Statemate Course

W.-P. de Roever & R. v. Hanxleden
K. Baukus & J. Lukoschus

CAU Kiel

The STATEMATE Toolset — Lesson IV

The notion of a reactive system and the language Statecharts were introduced in the last session. We explained the rationale behind the design decisions of Statecharts in relation to the specific nature of reactive systems.
This time, the role of models in a system development life cycle is described. We introduce three languages to characterize reactive systems from different views. This results in a brief description of the STATEMATE toolset

[HP98] Modeling Reactive Systems with Statecharts: The STATEMATE Approach, D. Harel, M, Politi. McGraw-Hill, 1998.

Systems Life Cycle

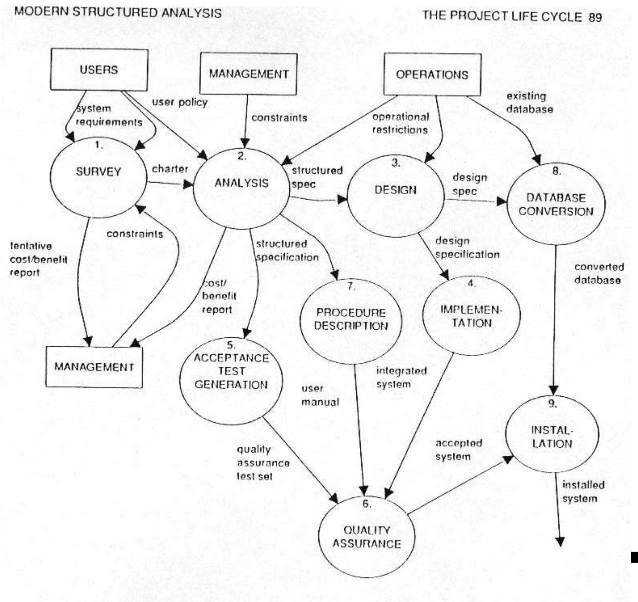
Specification in a systems life cycle

- Identify several phases in the development life cycle of a system
- **Classic waterfall model**: requirements analysis, specification, design, implementation, testing, and maintenance.
- Other approaches center around prototyping, incremental development, reusable software, or automated synthesis.

Requirements Analysis

- Most proposals contain a requirements analysis phase. Specification errors and misconceptions should be discovered in that early phase.
- Correcting errors in later stages is extremely expensive.
- Special languages are therefore used in the requirements analysis phase to specify a model of the system, and special techniques are used to analyze it extensively.

System's life cycle



System Model

- A good model is important for all participants in the system's development.
- Having a clear and executable model the functionality and behavior can be approved before investigating heavily in the implementation stages.
- The specification team uses modeling as the main medium for expressing ideas.

Methodology

A methodology provides guidelines for performing the processes that comprise the various phases. Concentrating on the modeling and analysis phase, a methodology consists of the following components:

- The methodology's underlying approach and the concepts it uses.
- The notation used, that is, the modeling languages with their syntax and semantics.
- The process prescribed by the methodology, that is, which activities have to be carried out to apply the methodology and in what order.
- The computerized tools that can be used to help in the process.

Reactive systems

The Statecharts language is especially effective for reactive systems.
A typical reactive system exhibits the following distinctive characteristics:

- It continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete.
- It must be able to respond to interrupts, i.e., high-priority events.
- Its operation and reaction often reflect stringent time requirements.
- It is very often based on interacting processes that operate in parallel.

Examples

On-line interactive systems: e.g., automatic teller machines, flight reservation systems

Computer-embedded systems: avionics, automotive, and telecommunication systems

Control systems: such as chemical and manufacturing systems.

The early warning system

A system model constitutes a tangible representation of the system's conceptual and physical properties and serves as a vehicle for the specifier and designer to capture their thoughts.

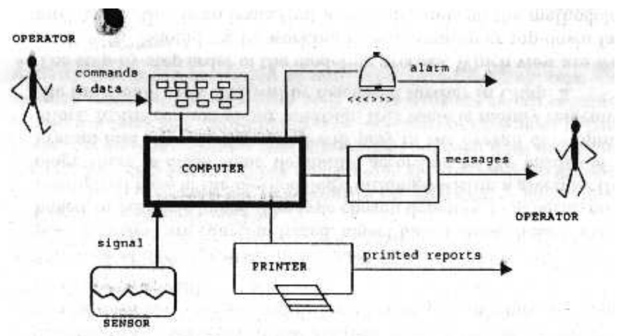


Figure 1.1 The early warning system (EWS).

Characteristics of models

Beside for communication, systems models should also be used for inspection and analysis.

When the model reflects some preexisting descriptions, such as requirements written in natural language, it is useful to keep track of how the components of the developing model are derived from the earlier descriptions.

STATEMATE

The modeling languages used in STATEMATE have been designed with several important properties in mind:

- to be intuitive and clear
- to be precise
- to be comprehensive
- to be fully executable

How to achieve these properties?

- To achieve clarity, elements of the model are represented graphically wher ever possible.
- For precision, all languages features have rigorous mathematical semantics
- Comprehension comes from the fact that the languages have the full expressive power needed to model all relevant issues, including the what, the when, and the how.
- For executability, the behavioral semantics is detailed and rigorous enough to enable the model to be executed (or be used to generate code).

Modeling Views

Building a model can be considered as a transition from ideas and informal descriptions to concrete descriptions that use concepts and predefined terminology.

Here, the descriptions used to capture the system specification are organized into three views: the **functional**, the **behavioral**, and the **structural**

Illustration

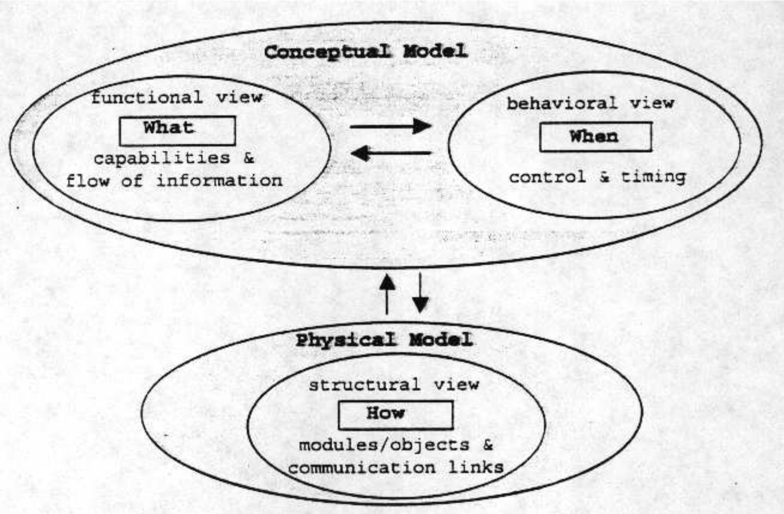


Figure 1.2 The three specification views.

The Three Views

Functional view : The functional view captures the "what". It describes the system's functions, processes, or objects, also called activities, thus pinning down its capabilities. This view includes the inputs and outputs of the activities.

Behavioral view : The behavioral view captures the "when". It describes the system's behavior over time, including the dynamics of activities, their control and timing behavior, the states and modes of the system, and the conditions and events that cause modes to change and other occurrences to take place.

Structural view : The structural view captures the "how". It describes the subsystems, modules, or objects constituting the real system and the communication between them.

Connection

While the two former views provide the conceptual model of the system, the structural view is considered to be its physical model.
The main connection between the conceptual and physical models is captured by specifying the modules of the structural view that are responsible for implementing the activities in the functional view.

Modeling Heuristics

Modeling heuristics are guidelines for how the notation should be used to model the system.

- The mapping between the methodology's concepts and the elements allowed in the notation.
- The type of decomposition to be used: e.g., function based, object based, mode based, module based, or scenario based.
- The step-by-step order of the modeling process: bottom-up or top-down

The Modeling Languages

The three views of a system model are described in our approach using three graphical languages.

- **Activity-charts** for the functional view,
- **Statecharts** for the behavioral view,
- and **Module-charts** for the structural view.
- Additional non-graphical information related to the views themselves and their inter-connections is provided in a **Data Dictionary**

Illustration

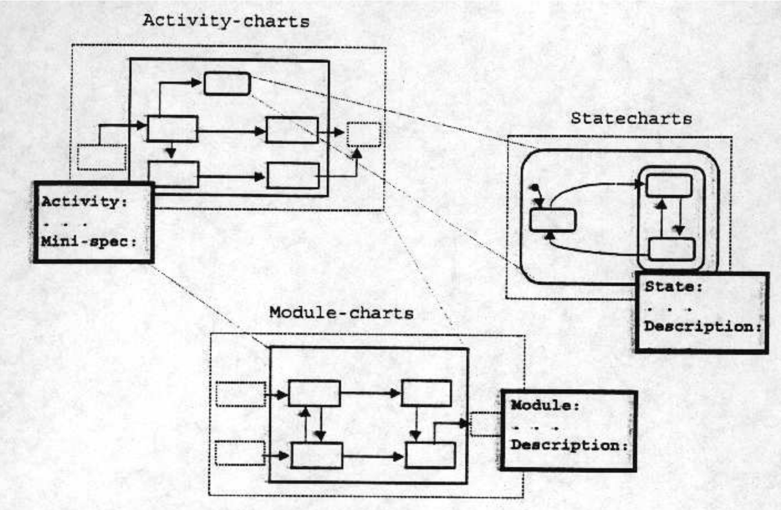


Figure 1.3 The modeling languages.

Activity-charts

Activity-charts can be viewed as multilevel data-flow diagrams. They capture functions, or activities, as well as data-stores, all organized into hierarchies and connected via the information that flows between them.

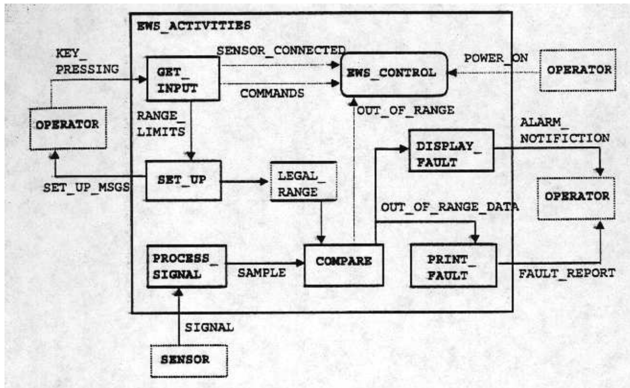


Figure 1.4 An activity-chart.

Non-graphical information

In addition to the graphical information, each element in the described has an entry in the Data Dictionary, which may contain non-graphical information about the element. For example, the activity entry contains fields called mini-spec and long description, in which it is possible to provide formal and informal textual descriptions of the activities workings.

Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES

Termination Type: Reactive Controlled
Mini-spec: st/TICK;;
TICK/ $SIGNAL_VALUE:=SIGNAL;
SAMPLE:=COMPUTE($SIGNAL_VALUE);
sc!(TICK,SAMPLE_INTERVAL)

Figure 1.5 An activit...

Statecharts

Statecharts constitute an extensive generalization of state-transition diagrams. They allow for multilevel states decomposed in an and/or fashion, and thus support economical specification of concurrency and encapsulation. They incorporate a broadcast communication mechanism, timeout and delay operators for specifying synchronization and timing information, and a means for specifying transitions that depend on the history of the system's behavior.

EWS-Statecharts

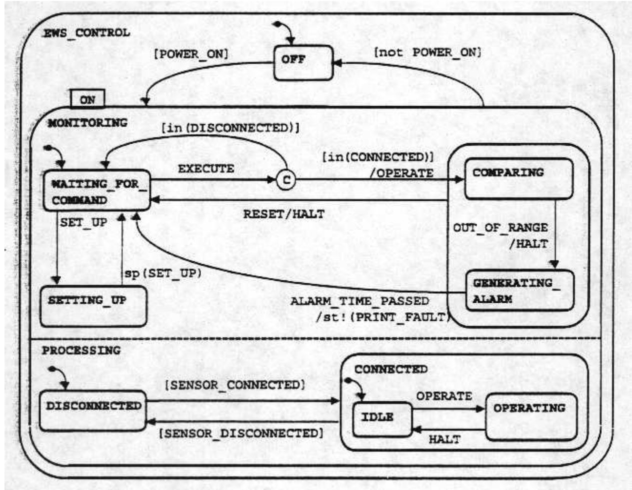
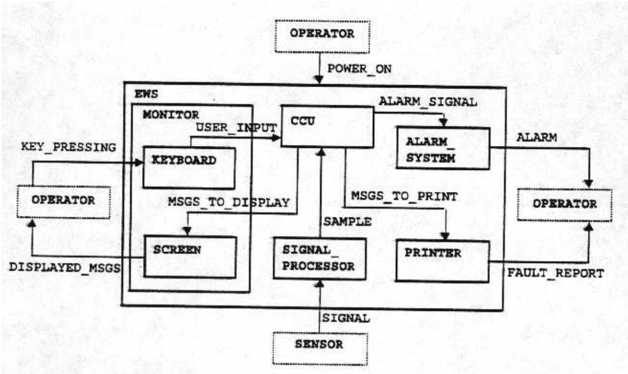


Figure 1.6 A statechart.

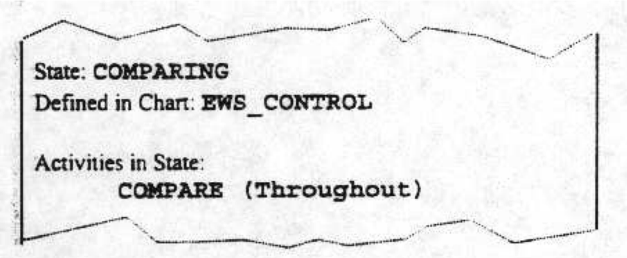
Module-charts

A module-chart can also be regarded as a certain kind of data-flow diagram or block diagram. Module-charts are used to describe the modules that constitute the implementation of the system, its division into hardware and software blocks and their inner components, and the communication between them.



Relationship between the languages

The relationship between the concepts of the three views are reflected in corresponding connections between the three modeling languages.
Most of these connections are provided in the Data Dictionary, and they tie the pieces together, thus yielding a complete model of the system under development.


State: COMPARING
Defined in Chart: EWS_CONTROL

Activities in State:
COMPARE (Throughout)

Figure 1.8 Specifying an activity throughout a state.

Handling large-scale systems

- The languages allow to split large hierarchical charts into separate ones:

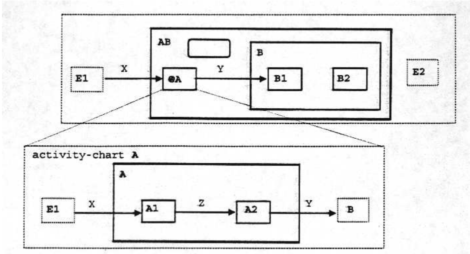


Figure 1.10 Splitting up charts.

- Also, coping with visibility and information hiding by setting scoping rules of elements in the model.
- Moreover, **generic charts** and **user-defined types**.

The STATEMATE toolset

STATEMATE has been constructed to “understand” the model and its dynamics. The user can then execute the specification by emulating the environment of the system under development and letting the model make dynamic progress in response.

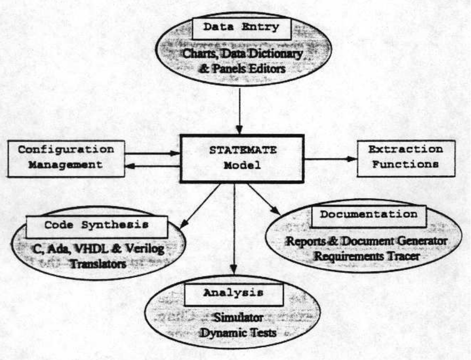


Figure 1.11 The STATEMATE toolset.

Configuration Management

Configuration managment

Basic idea of configuration management (CM) in the software development process is to guarantee the integrity (completeness and intactness) of the software product at any moment of the development

CM (cont’d)

Aims of CM:

- structure and discipline in the development process
- reusability of software

result of CM is

- better software quality and
- increased efficiency of the software development

CM is a management task that includes

- people responsible for it
- CM strategy/methods, plans
- support tools

Aspects of configuration management

- Identification: The imposed structure of the product provides access to parts of the product.
- Control: Product changes are authorized by a formal procedure that distinguishes different releases of the same product and its parts. Consistent releases constitute a baseline.
- Documentation: of status (releases, baselines).
- Verification: guarantees completeness and consistency
- Construction: of the product from its constituents
- process management: support the software life cycle.
- teamwork: several teams/developers of one product

CM informally

programming-in-the-many: $\Rightarrow$ potential for confusion. CM is meant to decrease the confusion. CM must

- identify,
- organize, and
- control

changes by different developers. The aim of CM is to increase quality by avoiding errors.

Architecture of a CM system

- repository: to provide consistency, releases and baselines;
- workareas: parallel development/test and parallel (sic!) changes of the same parts of the software
- Makefiles: construction and dependency checks

Tasks of a CM system

Typical CM activities from the viewpoint of a developer (= user)

- check out current product parts
- build product from checked out parts (if available)
- check in modifications
- compare own version to one in the repository
- update to actual status
- change the structure of the product: add or remove parts.

SCL programs

Skeleton

```
PROGRAM  template;

CONSTANT
 INTEGER step_counter := 0;
VARIABLE
 INTEGER I := 45;
 GLOBAL FILE f1;

INIT
END INIT;

SET BREAKPOINT [STEP] DO
END BREAKPOINT;

BEGIN
END;
END.
```


Declaration

```
-- CONSTANT
--%% To declare constants remove the comment sign in the previous line
--%% and declare their types as in the example in the next several lines.
--%%    INTEGER step_counter := 0;

-- VARIABLE
--%% In this section you declare variables
--%% and their types as in the example in the next several lines.
--%%    INTEGER I := 45;
--%%    GLOBAL FILE f1;
```


Init

```
-- INIT
--%% In this section you place SCL statements to be executed once,
--%% each time the SCP is invoked.
-- END INIT;


INIT
time_to_change:=0;
open(fout,'/tmp/klimatest.log', output);
write(fout,'Juhu','\n')
END INIT;
```

Breakpoints

```
-- SET BREAKPOINT [STEP] DO
--%% In this section you put statements to be executed conditionally,
--%% at each simulation step.  Duplicate this structure for each
--%% breakpoint definition.
--%% The condition STEP may be replaced by other condition/event expressions
-- END BREAKPOINT;

set breakpoint neue_temperatur=> every 1 do
  if (time_to_change <= cur_clock) then
    temp:=rand_iuniform(-20,40);
    write('Temperature changed to ',temp,' at ',cur_clock,'\n');
    time_to_change:=rand_iuniform(20,120)+cur_clock;
    write('Next change at ',time_to_change,'\n');
  end if;
end breakpoint;
```


Body

```
-- BEGIN
--%% This is the Main Section, in which you place any SCL statements,
--%% including Go statements.
--%% These statements are executed sequentially.
-- END;

BEGIN
write('Simulation started at: ',cur_clock,'\n');
while true loop
  go step;
end loop;
```