Introduction to SDL
(Specification and Description Language)

Julien Delange <julien.delange@esa.int>

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.
About this course

• Introduction, not a complete lecture
  • Cover most SDL concepts
  • For interested students, resources available on the internet
  • See the links section

• Focused on main SDL aspects and practical use
  • Basic components
  • Code generation, SDL specs usage

• Rely on RTDS when modeling software is used
  • Particularly for code generation
  • Thanks to Pragmadev for providing RTDS
Overview

• Introduction, history and rationale

• SDL language

• MSC diagrams

• Code generation

• Conclusion & links
Overview

- Introduction, history and rationale
- SDL language
- MSC diagrams
- Code generation
- Conclusion & links
Introduction and history

• International standard
  • First version late 80, refinements until late 2000

• Primary use for telecommunication purposes
  • Messages exchange
  • Timing concerns

• Deterministic specification of system behavior
  • System states and messages
  • Timing issues
Rationale for real-time systems

- System design = error prone
  - Need for system behavior specification

- Appropriate representation of system requirements
  - Avoid description of implementation concerns
  - Specify critical aspects (time, messages, etc.)

- Dedicated formalism
  - For system verification
  - For automatic implementation
A bit about existing SDL toolset

- SDL Rational SDL suite
  - Based on TAU, previous reference tool

- RTDS (french product !)
  - SDL/SDL-RT/MSC edition
  - Code generation targeting C and many OS

- Cinderella
  - SDL/SDL-RT/MSC edition
  - Code generation for C/C++
SDL ecosystem

- SDL *regular*
  - First version late 80, refinements until late 2000

- SDL-RT (Real-Time)

- UML profile for SDL
  - Standardized by ITU in 2007
Overview

- Introduction, history and rationale
- SDL language
- MSC diagrams
- Code generation
- Conclusion & links
SDL specification

- Hierarchical components assembly
  - Root component: system
  - Behavior specification: process

- Data types description

- Signals (with parameters) specification

- Behavior description
  - States/transitions/conditions of processes
SDL components (1)

- **Text**
  - Data/variables/signals definition
  - Package dependencies

- **Packages**
  - Organize components
  - Use with the `USE` clause in text sections

- **System**
  - Contain processes/blocks
  - Defines signals
SDL components

- **Block**
  - Contain blocks or processes

- **Comment**
  - Can also be inserted in text section using C-style comment (`/* ... */`)

- **Process**
  - Identification with a PID
  - Describe system functions behavior

```plaintext
BLOCK myblock
<declarations>

This is a comment

MyProcess(1)
```
Process behavior specification (1)

- **State**
  - Separated with signals or timers reception
  - Process abortion

- **Signal reception**
  - Special `SENDER` variable (PID of the latest sender)

- **Signal emission**
  - Specify recipient via `TO` clause
Process behavior specification (2)

- Process data management
  - Text section

- Timers (use the `Time` predefined type)
  - Set timers
  - Reset timers
  - Timer events as signal reception

- Decision
  - Similar to a `if`
Process behavior specification (3)

- All messages symbol: *
- Previous state: -
First example

- Ping-pong system
  - Ping process sends one integer
  - Pong returns the same integer

- Basic SDL usage
  - One block, several processes
  - No instance, only regular processes
  - Predefined data types
SDL block: example

Sent and received signals between processes

Processes

Text
(signals descriptions)
SDL example: ping process

DCL
val integer, recv integer;

Initial state
val := 1;

Data initialisation
\(\text{SET} \ (\text{NOW} + 5, \ \text{mytimer})\)

Timer initialisation

State

Timer event

Send signal

Receive signal

Data/timer processing

Idle

Wait_Pong

Idle

mytimer

sping(val)

spong(recv)

val := val + 1;

\(\text{SET} \ (\text{NOW} + 5, \ \text{mytimer})\)
SDL example: pong process

**DCL**
```
val integer;
```

1. Receive signal (put value into `val` variable)
2. Process data (variables)
3. Send signal (send value of the `val` variable)
Process semantics, constraints

- FIFO order for signals queues
  - \((\text{Requested message} \neq \text{available message}) = \text{ERROR}\)
  - Use conditions and previous states to avoid errors

- Avoid instant reaction
  - Transition must wait for some inputs or timers
  - Similar to Esterel/LUSTRE approaches
Extend the ping process

Cyclic timer that sends values 0, 1 and 2
Predefined data types

- Boolean
- Integer & Natural
- Real
- Character
- Time (point in time)
- Duration (time interval)
- Charstring
- Octet
Constraints data types: SYNTYPE

SYNTYPE <typename> 
[type definition] 
ENDSYNTYPE;

• Constraints on existing types : SYNTYPE

SYNTYPE NumberOfProducts = Integer
  CONSTANTS 0..200
ENDSYNTYPE;
Introduce new data types (1)

- **Structure**
  
  ```
  NEWTYPE article STRUCT
  name CHARSTRING;
  id INTEGER;
  ENDNEWTYPE;
  ```

- **Arrays**
  
  ```
  NEWTYPE articles ARRAY (NumberOfArticles, article)
  ENDNEWTYPE;
  ```
Introduce new data types (2)

- **Enum**

  ```
  NEWTYPE <typename>
  [type definition]
  ENDNEWTYPE;
  ```

  ```
  NEWTYPE MachineState
  LITERALS On, Off, Error
  ENDNEWTYPE;
  ```

- **Choice**

  ```
  NEWTYPE menu
  CHOICE
    two-course Integer;
    three-course Integer;
  ENDNEWTYPE
  ```
Using structs, enum, choice and arrays

- **Access to struct/choice/enum members : operator !**

  ```
  DCL myvariable article, tmp integer;
  myvariable!name := "computer";
  tmp := myvariable!id;
  ```

- **Array values and assignment**

  ```
  DCL myvariable articles, art article;
  art!name := "computer";
  art!id := 1;
  articles (2) := art;
  ```

```NEWTYPE article STRUCT
  name CHARSTRING;
  id INTEGER;
ENDNEWTYPE;```

```NEWTYPE articles
  ARRAY (NumberOfArticles, article)
ENDNEWTYPE;```
Introduce new data types

- Structure

```plaintext
NEWTYPE article STRUCT
    name CHARSTRING;
    id INTEGER;
ENDNEWTYPE;
```

- Arrays

```plaintext
NEWTYPE articles ARRAY (NumberOfArticles, article)
ENDNEWTYPE;
```
Data operators

- **Assignment:** \( := \)
- **Equality:** \( = \)
- **Non-Equality:** \( /= \)
- **Superior, inferior:** \( > < \)
- **Bit operator:** and, or, not, xor
More on data types ...

- Other operations/data types
- Compatibility with ASN.1
- Redefinition of data operator for new types
Integration of third-party code

• Declaration

```plaintext
PROCEDURE <name>
([IN/OUT paramname1 paramtype1,
IN/OUT paramname2 paramtype2])
EXTERNAL
```

• Procedure use

  • Use regular SDL procedure

• Potential assumptions breaks

  • No simulation available
  • Validate system according to ext. code
Third-party code in ping system

DCL
val integer,
recv integer;
PROCEDURE myproc (IN p int) EXTERNAL;

Idle
mytimer
myproc(val)
sping(val)
Wait_Pong

Call myproc with the value (val) to be sent to pong
Integration of third-party, binary level

- External procedure mapped in implementation language
  
  PROCEDURE myproc (IN val int, OUT val2 int)
  
  To C: Void myproc (int val, int* val2);
  
  To Ada: PROCEDURE MYPROC (IN val   : INTEGER;
                             OUT val2 : INTEGER);

- No consistency check !!!
  - Semantics/types to be validated

- User-provided object-code
  - Linked with RTDS generated code
Overview

• Introduction, history and rationale

• SDL language

• Message Sequence Chart (MSC) diagrams

• Code generation

• Conclusion & links
Message Sequence Charts

- Standardized by ITU (1993)
  - Smooth integration in vendor tools

- Conceptually similar to UML sequence chart
  - Seem to fit better for large-scale systems

- Trace system activity and agents interactions
  - Appropriate timing information
  - System agents (process, semaphore, ...)
  - Sent/received signals between agents
Relation between MSC and SDL

- **SDL**: system behavior specification
  - Does not specify exchanged messages
  - Cannot trace messages sent across a distributed system

- **MSC**: one execution scenario
  - Show all information
  - Relevant to one particular configuration

- Complementary formalisms
  - MSC for behavior analysis
  - Support language for SDL specs.
MSC usage

- Debugging
  - Generate MSC diagrams from system execution
  - Show activity, messages
  - Trace and find errors

- Test
  - Send unexpected signals to assess system robustness
  - Test various system configuration

- Performance analysis
  - Useless states
  - Irrelevant signals
MSC: representation (1)

- Time from top to bottom
- Show agent activity
  - Stop condition specification
  - Partial specification (continuous lifetime)
- Agents can be spawned by others
MSC : representation (2)

- Signal between agents
- Timers activity
  - Set timer
- Timer events
MSC: example

- Time
- Agents
- Timer expiration
- States
- Signals
MSC : going further

- More operations on signals
  - Lost signals
  - Saved signals

- Add comments

- More on time operations
  - Specify time interval

- See the links section !
Agenda

- Introduction, history and rationale
- SDL language
- MSC diagrams
- Code generation
- Conclusion & links
Code generation: target platforms

• Support for different OS
  • Linux, Solaris, Windows
  • Support for some RTOS

• POSIX compliance
  • Use pthread routines
  • Ease port to non-supported OS

• Interface with gdb debugger
Code generation

- General purpose files (for all SDL agents)
  - `RTDS_gen.h`: resources identifiers (PID, signals ID, etc.)
  - `RTDS_Start.c`: process, semaphores, data initialization
  - `RTDS_String.c`: OS-agnostic manipulation of strings
  - `RTDS_Set.c`: OS-agnostic set handling

- Process specific files
  - `<processname>.c|h`
  - Operations performed by process
  - Initialization and infinite loop
Code generation: `RTDS_gen.h`

- **Process identifiers**
  ```
  #define RTDS_process_ping     1
  #define RTDS_process_pong     2
  #define RTDS_process_RTDS_Env 3
  ```

- **Signals/timers identifiers**
  ```
  #define sping   1
  #define spong   2
  #define mytimer 3
  ```

- **States identifiers**
  ```
  #define Wait_Pong 1
  #define Idle      2
  #define RTDS_Idle 3
  ```
Code generation: RTDS_Start.c

- **RTDS_Start()**
  - Initialize data and environment (see var RTDS_currentContext)
  - Create resources (cf. RTDS_STARTUP_PROCESS_CREATE())

- **RTDS_MAIN()**
  - Initialize network stack (if necessary)
  - Call to RTDS_Start()
Code generation: `<processname>`.c

- **RTDS_TASK_ENTRY_POINT** `<processname>()`
  - Single entry point
  - Called by `RTDS_Start()`

- Process initialization
  - Local data
  - Timers, etc.

- Infinite loop (`for ( ; ; )`)
  - Signal and environment infos with globvar `RTDS_currentContext`
  - Signal handling (cf. `RTDS_MSG_QUEUE_READ`, `RTDS_MSG_SEND_<signal_name>_TO_NAME`)
  - States management (cf. `RTDS(SDL)_STATE_SET`)
  - Timer handling (`RTDS_SET_TIMER`)
Code generation: going further

- Define your own code generation templates
  - Adaptation to other OS and/or requirements
  - See $RTDS_HOME/share/ccg directory

- Template organization
  - Must define generic functions (task creation, etc.)
  - Target-specific options (boot, configure environment, etc.)
  - Integration within RTDS environment (configuration with GUI, etc.)
Agenda

• Introduction, history and rationale
• SDL language
• MSC diagrams
• Code generation
• Conclusion & links
Conclusion

- Formal specification of application concerns
- Ease system validation and test
- Extended existing toolset
  - SDL editor
  - MSC generation & editor tools
- Require integration for implementation
Useful links

- **ITU website:** [http://www.itu.int](http://www.itu.int)

- **SDL Data types:** [www.lcc.uma.es/~pedro/docencia/sc/UsingDataTypes.pdf](http://www.lcc.uma.es/~pedro/docencia/sc/UsingDataTypes.pdf)

About this course

• Introduction, not a complete lecture
• Cover most SDL concepts
• For interested students, resources available on the internet
• See the links section
• Focused on main SDL aspects and practical use
• Basic components
• Code generation, SDL specs usage
• Rely on RTDS when modeling software is used
• Particularly for code generation
• Thanks to PragmaDev for providing RTDS
Overview
• Introduction, history and rationale
• SDL language
• MSC diagrams
• Code generation
• Conclusion & links
Introduction and history
- International standard
- First version late 80, refinements until late 2000
- Primary use for telecommunication purposes
- Message exchange
- Timing concerns
- Deterministic specification of system behavior
- System states and messages
- Timing issues
Rationale for real-time systems

- System design = error prone
- Need for system behavior specification
- Appropriate representation of system requirements
- Avoid description of implementation concerns
- Specify critical aspects (time, messages, etc.)
- Dedicated formalism
- For system verification
- For automatic implementation
A bit about existing SDL tools set

- SDL Rational SDL suite
- Based on TAU, previous reference tool
- RTDS (French product!)
- SDL/SDL-RT/MSCEditation
- Code generation targeting C and many OS
- Cinderella
- SDL/SDL-RT/MSCEditation
- Code generation for C/C++
Julien Delange <julien.delange@esa.int>

SDL ECOSYSTEM

• SDL
• SDL Regular
• First version late 80, refinements until late 2000
• SDL-RRT (Real-Time)
• UML profile for SDL
• Standardized by ITU in 2007
Overview

- Introduction, history and rationale
- SDL language
- MSC diagrams
- Code generation
- Conclusion & links
SDL specification
• Hierarchical components assembly
• Root component: system
• Behavior specification: process
• Data types description
• Signals (with parameters) specification
• Behavior description
• States/transition/conditions of processes
Julien Delange <julien.delange@esa.int>

SDL components (1)

• Text
• Data/variables/signals definition
• Package dependencies
• Packages
• Organize components
• Use with the USE clause in text sections
• System
• Contain processes/blocks
• Defines signals

DCLV

VAL integer;

SYSTEM my system <declarations>

PACKAGE my pkg <declarations>

SIGNAL ok, code (integer);
Julien Delange <julien.delange@esa.int>

SDL components

• Block
• Contain blocks or processes
• Comment
• Can also be inserted in text section using C-style comment (* ... *)
• Process
• Identification with a PID
• Describe system functions behaviour

BLOCK my block <declarations>
Process behavior specification (1)

- State
- Separated with signals or timer reception
- Process abortion
- Signal reception
- Special SEND variable (PID of the latest sender)
- Signal emission
- Specify recipient via TO clause
Process behavior specification

- Process data management
- Text section
- Timers (use the Time predefined type)
- Set timers
- Reset timers
- Timer events as signal reception
- Decision
  - Similar to if
Process behaviour specification (3)

• All messages symbol: *

• Previous state: -
First example

• Ping-pong system
• Ping process sends one integer
• Pong returns the same integer
• Basic SDL usage
• One block, several processes
• No instance, only regular processes
• Predefined data types
TEXT (signal descriptions)

Processes

Sent and received signals between processes
Julien Delange <julien.delange@esa.int>

SDL example: ping process

Process data

( variables )

Initial state

Data initialization

Timer initialization

State

Timer event

Send signal

State

Receive signal

Data/timer process
Process semantics, constraints

- FIFO order for signals queues
- (Requested message ≠ available message) = ERROR
- Use conditions and previous states to avoid errors
- Avoid instant reaction
- Transition must wait for some inputs or timers
- Similar to Esterel/LUSTRE approaches
Extend the ping process
decision
Cyclic timer that sends values 0, 1 and 2
Predefined data types

- Boolean
- Integer & Natural
- Real
- Character
- Time (point in time)
- Duration (time interval)
- Character String
- Octet
JULIEN DELANGE < julien.delange@esa.int>

CONSTRAINTS DATA TYPES: SYNTYPE

SYNTYPE <typename>
[typedef]
END SYNTYPE;

• CONSTRAINTS ON EXISTING TYPES: SYNTYPE

SYNTYPE NUMBER OF PRODUCTS = INTEGER
CONSTANTS 0..200
END SYNTYPE;
NEW TYPE <typename>
[typedef]
END NEW TYPE;

• Structure
NEW TYPE article STRUCT
    name STRING;
    id INTEGER;
END NEW TYPE;

• Arrays
NEW TYPE articles
    ARRAY (NumberOfArticles,
            article);
END NEW TYPE;

Array index type
Array value type
NEW TYPE (type name)

END NEW TYPE;

• ENUM

NEW TYPE Machine State

LITERALS On, Off, Error

END NEW TYPE;

• CHOICE

NEW TYPE menu

CHOICE two-course Integer;
three-course Integer;

END NEW TYPE;
Using structs, enum, choice and arrays

• Access to struct/choice/enum members: operator!

DCL my variable article, tmp integer;
my variable! name := "computer";
tmp := my variable! id;

• Array values and assignment

DCL my variable article s, article;
article! name := "computer";
article! id := 1;
article s (2) := article;

NEWTYPE article STRUCT
  name CHAR STRING;
  id INTEGER;
END NEWTYPE;

NEWTYPE article s
  ARRAY (NumberOfArticles, article);
END NEWTYPE;
NEW_TYPE <typename>
[typedef]
END_NEW_TYPE;

STRUCTURE
NEW_TYPE article STRUCT
name CHAR STRING;
id INTEGER;
END_NEW_TYPE;

ARRAYS
NEW_TYPE articles ARRAY (NumberOfArticles, article)
END_NEW_TYPE;
Data operators

• Assignment: 
  =

• Equality: 
  =

• Non-equality: 
  /=

• Superior, inferior: 
  > <

• Bit operators: 
  and, or, not, xor
More on data types...

- Other operations / data types
- Compatibility with ASN.1
- Redefinition of data operator for new types
Integration of third-party code

Declaration

PROCEDURE <name> (IN/OUT param name1 param type1, IN/OUT param name2 param type2)

EXTERNAL

Procedure use

Use regular SDL procedure

Potential assumptions breaks

No simulation available

Validate system according to ext. code
Third-party code in ping system

call my proc with the value (val) to be sent to pong
Integrazione di terze parti, livello binario

• Esterno procedure mappate in implementation language

PROCEDURE myproc (IN val INT, OUT val2 INT)
To C:
VOID myproc (int val, int *val2);
To Ada:
PROCEDURE MYPROC (IN val INTEGER;
OUT val2 INTEGER);

• No consistence check ! ! !

• Semantics/types to be validated

• User-provided object code

• Linked with RTDS generated code
Overview

• Introduction, history and rationale
• SDL language
• Message Sequence Chart (MSC) diagrams
• Code generation
• Conclusion & links
Message Sequence Charts

- Standardized by ITU (1993)
- Smooth integration in vendor tools
- Conceptually similar to UML sequence chart
- Seem to fit better for large-scale systems
- Trace system activity and agent interactions
- Appropriate timing information
- System agents (process, semaphore, ...)
- Sent/received signals between agents
Relation between MSC and SDL

- SDL: system behavior specification
- Does not specify exchanged messages
- Cannot trace messages sent across a distributed system

- MSC: one execution scenario
- Shows all information
- Relevant to one particular configuration
- Complementary formalisms
  - MSC for behavior analysis
  - Supports language for SDL specs.
Julien Delange <julien.delange@esa.int>

MSC usage
• Debugging
• Generate MSC diagrams from system execution
• Show activity, messages
• Trace and find errors
• Test
• Send unexpected signals to assess system robustness
• Test various system configuration
• Performance analysis
• Useless states
• Irrelevant signals
JULIEN DELANGE <julien.delange@esa.int>

MSC : representation (2)

- Signal between agents
- Timer activity
- Set timer
- Timer events
J uli en  D e l a n g e  < j u l i e n  d o t  d e l a n g e  a t  e s a  d o t  i n t >

M S C  :  g o i n g  f u r t h e r

• More operations on signals
• Lost signals
• Saved signals
• Add comments
• More on time operations
• Specify time interval
• See the links section!
Agenda

• Introduction, history and rationale
• SDL language
• MSC diagrams
• Code generation
• Conclusion & links
Julien Delange <julien.delange@esa.int>

Code generation: target platforms

- Support for different OS
  - Linux, Solaris, Windows
  - Support for some RTOS
  - POSIX compliance
  - Use pthread routines
  - Ease port to non-supported OS
  - Interface with gdb debugger
Code generation

• General purpose files (for all SDL agents)
  • RTDS_gen.h: resource identifiers (PID, signal IDs, etc.)
  • RTDS_Start.c: process, semaphores, data initialization
  • RTDS_String.c: OS-agnostic manipulation of strings
  • RTDS_Set.c: OS-agnostic set handling
• Process specific files
  • <process name>.[c|h]
  • Operations performed by process
  • Initialization and infinite loop
Code generation:

- RTDS_gen.h

  - Process identifiers
    - #define RTDS_process_ping
    - #define RTDS_process_pong
    - #define RTDS_process_RTDS_ENV

  - Signal/timer identifiers
    - #define spin
    - #define spong
    - #define my_timer

  - States identifiers
    - #define Wait_Pong
    - #define Idle
    - #define RTDS_Idle

Code generation:

- RTDS_Start

  - Initialize data and environment (see var RTDS_currentContext)
  - Create resources (cf. RTDS_STARTUP_PROCESS_CREATE)

- RTDS_MAIN

  - Initialize network stack (if necessary)
  - Call to RTDS_Start
RTDS_TASK_ENTRY_POINT (process_name)
• Single entry point
• Called by RTDS_Start()
• Process initialization
• Local data
• Timers, etc.
• Infinite loop (for ; ; )
• Signal and environment info with globvar RTDS_currentContext)
• Signal handling (cf. RTDS_MSG_QUEUE_READ, RTDS_MSG_SEND <signal_name>_TO_NAME)
• States management (cf. RTDS_SDL_STATE_SET)
• Timer handling (RTDS_SET_TIMER)

Process Init
Receive signal
Action depending on received signal
Send signal
Infinite loop
• Define your own code generation templates
• Adaptation to other OS and/or requirements
• See $RTD_HOME/share/ccg directory
• Template organization
• Must define generic functions (task creation, etc.)
• Target-specific options (boot, configure environment, etc.)
• Integration within RTDS environment (configuration with GUI, etc.)
Agenda

• Introduction, history and rationale
• SDL language
• MSC diagrams
• Code generation
• Conclusion & links
• Formal specification of application concerns
• Ease system validation and test
• Extended existing toolset
• SDL editor
• MSC generation & editor toolset
• Require integration for implementation