see Figure 4.3) can be used to describe the process behaviour in terms of SDL process diagrams in SDL/GR syntax [10] in accordance with ITU Recommendation Z.100 [12].

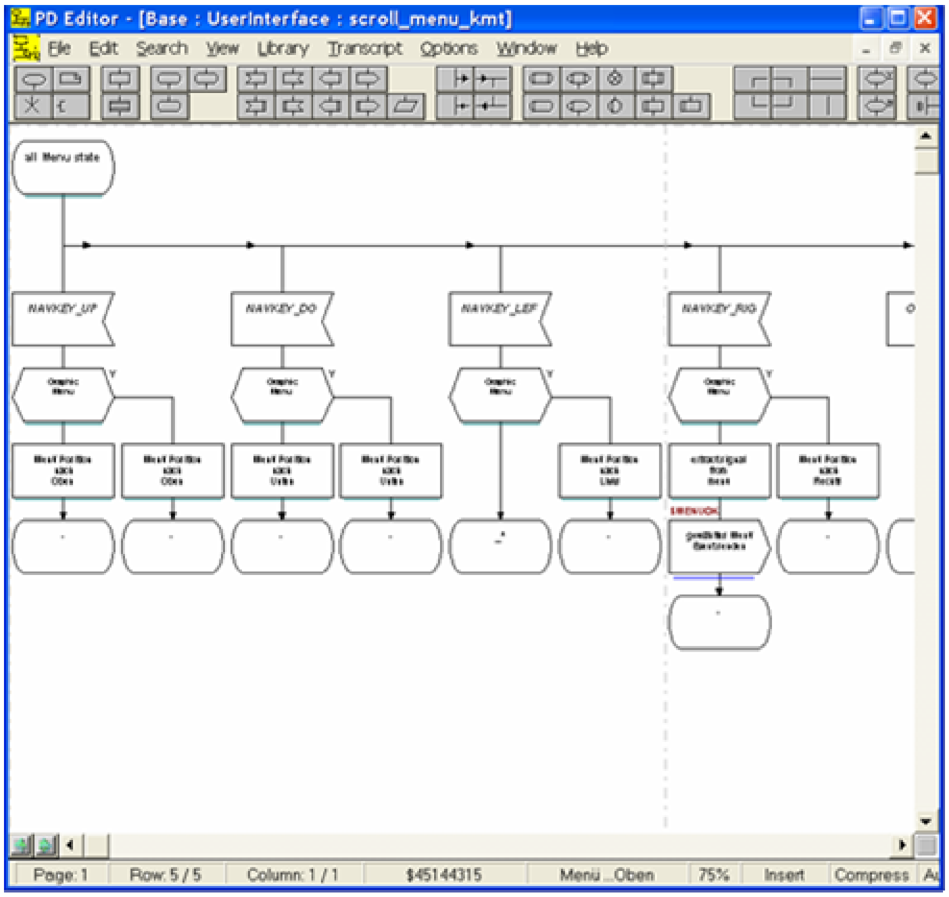


Figure 4.3: Process-Diagram Editor

4.3.1.2 MSC Editor

The MSC editor (see Figure 4.4) can be used to describe graphically and process scenarios in terms of message sequence charts according to Z.120 [12]. For example, test cases can be specified and simulated with the MSC editor.

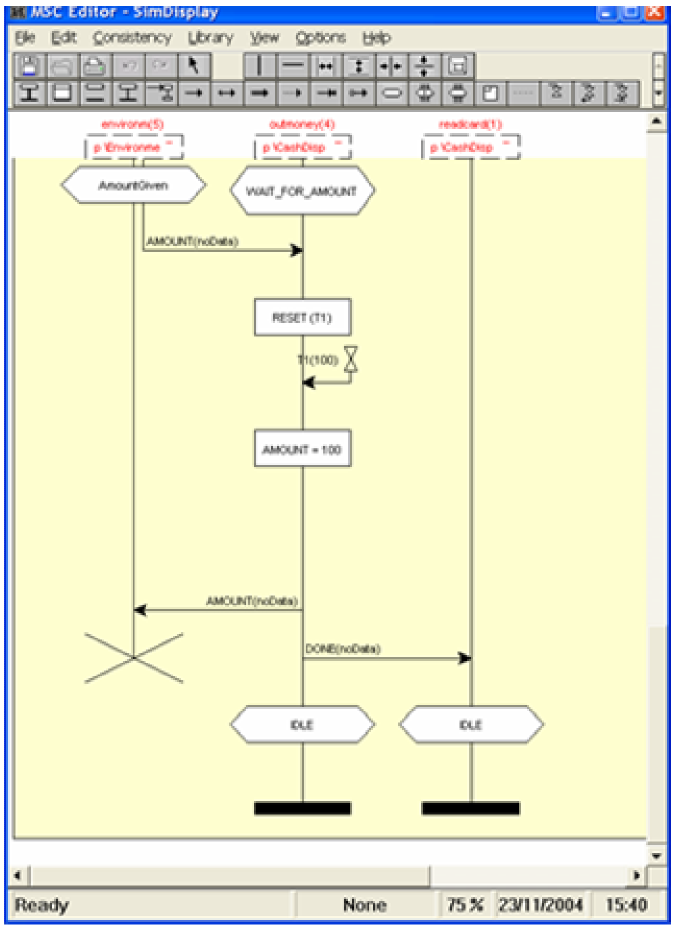


Figure 4.4: Message-Sequence-Charts Editor

4.3.1.3 Code Generators

SICAT code generator transforms an SDL process consisting of one or several process diagrams into source code.

In order to obtain a complete source module, the user must refine an SDL process diagram created during the design phase with the PD editor and add the corresponding code statements.

Data and instructions must be written in the syntax of the appropriate implementation language. The code portions are stored in the background, i.e., they are not visible in the diagram.

The basis for the code generation is the attributed tree generated by the Analyzer. The program structure is derived from the structure of the SDL diagram according to the transformation rules and completed with the code portions written by the user.

SICAT normally supports CHILL, C, C++, ASS386, JavaScript and lately ActionScript 2.0 languages. The rules for the transformation of the SDL design into the respective implementation language are described in the code generator templates. By modifying these templates, code generators for other languages can be derived easily.

4.3.2 Macromedia Flash

Macromedia Flash Professional 8 (see Figure 4.5) is the industry's most advanced authoring environment for creating interactive websites, digital experiences and mobile content [13].

With Flash Professional 8, creative professionals design and author interactive content rich with video, graphics, and animation for truly unique, engaging websites, presentations or mobile content [13].

Flash provides full design control and strong visual metaphors for developing graphic objects and animations, like movie clips, which are reusable pieces of animations and they represents objects that have a physical presence on the stage and can be accessed programmatically through their methods and properties.

Due to its player, Flash can run on the Web browser and on a variety of platforms like Windows, Macintosh, Unix, PDAs, and even cell phones.

ActionScript 2.0 is the scripting language used by Macromedia Flash. It is fully object oriented programming (OOP) language that provides several benefits to programmers, especially when creating large-scale Flash applications or presentations. OOP encourages reusability by class inheritance.

The modular nature of OOP lets the developer replace or swap components of an application without changing the application structure and the fundamental metaphors of OOP design closely mirror the natural world (objects that have internal states and can do certain things), which speeds and facilitates the application design process.

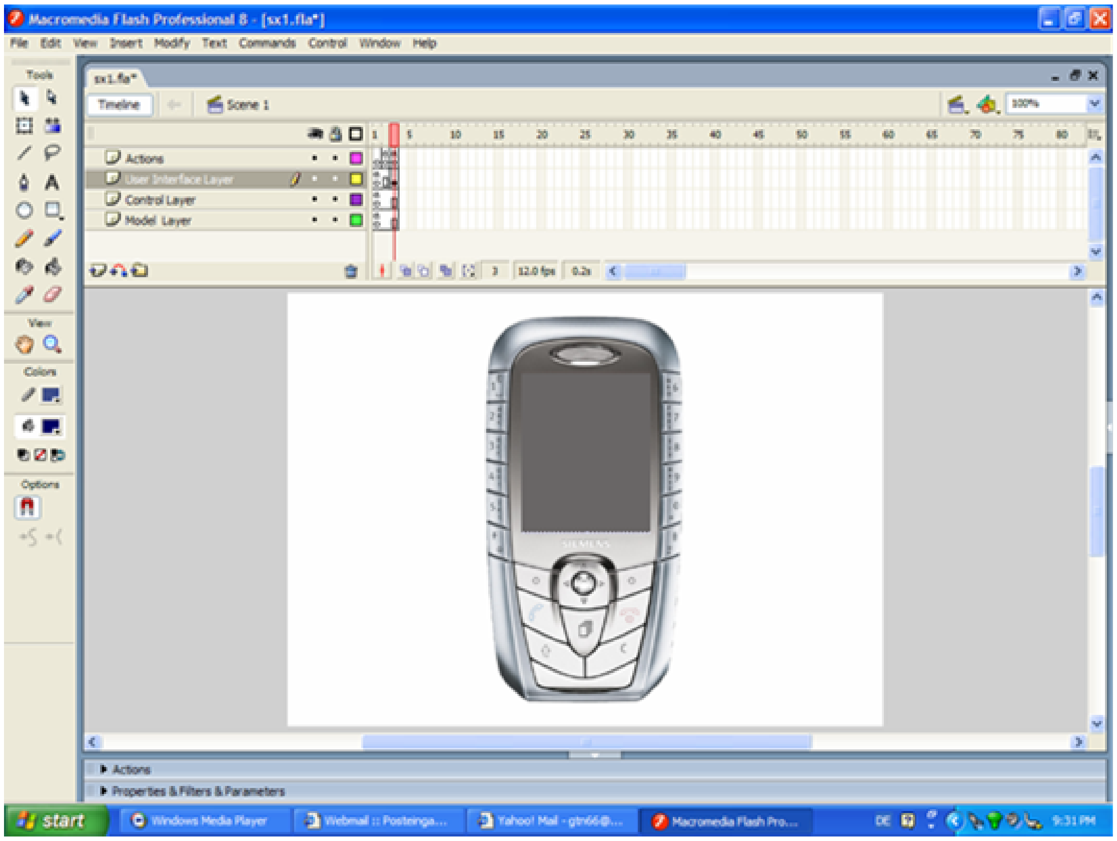


Figure 4.5: Macromedia Flash Professional 8 Environment

4.4 Summary

The use of software techniques requires the development of appropriate integration strategies that result in cohesive sets of techniques that effectively cover the development of the prototype and should be adaptable to already available in-house development process.

This set of integrated tools called GRAPE supports the visual construction, analyses, and transformation of device models, and the linking of these models with graphic objects across the development phases.

This development environment is the result of the use of a set of an adequate software techniques and integrated tools and components like SICAT toolset and Macromedia Flash Professional 8.

GRAPE not only provides rapid prototyping of any smart devices, but also supports the entire product lifecycle from product specification until computer based training (CBT), as shown in Figure 4.6 below.

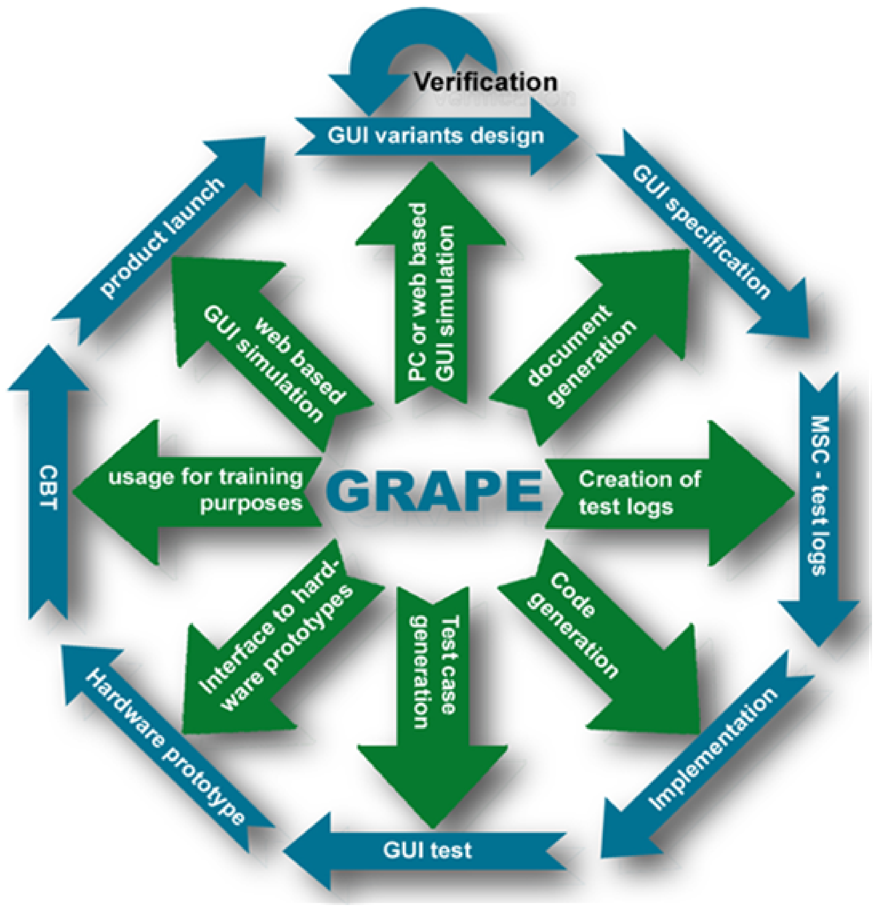


Figure 4.6: GRAPE supports the entire product lifecycle

5 Implementation of a Smart Device Using GRAPE

As mentioned in the previous chapter, prototypes of smart devices can be developed with GRAPE environment. On the one hand, the device behaviour can be specified with SICAT toolset using SDL State charts. ActionScript classes corresponding to SDL Models can be then generated automatically. In addition, these SDL models serve not only for code generation but also for documentation purposes of the device itself.

On the other hand, the visual appearance of the device and its graphical components can be designed with Macromedia Flash 8. This latter has been identified to be a powerful graphic design environment. It provides a flexible methodology to design graphic elements or objects, which are in terms of symbols in Flash terminology. The most important symbols are the so called "movie clips". Movie clips are reusable pieces of animations. With respect to object oriented principles, they are objects that have a physical presence on the stage and can be accessed programmatically through their methods and properties.

ActionScript 2.0 [81] is the scripting language used by Macromedia Flash 8. It is a formal and an object oriented programming language that supports full class inheritance and all the features that developers demand from a mature language. ActionScript has a predefined large set of classes for data types, for movie clips and other elements. With sub-classing the predefined classes, the ActionScript language can be abundantly extended. Especially by sub-classing movie clips they can contain not only written code but also graphics and animation.

In this way, the concept of component has been introduced by Macromedia Flash. A components is a custom class, which can be defined by the developer and which inherits the "MovieClip" class, whereat a movie clip with the desired items can be created and turned into a custom class. This custom sub-class itself can be inherited from other classes. In this manner the reusability can be enhanced widely.

In addition, Macromedia Flash environment provides a set of user interface components, like buttons, checkboxes, radio buttons, and so on. These base components were designed to be extended, and they provide a magnificent basis for rapid creating complex specific interface elements.

ActionScript language and the mechanism of component customization [82] make Flash content interactive, customizable and reusable. They provide an efficient way to do things in Flash rapidly from creating simple animations till designing complex, data-rich, interactive application interfaces. In addition, Flash provides full design control to maximize creativity, interactivity and components integration.

Further, developing a device prototype without a complete well thought out design is like constructing buildings on waggling fundament. This means an appropriate methodology must be applied to organize the process development of the device prototype with taking into account the main purpose of rapid prototyping and making the development tasks comfortable to implement and to be easily understandable. The applied methodology should also manage the complexity of the device development in an effective and easy way.

For this purpose a methodology for managing the complexity of device prototype development, based on SDL state charts and the UCM architecture has been chosen.

This methodology is valuable for the raison that development will become more efficient and the way of design, implementation, and documentation of the device prototype can be standardized.

At the beginning, this chapter will present a scaleable architecture, named the “UCM” architecture, for designing the device prototype and managing its complexity.

Then the transfer of SDL models into code will be explained. In other words, code generation from SDL models into ActionScript classes will be illustrated. Further, the construction of the graphic elements of the device prototype with Macromedia Flash 8 will be shown.

Finally, the implementation of the architecture will be demonstrated and the combination of all components of the prototype together will be then presented.

5.1 UCM Architecture

The UCM architecture is a "top down" approach, explained by Horrocks [2], which supports a centralized control access of the system information and facilitates the implementation and inspection of how the interface is coordinated in each context [3].

UCM stands for "**U**ser Interface - **C**ontrol Object - **M**odel", which describes a three-tiered system to conceptualize the way a software program or device can be decomposed. UCM is similar to the design pattern Model-View-Controller (MVC) [83].

The UCM architecture is a scaleable architecture based on an engineering practice. It separates the device elements into three layers [2][3], as shown in Figure 5.1 below:

- *User Interface*: A layer for user interface elements like buttons, display, etc
- *Control Object*: A layer containing the control object (in some applications there are more than one object). The control object is a mechanism that coordinates the interface elements and mediates the interface and the device internal functionality
- *Model Layer*: This layer containing processes and computations, which involve the internal functionality of a device. The model layer is not concerned with how the device behaviour is presented to the user, because that is the role of the interface

Within the device organization, the UCM architecture requires a separation between the device interface and the underlying behaviour layers [3].

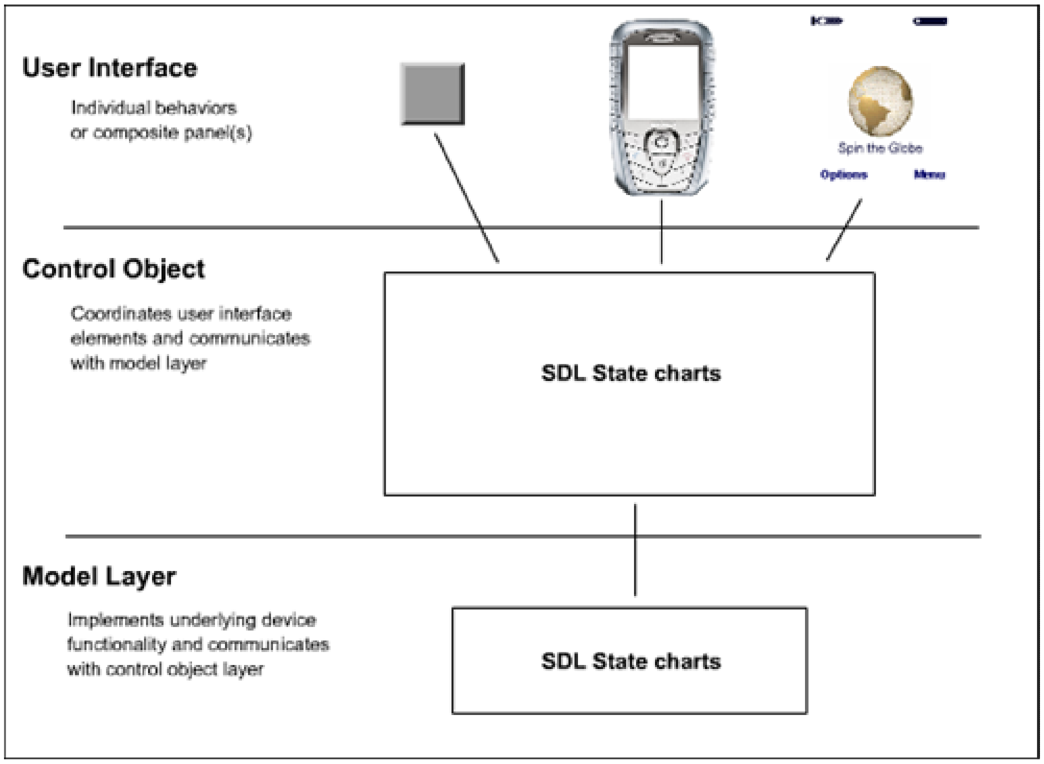

User Interface
Individual behaviors or composite panel(s)
Spin the Globe
Options
Menu
Control Object
Coordinates user interface elements and communicates with model layer
SDL State charts
Model Layer
Implements underlying device functionality and communicates with control object layer
SDL State charts

Figure 5.1: UCM Architecture Layers

A characteristic of the top down approach is that a central control object receives and handles events from a group of related interface elements. The control object in turn communicates with the underlying system model object on the model layer.

In the following the difference between the “bottom up” and the “top down” approaches will be briefly explained through an example. Let’s suppose the used approach is the “bottom up” one and the impact of a button on a mobile phone should be programmed. When the user presses the button, the button event handler must determine in which mode the phone display is, such as “displaying the entry display” or “displaying the main menu rows”, and then invoke the appropriate actions. Each button event handler (there are at least 9 buttons) must go through the same process, so that to understand the extent of what happens, for example, in the “displaying the main menu rows” a look through all button event handlers must be taken to determine what each does, if anything in this mode should be accomplished.

In the top down approach, each button event handler, which is placed in the user interface layer, simply reports the button event to the control object. The control object has to determine the context or mode of the mobile phone and then to invoke the appropriate actions [3].

The subtle difference is that in the top down approach all code for a particular context, and hence all events that can be handled, is located in a single place, whereas in the bottom up approach the code is distributed across all button event handlers [3].

In other words, by centralizing the code, the top down approach makes the coordination of the interface in each context more apparent. Further, making changes or additions to event handlers in one context do not affect the interface behaviour while the device is in different context.

By centralizing the event handling makes the design and implementation easier because the developer can immediately see all, and only those, events processed for each context.

To design each layer efficiently, SDL state charts have been chosen. In this way, the control object can keep track of the context of the device and the communication between the UCM layers can take place with sending messages and method invocation. This means that the control object can, for example, send messages on the one hand to interface elements such as setting a certain value. The interface elements are only responsible for performing their immediate function, such as button press, display darkening and so on, and inform the control object about what they did, not to take on any of the device processing by themselve.

On the other hand, the control object can send messages to the model layer objects. But only the control object layer, as shown in Figure 5.1, is allowed to communicate with the model layer. This centralizes access to model layer information and functionality, making it perfectly clear what information the interface needs, when that information is sought, an how the internal functions can communicate with the interface.

In other words, the control object layer and model layer are communicating extended state machines modelled with SDL state charts. From these models, ActionScript classes can be generated directly.

Principles of code generation will be now explained in the next following subsection.

5.2 Principles of Code Generation

The device behaviour can be specified and modelled with SDL state charts using the PD Editor of SICAT Toolset. From these models target code can be then generated.

As the transformation rules of SDL models into the respective implementation language were described in the code generator templates, code generators for other languages can be derived easily by modifying these templates accordingly. That's why an ActionScript code generator named with "Flash Code Generator" has been implemented and integrated in SICAT environment without a great outlay (see Figure 5.2 below).

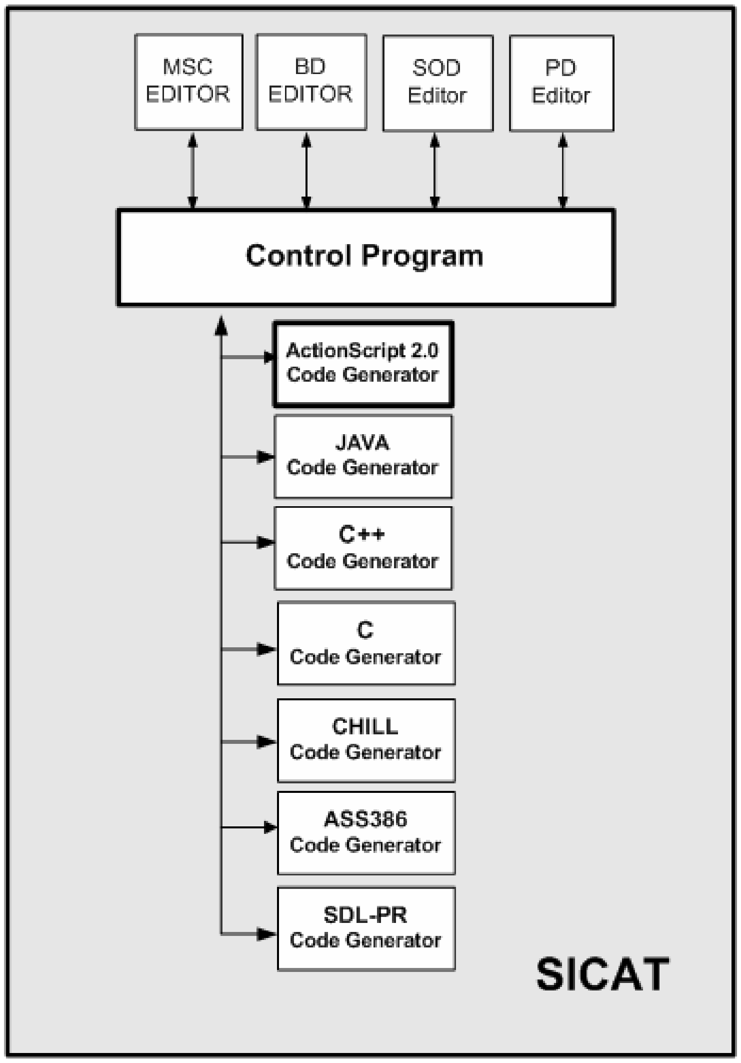


Figure 5.2: Integration of Code Generators into SICAT Toolset

After successful analysis the new Flash code generator of SICAT generates code in programming language ActionScript 2.0 from SDL process diagrams, which had been already created with the PD editor of SICAT.

Code generation can be started for an entire SDL process or for an individual process (-type) module. For each SDL process, which is mapped onto a class, an ActionScript file with ".as" extension will be then generated

The contents of the generated ActionScript file are determined by the structure of the process diagram, contents of the textual description levels (see Figure 5.4) of the individual graphic symbols and by setting the profile parameters of the code generator. Currently, the Flash code generator supports the SDL'92 features.

Figure 5.3 illustrates the control and data flow of the Flash Code Generator from the syntax and semantic analysis of the SDL process diagrams until the generation of the corresponding ActionScript classes.

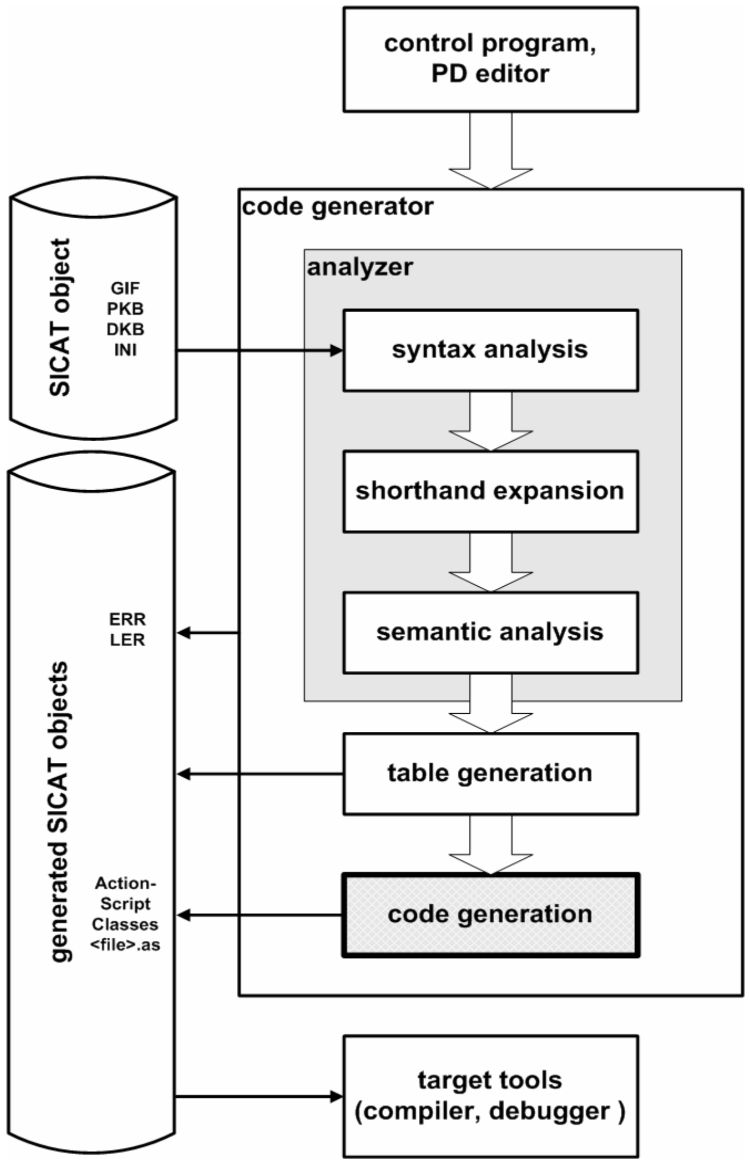


Figure 5.3: Control and Data Flow of the Flash Code Generator

5.2.1 ActionScript SDL Base Classes

As ActionScript 2.0 is fully object oriented programming (OOP) language and the OOP encourages reusability by class inheritance, a set of ActionScript base classes have been implemented.

For instance, the “SDLProcess” class describes, with taking into consideration a certain degree of abstraction, an SDL process and provides essential methods to implement the structure of an SDL process diagram, for example, sending signals. Then the generated class of a concrete SDL process of a certain device behavior will inherit this base class.

As the “SDLProcess” base class inherits the predefined “MovieClip” class of ActionScript 2.0, the generated class can be then easily integrated into Macromedia Flash 8, and will possess all the abilities of the “MovieClip” class automatically.

5.2.2 Mapping of SDL Processes onto ActionScript Constructs

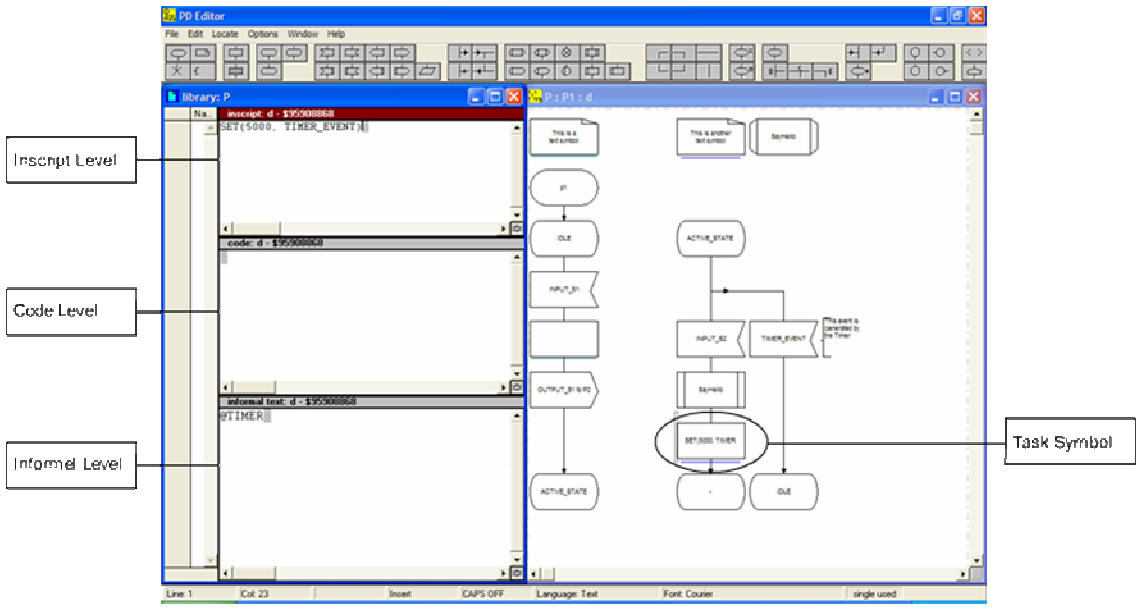
The contents of the generated ActionScript file are determined by the structure of the process diagram, by the contents of the textual description levels (see Figure 5.4) of the individual graphic symbols and by setting the profile parameters for code generation.

The individual graphic symbols of a process diagram support three textual description levels:

- The inscript level for labeling the symbol in the process diagram
- The code level can be used to add ActionScript statements, which will be then incorporated into generated code
- The informal text level which is used to incorporate additional textual information or keywords to control code generation

For instance, as shown in Figure 5.4, the task symbol represents a timer because its “informal text level” contains the keyword “@TIMER” and at its “inscript text level” the expression “SET(5000, TIMER_EVENT)” can be seen. This expression means that the timer will be timed out after five seconds and then the event named “TIMER_EVENT” will be triggered. The task symbol, at its simplest role, can contain ActionScript statements at its “code text level”. These statements will be then included into target source.

There are a set of keywords, which can be used not only to assign many roles to SDL symbol semantically, but also to control code generation layout concerning comment headers, code formatting and so on. While analyzing the diagram structure, the code generator will take into consideration the keywords and will generate code accordingly.



```
SET(5000, TIMER_EVENT);
```

```
@TIMER
```

Figure 5.4: The Three Text Levels of an SDL Symbol (i.e. Task Symbol)

SDL process diagrams are mapped onto ActionScript construct as follows:

- Each SDL process is mapped to an ActionScript class with the name <process name>. All process classes inherit the SDL Process base class
- Start transition is mapped to the constructor of the process class and other method for initialisation purposes and defining the state machine event handler and the starting transition
- SDL procedure diagrams are mapped onto member methods of the process class

Let's observe the process diagram shown in Figure 5.6 to illustrate how an SDL process diagram can be transformed into ActionScript class. The SDL process is named in this case with "p1". On the diagram two text symbols can be seen. The first one, which can be placed everywhere on the diagram, serves to insert general information about the author or the version of the diagram and some additional information by adding "#AUTHOR", "#VERSION" and "#INFORMATION" keywords at the "informal text level" of the text symbol.

The generated code, corresponding to this text symbol, as shown in Figure 5.6 (rectangle 1), is the following:

```
////////////////////////////////////////////////////////////////////////////////////////////
///   @brief    ActionScript module generated by GRAPE
///             ActionScript Code Generator for Rapid Prototyping,  CG-Version:
///
///          project    : <Path>
///          process  : P
///          module   : P1
///          diagram  : d
///
///   @author    That author
///   @version   This version
///   @file         p1.as
///   @date       Date  : 12/11/2006, Time  : 10:39:17 AM
///   @info        Some information
////////////////////////////////////////////////////////////////////////////////////////////
```

The second text symbol, which must be placed exactly above the process start symbol on the diagram, serves to declare class attributes (global variables), if needed, by using the keyword "#DECLARATIONS" placed at the "code text level" of the text symbol. The corresponding generated code, as shown in Figure 5.6 (rectangle 2), is:

```
public var m_aVariable = 0;
```

The generated code corresponding to the start transition on the diagram, as shown in Figure 5.6 (rectangle 3), is as follows:

```
// Constructor
function p1(timeL)
{
 this.timeLine = timeL;
 this.currentState = "";
 ...
}

function Start()
{
 currentState = "IDLE";
}

public function handleEvent(event)
{
 EventHandler(event.type, event.val);
}

function EventHandler(event, param):String
{
 switch (currentState)
 {
    case "ACTIVE_STATE":
 return State_ACTIVE_STATE(event, param);
    case "IDLE":
 return State_IDLE(event, param);
 }
}
```

This code represents a constructor of the class and some methods, which achieve some initialisations and define the state machine event handler and the starting transition.

The generated code of the transition between the “IDLE” and “ACTIVE_STATE” states, as shown in Figure 5.6 (rectangle 4), is the following:

```
// STATE: IDLE
function State_IDLE(event, param):String
{
 switch(event)
 {
   case "INPUT_S1":
      m_aVariable = 1;
      trace("Variable has been changed!!!");
      //====> OUTPUT_S1 to P2
      SendEvent("OUTPUT_S1", _root.P2);
      //====> ACTIVE_STATE
      currentState = "ACTIVE_STATE";
      return "";
 }
 return event;
}
```

The generated code means that to each transition on an SDL process diagram, a class method will be generated. A similar code will be generated from the transition between “ACTIVE_STATE” state and other states, as shown in Figure 5.6 (rectangle 6), but in this case the “switch-case” structure will be accordingly enlarged depending on the number of events that affect the transition.

Finally, a procedure declaration symbol named with “SayHello” can be seen on the process diagram. This symbol represents a separate procedure diagram, which is not shown in Figure 5.6, but for the sake of completeness, this diagram is illustrated in the following figure:

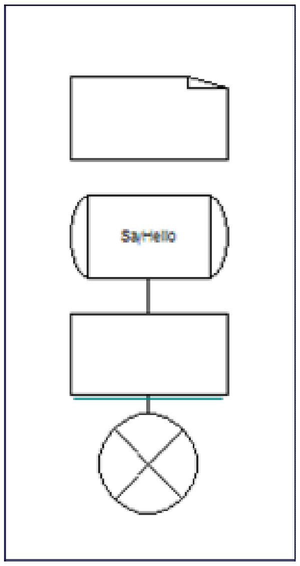


Figure 5.5: Procedure Diagram (SayHello)

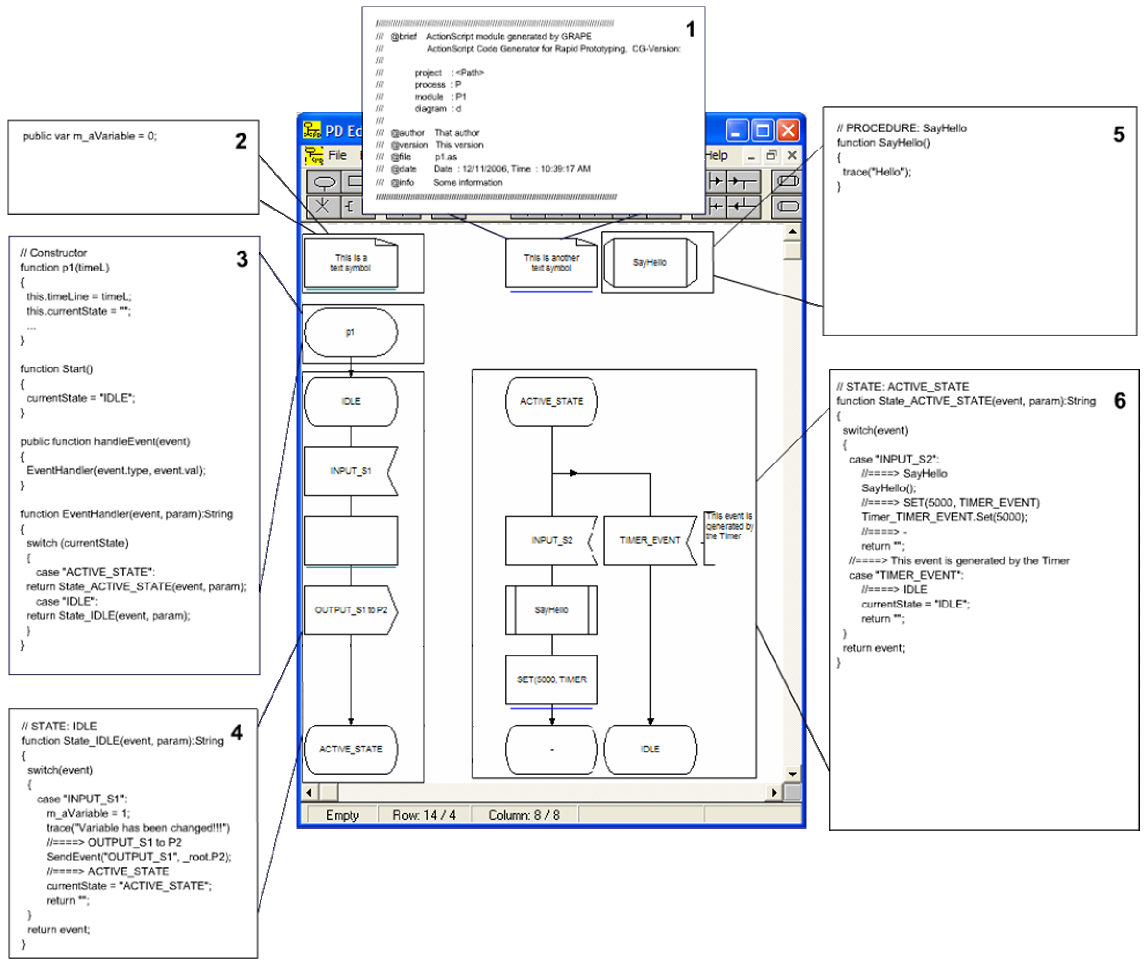
The “code text level” of the task symbol on the procedure diagram, shown in Figure 5.5, contains the following ActionScript statement “trace("Hello");”. This statement does nothing else than printing the word “Hello” on the screen.

The generated final code corresponding to the procedure declaration symbol, as shown in Figure 5.6 (rectangle 5), is about as follows:

```
// PROCEDURE: SayHello
function SayHello()
{
  trace("Hello");
}
```

This means that to each procedure declaration symbol on a process diagram a class method will be generated. The content of the method body depends on the structure of the procedure diagram.

The final result, after starting the code generator on the corresponding process object from SICAT Control Program, is an automatically generated ActionScript file named “p1.as”.



```
////////////////////////////////////////////////////////////////////////////////
/// @brief    ActionScript module generated by GRAPE
///           ActionScript Code Generator for Rapid Prototyping,  CG-Version:
///
///        project   : <Path>
///        process   : P
///        module    : P1
///        diagram   : d
///
/// @author   That author
/// @version  This version
/// @file     p1.as
/// @date     Date  : 12/11/2006, Time  : 10:39:17 AM
/// @info     Some information
////////////////////////////////////////////////////////////////////////////////
```

```
public var m_aVariable = 0;
```

```
// Constructor
function p1(timeL)
{
 this.timeLine = timeL;
 this.currentState = "";
 ...
}

function Start()
{
 currentState = "IDLE";
}

public function handleEvent(event)
{
 EventHandler(event.type, event.val);
}

function EventHandler(event, param):String
{
 switch (currentState)
 {
   case "ACTIVE_STATE":
 return State_ACTIVE_STATE(event, param);
   case "IDLE":
 return State_IDLE(event, param);
 }
}
```

```
// STATE: IDLE
function State_IDLE(event, param):String
{
 switch(event)
 {
   case "INPUT_S1":
      m_aVariable = 1;
      trace("Variable has been changed!!!")
      //====> OUTPUT_S1 to P2
      SendEvent("OUTPUT_S1", _root.P2);
      //====> ACTIVE_STATE
      currentState = "ACTIVE_STATE";
      return "";
 }
 return event;
}
```

```
// PROCEDURE: SayHello
function SayHello()
{
 trace("Hello");
}
```

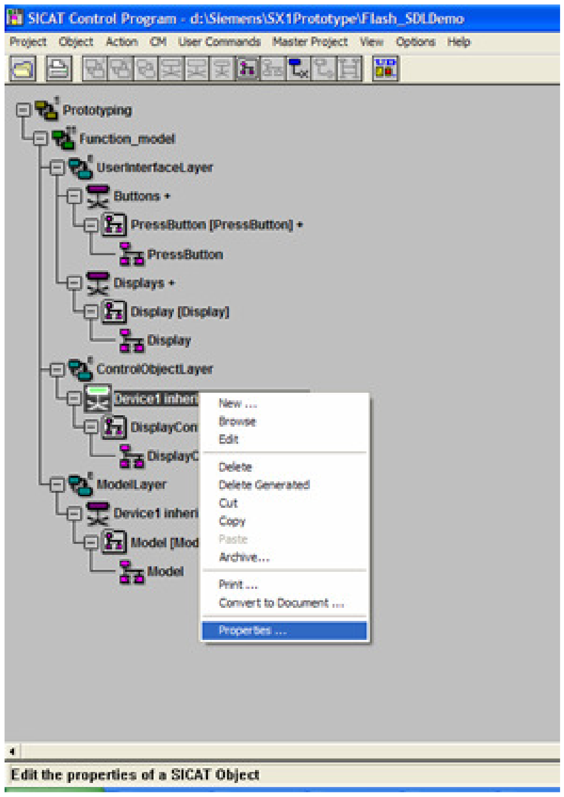
```
// STATE: ACTIVE_STATE
function State_ACTIVE_STATE(event, param):String
{
 switch(event)
 {
  case "INPUT_S2":
     //====> SayHello
     SayHello();
     //====> SET(5000, TIMER_EVENT)
     Timer_TIMER_EVENT.Set(5000);
     //====> -
     return "";
  //====> This event is generated by the Timer
  case "TIMER_EVENT":
     //====> IDLE
     currentState = "IDLE";
     return "";
 }
 return event;
}
```

Figure 5.6: Transformation of a SDL Model Example into ActionScript Class

5.2.3 How to Generate Code from SICAT

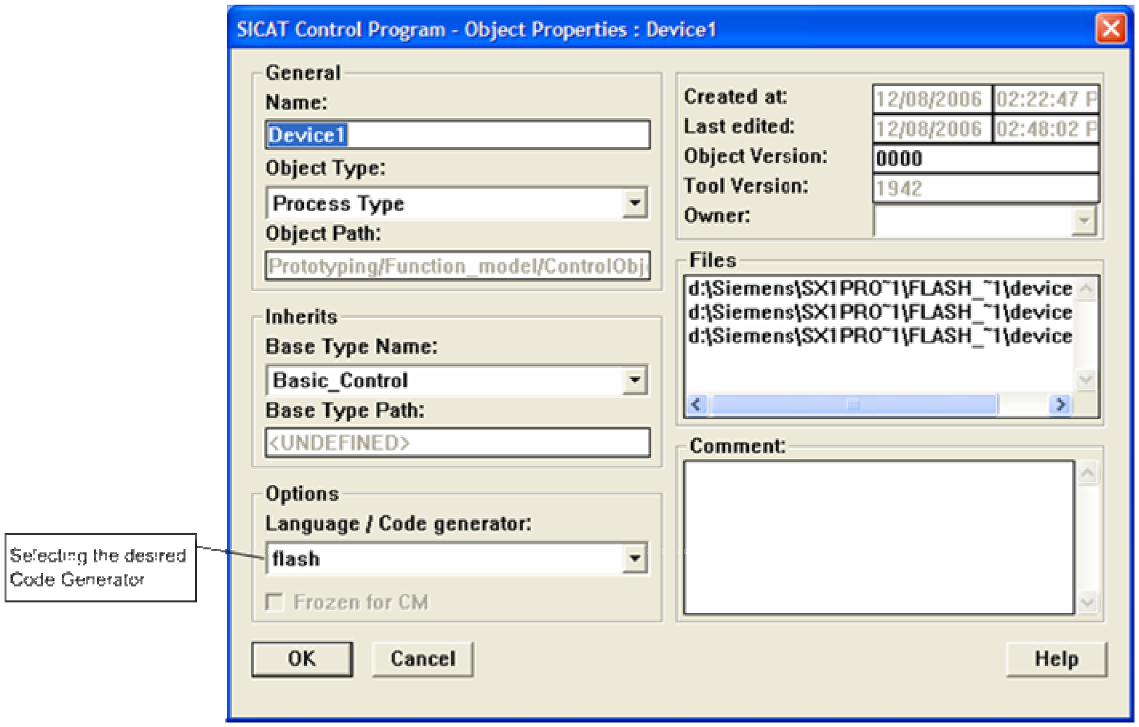
The Flash code generator can be started from the SICAT control program. In order to achieve code generation with the desired code generator, the intended code generator must be installed within SICAT toolset and some settings steps have to be done first and only once.

For this purpose, the process module or the process symbol for which code shall be generated must be selected. As shown in Figure 5.7 below, after selecting the object and by clicking on it with the right mouse button, a pop up menu will arise. Then by clicking on the menu item named with "Properties" the corresponding properties dialog of the SICAT object will appear, as shown in Figure 5.8.



Edit the properties of a SICAT Object

Figure 5.7: Properties Settings of SICAT Object from Control Program


SICAT Control Program - Object Properties : Device1

General
Name: Device1
Object Type: Process Type
Object Path: Prototyping/Function_model/ControlObj

Inherits
Base Type Name: Basic_Control
Base Type Path: <UNDEFINED>

Options
Language / Code generator: flash
Frozen for CM

Created at: 12/08/2006 02:22:47 P
Last edited: 12/08/2006 02:48:02 P
Object Version: 0000
Tool Version: 1942
Owner:

Files
d:\Siemens\SX1PRO~1\FLASH_~1\device
d:\Siemens\SX1PRO~1\FLASH_~1\device
d:\Siemens\SX1PRO~1\FLASH_~1\device

Comment:

Figure 5.8: Selecting the Flash Code Generator

Then, the associated SICAT object must be assigned to "Flash" as the target language for code generation by selecting the item “flash” from the options combo box.

Finally, after setting the corresponding properties, code generation can be achieved from SICAT Control Program. For this purpose, the process module or the process symbol must be selected and the menu item named “Generate Code” from the menu “Action” of the menu bar has to be selected, as shown in Figure 5.9.

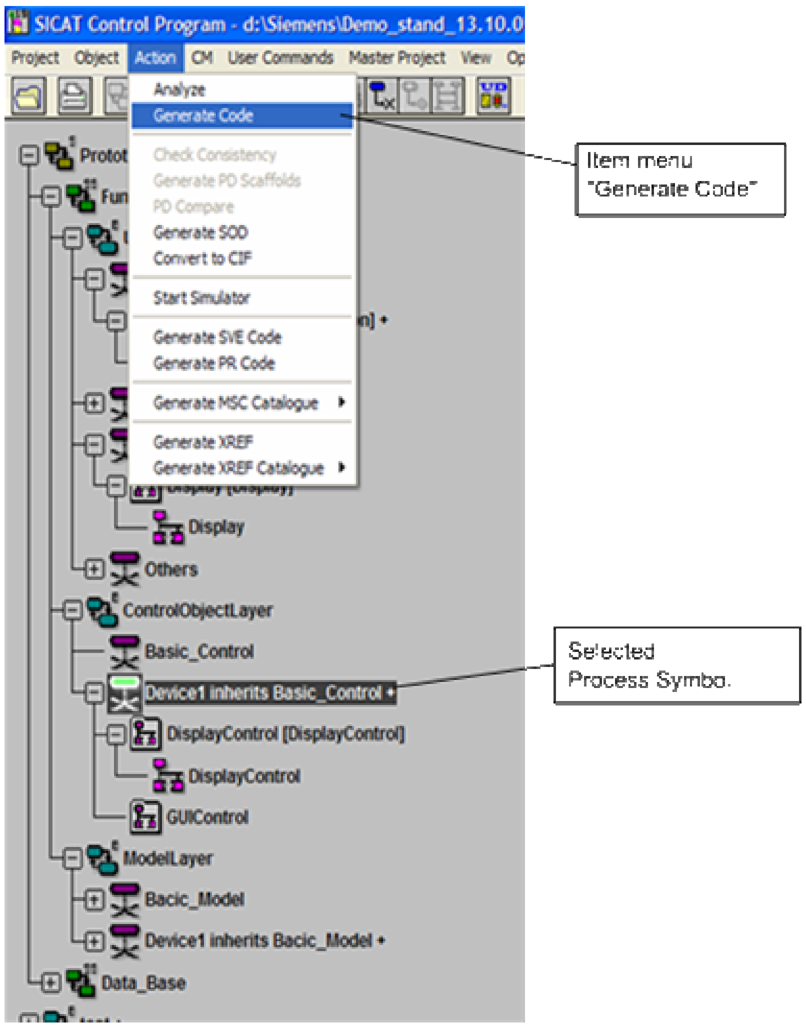


Figure 5.9: Code Generating from SICAT Control Program

5.3 Construction of the Device Graphic Elements

As mentioned previously, the visual appearance of the device and its graphical components can be designed with Macromedia Flash 8. This latter has been identified to be a powerful graphic design environment. It provides a flexible methodology to design graphic elements or objects, which are in terms of symbols in Flash terminology.

The most important symbols are the so called “movie clips”. Movie clips are reusable pieces of animations. With respect to object oriented principles, they are objects that have a physical presence on the stage and can be accessed programmatically through their methods and properties. The device graphic elements are nothing else than movie clips symbols. They have been kept simple to show how easy it can be to design graphic elements with the mentioned environment.

Flash is an authoring tool that designers and developers use to create presentations and interactive applications. Flash projects can include simple animations, sound effects, video content and complex applications, which enable comfortable interaction with the end user.

Each piece of content made with Flash is called applications, even though they might only be a basic animation. Flash applications can be made by including pictures, sound, video, and special effects.

Flash is extremely well suited for creating content for deployment over the internet because its files are very small. Flash achieves this through its extensive use of vector graphics. Vector graphics require significantly less memory and storage space than bitmap graphics because they are represented by mathematical formulas instead of large data sets.
Bitmap graphics are larger because each individual pixel in the image requires a separate piece of data to represent it [84].

In order to build a Flash application, graphics can be created with the Flash drawing tools, as shown Figure 5.10. The application can be then ameliorated by importing additional media elements into the Flash document.

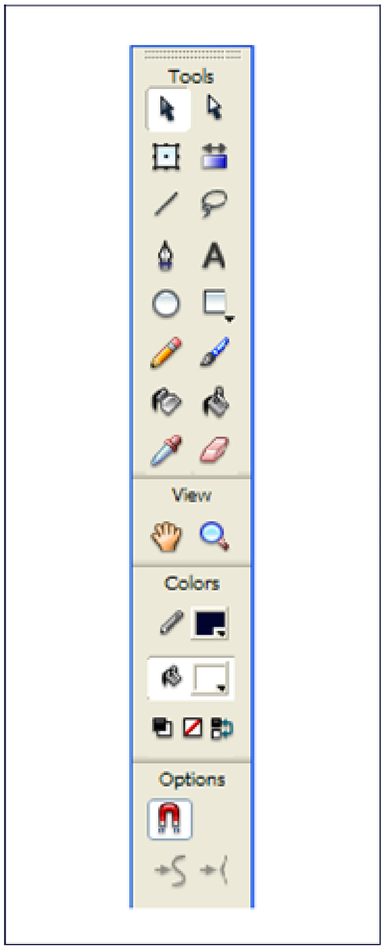


Figure 5.10: Flash Drawing Tools

Authoring Flash content is nothing else than working in a Flash document file. Flash documents have the file extension “.fla” (FLA) [84].

A Flash document has four main parts [84]:

The first part is called “stage”. The stage is the place, where graphics, videos, buttons, and so on appear during playback. The stage, as shown in Figure 5.11, is the quadratic white surface. Its dimensions and colour can be changed using the properties panel (see Figure 5.12) adequately.

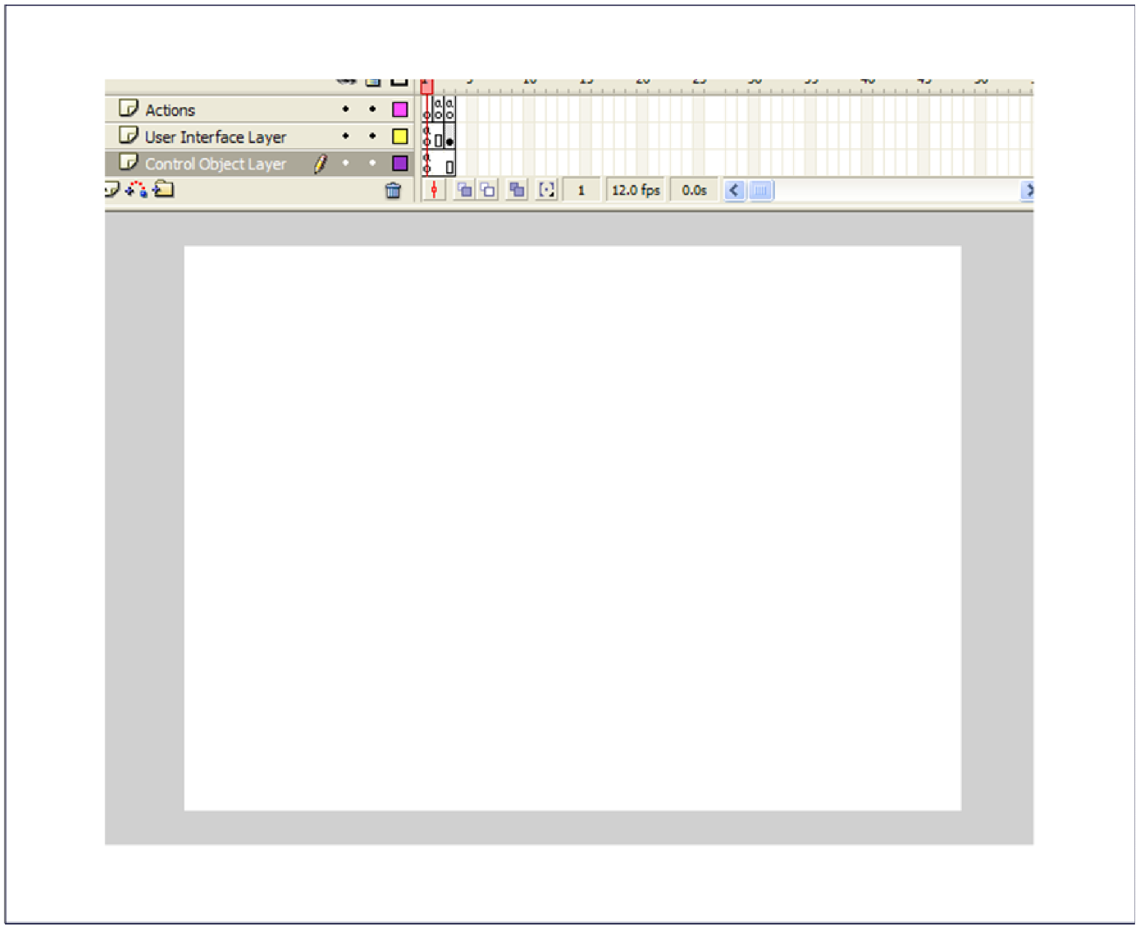
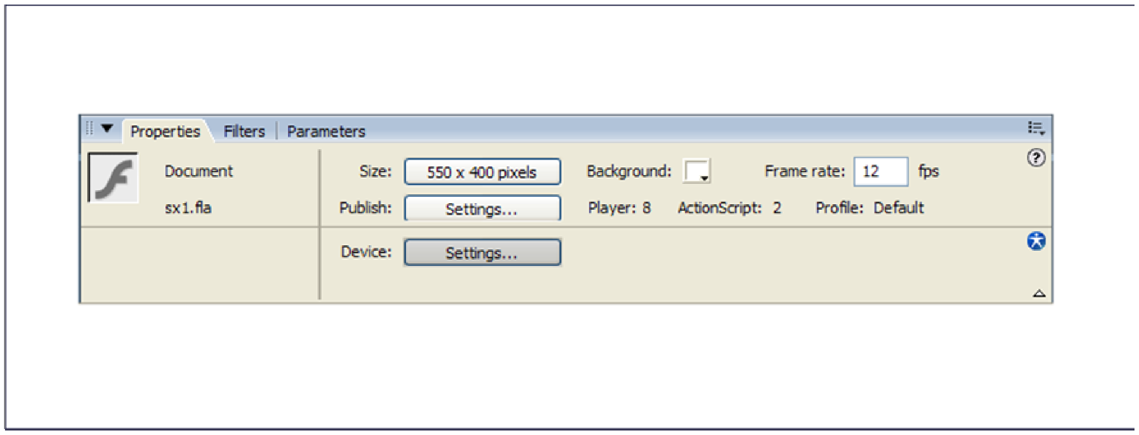


Figure 5.11: The Stage of Macromedia Flash 8



Document

sx1.fla

Size: 550 x 400 pixels

Background:

Frame rate: 12 fps

Publish: Settings...

Player: 8

ActionScript: 2

Profile: Default

Device: Settings...

Figure 5.12: The Properties Panel of Macromedia Flash 8

The second part is named with “timeline”. The timeline responds the question of when the graphics or other elements of an application should appear. The timeline, as shown in Figure 5.13, is the vertical red line.

Further, the timeline is used to specify the layering order of graphics on the stage. It means graphics placed in higher layers appear above of graphics placed in lower layers.

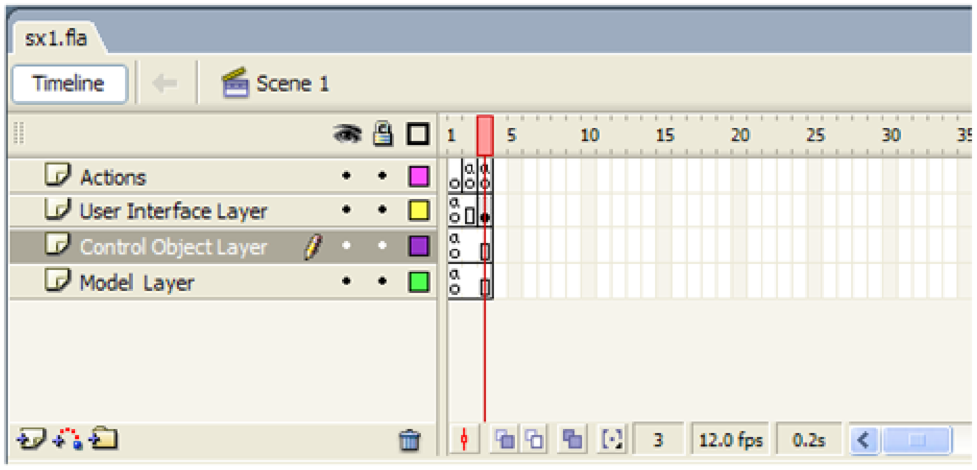


Figure 5.13: The Timeline of Macromedia Flash 8

The third part is called "library panel". This panel, as shown in Figure 5.14, displays a list and provides an overview of the media and graphics elements inserted in a Flash document.

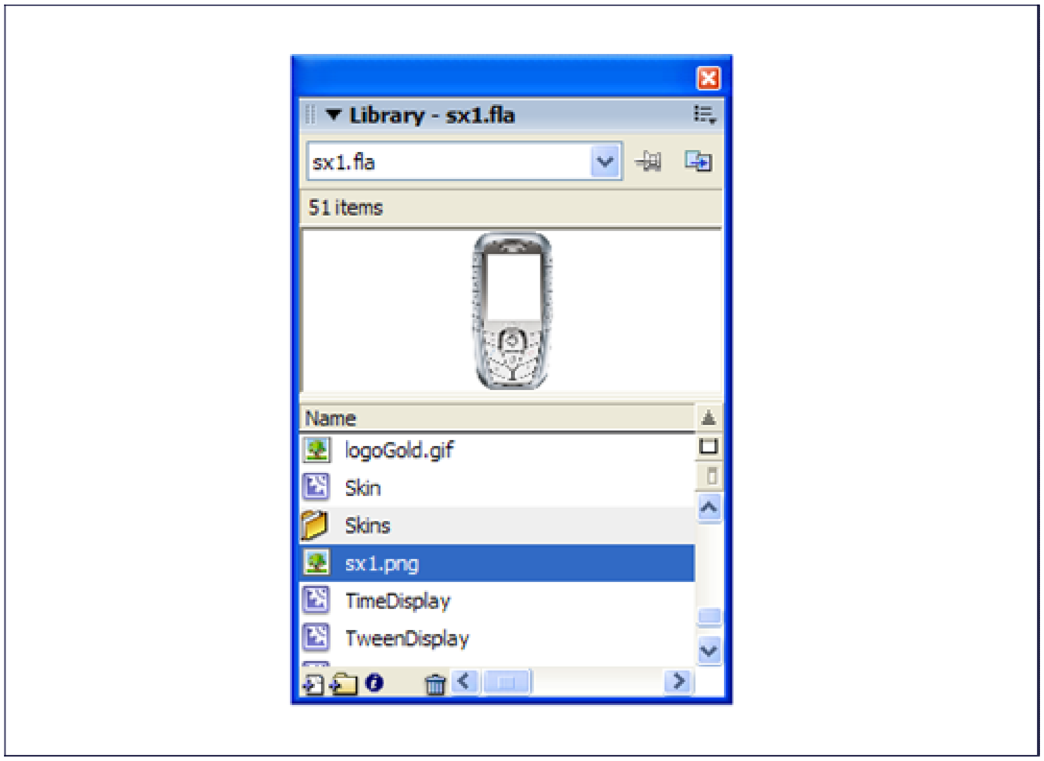
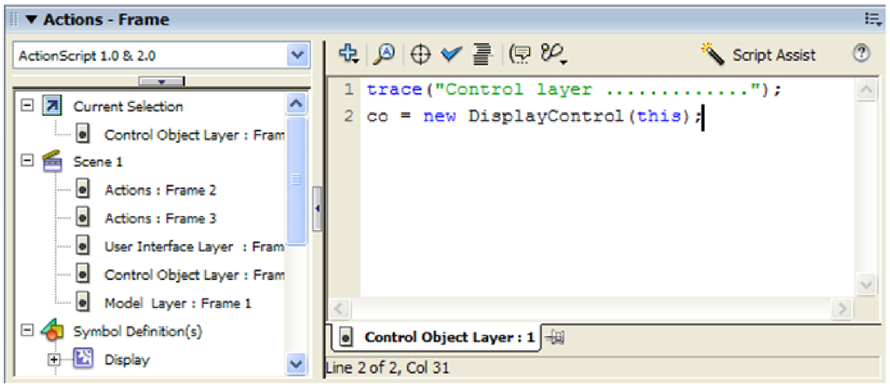


Figure 5.14: The Library Panel of Macromedia Flash 8

The fourth part is the ActionScript editor. This editor, as shown in Figure 5.15 below, enables designers and developers to add ActionScript code, which allows them to control media and graphics elements inserted in a Flash document. In this way the interactivity can be guaranteed.

ActionScript can be also used to embed logic in applications. Logic enables applications to behave in different ways depending on the user actions or other conditions.

Flash includes many versions of ActionScript. Only ActionScript 2.0 will be exclusively used to develop the device prototype.



```
trace("Control layer ............");
co = new DisplayControl(this);
```

Line 2 of 2, Col 31

Figure 5.15: The ActionScript Editor of Macromedia Flash 8

Flash includes many features that make it powerful and easy to use, such as pre-built drag-and-drop user interface components, built-in behaviours that add easily ActionScript to the Flash document and special effects that can be associated to media objects.

After finishing authoring a Flash document, it can be then published by selecting the menu item named “Publish” from the menu “File” of the menu bar of Macromedia Flash development environment.

This command action creates a compressed version of the Flash document with the extension .swf (SWF), which can be then interpreted and executed by the Flash Player in a web browser or as a stand-alone application.

To construct the visual appearance of the device prototype many movie clips symbols have been created. These symbols are based on primitive shapes and on the user interface components provided by Flash environment and which will be explained in the next following subsections.

5.3.1 Flash Symbols and Library Assets

A symbol is a graphic, button, or movie clip that can be created using Macromedia Flash 8. Symbols have to be created only once and can then be used rather in the recent Flash document or other documents.

A symbol can include artwork that has been imported from other applications. Further, any symbol, that has been already created, becomes automatically part of the library for the current document.

If a symbol is of the type “movie clip” or “button”, it can be instantiated as an object. Thereby it can be accessed and controlled programmatically through their methods and properties

An instance is a copy of a symbol located on the stage or nested inside another symbol. The properties of each instance can be changed individually and independent of other instances of the same symbol.

Further, using symbols in Flash documents can reduces significantly the size of the swf file, because saving several instances of a symbol requires less storage space than saving multiple copies of the symbol contents.

Symbols can be also shared between documents as shared library assets during authoring or at runtime. For runtime shared assets, assets can be linked in a source document to any number of the destination document without importing the assets into the destination document.

For assets shared during authoring, symbols can be updated or replaced with any other symbol available on the local network.

5.3.1.1 How to Create a Flash Symbol

A symbol is a reusable object, and an instance is an occurrence of a symbol on the stage. Using instances several times does not increase the file size, that's why the document file size remains, in many cases, small.

Symbols have many benefits. For example, symbols simplify editing a document because when a symbol had been edited, all instances of the symbol update to reflect the changes. Further symbols allow creating sophisticated interactivity in an easy way.

There are three types of symbol: graphic, button and movie clip. Each symbol has its unique timeline, stage and optionally layers. While creating a symbol, its type can be chosen, depending on the use purpose of the symbol in a document.

For instance, graphic symbols can be used for static images and to create reusable pieces of animation that are tied to the main timeline. Graphic symbols operate synchronously with the main timeline of a document.

Button symbols, on the other hand, can be used to create interactive buttons that capture mouse events and handle them appropriately.

Movie clip symbols are the most important symbols and they can be used to create reusable pieces of animation. Movie clips have their own multi-frame timeline that is independent from the main timeline of a document.

Movie clip instances can be placed inside a document or can be nested in other symbols. For example, a movie clip symbol can be placed in the timeline of a button symbol to create animated buttons.

Flash provides built-in components, as shown in Figure 5.16. They are nothing else than parameterized movie clips that can be used to add user interface elements, such as buttons, check boxes, or scroll bars, to a Flash documents. These components can be also non-visible component like the "XMLConnector", which facilitate the access to application data. For example, GUI-Components can be selected and added to the stage of a document to construct the graphical user interface of an application.

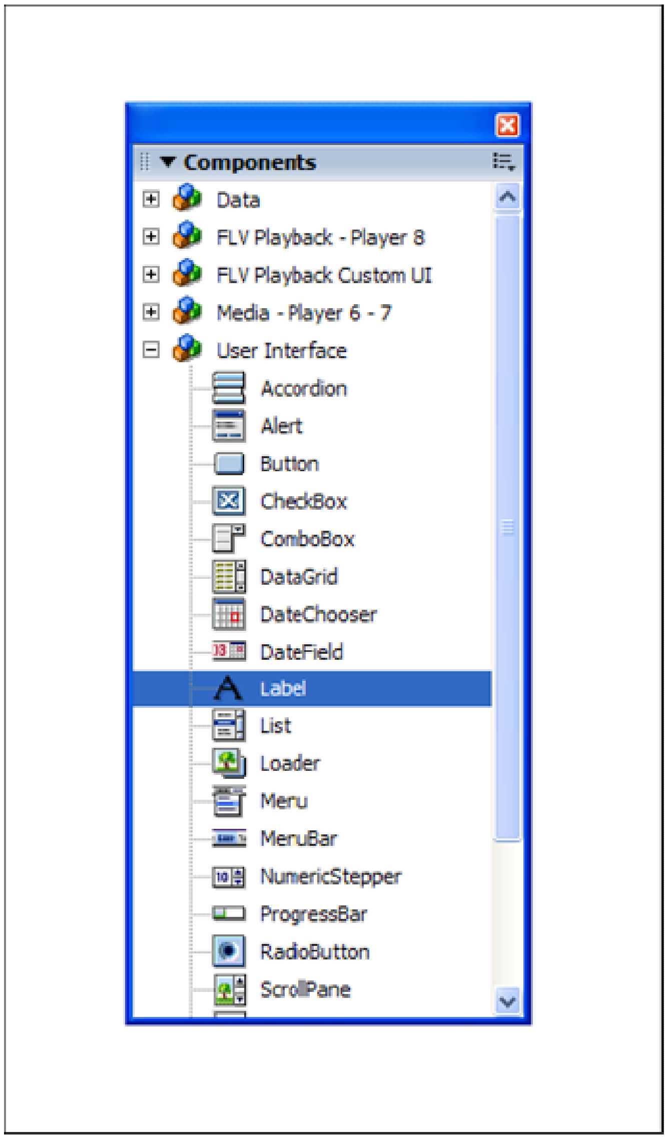
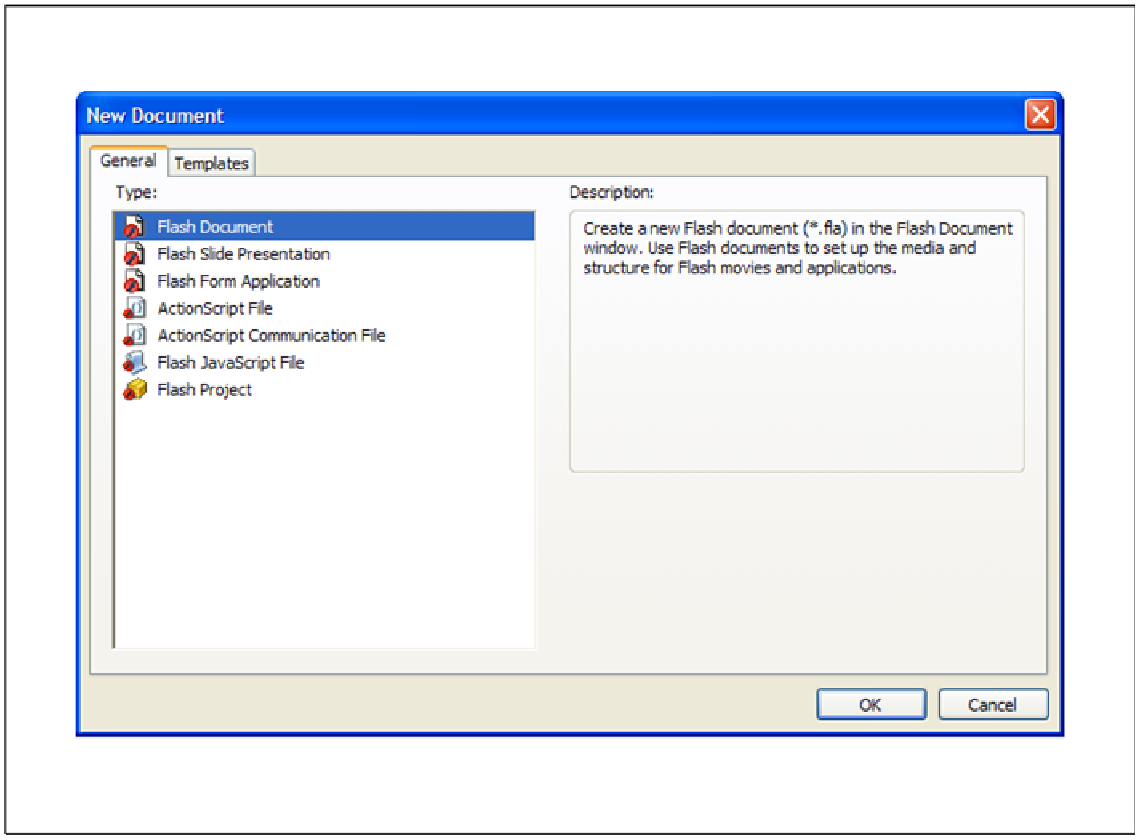


Figure 5.16: Components Panel

In order to create a symbol, a Flash document must be first created. This document represents the working space for designing and developing Flash application.

A Flash document can be created newly, or a document, that has been previously saved, can be opened, elaborated and ameliorated. Under Windows platform, a new Flash document can be created by selecting the sub-menu item “New” from the menu “File” of the menu bar of Macromedia Flash 8. Then the new document dialog box, as shown in Figure 5.17, will appear.

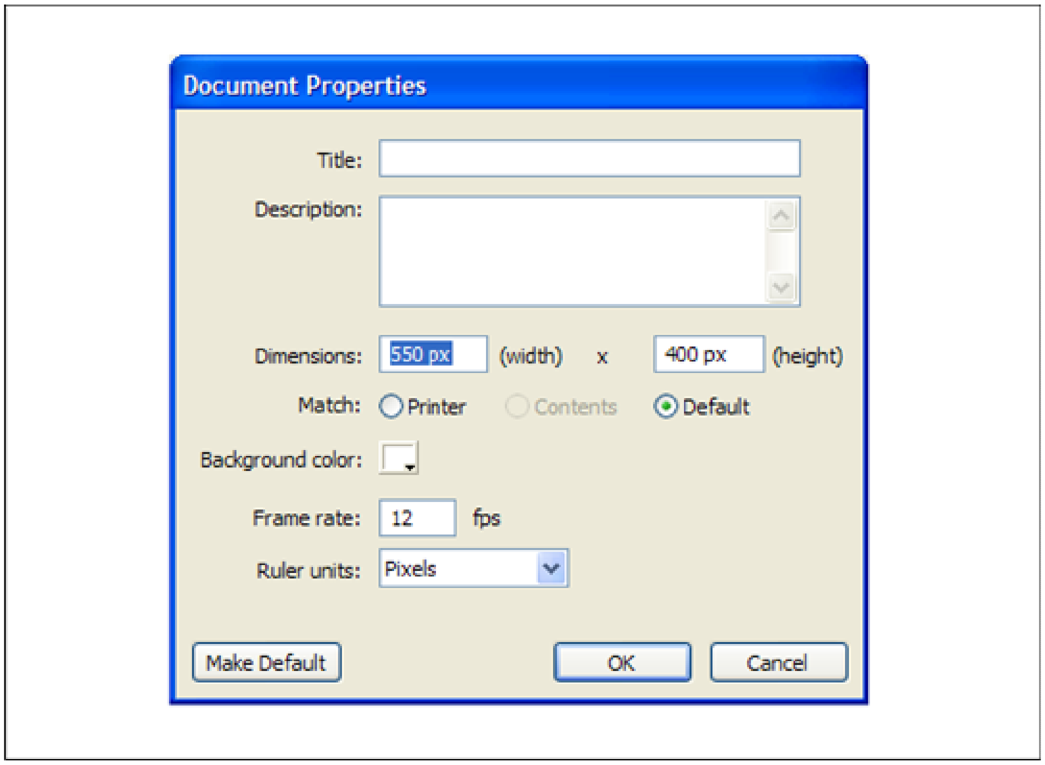


Create a new Flash document (*.fla) in the Flash Document window. Use Flash documents to set up the media and structure for Flash movies and applications.

Figure 5.17: New Document Dialog Box

From this dialog a Flash document option can be selected and opened. In addition standard templates that come with Macromedia Flash or templates that have been already created or saved, can be then selected and opened.

After opening a Flash document or a template, its properties can be changed and respectively adjusted adequately. To set the size, frame rate, background colour, and other properties of a new or existing document, the document properties dialog box, as shown in Figure 5.18, can be used for this purpose.



Title:
Description:
Dimensions: 550 px (width) x 400 px (height)
Match: Printer Contents [✓ Default]
Background color:
Frame rate: 12 fps
Ruler units: Pixels

Figure 5.18: Documents Properties Dialog

The property inspector, as shown in Figure 5.12, can be also used to set properties for an existing document. The property inspector makes it easy to access and change the most commonly used attributes of a document.

After successfully creating or opening a Flash document, a symbol can be now created. There are two common ways to create symbols. The first one is to select already placed objects on the stage and then converting them to a symbol or, the second way, an empty symbol can be created firstly and the content can be then imported or placed on the stage. In both cases the symbol must be in editing mode.

To convert selected, already placed, elements on the stage to a symbol, the submenu “Convert to Symbol” of the menu “Modify” must be selected. Then the “Convert to Symbol” dialog box will appear. In this dialog the name of the symbol can be given, its type can be selected and other properties can be determined.

Flash adds the symbol to the library automatically. This symbol can be then selected and added on the stage and becomes an instance of the symbol.

Analogously, a new empty symbol can be created by selecting the submenu item called "New Symbol" from the menu "Insert" of the menu bar. Then the "Create New Symbol" dialog box, as shown in Figure 5.19, will appear in a basic mode. In this dialog the name of the symbol can be given, its type can be selected and other properties can be determined.

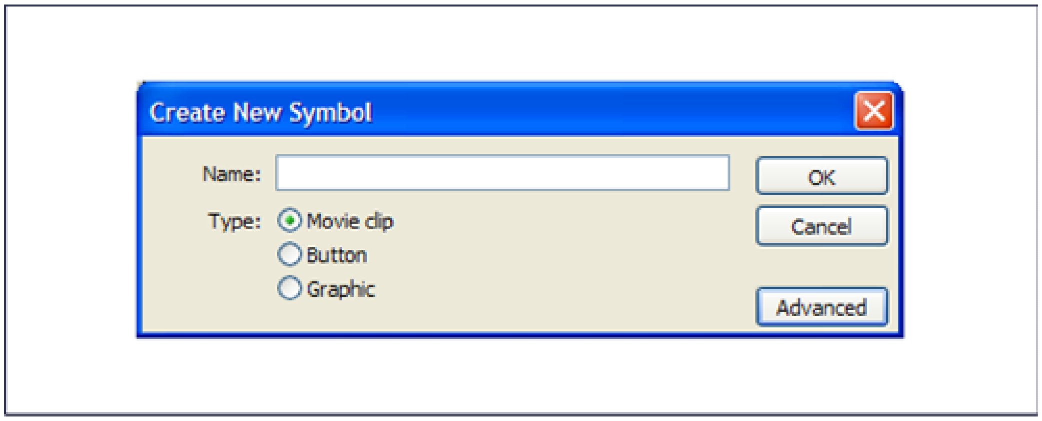
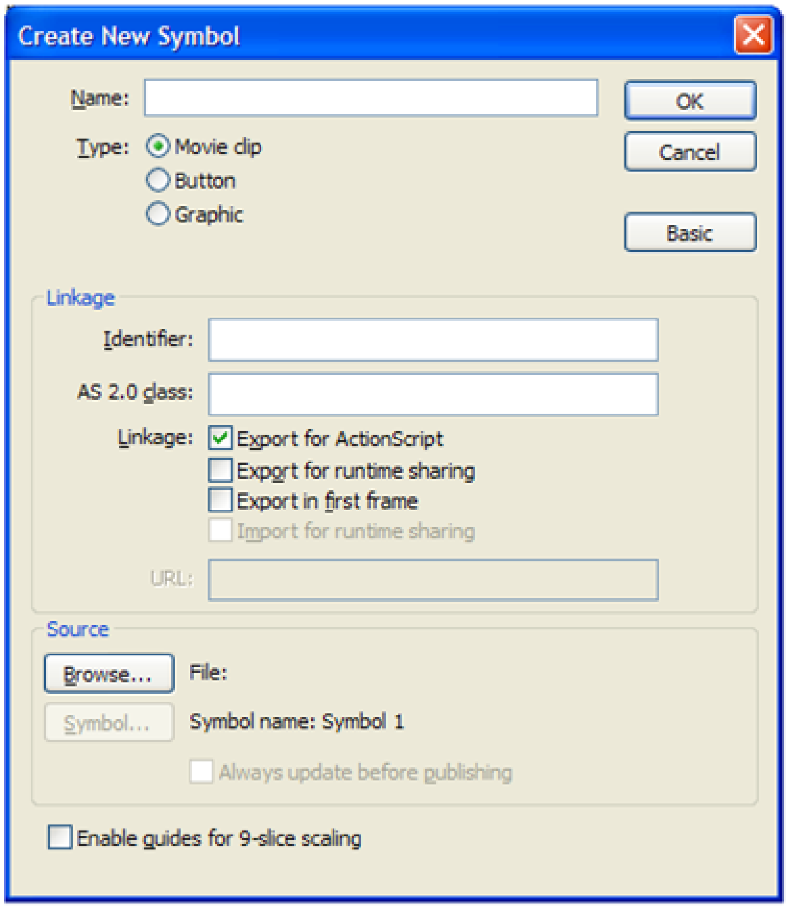


Figure 5.19: Create Symbol Dialog Box (Basic Mode)

Macromedia Flash adds the created symbol to the library and switches to symbol-editing mode. In this mode, the symbol can be authored, as a normal document, by adding and respectively importing content to the stage.

By clicking on the button named with "Advanced" of the "Create New Symbol" dialog shown in the figure above, the same dialog will appear in advanced mode, as shown in Figure 5.20. This dialog allow developers not only to assign ActionScript classes to the symbol, which will be linked at runtime but also to determine some other powerful properties to symbol, like guides for 9-slice scaling.


Create New Symbol

Name:

Type: [✓ Movie clip] Button Graphic

Linkage

Identifier:

AS 2.0 class:

Linkage: [✓ Export for ActionScript]
Export for runtime sharing
Export in first frame
Import for runtime sharing

URL:

Source

File:

Symbol name: Symbol 1

Always update before publishing

Enable guides for 9-slice scaling

Figure 5.20: Create Symbol Dialog Box (Advanced Mode)

A movie clip symbol is a powerful symbol and is analogous in many ways to a document within a document. This symbol type has its own timeline independent of the main timeline of a document. Movie clips can be added within other movie clips and buttons to create nested movie clips.

For instance, one of the movie clips of the device prototype is the so called “BatteryIcon”, as shown in figure below and encircled with a red ellipsis.

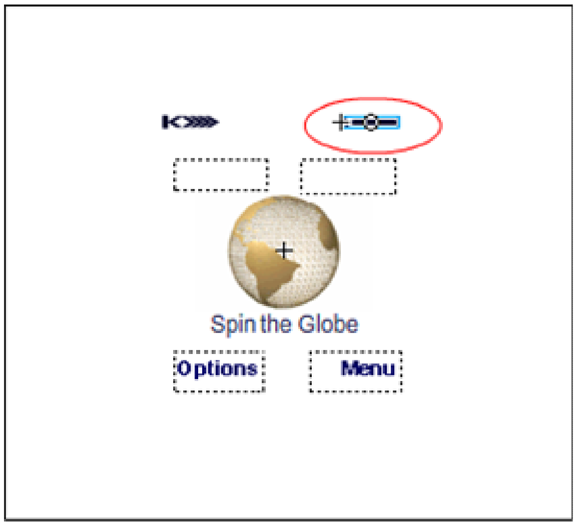


Figure 5.21: Battery Icon Movie Clip Symbol

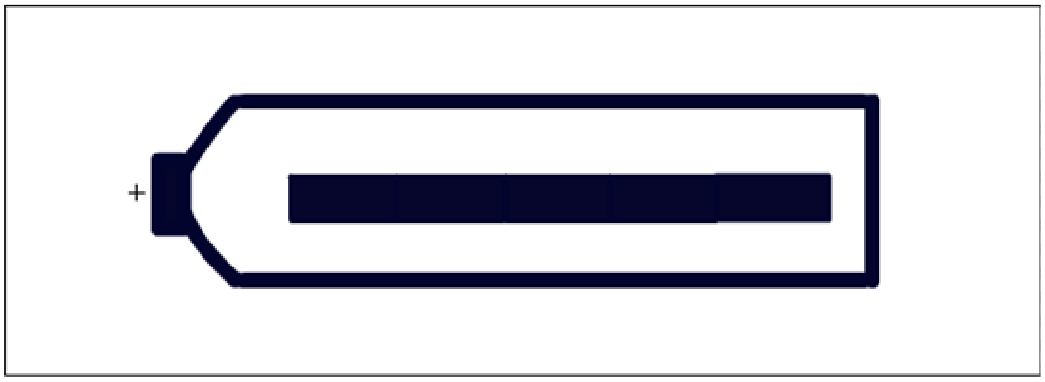
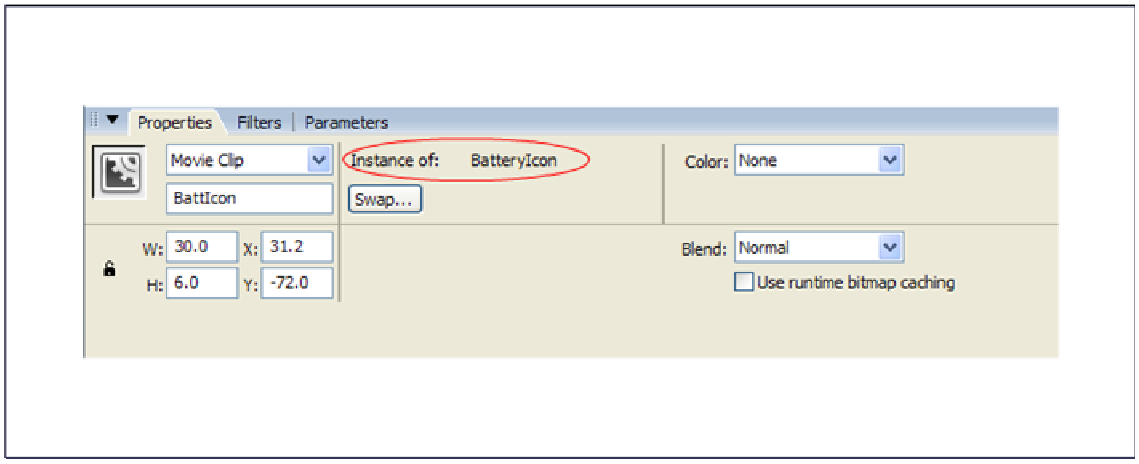


Figure 5.22: Battery Icon Movie Clip (Zoomed)

Within the “BatteryIcon” movie clip many other movie clips, in this case 5 instances of the so called “BattCapacity” movie clips framed in a tube liked shape, are nested (see Figure 5.22). The “BattCapacity” movie clips are nothing else than a primitive shape in terms of black rectangles.



Movie Clip
BattIcon
Instance of: BatteryIcon
Color: None
W: 30.0 X: 31.2
H: 6.0 Y: -72.0
Blend: Normal
Use runtime bitmap caching

Figure 5.23: Properties Inspector of Movie Clips Instances

After creating a symbol, it can be instantiated in a document or nested in other symbols. When a symbol has been modified, Flash updates all instances of that symbol, but when modifying an instance, the updates will affect only that changed instance.

While creation, Flash gives movie clip and button instances default instance names. From the property inspector custom names can be assigned to instances.
For example, as shown in Figure 5.23, an instance of the “BatteryIcon” (encircled with a red ellipsis) is named with “BattIcon”.

Programmatically speaking, the instance name can be then used to refer to an instance of an object of type movie clip in ActionScript. That’s why the name of an instance must be unique in order to control it at runtime.

In order to create a new instance of a symbol, a layer of the main timeline of the current document must be selected firstly. After selecting the desired symbol from the library panel, it can be placed on the stage of the current document by simply dragging and dropping it.

To assign a custom name to an instance, the corresponding symbol placed on the stage must be selected. Then the name can be entered in the instance name text field on the left side of the property inspector, as shown in Figure 5.23.

Further the property inspector can be used to specify color effects, assign actions, set the graphic display mode, or change the behavior of the instance.

The behavior of the instance is the same as the symbol behavior, unless the behavior may be specified otherwise depending on requirements and necessity. As mentioned previously, any changes performed on the instance affect only the instance itself, and not the symbol.

5.3.1.2 Device Prototype Symbols

To design the visual appearance of the device prototype, many Flash symbols have been created. The most of them are of type movie clip. The design complexity of these symbols vary from very simple, in terms of primitive geometric shapes, till sophisticated, in terms of animation and visual effects.

In order to improve reusability, these symbols of the type movie clips can be transformed to components easily. Macromedia Flash components are movie clips with parameters that allow developers and designers to modify their appearance and behaviour.

A component can be a simple user interface control, such as a radio button or a check box. It can also be non-visual, like the “XMLConnector” that allows access data application saved in XML files.

In other words, the essential purpose of components is to define parameterized movie clips that can be programmatically used in a natural, object oriented way. A component starts with a movie clip that can be linked to an ActionScript class definition, which provides properties and methods to control the movie clip itself. When the component is placed in a movie clip, the class constructor is invoked.

Reference [85] demonstrates how to create a custom component and how to install it with the extensions manager of Macromedia Flash 8. This reference is also highly recommended to interested reader, to get information about this powerful feature provided by Flash.

As mentioned already, many movie clips have been created in order to achieve designing the visual part of the device prototype.

The simplest symbol of the device prototype is the so called “Arrow”, as shown in Figure 5.24. This movie clip is nothing else than a black triangle shape, which has the purpose of pointing to the direction of the menu navigation either to down or to up.

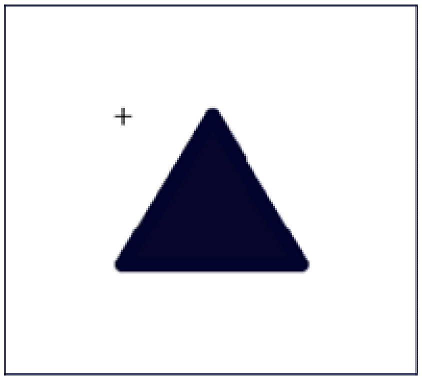


Figure 5.24: Arrow Movie Clip (Zoomed)

Another simple movie clip is the so called “ConnecPower”, as shown in Figure 5.25 below.

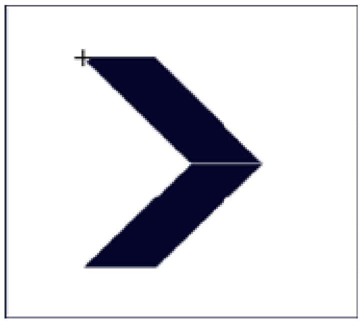


Figure 5.25: Connection Power Movie Clip (Zoomed)

Many instances of the “ConnecPower” movie clip are used to construct the movie clip named with “ConnectivityIcon” shown in Figure 5.26 below. The instances are named with a unique name, in order to control them and to simulate the connectivity power by showing and hiding the visual appearance of the appropriate instance. For example, to hide respectively show the visual appearance of the one of the instances, the property “_visible” can be set to “false” respectively to “true”, hence the instances are movie clips objects.

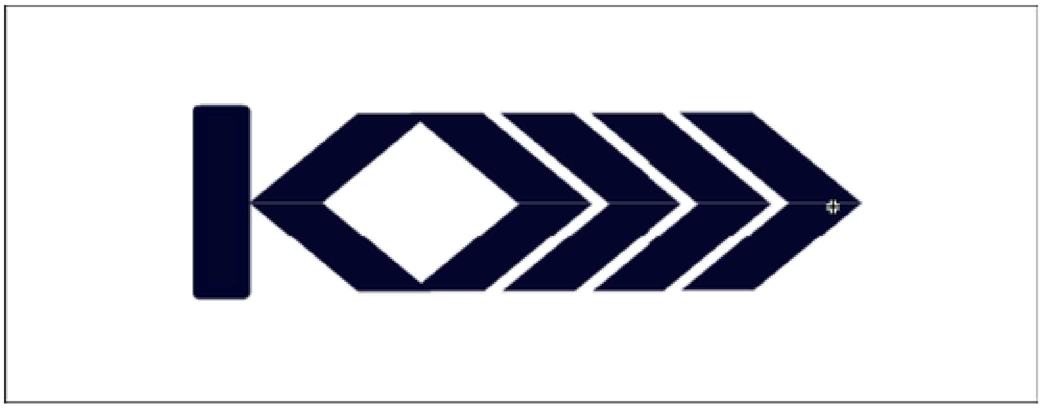


Figure 5.26: Connectivity Icon Movie Clip (Zoomed)

One of the movie clips of the device prototype is an animated sequence that has been converted to movie clip symbol. Macromedia Flash 8 offers several ways to include animation and special effects in a document. Timeline effects, such as blur, expand, and explode, make it easy to animate an object.

Further, with timeline effects a previously time-consuming task that required more advanced knowledge of animation can be achieved easily in a few steps.

The subject of this movie clip is a so called tweened animation, as shown in Figure 5.27.
To create tweened animation, a starting and ending frames must be created. Flash will then create the animation for the frames in between. Flash varies the object's size, rotation, color, or other attributes between the starting and ending frames to create the appearance of the motion.

Animation can be also created by changing the contents of successive frames in the timeline. An object can move across the stage, increase or decrease its size, rotate, change color, fade in or out, or change shape. Changes can occur independently of or in concert with other changes.

For example, the starting frame of the so called "TweenDisplay" is a solid dark rectangle and the ending frame is the same rectangle but transparent. The animation illustrated the disappearance of the rectangle slowly, but in a desired speed, which is in this case one second because the frame rate of the timeline is set to 12 frames per seconds and 12 frames are used in between.

The "TweenDisplay" movie clip is used to simulate the effect of the opening of the device display, when the used the on button pressed for a while.

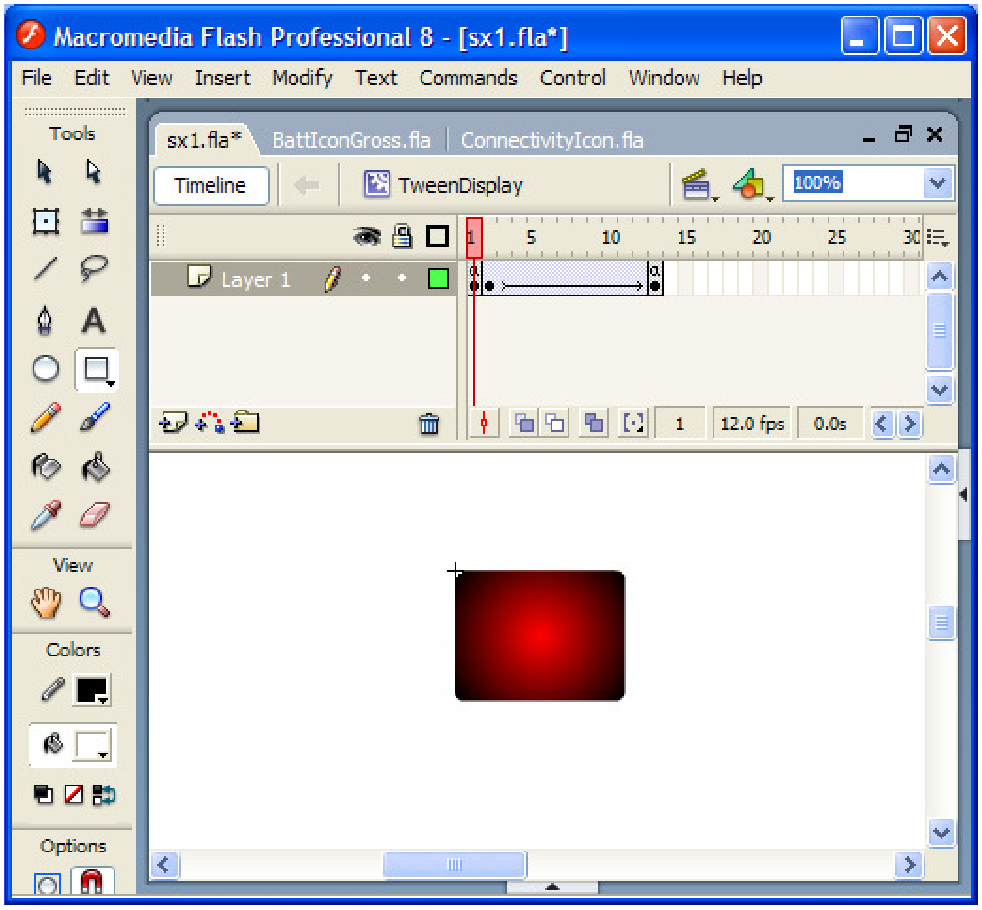

Macromedia Flash Professional 8 - [sx1.fla*]
File Edit View Insert Modify Text Commands Control Window Help
Tools
sx1.fla* BattIconGross.fla ConnectivityIcon.fla
Timeline
TweenDisplay
100%
Layer 1
1 5 10 15 20 25 30
1 12.0 fps 0.0s
View
Colors
Options

Figure 5.27: Tween Display Movie Clip

In order to label the device display, such as the menu items and displaying time and date, the label component provided by Flash or the text tool has been used.

There are three types of text. It can be static, dynamic or input text. The type of a text can be selected from the property inspector, as shown in Figure 5.28.

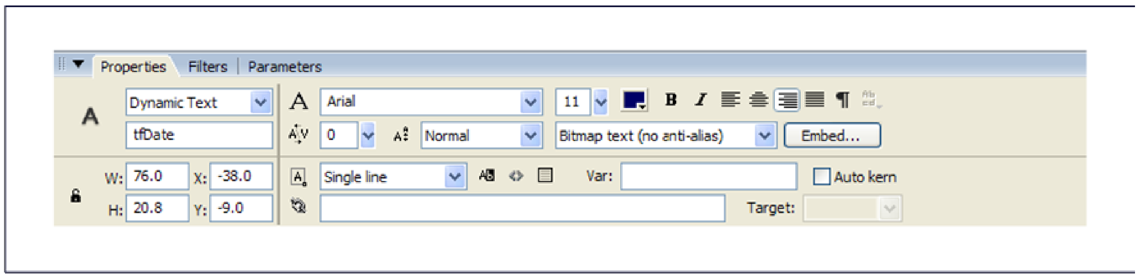
Dynamic text fields can display text dynamically. Its content can change and can be programmatically controlled at runtime. Input text fields are similar to those dynamic ones, they allow users to enter text. They can be used while designing forms, surveys etc.
But only static text can not be changed.

Like movie clip instances, text fields, of type dynamic or input, can be instantiated. Their instances are ActionScript objects that can be controlled through their properties and methods.

Macromedia Flash provides various ways to display, format and manipulate text in an easy manner. This can be done by setting the text attributes which are font, point size, style, color, tracking, kerning, alignment and many other attributes. Further, timeline effects allow developers and designers to apply pre-built animation effects to text, such as bouncing, fading in or out, and exploding or what ever.

While working with Flash FLA files, Flash substitutes fonts in the FLA file with other fonts installed in the local system. When the specified fonts are not in the system, options can be selected to control the fonts which should be substituted. Furthermore, Flash allows designers to create a symbol from a font so that it can be exported as a part of a shared library and can be then used in other Flash documents.

Formatting input and dynamic text and creating scrolling text fields can be easily done using ActionScript statements. In addition, dynamic and input text fields can handle events, which can be captured to achieve some tasks, for example to trigger scripts.



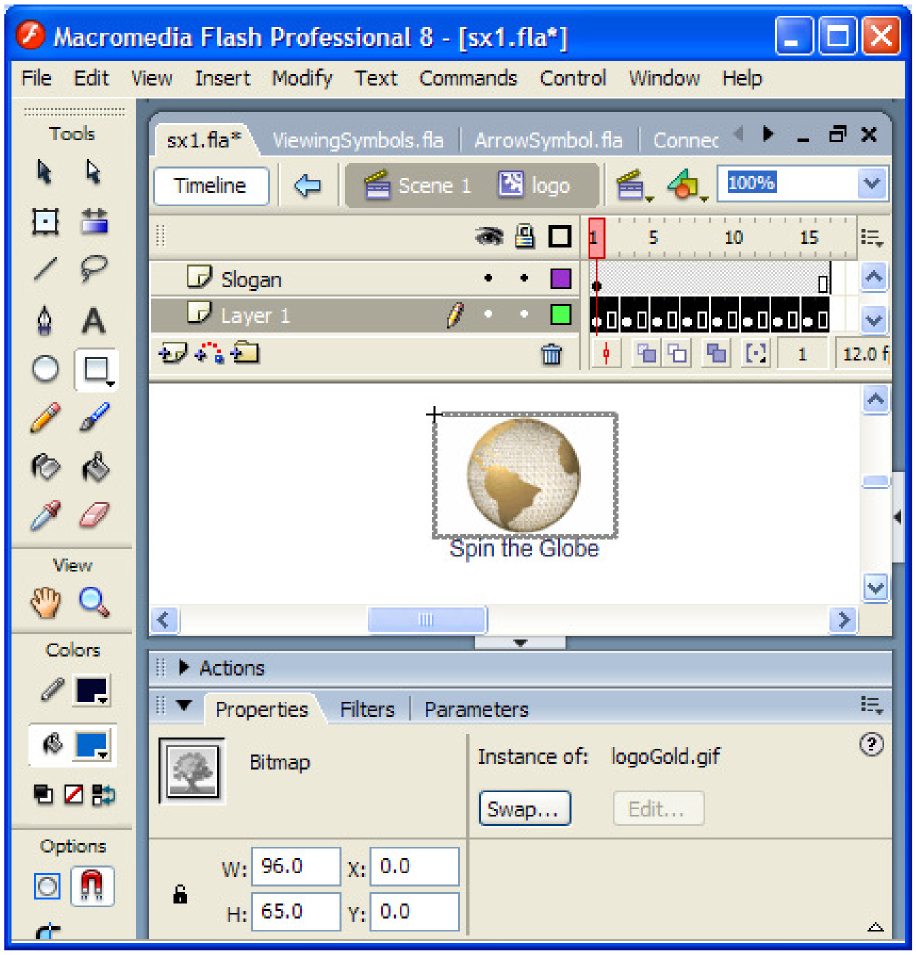
Dynamic Text
tfDate
Arial
11
0
Normal
Bitmap text (no anti-alias)
W: 76.0 X: -38.0
H: 20.8 Y: -9.0
Single line
Var:
Auto kern
Target:

Figure 5.28: Property Inspector of Dynamic Text

While authoring a document with Flash, assets can be imported into either the stage or to the library of the current document. Assets can be, in this case, sound, video, bitmap images and other graphic formats, such as PNG, JPEG, AI, and PSD.

Imported graphics are stored in the document's library. The library stores both the imported assets into the document and symbols that have been created by Flash.

For example, the graphics that have been imported to the device prototype Flash document are bitmaps file that represent the globe, as shown in Figure 5.29 below. By selecting them from the stage, they can be converted, as mentioned previously, to a movie clip symbol. This movie clip is named with “logo” and can be then instantiated and controlled at runtime.


Macromedia Flash Professional 8 - [sx1.fla*]
File Edit View Insert Modify Text Commands Control Window Help
Tools
View
Colors
Options
sx1.fla* ViewingSymbols.fla ArrowSymbol.fla Connec
Timeline
Scene 1 logo
100%
5 10 15
Slogan
Layer 1
1 12.0 f
Spin the Globe
Actions
Properties Filters Parameters

Bitmap

Instance of: logoGold.gif

Swap... Edit...

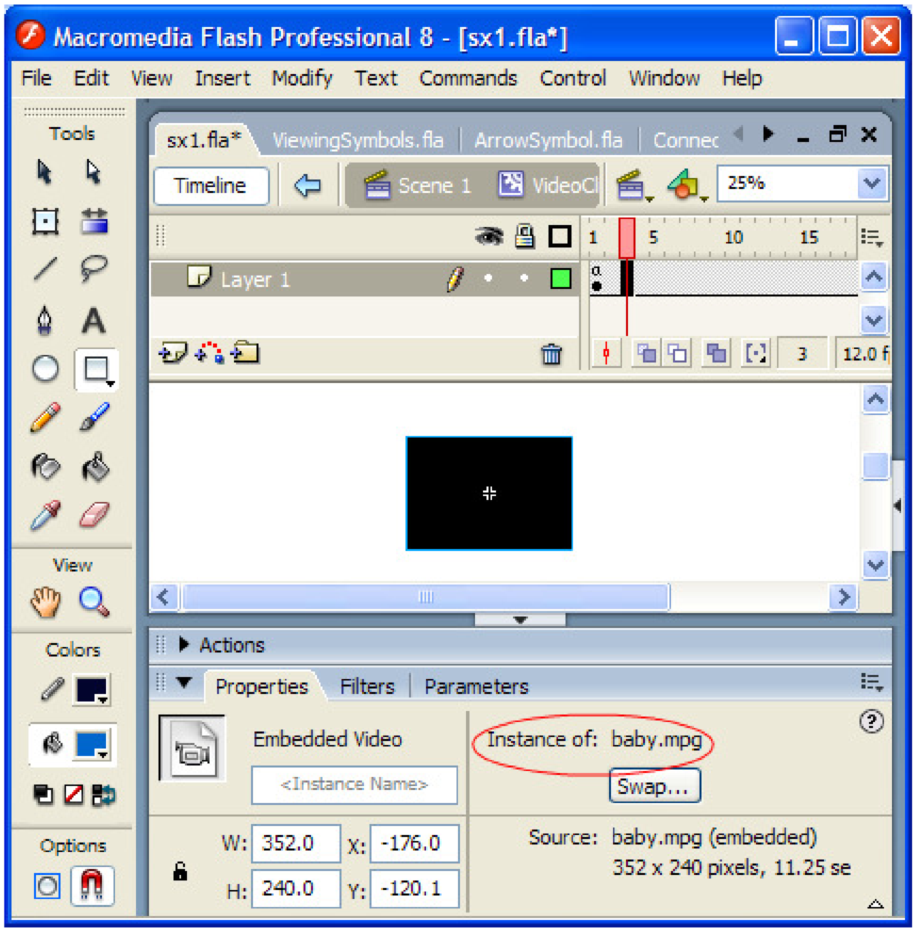
W: 96.0 X: 0.0
H: 65.0 Y: 0.0

Figure 5.29: The Logo Movie Clip

Macromedia Flash 8 is a powerful tool for incorporating video footage into web-based presentations. Flash Video offers technological and creative benefits that create immersive, rich experiences that fuse video together with data, graphics, sound, and interactive control. Flash Video enable putting video on a web page in a format that almost anyone can view easily.

Video clips can be imported into Flash as embedded files. Like any imported asset, an embedded video file becomes part of the Flash document. Video clips can be also converted to a movie clip symbol and can be then instantiated and treated as an object in the sense of object oriented way. Its properties can be then modified, as usual, using the property inspector.

The video can be used to simulate, for example, the multimedia message system (MMS) for mobile phones. For example, an instance of a video clip, as shown below in Figure 5.30 encircled within a red ellipsis, has been used for the device prototype to simulate playing video clip on the device.


Macromedia Flash Professional 8 - [sx1.fla*]
File Edit View Insert Modify Text Commands Control Window Help
Tools
sx1.fla* ViewingSymbols.fla ArrowSymbol.fla Connec
Timeline
Scene 1
VideoCl
25%
Layer 1
View
Colors
Options
Actions
Properties Filters Parameters

Embedded Video
<Instance Name>
Instance of: baby.mpg
Swap...
W: 352.0 X: -176.0
H: 240.0 Y: -120.1
Source: baby.mpg (embedded)
352 x 240 pixels, 11.25 se

Figure 5.30: Instance of a Video Clip

To achieve designing the visual appearance of the device prototype, many other Flash symbols have been created. To get an overview about all the symbols which have been used in the device prototype's Flash document, the icon symbol, as shown in Figure 5.31 encircled within a red circle, must be clicked with the mouse button. The pop up menu, as shown in Figure 5.31 encircled within a red ellipsis, will appear and the selected symbol can be then edited.

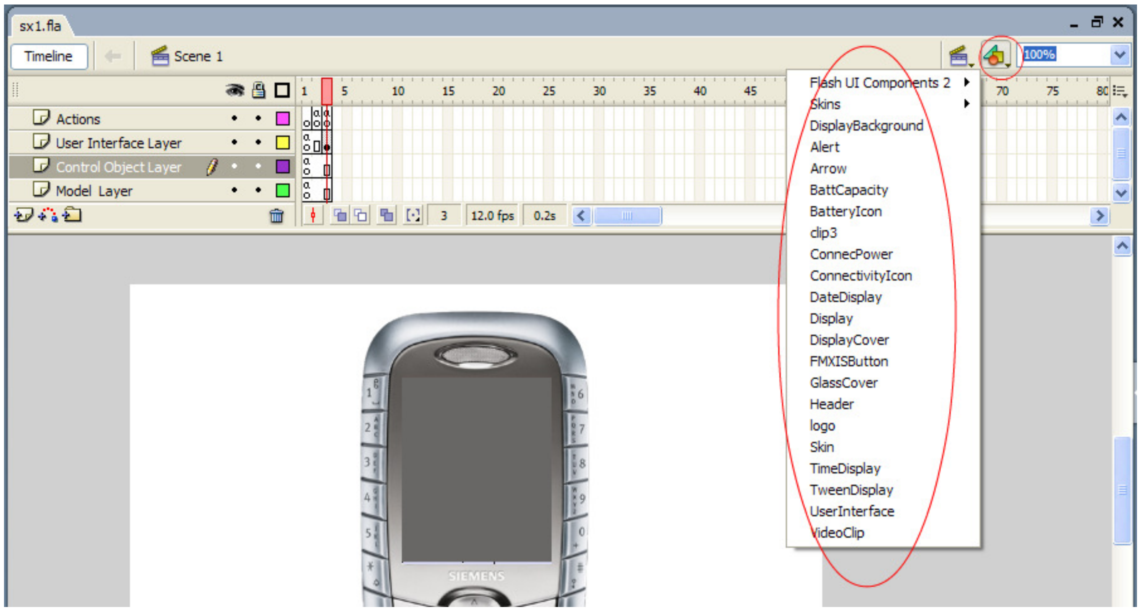


Figure 5.31: Overview of all Symbols in a Flash Document

5.3.1.3 The Skin Movie Clip of the Device Prototype

The skin movie clip represents the real visual appearance of the device prototype. It is composed of a photo-realistic of the device itself and the device periphery.

With periphery is meant, for example, keyboard, touch screen, buttons or whatever, which enable the end user to interact with the device using events sending. Within the scope of device simulation or animation the events are nothing else than mouse or keyboard events hence the virtual device is an application. These events should be captured and appropriately handled. For this purpose and as known that Macromedia Flash provides a way to create button symbol, the periphery can be designed easily.

In order to achieve designing the visual appearance, a photo-realistic image of the device prototype, in this case Siemens mobile phone "SX1" [87], has been imported to the stage and placed on the first layer named with "PhoneImage" of the timeline, as shown in Figure 5.32.

Further, to design the periphery of the virtual device prototype, the button component called "FMXISButton" [86] has been used. This component has been instantiated several times and placed, in a transparent way, over the phone image in a separate upper layer called "Buttons" of the timeline, as shown in Figure 5.32.

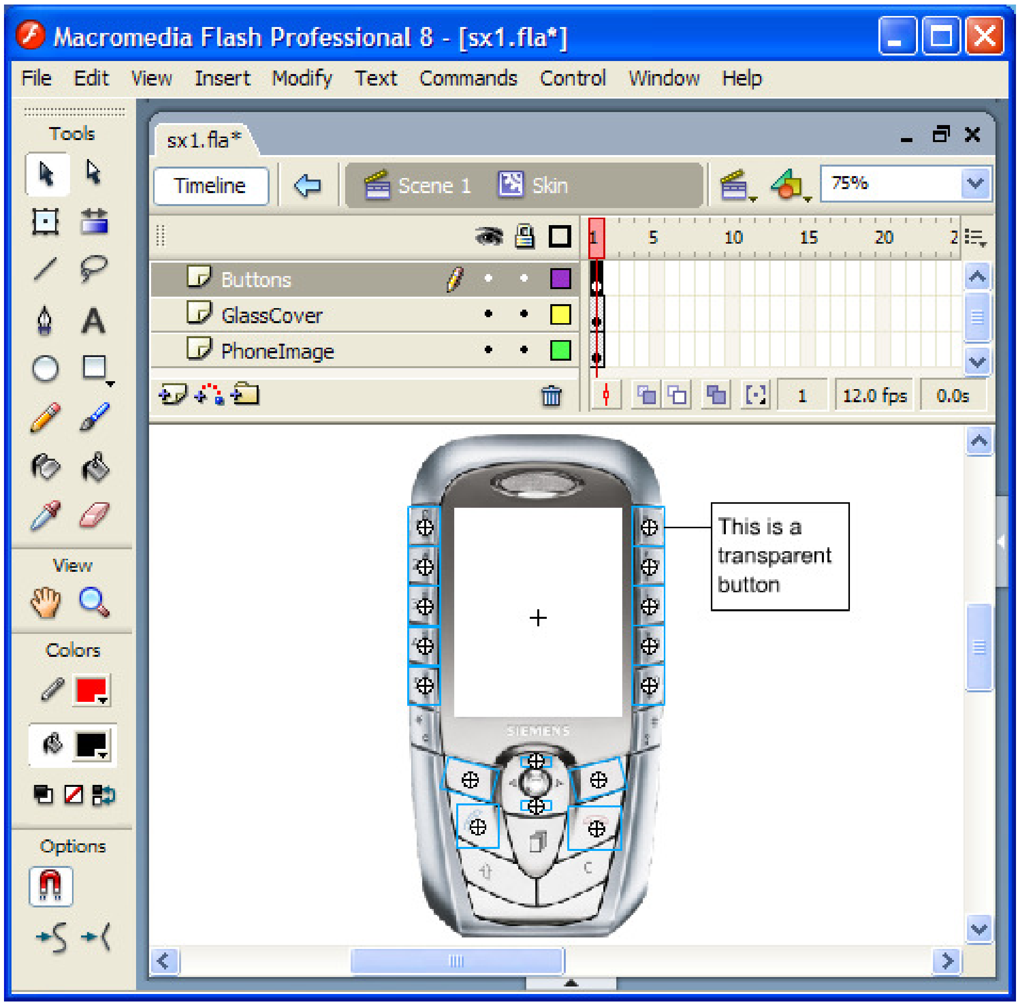
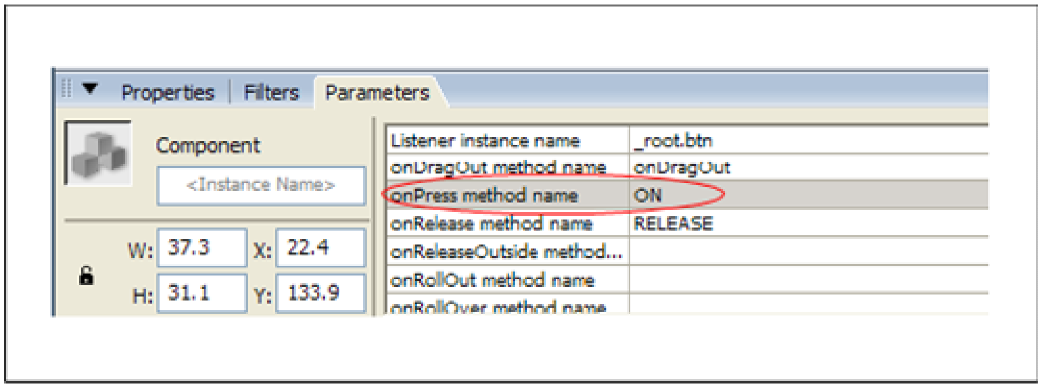

Macromedia Flash Professional 8 - [sx1.fla*]
File Edit View Insert Modify Text Commands Control Window Help
Tools
View
Colors
Options
sx1.fla*
Timeline
Scene 1
Skin
75%
Buttons
GlassCover
PhoneImage
1
12.0 fps
0.0s
This is a transparent button
SIEMENS

Figure 5.32: The Skin Movie Clip of the Device Prototype

The button component has a very useful parameter called “onPress method name”, as shown in Figure 5.33 encircled within a red ellipsis. The value of this parameter is an event name, which can be generated when user presses the button. To each button component an event name can be set and generated individually. In this way we can name the event that should be generated depending on the needs and necessity.

Freely naming the events is very important to activate the SDL state charts and to link their generated ActionScript classes from SDL models with the Flash graphic elements. For instance, to switch to device prototype the end user has to press the button “ON” for two seconds duration. As a Flash object, this button is an “FMXISButton” component, which can generate the event “ON”, as shown in Figure 5.33 encircled within a red ellipsis.



Listener instance name	_root.btn
onDragOut method name	onDragOut
onPress method name	ON
onRelease method name	RELEASE
onReleaseOutside method...	
onRollOut method name	
onRollOver method name	

Figure 5.33: Button Component Parameters Inspector

This generated "ON" event is used in the SDL model, as illustrated in Figure 5.34 encircled within a red ellipsis below. This event has the purpose to enable the transition between "Idle" state and the so called "InterpretEvent" state.

As events are essential parts of state charts and, on the other hand, events can be freely named and generated with the button component, the linkage between SDL models and Flash elements can be done easily, which is naming the intended events the same in the SDL models side and in Macromedia Flash side.

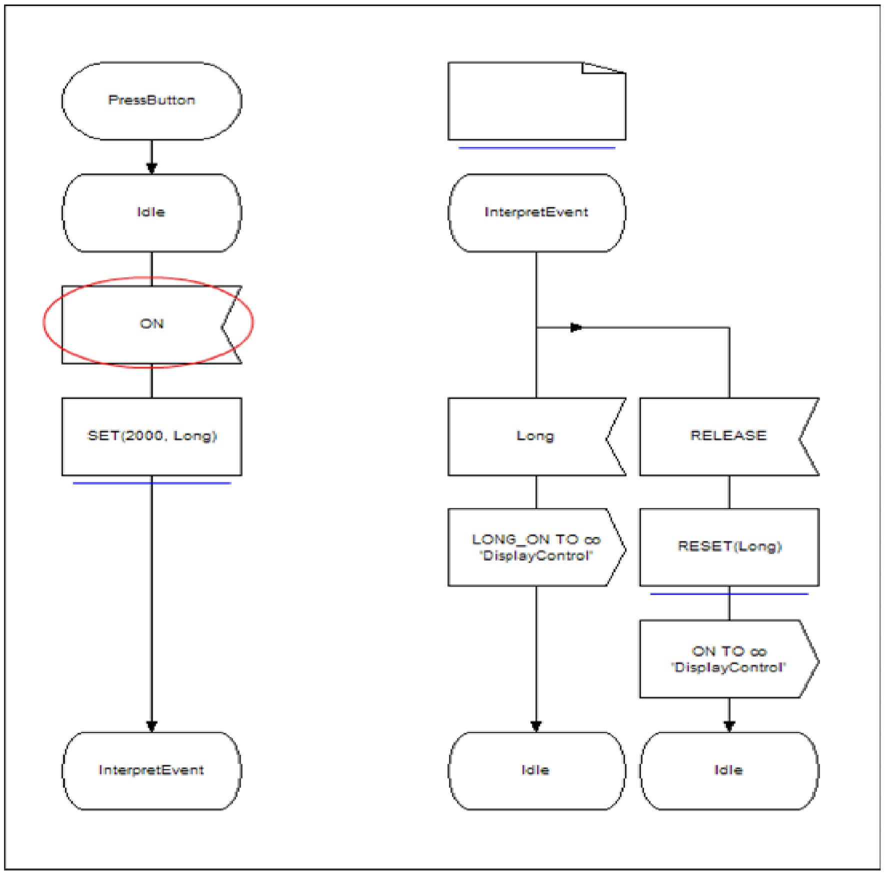


Figure 5.34: The Use of a Generated Button Event in SDL State Charts

5.3.1.4 The Display Movie Clip of the Device Prototype

The display is the most important and interesting part of the visual appearance of the device prototype. Its design needs preliminary considerations before being done.

In other words, the interaction between the end user and the device can be noticed through the changes of the display looks. It means, when the user pressed a certain button the logo display, for example, will be shown or when pressing another button the main menu display will appear and so on. Keeping in mind the states in a state charts and taking into consideration the changing of the display looks an idea had arisen, which is to map the diverse display looks to a diverse states. In this way the different looks of display can be coordinated to state transitions. It means, when a state has been activated or entered in a state charts the corresponding display look must be shown.

That's why the timeline of the display movie clip has many key frames. The stage of each key frame contains the graphic elements of the corresponding display look. To transition between the frames, it means to change between the display looks, the timeline control methods "gotoAndPlay()" or "gotoAndStop()" of ActionScript provided by Macromedia Flash can be used.

For example, in the first frame of the timeline, the first display look (state) has been placed on the stage, as shown in Figure 5.35 below.

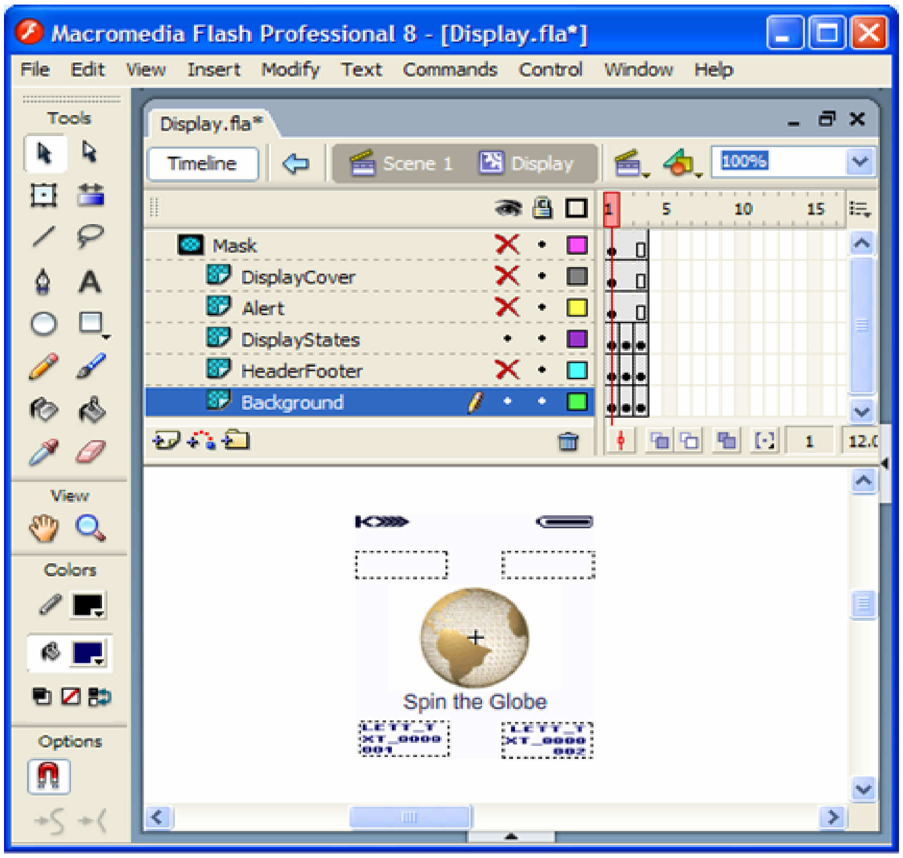


Figure 5.35: First Display State

Analogously to the first one, the second respectively the third display look (state) has been placed on the stage of the second respectively the third key frame of the time line, as shown in Figure 5.36 respectively in Figure 5.37. Then the corresponding display look can be drawn and the needed assets can be imported.

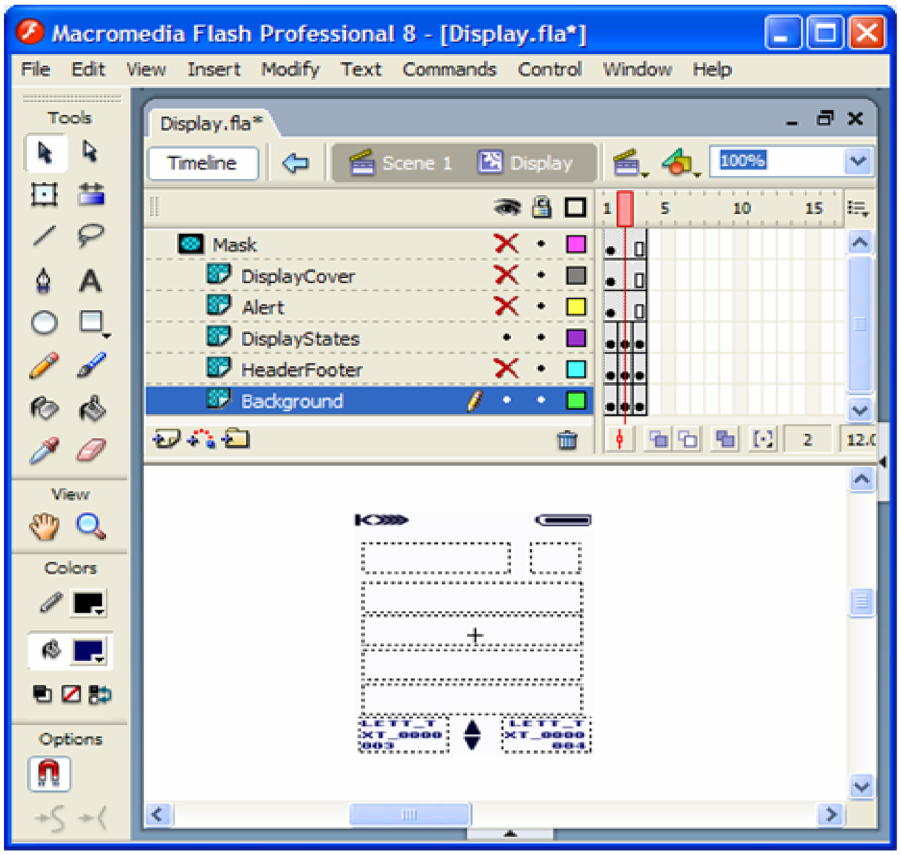


Figure 5.36: Second Display State

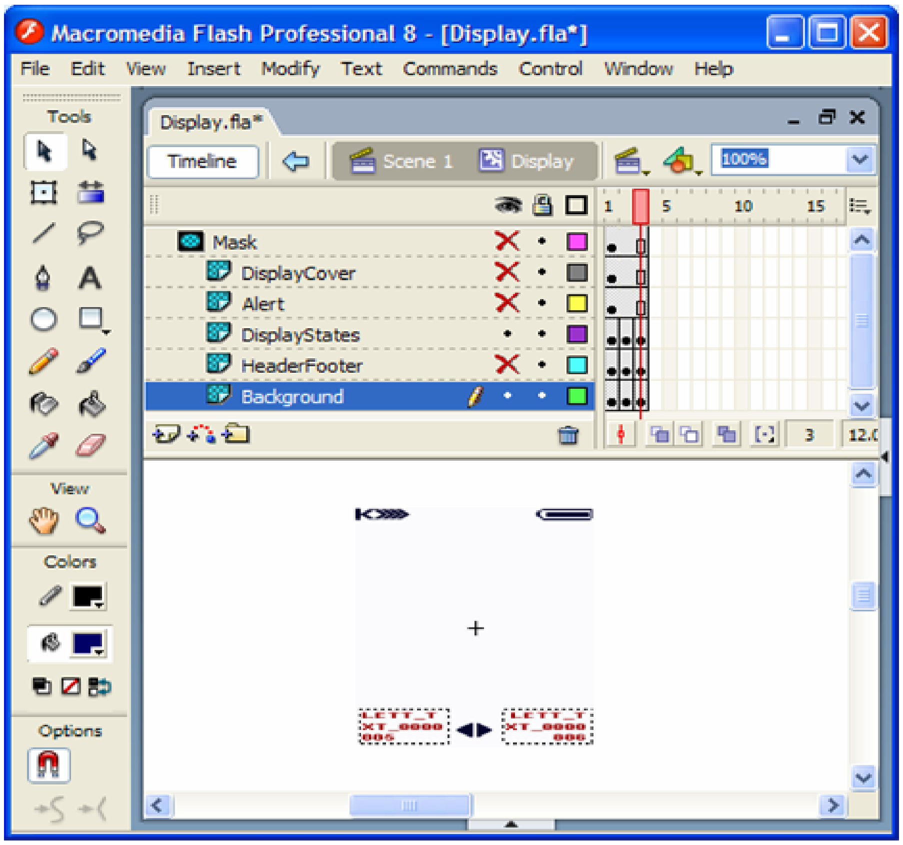


Figure 5.37: Third Display State

In the case that another display look is needed, it’s easy to extend the state charts with another state. The new display look can be drawn easily in the next following key frame of the timeline of the display movie clip symbol.

5.4 Implementation of the UCM Layers

The UCM architecture, that has been presented previously, forms the basis of a scaleable architecture for developing device prototypes. The architecture defines a separation between the device interfaces and the underlying device behaviour layers. Applied to the creation of device interfaces, the architecture promotes good design practice because it centralizes the coordination of the user interface, making the interface behaviour easier not only to understand and design but also to validate. Further, the architecture dictates controlled access to system information.

In the following subsections, the implementation of each layer of the UCM architecture will be presented and illustrated.

5.4.1 Implementation of the User Interface Layer

The user interface corresponds to the virtual visual appearance of the device prototype. Accomplishing the implementation of this layer is nothing else than designing and constructing the interface elements using Macromedia Flash environment.

The starting point is to place these elements on the stage, label them, initialize their properties and prepare them for coordinated use. For this purpose, the display and the skin movie clip, as explained in the previous subsections, have been used to construct the whole user interface.

In order to achieve the design of the user interface, a new Flash document has been opened at the beginning. By selecting the sub-menu item “New Symbol” from the menu item “Insert” a new symbol of type movie clip named with “UserInterface” has been created, as shown in Figure 5.38. The first layer of the timeline of the movie clip has been renamed to “Display” in order to give designers and developers information about the kind of the content of this layer. At the first frame of the “Display” layer an instance of the display movie clip has been placed on the stage and has been named with “theDisplay”. In this manner the display instance can be controlled programmatically, as explained previously, within the so called “UserInterface” movie clip.

Analogously, another layer has been inserted above the “Display” layer, and has been renamed to “Skin”. At the first frame of the “Skin” layer an instance of the skin movie clip has been placed on the stage and has been named with “theSkin”, in order to control it at runtime.

In other words, as simple as it sounds, the whole user interface of the device prototype is nothing else than a Flash movie clip symbol that can be instantiated everywhere, every time and in every way.

Further, for reusability purposes, the user interface can be converted easily to a component, as explained in [82], assigned to an appropriate ActionScript class and installed in the components panel of the Macromedia Flash environment.

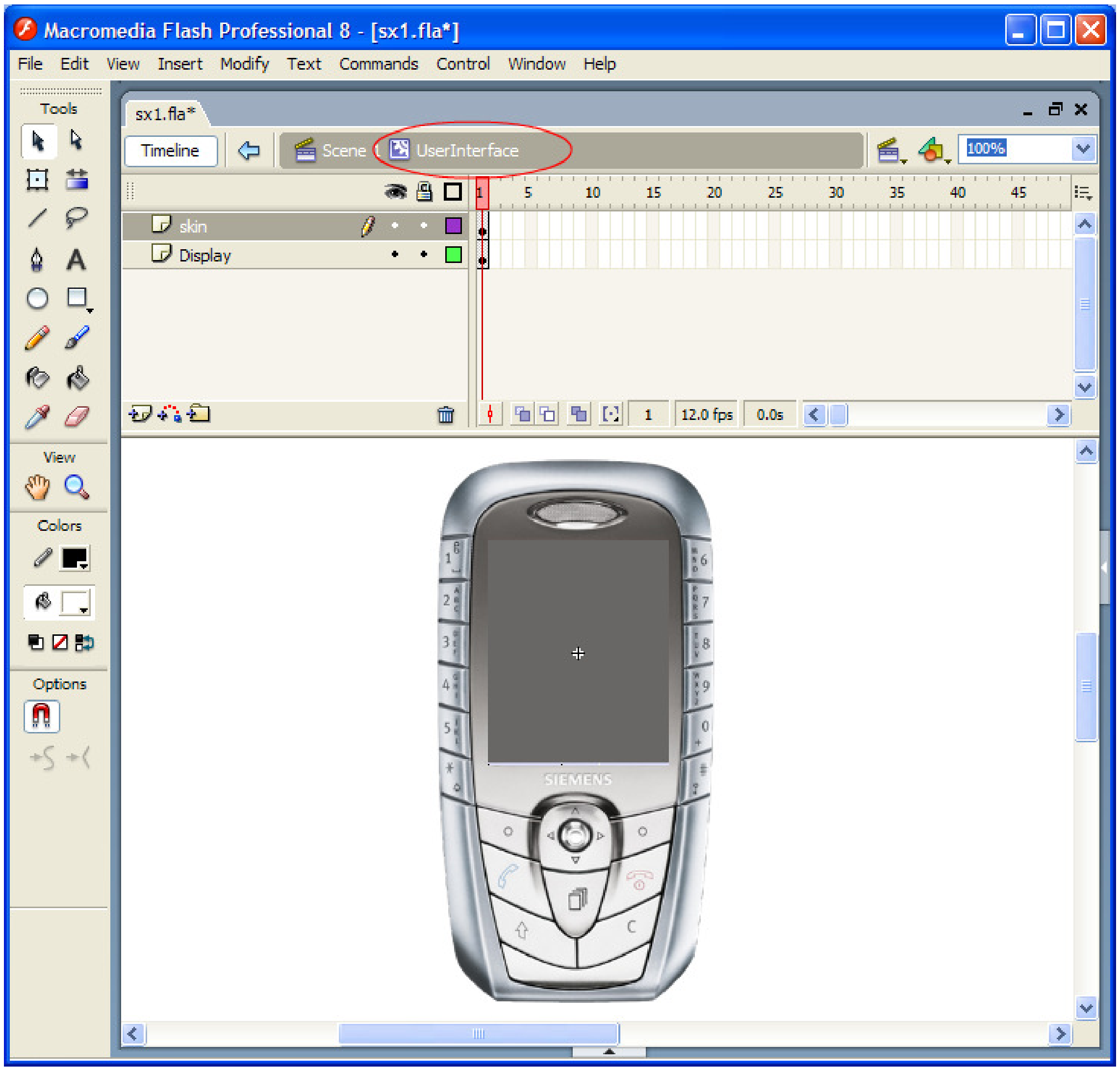

Macromedia Flash Professional 8 - [sx1.fla*]
File Edit View Insert Modify Text Commands Control Window Help
Tools
View
Colors
Options
sx1.fla*
Timeline
Scene 1
UserInterface
100%
skin
Display
12.0 fps
0.0s
SIEMENS

Figure 5.38: User Interface Movie Clip

5.4.2 Implementation of the Control Object Layer

The control object layer consists of the framework necessary to coordinate the user interface. It means, it has the responsibility, on the one hand, to coordinate the interface elements, and on the other hand to mediate the interface and the device internal functionality, which is the model layer.

To design this layer efficiently, SDL state charts have been chosen to describe the behaviour of the device while interaction with the end user.

In order to achieve designing the SDL models of the control object layer, the second most important part of GRAPE, namely the SICAT tool set, particularly the Process Diagram (PD) Editor, has been used.

As the control object layer is an extended state machine, preliminary considerations about its states should be well thought out. Certainly, the states of the display movie clip, as explained previously, must be taken into consideration while modelling the control object layer. It means the states of the SDL model of the control object should be attuned to the states of the display. Because, while interaction, the behaviour of a device can be noticed mostly on the display changes.

As the device can be switched on or off, two states to represent this context of “switched on” and “switched off” are needed. These states are named with “OFF” and “ACTIVE”. Two additional states, namely “Navigation”, “Entertain” are necessary to describe the different display states. Obviously, other states can be needed, while designing, for encapsulation or coherence purposes.

The entire SDL model of the control object layer is illustrated in the following figures: Figure 5.39, Figure 5.40 and Figure 5.41.

Moreover, more detailed models can be seen on the companion CD-ROM (please refer to Appendix C).

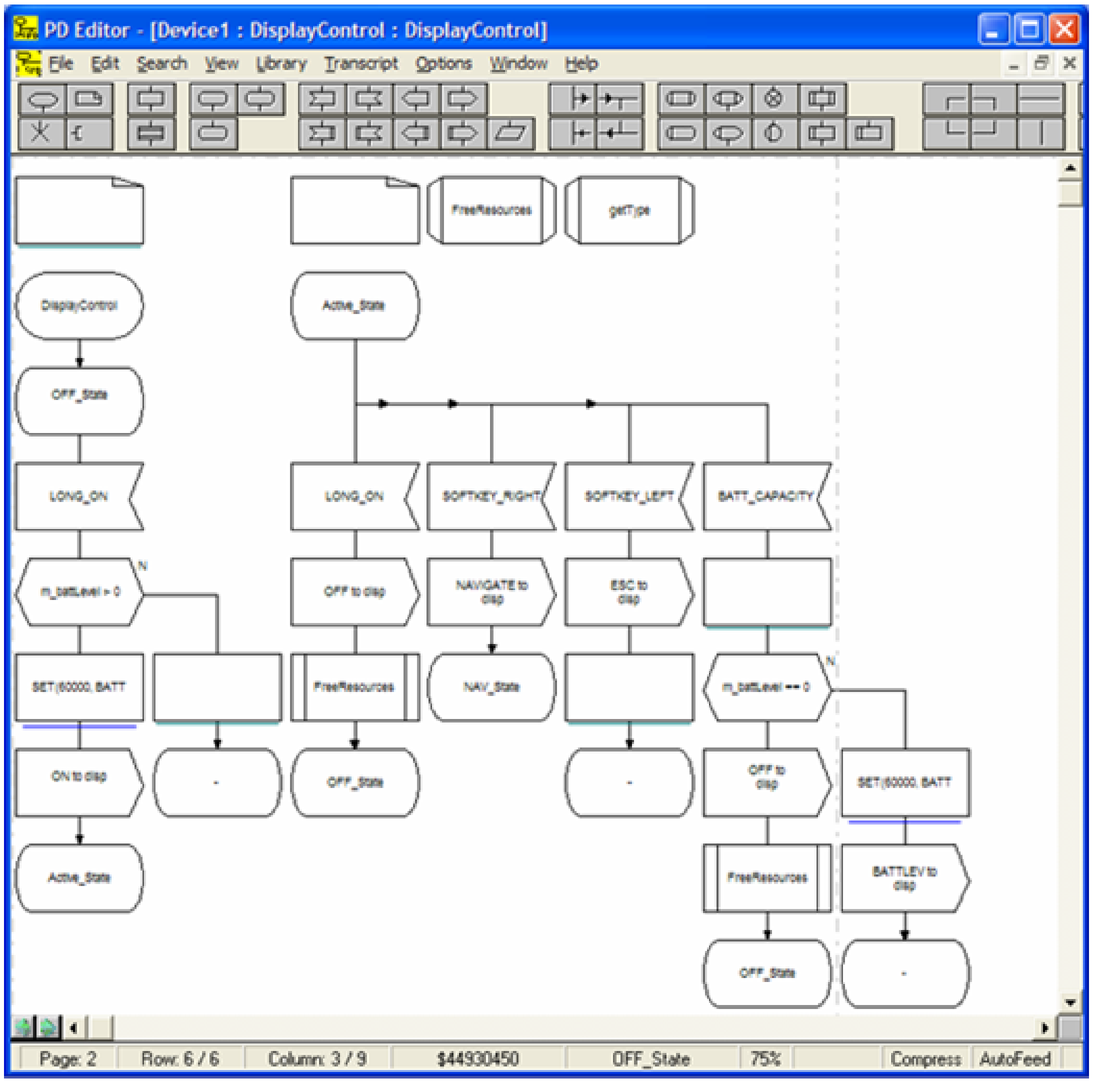


Figure 5.39: SDL Model of The Control Object Layer (Part 1)

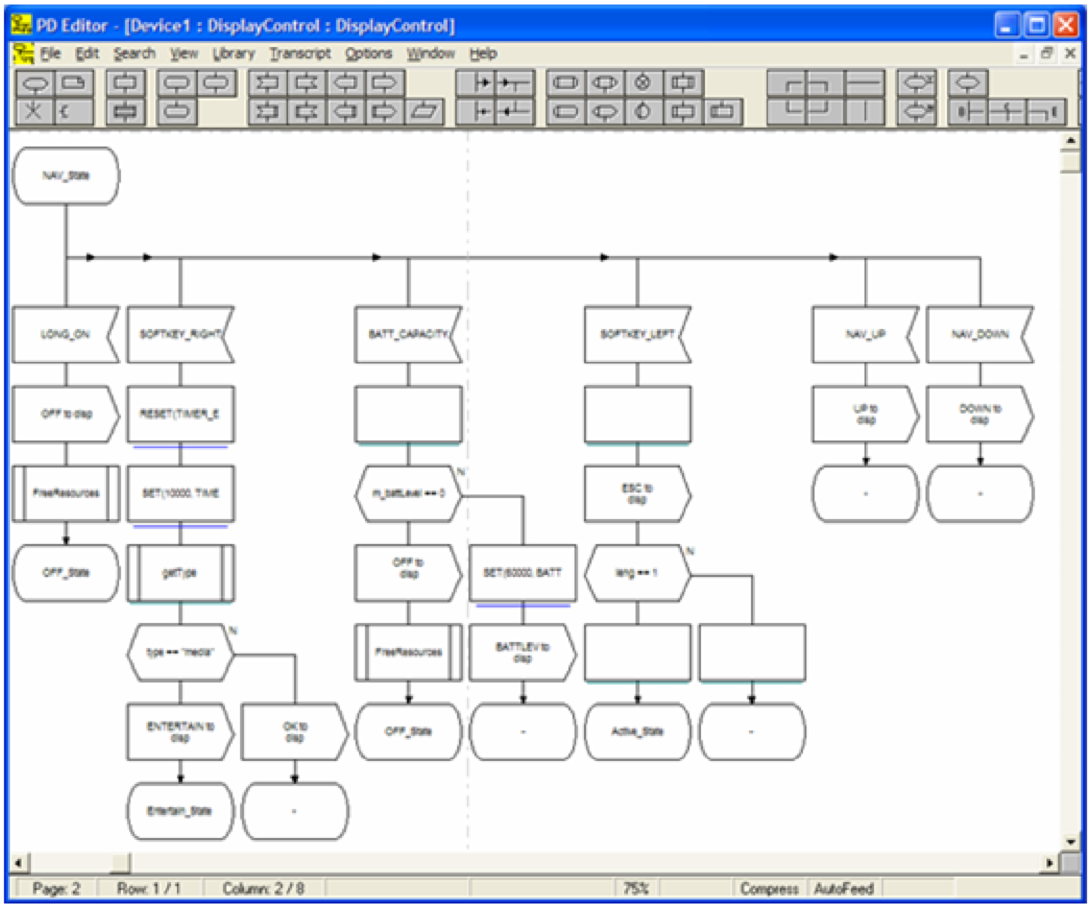


Figure 5.40: SDL Model of The Control Object Layer (Part 2)

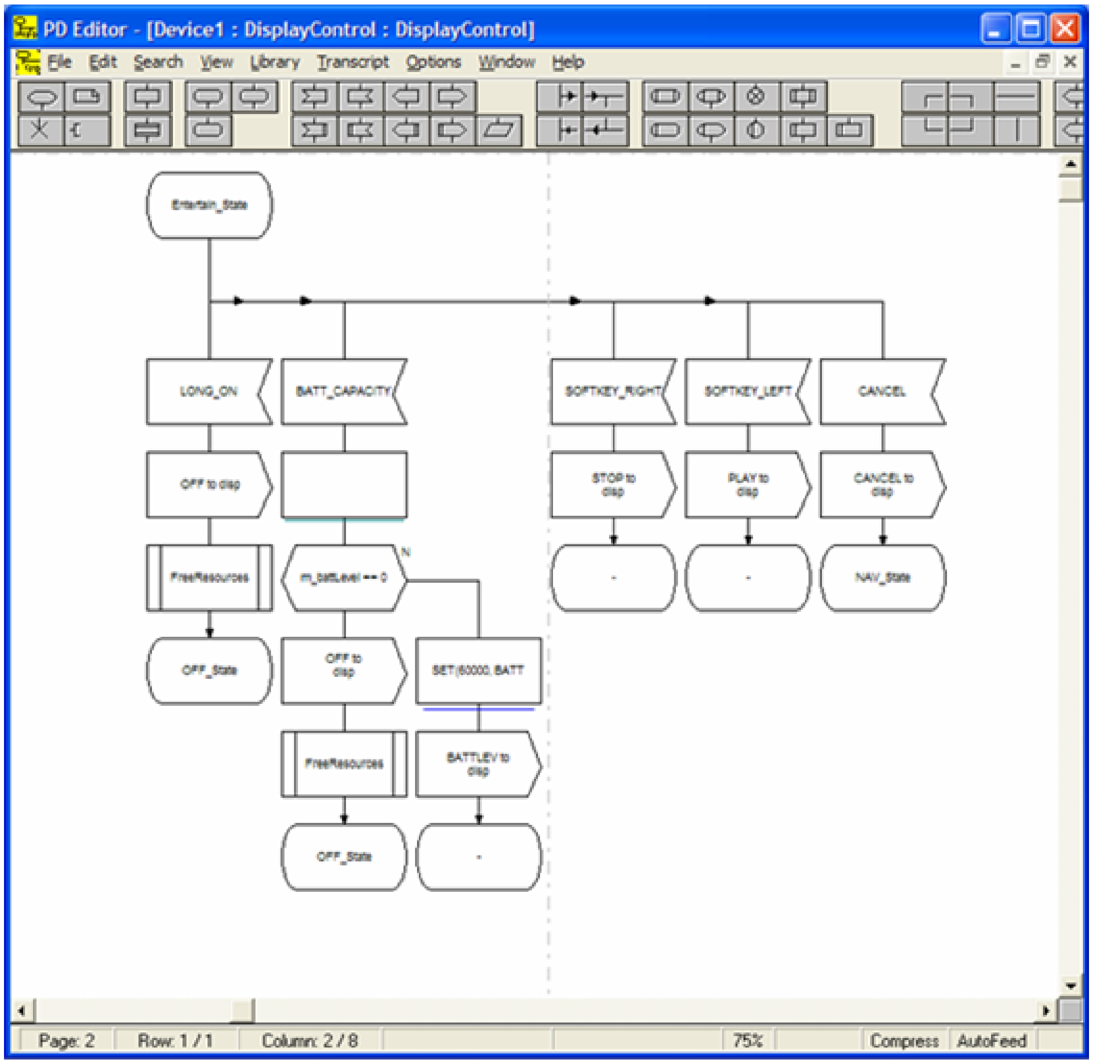


Figure 5.41: SDL Model of the Control Object Layer (Part 3)

To summarize: the control object layer is a set of an extended state machines modelled with SDL state charts. From these models, ActionScript classes can be generated automatically. These classes can be then bound to Flash document of the device prototype.

In this way, the control object can keep track of the context of the device and the communication between the UCM layers can take place with sending messages and method invocation. This means that the control object can, for example, send messages, on the one hand, to interface elements such as setting a certain value. The interface elements are only responsible for performing their immediate function, such as button press, display darkening and so on, and inform the control object about what they did, not to take on any of the device processing by themselve.

5.4.3 Implementation of the Model Layer

The model layer is essentially responsible for the internal functionality of the device and provides methods that can be invoked by the control object layer.

The model layer is the code necessary to support some internal system functionalities such as battery level, menu items that are managed in XML file, device initialisation and so on. This code can be in terms of ActionScript methods.

To design this layer efficiently, SDL state charts have been chosen to describe the internal system behaviour and functionality of the device.

Like the control object layer, the SDL models of the model layer can be designed using the Process Diagram (PD) Editor of SICAT Tool set.

The model layer is also an extended state machine that reacts to the device being switched on and off. The states of the SDL model of the model layer should be attuned to the states of the control object layer, because the model layer process is slightly expanded from that of the control object.

As the device can be switched on or off, two states to represent this context of “switched on” and “switched off” are needed. Other states can be needed, while designing, depending on needs and necessity.

The entire SDL model of the model layer is illustrated in the following Figure 5.42. Moreover, more detailed models can be seen on the companion CD-ROM (please refer to Appendix C).

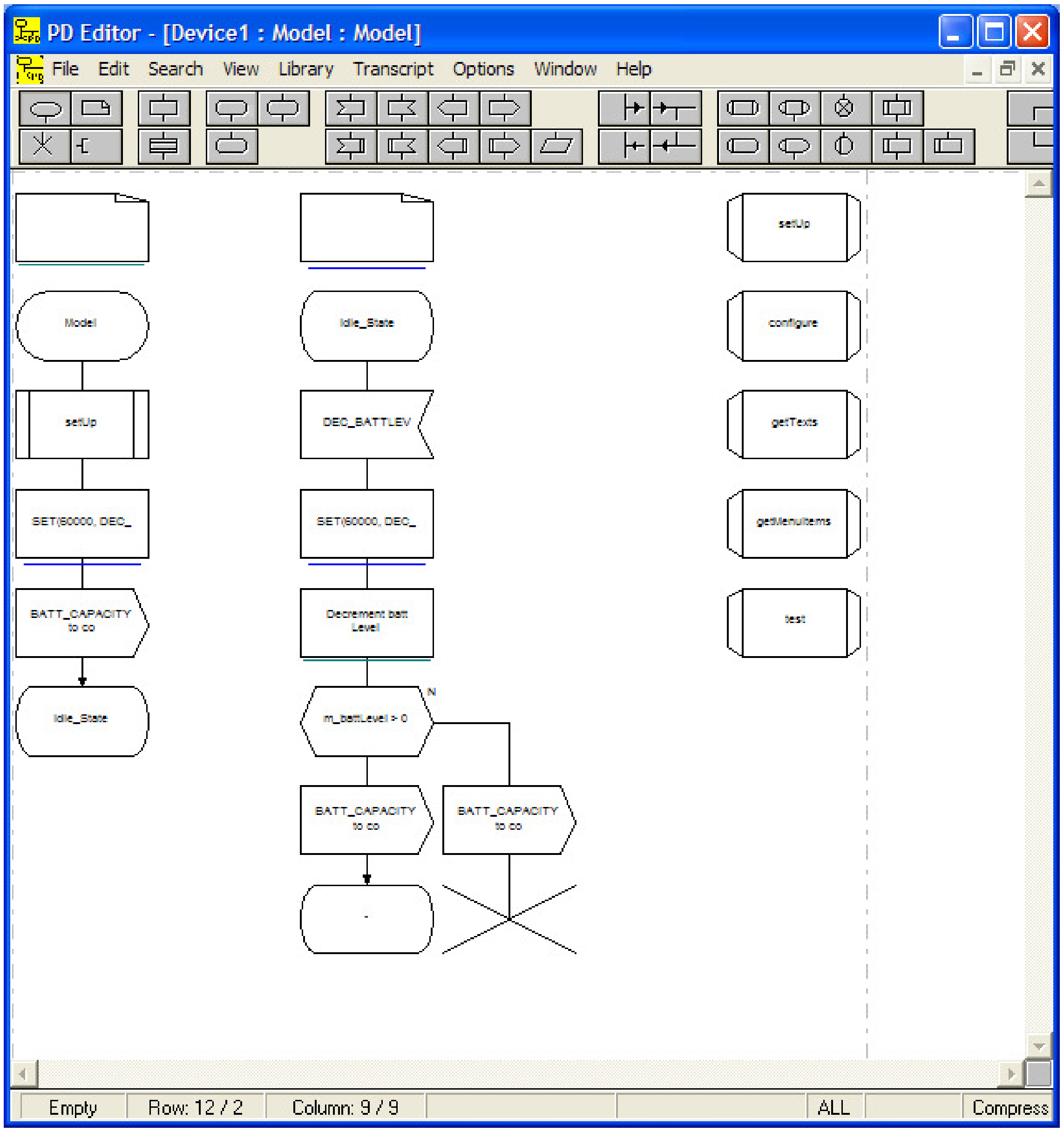


Figure 5.42: SDL Model of the Model Layer

Finally, the model layer is an extended state machine modelled with SDL state charts. From these models, ActionScript classes can be generated automatically. These classes provide methods that describe the internal system functionality and other system initialisation mechanism and can be then bound to Flash document of the device prototype.

5.5 Putting It All Together

As mentioned previously, GRAPE is based on two major components. The first one is called “SICAT”, which is responsible for describing the behaviour of the device, for generating code and supports the documentation of the device. The second one is “Macromedia Flash Professional 8”, which achieves the graphical design, manages the graphic elements and creates the virtual interactive device prototype.

The starting point is to design the visual appearance of the device prototype and to place the graphic elements on the stage, label them, initialize their properties and prepare them for coordinated use. Then the device behaviour can be modelled using SDL state charts.

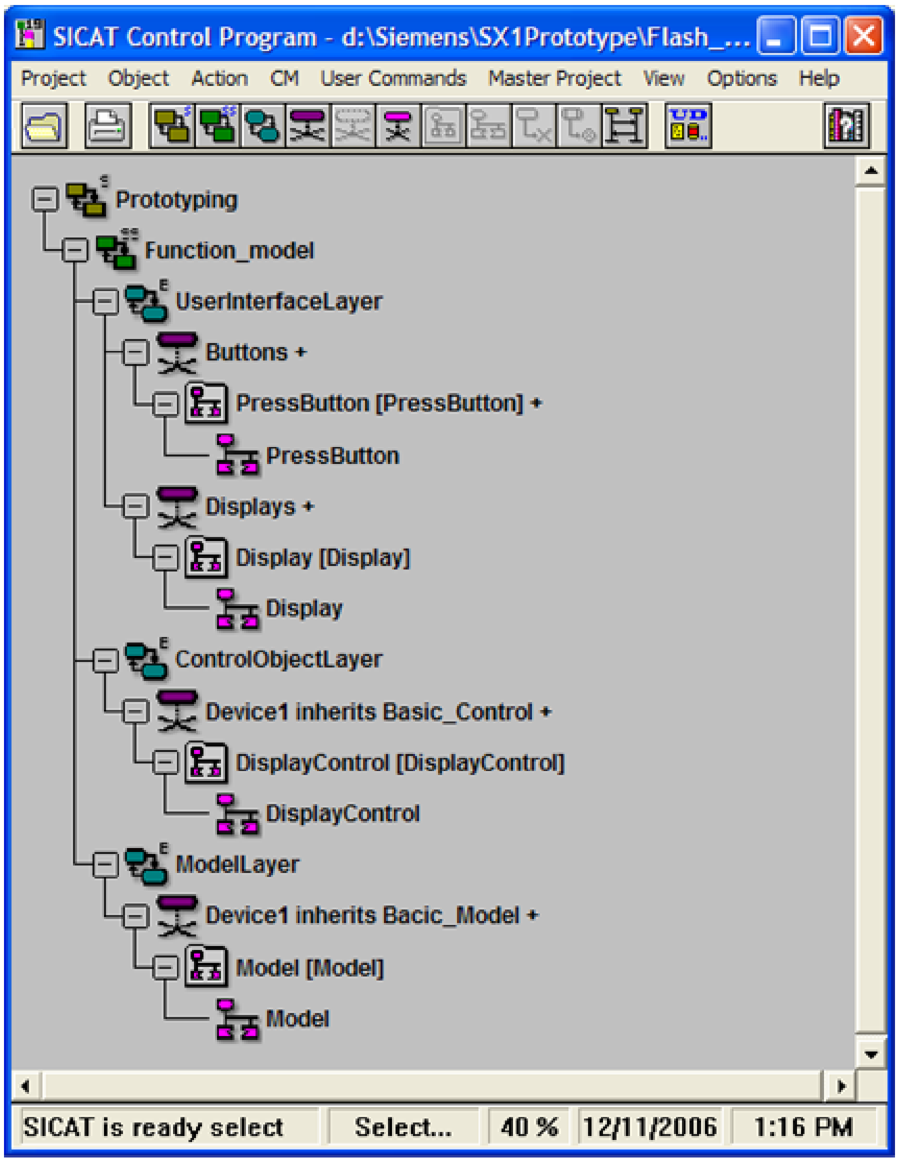
The behaviour of the device has been modelled using SICAT Toolset. This latter manages the SDL diagrams created by the Process Editor diagram and provides a tree overview of the structure of the description of the so called device prototype system[6] using the Control Program of SICAT, as shown in Figure 5.43.

In order to achieve modelling the whole device prototype behaviour, four different process diagrams have been created and which will be described as follows.

The first diagram is the so called “DisplayControl” of the Block Object named with “ControlObjectLayer” on the Control Program of SICAT, as shown Figure 5.43.
This diagram represents the SDL model of the control object layer of the UCM architecture and which has been explained in the subsection 5.4.2. The diagram can be opened by double clicking on the Process Object Type from the Control Program of SICAT. From the SDL model an ActionScript class can be generated. This class is called “DisplayControl” which can be instantiated to represent the control object layer. Its code is saved in “DisplayControl.as” file (please refer to Appendix C).

The second diagram is called “Model” of the Block Object named with “ModelLayer” on the Control Program of SICAT, as shown Figure 5.43.
This diagram represents the SDL model of the model layer of the UCM architecture.
The diagram can be opened by double clicking on the Process Object Type from the Control Program of SICAT. From the SDL model an ActionScript class can be generated. This class is called “Model” which can be instantiated to represent the model layer. Its code is saved in “Model.as” file (please refer to Appendix C).

[6] System is a term of SDL terminology



Prototyping
Function_model
UserInterfaceLayer
Buttons +
PressButton [PressButton] +
PressButton
Displays +
Display [Display]
Display
ControlObjectLayer
Device1 inherits Basic_Control +
DisplayControl [DisplayControl]
DisplayControl
ModelLayer
Device1 inherits Bacic_Model +
Model [Model]
Model

SICAT is ready select | Select... | 40 % | 12/11/2006 | 1:16 PM

Figure 5.43: Device Prototype System on SICAT Side

The third diagram is called “PressButton” of the Block Object named with “UserInterfaceLayer” on the Control Program of SICAT, as shown Figure 5.43.

This diagram, as shown in Figure 5.44, represents an SDL model of a press button and describes the behaviour when the user pressed the button for longer than a certain amount of time (i.e. 2 Seconds). In this way the opening behaviour of the device can be simulated.

The press button generates the events “ON” respectively “RELEASE” when it is pressed respectively released. As shown in Figure 5.44, when the user presses the button the transition between “Idle” and “InterpretEvent” states is triggered, and a timer is started to see if the user holds down the button sufficiently long. If that timer elapses after 2 seconds a timer event called “Long” will be generated. If the state machine is in state “InterpretEvent ” and the event “Long” arrives, the event named with “LONG_ON” will be forwarded to the control object instance called “co”.

This diagram is related to the button component that has been explained in subsection 5.3.1.3, and is part of the user interface layer of the UCM architecture. The diagram can be opened by double clicking on the Process Object Type from the Control Program of SICAT. From the SDL model an ActionScript class can be generated. This class is called “PressButton” which can be instantiated. Its code is saved in “PressButton.as” file (please refer to Appendix C).

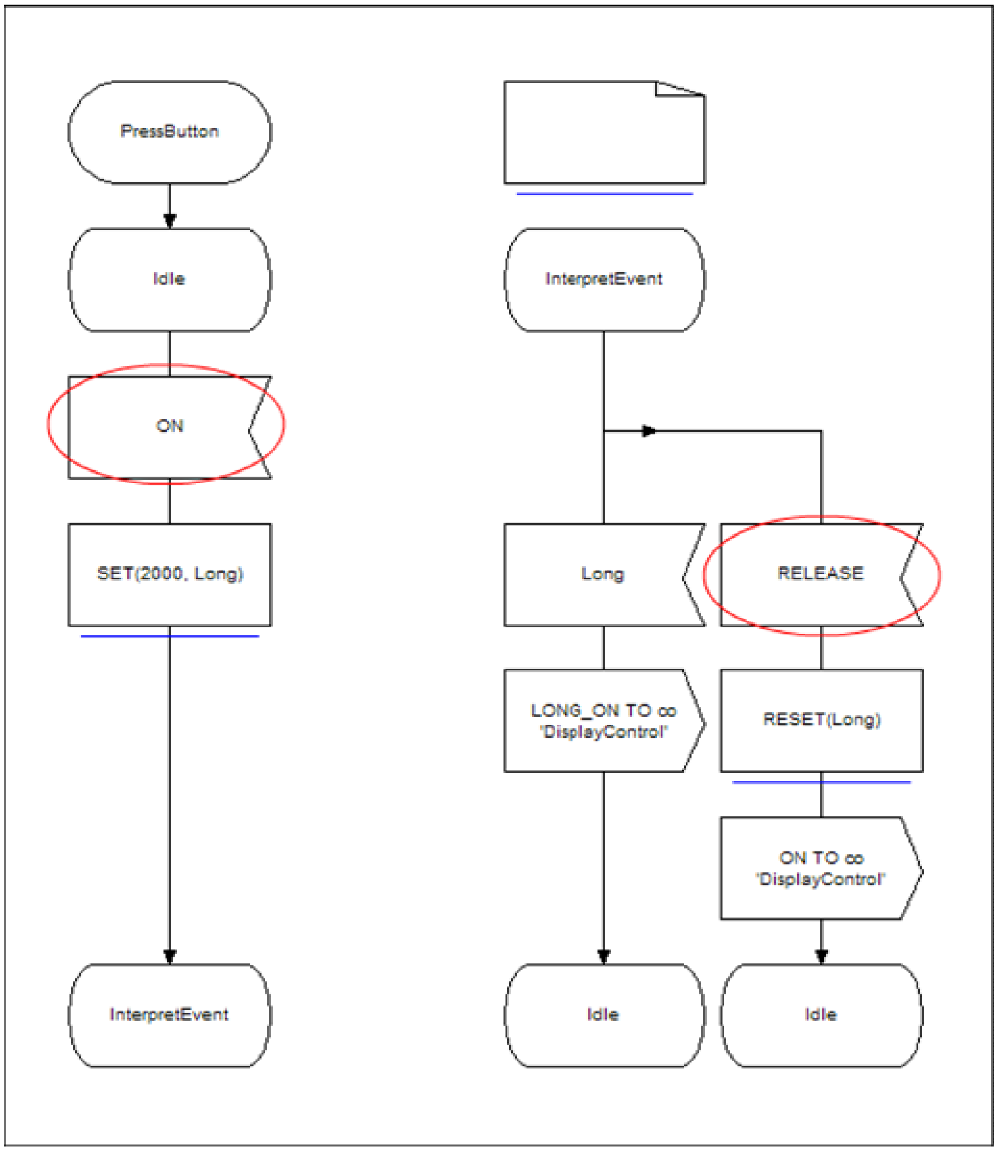


Figure 5.44: SDL Model of The Behavior of Press Button

The fourth diagram is called “Display” of the Block Object named with “UserInterfaceLayer” on the Control Program of SICAT, as shown Figure 5.43.

This diagram represents an SDL model of the display and describes the different states of the display movie clip. The states of this state machine are attuned to the states of the SDL models of the control object layer. In this way, the communication between the control object layer and the display can take place by sending signals between each other.

This diagram is related to the display movie clip that has been explained in subsection 5.3.1.4, and is part of the user interface layer of the UCM architecture. The diagram can be opened by double clicking on the Process Object Type from the Control Program of SICAT. From the SDL model an ActionScript class can be generated. This class is called “Display” which can be instantiated. Its code is saved in “Display.as” file (please refer to Appendix C).

Let’ summarize in between: In order to achieve modelling the whole device prototype behaviour four different process diagrams have been created. From these SDL models ActionScript classes have been generated. The use of theses classes will then demonstrated as follows.

Let’s turn back to the starting point which is designing the visual appearance of the device prototype and to placing the graphic elements on the stage, labelling them, initializing their properties and preparing them for coordinated use.

To achieve the task of interface panel arrangement, Macromedia Flash 8 environment has been used. At the beginning, a new Flash document has been opened and saved under the name “sx1.fla” (Please refer to Appendix D to see it).

The first layer, at the bottom, as shown in Figure 5.45, of the main timeline of the document, and named with “Model Layer” represents the model layer of the UCM architecture. At its first key frame an instance of the “Model” generated class has been declared. This frame contains only ActionScript code, which is “*model = new Model(this);*”. In this way the state machine of the model layer can be controlled programmatically through the instance called “model”.

Analogously, the second layer called “Control Object Layer” represents the control object layer. At its first key frame an instance of the “DisplayControl” generated class has been declared. This frame contains only ActionScript code, which is
“*co = new DisplayControl(this);*”. In this way the state machine of the control object layer can be controlled programmatically through the instance called “co”.

The third layer named with “User Interface Layer” represents the user interface layer of the UCM architecture. At its first key frame instances of “PressButton” and “Display”generated classes have been declared. This frame contains only ActionScript code, which is
“*btn = new PressButton(this); disp = new Display(this);*”. In this way the state machine of the press button respectively the display can be controlled programmatically through the instances called “btn” respectively “disp”.

In this manner all the state machines concerning the behaviour description have been bound timely. These state machines have to be activated now. This has been done in the fourth layer called "Actions" at its second key frame by called the method "Start()" to the corresponding instance object. For example, with ActionScript statement "*co.Start();*" the state machine of the control object layer can be activated.

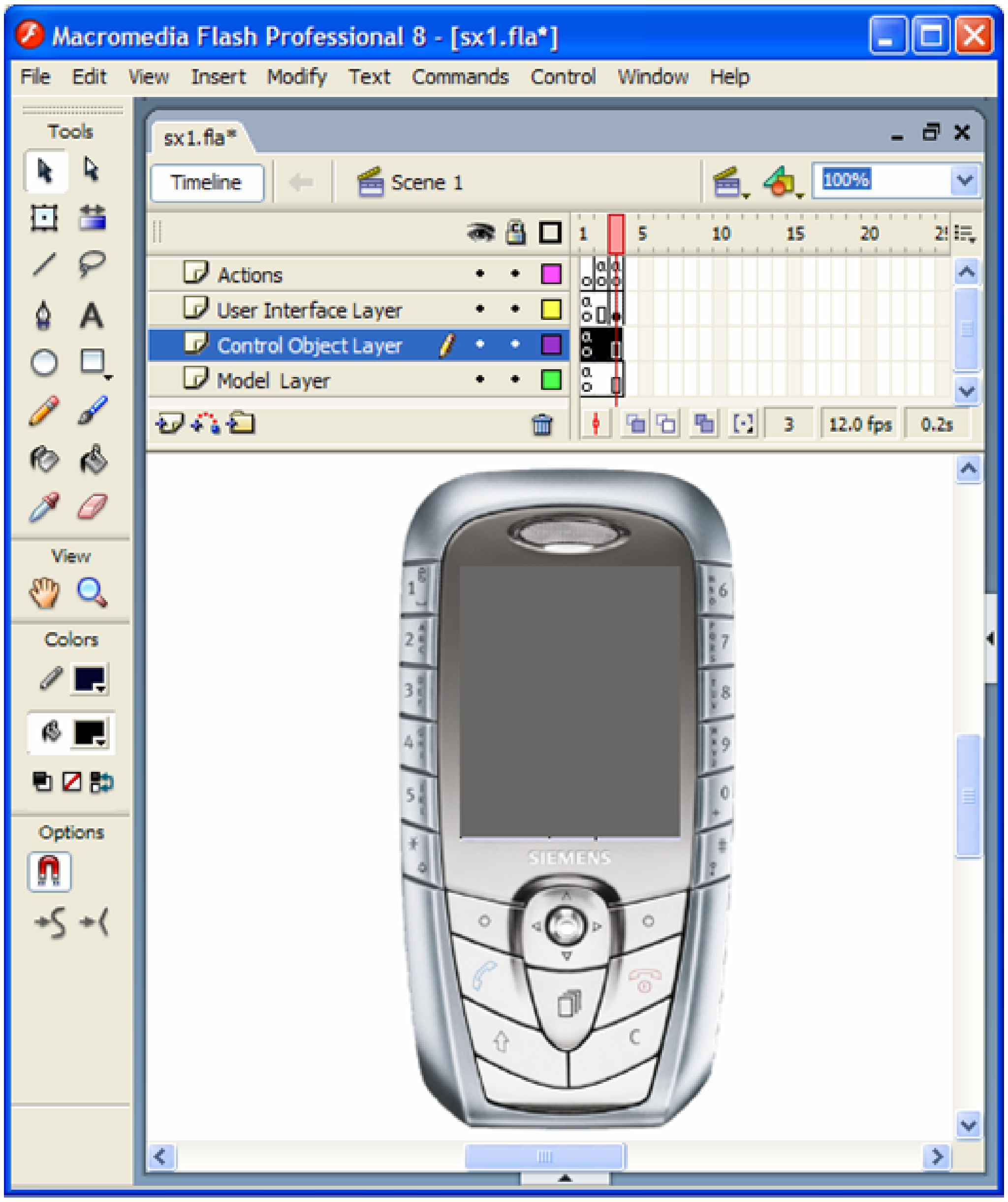


Figure 5.45: Device Prototype on Macromedia Flash 8 Side

After finishing binding and activating the state machines concerning the behavior description of the device, something must be done now with the visual appearance of the device prototype. That's what has happened by simply placing an instance of the user interface movie clip symbol that has been explained in subsection 5.4.1 and named with "theUI", on the stage in the second key frame of the so called "User Interface Layer" layer.

With this last task the design of the functional device prototype has been finished and it needs now to be started to enable interaction with it. Running the device prototype is the topic of the following subsection.

5.5.1 How to Run the Device Prototype

In order to run the Device Prototype, the Flash document named with "sx1.fla" has to be opened with Macromedia Flash 8 firstly. From the menu item called "Test Movie" of the menu "Control" or typing the shortcuts "Ctrl+Enter", the prototype movie clip can be started.

Then the prototype will appear, as shown in Figure 5.46. By pressing the right mouse button on the "ON" button of the device for at least 2 seconds, the device prototype will be switched on. Furthermore, the end user can interact with the device prototype by pressing the corresponding button on the device. For example, to navigate between the menu rows of the display up respectively down, the button "up" respectively "down" can be pressed. The device prototype can be switched off by pressing again the right mouse button on the "ON" button of the device for at least 2 seconds.

Further, Macromedia Flash provides a way to publish a Flash movie automatically by embedding it in an HTML site. This can be done by simply selecting the menu item called "Publish" from the menu "File".

The generated HTML site, containing the device prototype as an embedded Flash movie, can be then ameliorated depending on needs and can be transformed to a presentation or whatever of the device. The HTML site can be, at the end, placed on a server, which enables the end user to access it by typing the corresponding URL on its local browser and starting experimenting and interacting with the virtual device prototype.

Finally, the Flash document of the device prototype and its relevant files can be seen on the companion CD-ROM (please refer to Appendix C).

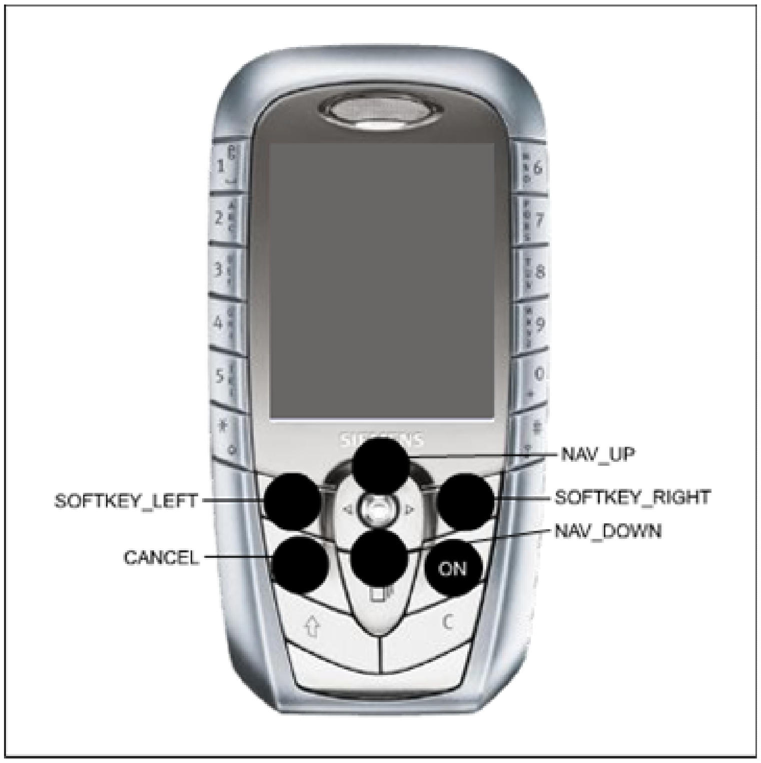


Figure 5.46: The Device Prototype

5.6 Summary

Prototypes of smart devices can be developed with GRAPE environment. On the one hand, the device behaviour can be specified with SICAT toolset using SDL State charts. ActionScript classes corresponding to SDL models can be then generated automatically.

On the other hand, the visual appearance of the device and its graphical components can be designed with Macromedia Flash 8. This latter has been identified to be a powerful graphic design environment. It provides a flexible methodology to design graphic elements or objects, which are in terms of symbols in Flash terminology.

In order to design virtual device prototypes, a methodology for managing the complexity of device prototype development, based on SDL state charts and the UCM architecture, has been chosen.

The UCM architecture is a “top down” approach, explained by Horrocks [2], which supports a centralized control access of the system information and facilitates the implementation and inspection of how the interface is coordinated in each context.

UCM stands for “**U**ser Interface - **C**ontrol Object - **M**odel”, which describes a three-tiered system to conceptualize the way a software program or device can be decomposed.

The UCM architecture is a scaleable architecture based on an engineering practice. It separates the device elements into three layers.

The principles of code generation, the construction of the device graphic elements and the implementation of each UCM layer were the topic of this chapter. Binding and managing all these tasks together, to create a virtual device prototype has been explained. Its result in terms of files and code can be seen on the companion CD-ROM (please refer to Appendix C).

6 Summary and Future Work

GRAPE, (**G**UI **Ra**pid **P**rototyping **E**nvironment) is a software development environment intended for rapid GUI prototyping of smart devices. Its conception and design has been achieved within the scope of this Thesis.

The most significant benefit, which GRAPE can provide for rapid prototyping, is saving time, costs and resources. GRAPE holds many others important and tangible benefits, which can be summarised as follows:

- Quick time to market
- Lower device prototype development costs
- High quality stakeholder feedback at an early stage (requirement review)
- Enabling usability and design review (look and feel)
- Automatically generated code and documentation
- Prototype deployment via Internet (Flash movie clip)
- Fully customizable features (in-house tooling)

During the design and conception of GRAPE some areas for future work have been identified. Some of these ideas can be implemented and adapted to GRAPE in order to achieve the most important milestone of GRAPE, which is using Macromedia Flash as an embedded device UI. It means the models and graphic elements created by GRAPE can be reused in the real target device.

Macromedia provides a Flash player which has been ported, for example to x86, ARM, MIPS, and many other processors, and to Windows XP Embedded, Windows CE, Linux, QNX, and BeOS embedded operating systems. Since the Flash player runs at application level and does not talk directly to hardware, the porting can be feasible. Further, Macromedia also offers an SDK with source code, a test suite, and documentation for large volume device manufacturers.

Bibliography

[1] **Miro Samek**. ***Practical Statecharts in C/C++***. CMPBooks, CMP Media LLC San Francisco, CA USA, 2002.

[2] **Ian Horrocks**. ***Constructing the User Interface with Statecharts***. Addison-Wesley, Harlow, England, 1999.

[3] **Jonathan Kaye, David Castillo**. ***Flash MX for Interactive Simulation***. Delmar Learning, Clifton Park, NY USA, 2003.

[4] **Laurent Doldi**. ***SDL Illustrated***: Visually design executable models. TMSO, Toulouse, FRANCE, 2001.

[5] **Laurent Doldi**. ***UML 2 Illustrated***: Developing Real-Time and Communications Systems. TMSO, Toulouse, FRANCE, 2001.

[6] **B. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven. *Prototyping, An Approach to Evolutionary System Development.*** Springer-Verlag, Berlin Heidelberg, Germany, 1992

[7] **John Connel, Linda Shaffer. *Object-Oriented RAPID PROTOTYPING***. Prentice-Hall, Englewood Cliffs, New Jersey USA, 1995

[8] **Todd Grimm*.User's Guide to Rapid Prototyping.*** Society of Manufacturing Engineers (SME), Michigan USA, 2004

[9] **Sheshire Henbury. *Rapid Prototyping.***
URL: http://www.cheshirehenbury.com/rapid/what.html

[10] **SDL Forum Society. *Specification Description Language.***
URL: http://www.sdl-forum.org/SDL/index.htm

[11] **SDL Forum Society. *Message Sequence Charts.***
URL: http://www.sdl-forum.org/MSC/index.htm

[12] **International Telecommunication Union. *ITU Recommendation.***
URL:http://www.itu.int/home/index.html

[13] **Adobe.com. *Macromedia Flash Professional 8.***
URL:http://www.adobe.com/products/flash/flashpro/

[14] **Walt Scacchi. *Process Models in software engineering.*** In J.J. Marciniak (ed.), Encyclopedia of Software Engineering, 2nd Edition, John Wiley and Sons, Inc, New York, December 2001

[15] **MacCormack A., *Product-Development Practices that Work: How Internet Companies Build Software.*** Sloan Management Review, 75-84, Winter 2001

[16] **Balzer, R., T. Cheatham, and C. Green. *Software Technology in the 1990's: Using a New Paradigm.*** *Compute*r,16,11, 39-46, 1983

[17] Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven. ***Approaches to Prototyping.*** Springer-Verlag, New York, 1984

[18] Hekmatpour, S., ***Experience with Evolutionary Prototyping in a Large Software Project***. ACM Software Engineering Notes, 12,1, 38-41 1987

[19] John Crinnion. ***Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology.*** Plenum Press, New York, 1991. Page 17

[20] S. P. Overmyer. ***Revolutionary vs. Evolutionary Rapid Prototyping: Balancing Software Productivity and HCI Design Concerns***. Center of Excellence in Command, Control, Communications and Intelligence (C3I), George Mason University, 4400 University Drive, Fairfax, Virginia

[21] Alan M. Davis. ***Operational Prototyping: A new Development Approach.*** IEEE Software, September 1992. Page 71

[22] Software Productivity Consortium. ***Evolutionary Rapid Development***. SPC document SPC-97057-CMC, version 01.00.04, June 1997. Herndon, Va. Page 6

[23] E. Bersoff and A. Davis. ***Impacts of Life Cycle Models of Software Configuration Management***. Comm. ACM, Aug. 1991, pp. 104-118

[24] Ian Sommerville. ***Software engineering, Chapter 8, 6.Auf*..**Addison-Wesley, Munich Germany, 2001

[25] Nikolai N. Mansurov, Dmitri Vasura. ***Scenario-Based Approach to Rapid Prototyping of Human-Machine Systems.*** Russian Academy of sciences, Moscow, 2001

[26] David P. Wood, Kyo C. Kang. ***A Classification and Bibliography of Software Prototyping.*** Technical Report CMU/SEI-92-TR-013, ESD-92-TR-013, October 1992, NTIS, U.S., Department of Commerce, Springfield, VA 22161

[27] Vasian Cepa.***Product Line Development for Mobile Device Applications with Attribute Supported containers.*** TU Darmstadt, Germany, 2005

[28] Edward A Lee. ***Embedded Software***. Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002

[29] ***Report of the Defense Science Board Task Force on Military Software.*** Office of the Under Secretary of Defense for Acquisition, US DoD, September 1987.

[30] Monkevich, O. ***SDL-based Specification and Testing Strategy for Communication Network Protocols.*** In: Proc. 9 th SDL Forum, Montreal,Canada, June 21-26, 1999

[31] Boehm, B. W., et al. ***Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.*** IEEE Transactions on Software Engineering, vol 1, no 1, March 1975.

[32] Tavolato, P., K. Vincena. ***A Prototyping Method and Its Tool.*** In Approaches to Prototyping, R. Budde et al., eds., Berlin: Springer-Verlag, 1984. pp. 434-446

[33] Bugs in the Program: ***Problems in Federal Government Computer Software Development and Regulation.*** Staff Study by the Subcommittee on Investigations and Oversight, One Hundred First Congress, April 1990

[34] Software Technology Plan: Vol. II Plan of Action. Draft 5, August 15, 1991

[35] Basili, V., and Weiss, D.. ***Evaluation of a Software Requirements Documents by Analysis of Change Data.*** Proceedings of the 5th ICSE, 1981

[36] W. Royce. ***Managing the Development of Large Software Systems.*** IEEE WESCON, August 1970, pp 1-9.

[37] Mahil Carr, June Verner. ***Prototyping and Software Development Approaches***. City University of Hong Kong 83 Tat Chee Avenue Hong Kong, College of Information and Technology Drexel University 4131 Chestnut St Philadelphia PA 19104, 97/04

[38] Vicki L. Sauter. ***Prototyping.*** URL: http://www.umsl.edu/~sauter/analysis/prototyping/proto.html. University of Missouri, St. Louis, College of Business Administration, 8001 Natural Bridge Road, St. Louis , MO 63121-4499, USA, May 1999

[39] Sage, A.P., J.D. Palmer. ***Software Systems Engineering.*** John Wiley and Sons 1990.

[40] DeSoi J. F., Lively W. M., Sheppard S.V..***Graphical Specification of User Interfaces With Behavior Abstraction.*** Department of Computer Science, Texas A&M University, CHI 89 Proceedings May 1989

[41] Clif Kussmaul, Roger Jack. ***User Interface Prototyping: Tips Amd Techniques.*** Consortium for Computing Sciences in Colleges (JCSC 21, 6), June 2006

[42] Arthur W. Mansky. ***Improving the Design of User Interfaces Through Rapid Prototyping***. Vitro Corporation Silver Spring, MD 20906

[43] J. Reese, R. Twiddy, L. Buchanan, M. Tarka, and K. C. Leung. ***GUIDES: A Tool for Rapid Prototyping of User-Computer Interfaces***. Proceedings of ACM Computer science Conference March 1985

[44] Vicki L. Sauter. ***The Analysis and Prototyping of Effective Graphical User Interfaces.*** URL: http://www.umsl.edu/~sauter/analysis/prototyping/intro.html . University of Missouri, St. Louis, College of Business Administration, 8001 Natural Bridge Road, St. Louis , MO 63121-4499, USA, August 2000

[45] Maha Boughdadi, Robert Busser. ***An Industrial Application of an Integrated UML and SDL Modeling Technique***. Compsac, p. 54, Twenty-Third Annual International Computer Software and Applications Conference, 1999

[46] J. Marrenbach. ***Rapid Development and Evaluation of Interactive Systems.*** Proceedings of the 5th ERCIM Workshop User Interfaces for All, Volume Report 74, pp. 81-86, Dagstuhl 1999

[47] J-P. Babau and A. Alkhodre. ***A development method for PROtotyping embedded SystEms by using UML and SDL (PROSEUS).*** In workshop SIVOEES 2001 - ECOOP, Budapest, 2001.

[48] Sasu Tarkoma. ***Specification Languages and Their Use (Case: AsmL)***. Seminar on Generative Programming University of Helsinki, Department of Computer Science, Spring 2003

[49] L. F. Pires, M. van Sinderen, C. R. G. de Farias, J. P. A. Almeida. ***Use of Models and Modelling Techniques for Service Development.*** Proceedings of the 3rd IFIP International Conference on e-Commerce, e-Business and e-Government (I3E 2003) 22-25 September 2003, Guarujá, Brazil.

[50] J. Floch, R. Brk. ***Using SDL for modeling behavior composition.*** In: Proc. of the 11th Int. SDL Forum, Stuttgart, Germany, LNCS 2708, Springer (2003)

[51] Øystein Haugen, Birger Møller-Pedersen, Thomas Weigert. ***Structural Modeling With UML2.0. Classes, Interactions and State Machines.*** Ericsson, Motorola, Inc.

[52] Nikolai N. Mansurov, Robert L. Probert. ***Improving time-to-market using SDL tools and techniques.*** Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 35 , Issue 6 (May 2001)

[53] Ananda Amatya. ***Modelling Software Development Using UML.*** Department of Computer Science. University of Warwick. Coventry, CV4 7AL, UK.

[54] ISO 13407 - Draft: ***User centred design process for interactive systems.*** International Organisation for Standardisation, Genf, 1998.

[55] G. Guizzardi, L. Ferreira Pires, M. van Sinderen. ***On the role of Domain Ontologies in the Design of Domain-Specific Visual Languages.*** In: 2nd Workshop on Domain-Specific Visual Languages, ACM OOPSLA, 2002.

[56] J. Marrenbach. ***Modelling Complex Systems using Statecharts applied to the User Interface of an Advanced Flight Management System.*** In: Proc. of the IEEE International Conference on Intelligent Engineering Systems INES´98, Vienna, Austria, pp. 43-48.

[57] D. Harel. ***Statecharts: A Visual Formalism for Complex Systems.*** In: Science of Computer Programming, North Holland, Elsevier Science Publishers, Vol. 8 pp. 231–274.

[58] Yan Jin, Robert Esser, Jörn W. Janneck. ***Describing the syntax and semantics of UML Statecharts in a heterogeneous modelling environment.*** Proceedings DIAGRAMS 2002, 2002.

[59] Iftikhar Azim Niaz, Jiro Tanaka. ***Mapping UML Statecharts to JAVA Code.*** Institute of Information Sciences and Electronics University of Tsukuba. Tennoudai 1-1-1, Tsukuba, Ibaraki 305-8573 Japan

[60] The Object Management Group. ***OMG Unified Modeling Language Specification.*** URL: http://www.uml.org/

[61] Jonathan Kaye, David Castillo. ***Flash MX for Interactive Simulation (Chapter 7)***. Delmar Learning, Clifton Park, NY USA, 2003.

[62] The International Telecommunications Union-Telecommunications (ITU-T) (formerly CCITT) Recommendation Z.100. ***Specification and Description Language (SDL)*****.** Geneva, March 1992.

[63] The O.M.G. Group. ***Unified Modeling Language Specification, UML version 2.0*****.** URL: http://www.omg.org/technology/documents/formal/uml.htm

[64] International Engineering Consortium (IEC). ***Specification Description Language*****.** URL: http://www.iec.org/online/tutorials/sdl/index.html

[65] Rick Reed. ***SDL-2000 for New Millennium Systems*****.** Telektronikk 4.2000, p 20-35. URL: http://www.telenor.com/telektronikk/

[66] ITU-T Standardization Sector. URL:http://www.itu.int/publications/sector.aspx?sector=2

[67] Jun Gong, Peter Tarasewich. ***Guidelines For Handheld Mobile Device Interface Design.*** College of Computer and Information Science, Northeastern University 360 Huntington Ave, 161CN, Boston, MA 02115 USA.

[68] N Kohtake, K Matsumiya, K Takashio, H Tokuda. ***Smart Device Collaboration for Ubiquitous Computing Environment.*** Workshop of Multi-Device Interface for Ubiquitous Peripheral, Keio University Japan 2003.

[69] Celine Pering. ***Interaction design prototyping of communicator devices: towards meeting the hardware-software challenge.*** Interactions, v.9 n.6, p.36-46 (portal.acm.org), November & December 2002.

[70] Loren Terveen, Elena Papavero, Mark Tuomenoksa. ***DynaDesigner: A Tool for Rapid Design and Deployment of Device-Independent Interactive Services.*** AT&T Bell Laboratories, CHI '95 Mosaic of Creativity. May 7-11 1995.

[71] Flash Developer Center. URL: http://www.adobe.com/devnet/flash/

[72] ActionScript and Object-Oriented Programming. URL: http://www.adobe.com/devnet/flash/actionscript.html

[73] Flash ActionScript 2.0 . URL: http://www.adobe.com/devnet/flash/articles/actionscript_guide_07.html

[74] Microsoft Windows Metafiles (WMF). URL: http://msdn2.microsoft.com/en-us/library/ms536391.aspx

[75] Microsoft Visual Studio .NET. URL: http://msdn2.microsoft.com/en-us/library/fx6bk1f4(vs.71).aspx

[76] ActiveX Controls. URL:http://msdn.microsoft.com/library/default.asp?url=/workshop/components/activex/intro.asp

[77] How to: Creating a Wizard With Visual Studio.
URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxconcreatingwizard.asp

[78] Siemens Gigaset Phones.
URL: http://gigaset.siemens.com/shc/1,1935,hq_en_0_11729_rArNrNrNrN,00.html

[79] Working with Movie Clips.
URL:http://livedocs.macromedia.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00001399.html

[80] Using the Stage.
URL:http://livedocs.macromedia.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00000033.html

[81] ActionScript 2.0 Language Reference. URL:
http://livedocs.macromedia.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=Part4_ASLR2.html

[82] Creating a Component Movie Clip. URL:
http://livedocs.macromedia.com/flash/8/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00003024.html

[83] Designs Patterns: Model-View-Controller (MVC). URL:
http://java.sun.com/blueprints/patterns/MVC.html

[84] Flash 8, Online Documentation. URL:
http://livedocs.macromedia.com/flash/8/main/wwhelp/wwhimpl/js/html/wwhelp.htm?href=00001386.html

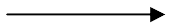
[85] Creating Flash Components. URL:
http://www.adobe.com/support/flash/applications/creating_comps/

[86] Jonathan Kaye, David Castillo. ***The Flash Simualtion Website*.** URL:
http://www.flashsim.com/

[87] BenQ-Siemens Mobile Phones Portal. ***Siemens SX1 Device Model.*** URL:
http://www.benq-siemens.com/cds/frontdoor/0,2241,hq_en_0_130289_0_xcs%253A130989_xcp%253A132668,00.html

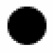
Appendix A: Guide to notation of UML State charts

Transition



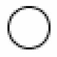
A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain specified actions when a specified event occurs, if specified conditions are satisfied. On such a change of state the transition is said to "fired". The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are available within actions specified on the transition or within actions initiated in the sub-sequent state. Events are processed one at a time. If an event does not trigger any transition, it is simply ignored. If it triggers more than one transition within the same sequential region (i.e., not in different concurrent regions), only one will fire; the choice may be nondeterministic if a firing priority is not specified.

Initial Pseudo State



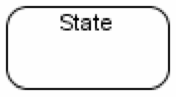
The initial element is used by State Machine diagrams. The initial element is a pseudo-state used to denote the default state of a composite state; there can be one initial vertex in each region of the composite state.

Entry Point



Entry points are used to define the beginning of a state machine. An entry point exists for each region, directing the initial concurrent state configuration.

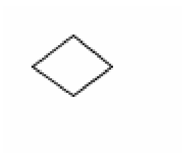
State



A state represents a situation where some invariant condition holds; this condition can be static, i.e. waiting for an event, or dynamic, i.e. performing a set of activities. State modeling is usually related to classes, and describes the allowable states a class or element may be in and the transitions that allow the element to move there. There are three types of states: simple states, composite states and submachine states.

Furthermore, there are pseudo-states, resembling some aspect of a state, but with a pre-defined implication. Pseudo-states are used to model complex transitional paths, and classify common state machine behavior.

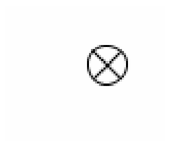
Choice



A state diagram expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides shorthand for showing decisions.

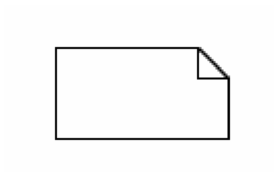
The choice pseudo-state is used to compose complex transitional paths, where the outgoing transition path is decided by dynamic, run-time conditions. The run-time conditions are determined by the actions performed by the state machine on the path leading to the choice.

Exit Point

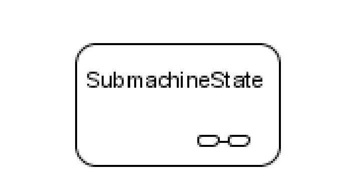


Exit points are used in submachine states and state machines to denote the point where the machine will be exited and the transition sourcing this exit point, for sub-machines, will be triggered. Exit points are a type of pseudo-state used in the state machine diagram.

Note

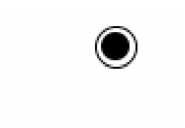


Substate



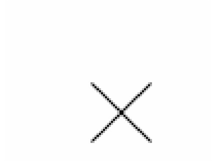
You can add a sub-machine element to a State Machine diagram. A sub-machine element is a pointer to a child State Machine diagram.

Final State



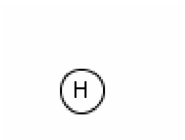
There are two nodes used to define a final state in an activity, both defined in UML 2.0 as of type "final node". The final element, shown above, indicates the completion of an activity - upon reaching the final, all execution in the activity diagram is aborted. The other type of final node, flow final, depicts an exit from the system but has no effect on other executing flows in the activity.

Terminate



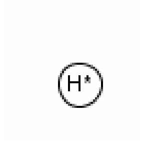
The terminate pseudostate indicates that upon entry of its pseudostate, the state machine's execution will end.

History



There are two types of history pseudo-states defined in UML, shallow and deep history.
A shallow history sub-state is used to represent the most recently active sub-state of a composite state; this pseudo-state does not recurse into this sub-state's active configuration, should one exist.

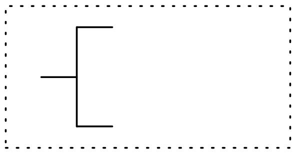
Deep History



A deep history sub-state, in contrast, reflects the most recent active configuration of the composite state. This includes active sub-states of all regions, and recurses into those sub-states' active sub-states, should they exist. At most one deep history and one shallow history can dwell within a composite state.

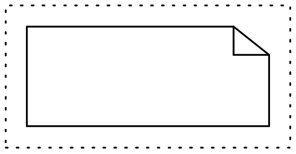
Appendix B: Guide to notation of SDL 92 State charts Supported by SICAT

Comment Symbol

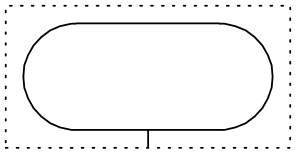


The Comment symbol can be used to insert a comment into the diagram. One or more comment symbols can be placed at to the right of any SDL symbol.

Text-Symbol

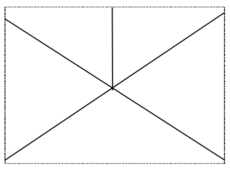


Process Start Symbol

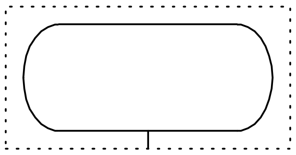


The Process Start symbol defines the starting point of a process whereby each process is assigned exactly one start symbol.

Process Stop Symbol

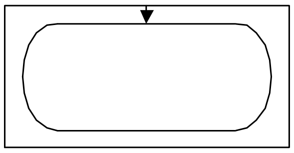


State Symbol



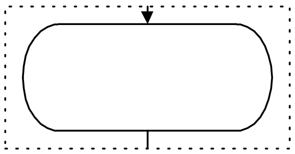
The State symbol is used to specify a state.

Next State Symbol



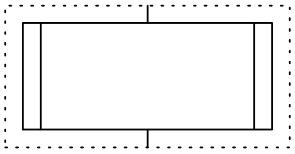
The Next State symbol is used to specify the end of a transition and the transition into another state.

State Continue Symbol



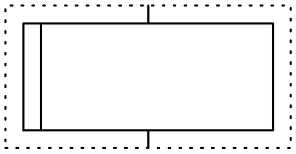
The State Continue symbol is a combination of the Next State symbol and a State symbol.

Procedure Call Symbol



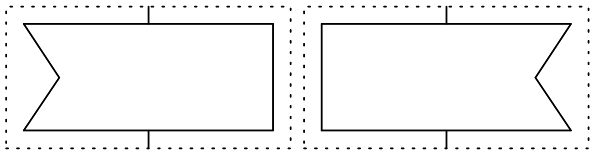
The Procedure Call symbol specifies a procedure call.

Macro Call Symbol



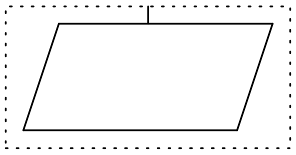
The Macro Call symbol specifies a macro call.

Input Right and Input Left Symbol



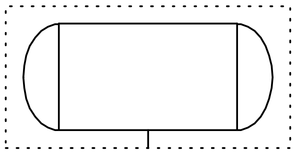
The two Input symbols are used for specifying the start of a transition: the transition is executed if the event identified in the Input symbol occurs.

Save Symbol



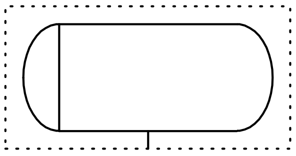
The Save symbol is used for buffering events in a Save queue.

Procedure Start Symbol



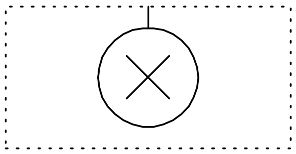
The Procedure Start symbol specifies the start of a procedure definition.

Macro Inlet Symbol



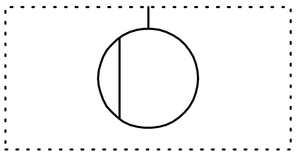
A Macro can be handled as a standard macro or a procedure macro. A standard macro can be switched to a procedure macro with the keyword "@PROCEDURE" in the informal text level of the Macro Inlet symbol.

Procedure Return Symbol



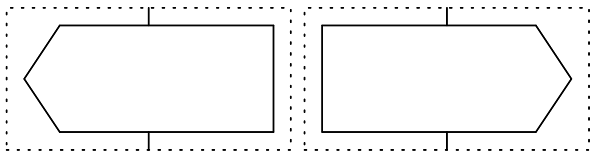
The Procedure Return symbol specifies the end of a procedure definition.
If the procedure has to supply a function value, the Procedure Return symbol has to contain the statement "return <function value>;" as last statement in the code text level.

Macro Outlet Symbol



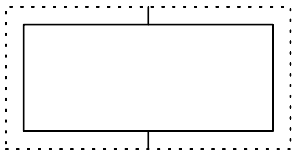
The Macro Outlet symbol specifies the end of a macro definition.
If the Macro is of Type "Procedure Macro" (keyword in Macro Inlet symbol) the Macro Outlet symbol is handled like a Procedure Return symbol.

Output Right and Output Left Symbol



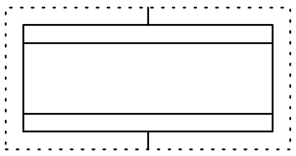
The Output symbol is used to specify the sending of a message to a process.

Task Symbol



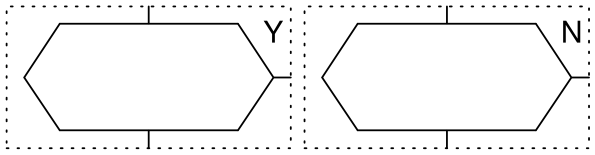
The Task symbol is used to specify actions. These actions must be specified as ActionScript statements in the code text level.

Process Create Symbol



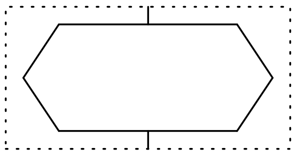
The Process Create symbol is used for starting a new process.

Decision Right and Decision Below Symbol



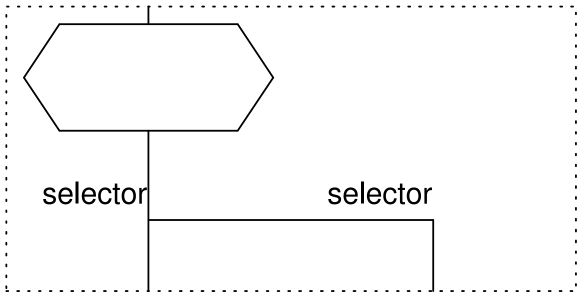
The decision symbol is used for specifying branches when some conditions are satisfied.

Selection Symbol



A case expression consists of a Selection symbol and several Branch symbols.

Example: Selection Structure:



Appendix C: The Companion CD-ROM

The CD-ROM accompanying this thesis includes the entire project of the device prototype. This project comprises as well all generated ActionScript classes and Flash movies and documents as the SDL models of the device prototype.

As far as prerequisite tools are affected, the reader must have a copy of Macromedia Flash Professional 8 to explore the content of the flash resources, generated classes and to run the device prototype, which is nothing else than a flash movie .

For this purpose, A free 30-day trial version of Macromedia Studio 8 can be downloaded from http://www.adobe.com/products/studio.