

# Possible Tailoring of the UML for Systems Engineering Purposes

Ingmar Ogren M SC

iog@toolforsystems.com

Tofs corporation, Fridhem 2, SE 76040 Sweden

Phone: +46 176 54580, Fax +46 176 54441

© John Wiley and sons publishers

(Published in the Journal of Systems Engineering)

## 1 Abstract

The Systems Engineering discipline needs a common Systems Engineering Modeling Language (SEML). One way to create a SEML is to start with the Unified Modeling Language (UML), created by Rational Inc. and originally intended for software. Some requirements on a SEML are analyzed together with the UML. The result is that a subset of the UML, primarily the Component diagram, can be combined with a pseudo code subset of the programming language Ada 95 to satisfy the requirements stated. Specially requirements concerning management of system-level missions and abilities combined with requirements on formality and understandability causes concern with the unmodified UML, which is why the UML needs to be both reduced and extended. The Component diagram supports architectural descriptions with system components ordered, according to dependency, with the system's missions as top components. The result is a SEML, compatible with the UML, readily understandable for analyzers and end users, and sufficiently formal for automated consistency checks.

## 2 Introduction

Every other major engineering area (building, electricity, mechanics, electronics, hydraulics, etc.) has an international common language as a basis for mutual understanding among practitioners. For the time being Systems Engineering is different with various languages used to describe systems. This situation is not very satisfactory since it hampers progress and understanding among practitioners and system users. Until recently the Software Engineering community had a similar situation. Here the problem has now, to some extent, been solved by the advent of the Unified Modeling Language (UML) [5, 6, 7]. For the Systems Engineering area, there are several alternative ways to establish a common modeling language, for example:

1. Use an existing notation, such as for example the IDEF0 notation [1] and tailor it to the needs of the Systems Engineering community
2. Accept the existing UML, not only for software, but also for systems engineering
3. Tailor the UML to meet the requirements for Systems Engineering
4. Invent a new modeling notation for Systems Engineering.

To select an alternative, you need an understanding of what the requirements on an efficient Systems Engineering Modeling Language (SEML) might be. A set of possible requirements is:

1. The SEML shall include modeling of system components of categories Operator, Software and Hardware
2. The SEML shall include modeling of a system's missions and abilities
3. The SEML shall support modeling of a system's structure
4. The SEML shall support modeling of a system's behavior
5. The SEML shall be simple enough for practitioners to learn in a short time and its notation should not be too difficult to explain to end-users.
6. The SEML shall include sufficient formalism to allow automatic analysis.

The alternatives and the requirements are listed and commented in table 1. The table indicates that the two existing alternatives both have problems to meet the requirements listed. To define and market a

completely new modeling notation would be a chancy enterprise, why the continued discussion concentrates on the alternative “Modified UML”.

	IDEF0	UML unmodified	UML modified	New notation
1. Model components	Yes	To some extent	Possibly	Possibly
2. Model missions and abilities	Yes	No	Possibly	Possibly
3. Model system structure	Yes	Yes	Possibly	Possibly
4. Model system behavior	Requires some additional notation to define activity content	Yes, multiple ways	Possibly	Possibly
5. Simplicity	Difficult to distinguish controls from inputs	No, to many diagrams	Possibly	Possibly
6. Formality	Yes for structure but requires a formal notation to define activity content	Requires some extensions as defined by UML tools.	Possibly	Possibly

**Table 1 Alternatives and requirements for an SEML**

### 3 The Unified Modeling Language (UML)

The UML was created by the “Three amigos” at Rational Software. Each of the three has an extensive background of Software Engineering and to some extent also Systems Engineering:

- Grady Booch [2] with a background from US Air Force and with long experience as “method guru” at Rational.
- Ivar Jacobsson [8], inventor of the “Use cases” has taught software engineers to start their work by considering how the software is intended to be used. He has an Ericsson background.
- James Rumbaugh [4], who took part in development of the popular Object Modeling Technique (OMT), while being employed by General Electric.

The UML was defined by the “three amigos” through taking all of their favorite notations and including them in the new modeling language (with some exceptions). The result is a rich language, which has something for most software engineers. This is why the UML has rapidly become popular in the software world. The richness is also a problem since some concepts can be described in alternative ways within the language. Consequently most practitioners prefer to use a subset of the UML, rather than the full language. To understand the different diagrams of the UML, you can categorize the diagrams after what they support. Below the diagrams are listed as supporting requirements, structural, communication or behavioral work. For details of the diagrams, see references [5, 6, 7]

#### The Use Case diagram

The Use case diagram shows the interaction between a user (human or other) and a software system. This is basically a dependency relation where the user depends on the system.

The Use Case diagram primarily supports requirements work.

#### The Class diagram

The Class diagram is basically an enhanced entity-relationship diagram where the entities are classes with name, attributes and actions. A “software class” can then be seen as a “template”

from which instances are created to implement the software. The diagram includes several kinds of relations such as generalization, association, aggregation. The diagram can be used for conceptual modeling, for specification and for documenting a software solution. The Class diagram primarily supports structural work.

#### The Sequence diagram

Message sequence charts have a long tradition for visualizing the interaction between concurrent processes communicating by way of messages. These charts are included in the UML as Sequence diagrams.

The Sequence diagram supports communication.

#### The Collaboration diagram

The Collaboration diagram resembles Structured Analysis diagrams with its “boxes and arrows”. It basically contains the same information as the Sequence diagram why it also supports communication.

#### The Component diagram

The Component diagram shows dependencies between components and can also be drawn to show aggregation (components included in other components). It supports modeling of system structures, showing how the different components in a system depend on each other.

#### The Package diagram

The Package diagram shows dependencies between major system components. Consequently it is an alternative for modeling of system structures.

#### The State diagram

The State diagram is well proven to visualize behavior. It is used to model behavior within a system component.

#### The Activity diagram

The Activity diagram is an enhancement of the well-proven flow chart. Like the State diagram it can be used to visualize behavior.

#### The Deployment diagram

The Deployment diagram shows how different software objects are distributed on hardware nodes. Consequently it is used to supplement structural descriptions.

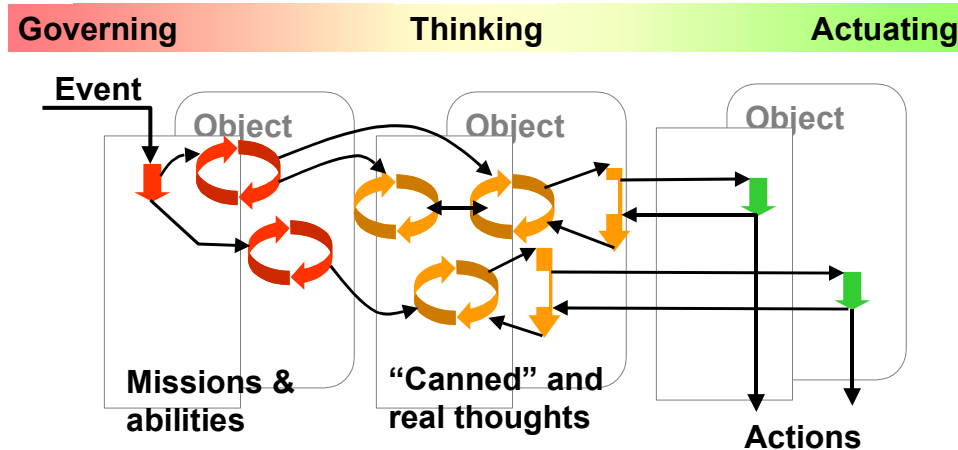
The listing above shows some of the richness of the UML. It is obvious that definition of a simple SEML from the UML requires that you limit the UML to a subset of the language, possibly with some extensions, wherever the UML does not completely meet the requirements for a SEML. To find the right subset you must first return to the basics of Systems Engineering.

## 4 Back to basics

The first concern in system modeling is to find a notation to model the system’s architectural structure. You must then decide on how to relate the different structural elements to each other. Some alternatives are:

- Aggregation with the main relation being “contained in”. Used in “Structured Analysis” [11]
- Inheritance with the main relation being “is derived from”. Supported by the UML Class diagram.
- Communication, with the main relations being “sends to”, “receives from” and “invokes”. Used in “Structured Analysis” and SADT with the IDEF0 notation [1]
- Dependency with the main relation being “depends on”. Supported by the UML Component and Class diagrams with the Component diagram being the simplest since it contains only system components (objects) and their dependencies.

Since you need to model, not only system components, but also abstractions such as Missions and Abilities and since you want to keep the modeling notation simple, the obvious choice, within the UML, is the Component diagram.



**Figure 1 Objects and actions in a system**

Figure 1 shows a system outlined as a UML component diagram with components (objects) on different levels and with actions indicated in these objects. From the figure can be understood:

- At the top level, a system can complete one or more missions through use of the system's abilities.
- External events may trigger the system to take action, using its abilities within its mission space.
- An essential part of any non-trivial system's actions is composed from human thought either as the "real" thoughts of the system's operators or as the "canned thoughts" of software developers, expressed as computer software.
- Actions may be continuous or "single shot".
- Actions may invoke other actions.
- Actions may communicate with other, concurrently active, actions.
- A system will normally deliver actions, which interact with the external world, physically or in some other way.

Note in Figure 1 that the UML Component diagram has been "extended" beyond its stated use to show "how software components depend on each other" in two ways:

- "Upwards" towards the system's operators, and ultimately missions/abilities to include the Operator roles ("real thoughts") and the Missions as objects with Abilities being actions in the Mission objects. This extension makes the separate UML Use Case diagrams unnecessary since the Use cases will be represented as interfaces between Operator objects and objects supporting them.
- "Downwards" towards the system's hardware to include Hardware objects. This extension makes the UML Deployment diagrams unnecessary and also allows for inclusion of non-computer hardware in the system model.

As discussed above it is possible to meet the requirements for structural (architectural) modeling, with inclusion of Missions and Abilities through extension of the UML Component diagram. We then still have to meet the requirements on behavioral modeling and formalism. UML here offers the State and Activity

diagrams, both being well proven. However it is doubtful if any of these meet the requirements on formalism and understandability, since they do not include typing to support variables and since non-experts do not immediately understand them. For those, who can read and write the UML behavioral diagrams, they can contribute significantly towards understanding behavior.

An alternative is to start from natural language and find a supplementing standard. Ada 95 [12] is special among programming languages, since it is “talkative” and close to natural language as compared to most other programming languages. Consequently a pseudo code subset of Ada 95 can meet the requirements on formalism and understandability and qualify as “Formalized English”. Such a simplified subset of Ada 95 is the Odel design language [9]. Besides simplifications, Odel includes extensions to manage real concurrency as required for Systems Engineering purposes.

Concurrency also needs to be clarified in system models with visualization of the messages between concurrent processes. This is traditionally done with message sequence charts, which are included in the UML as Sequence diagrams.

The conclusion of this discussion is that a combination of UML Components diagrams, UML Sequence diagrams and “Formalized English” (Odel), derived from the Ada 95 programming language, could be used to create a useful notation for systems modeling. To show how this can be done, a couple of system examples are presented, selected from projects where the technique has been applied. One is a simple technical system, which controls the windows of an automobile and one is a more abstract example from the Defense domain, which concerns establishment of “Dominant Battlefield Awareness”.

## 5 Examples of system models

### 5.1 Car window control

#### 5.1.1 Example overview

The example originates from a graduation work by Peter Junermark at Chalmers University of Technology in Goteborg. It concerns a system for controlling the windows of an automobile with some child-safety features included. A overview text of the example is:

*The window control system has the mission to allow the driver to control all windows and each passenger to control windows within reach, while ensuring child safety through not allowing rear windows to open more than 2" when "childproof button" is pressed.*

*The car is equipped with a CAN bus (CAN = Controller Area Network) and a number of programmable ECUs (ECU = Electronic Control Unit), connected to the CAN bus. One ECU is installed in the instrument board and one in each door as shown in figure 2.*

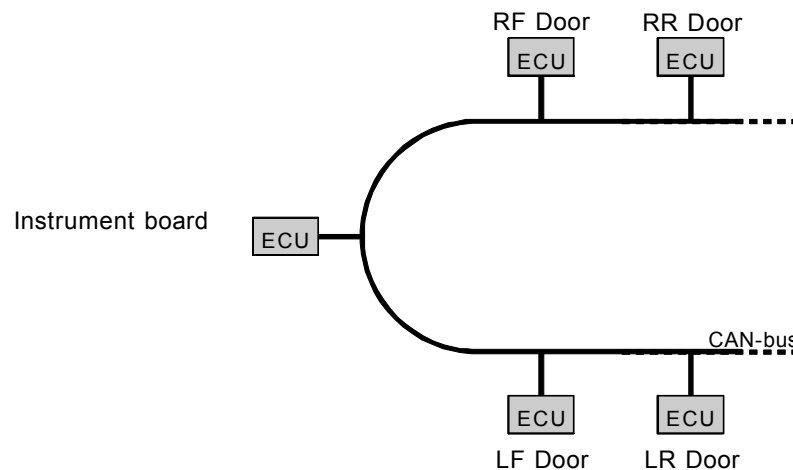


Figure 2 The CAN bus in Car window control

Note: The CAN bus is only modeled as a possibility to transfer messages between actions hosted by the different ECUs.

The system allows the driver to control all windows and it allows passengers to control windows within reach.

The driver can press a "childproof button". When this button is pressed, the rear windows are not allowed to be open more than 2" (This is a safety-critical requirement).

A system overview, which includes the driver as an operator role, is shown in figure 3.

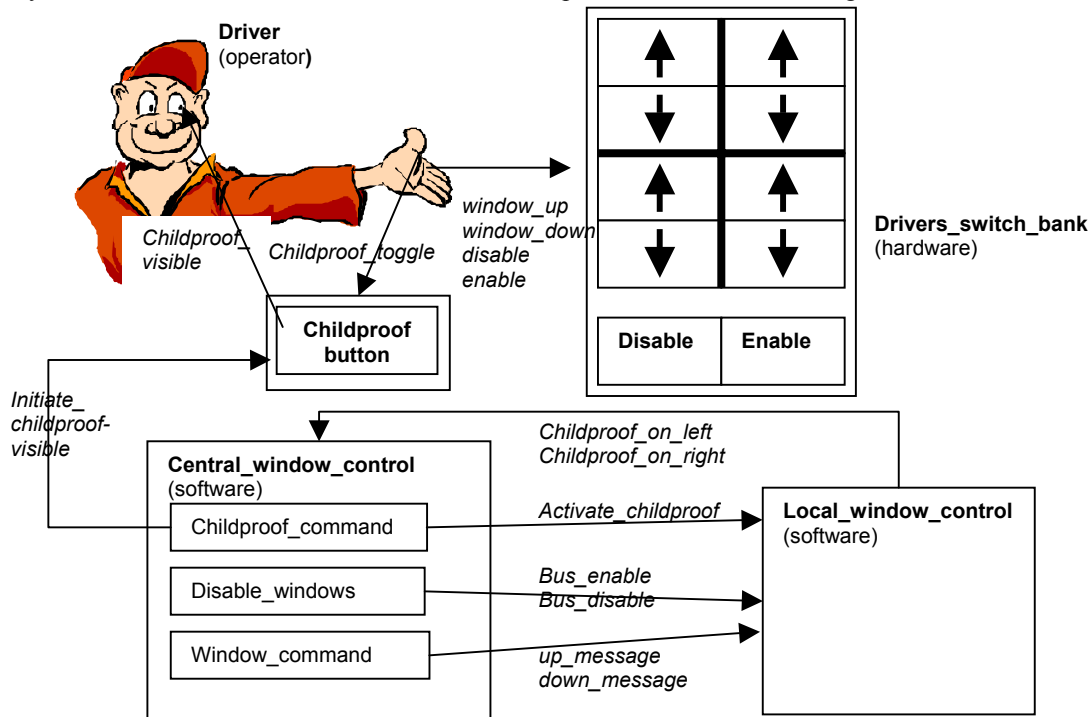


Figure 3 Overview of Car window control with messages

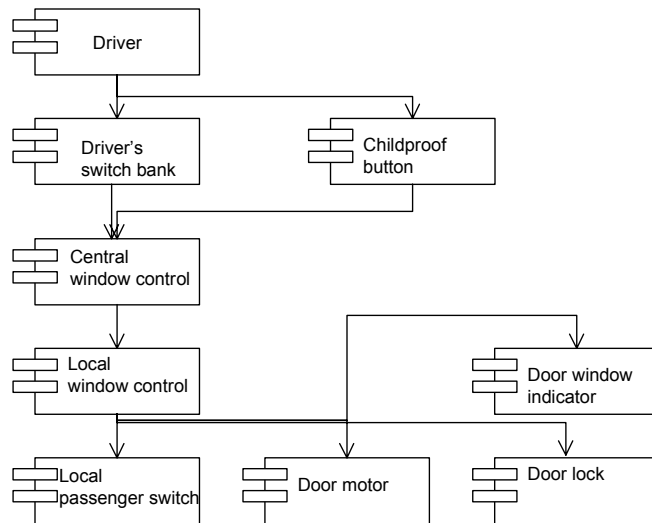
### 5.1.1 Tailor and apply the Component diagram

Grady Booch published the component diagram in his book of 1983 [2]. It has then been developed in different ways, one of these being the current UML version. UML expects you to use this diagram solely for illustration of dependency among software components. A first step towards making it useful for systems engineering is consequently to allow not only software components but also human operators and hardware components in the diagram.

For a UML component diagram for the Car window control example, see figure 4.

From a systems engineering point of view, this diagram has some problems:

- The dependency lines are drawn in a way that do not intuitively connect to the “offered interfaces” of the objects (as they did in Booch’s original version).
- It is unclear which objects are hardware, software or operators.
- The diagram is “flat” with multiple object levels, resulting in possible confusion when trying to draw larger systems.
- Objects inside and outside the current system (subsystem) are drawn in the same way with possible confusion as a result.



**Figure 4 A Component diagram for the Car window control example**

To overcome these problems a first step is to accept the modified notation presented by the HOOD User's group in [3] and then add some tailoring, useful for systems engineering purposes. This notation introduces a large box to the left in each object to list actions (operations), which the object offers. The recommended tailoring includes:

- Distinguish between the current object and its support object through drawing the objects with different size.
- Limit each diagram to two levels. This makes the arrows unnecessary since the current object will always depend on the support objects shown.
- Allow “objectification” of missions.
- Introduce a notation to distinguish the different object categories: mission (mi), operator (op), software (sw) and hardware (hw).
- Introduced a drawing style to distinguish between objects included in and outside the current system (subsystem), through placing the objects inside and outside of the current object respectively.
- Include “little clocks” in the object symbols to show development status for each object.
- Use an indented list to supplement the diagrams, to get a system-level understanding of the dependencies and to create a clear overview of multiple levels.

The result of this tailoring is a set of diagrams as shown in figure 5 and an indented list to show the complete dependency. The indented list is then an alternative way to present the Component diagram, called a “Tree graph”. It is primarily helpful to show the dependencies within multiple system levels.

The indented list for the example, as shown in Figure 5, is:

```

Driver
| Drivers_switch_bank
| | Central_window_control
| | | Local_window_control
| | | | Local_passenger_switch
| | | | Door.Motor
| | | | Door.Window_indicator
| | | | Door.Lock
| Childproof_button
| | Central_window_control -"-
  
```

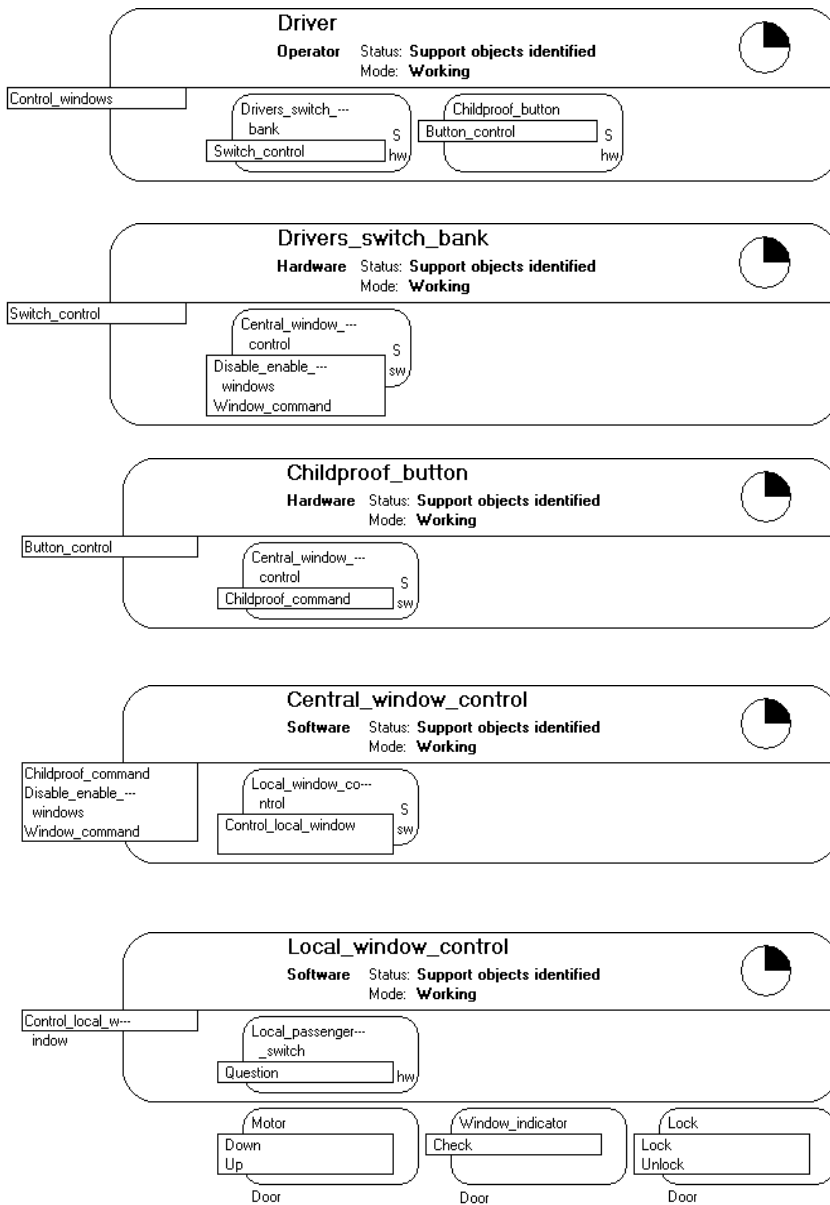


Figure 5 One component diagrams for each level

The result is that, through tailoring the UML Component diagram, a notation has been created that allows both an overview and a detailed view and which can be used not only for software, but also for complete systems. Consequently this UML diagram has been extended to support systems engineering. However the component diagram still only gives a structural (architectural) view and says nothing about the behavior within the objects. To model behavior, we need a notation which is detailed, formal and readily understandable. One such notation is Ada-based pseudo code.

### 5.1.1 Behavioral modeling with pseudo code

As an example is shown part of the Driver's behavioral description from the example, written in the Odel (Object Design Language [9]) pseudo code notation:

```
action Control_windows is
visibility: Offered
purpose: { This is a human action where the Driver controls the car's windows and
childproofing state }
messages:
childproof_visible, {received from the Childproof button hardware}

childproof_toggle, {sent to the Childproof button hardware}

{sent to the Drivers switch bank hardware}
disable,
enable,
window_down,
window_up

variables:
childproof_status: Boolean

{Enable means that the passengers may
open/close the windows, disable that
only the Driver may control the windows}
enable_button_pressed: Boolean
disable_button_pressed: Boolean

driver_window_down: Boolean
driver_window_up: Boolean
other_front_window_down: Boolean
other_front_window_up: Boolean
rear_left_window_down: Boolean
rear_left_window_up: Boolean
rear_right_window_down: Boolean
rear_right_window_up: Boolean

childproof_button_pressed: Boolean

driver_leaves_car: Boolean

begin
-- The Driver controls the windows in the car by pressing the buttons
-- shown in the picture above
-- Communication with other processes in this system is performed
-- by messages that are sent from this process
-- This action goes on (described as a loop) while the driver is in
-- the car

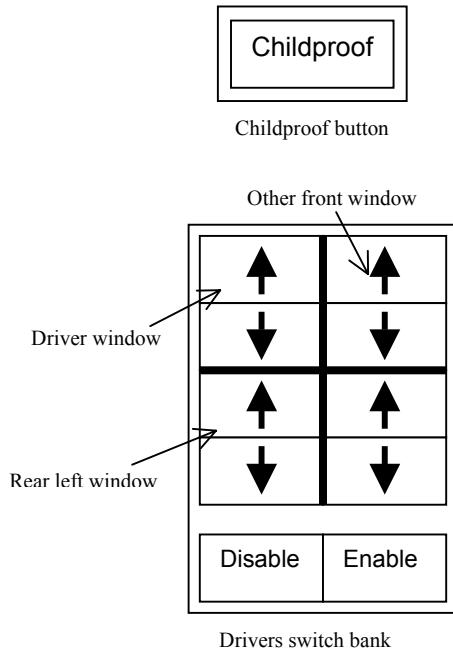
loop
{the Driver receives the message childproof_visible (in the variable
childproof_status) from the Childproof button = the Driver can see
if the Childproof button is lighted or not.
childproof_status = true - the Childproof button is lighted
childproof_status = false - the Childproof button is not lighted}
receive childproof_visible(childproof_status)

if driver_window_down = true then
-- The Driver presses the down button for the driver window
send window_down(driver)
-- A message is sent to the hardware object Drivers_switch_bank
```

```

-- The possible values of the message window_down depend on the
-- type of the message, which is seat_position.
-- seat_position is an enumeration type with possible values:
-- driver, other_front, rear_left or rear_right
end if

```



Above are the control buttons in the car that are used by the Driver

**Figure 6 Example of a figure to clarify pseudo code**

As seen above behavior can be defined in a pseudo code notation or “Formalized English” containing:

- Control structures using reserved English words
- Variables of defined types for parameters, messages and local variables
- Informal comments for clarification
- Figures for further clarification, for example of user interfaces, as shown in figure 6.

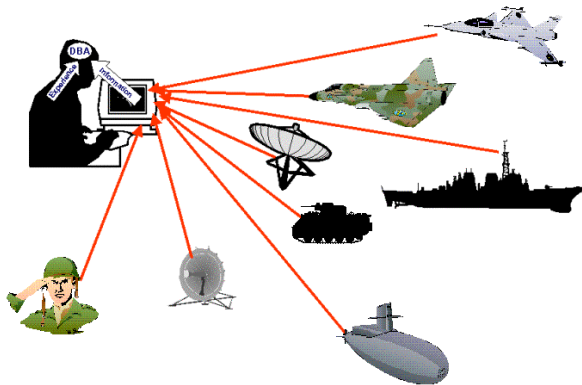
The result is a notation, which is a compromise between too informal natural language and mathematical formalism, the latter possibly being difficult to understand for application experts, concerned by systems engineering work. The “Car window control” example represents a small technical system, why we also need to study how the notation can be used in a larger more “organizational” system.

## 5.2 Dominant Battlefield Awareness

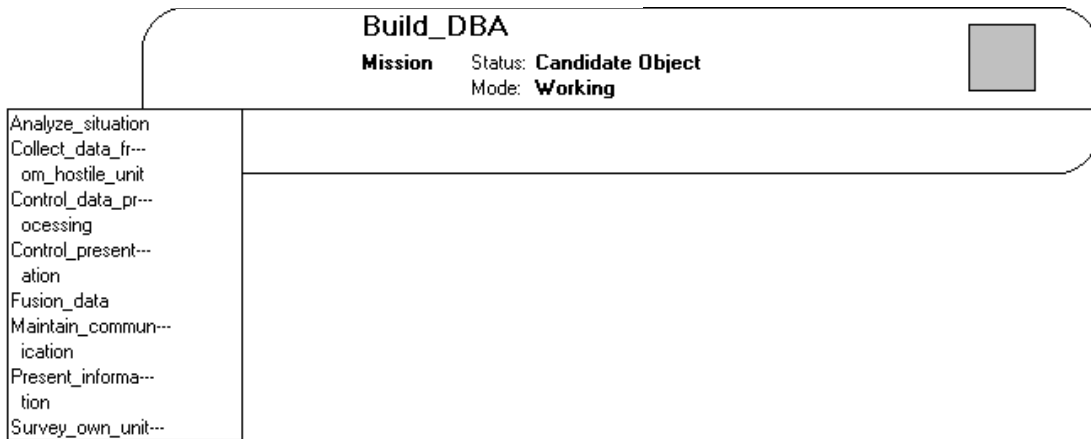
As shown in Figure 7, “Dominant Battle Field Awareness (DBA)”, means that an experienced commander combines his experience with information from multiple sources to build the necessary awareness to be able to take well-founded decisions. Although the concept originates from the defense domain the principles are fully applicable to several civilian domains.

A mission, selected to show the modeling principles is “Build DBA”, which requires abilities such as “analyze situation”, “Collect data from hostile unit”, etc. Figure 8 shows a top-level component diagram for the system, with the abilities required included as top-level actions.

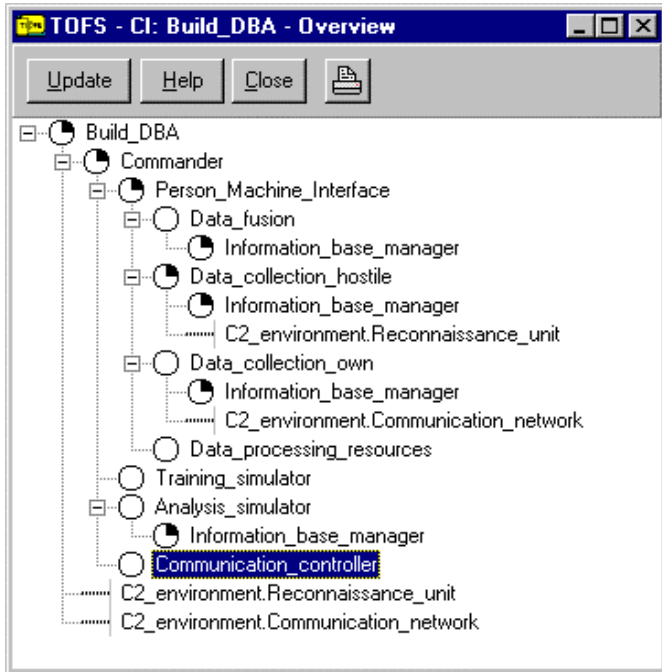
The system includes objects of categories Mission (Build DBA), Operator (Commander), Software (Person-machine interface and others) and Hardware (Data processing resources). A tree graph for the sub-system is shown in Figure 9.



**Figure 7 DBA requires large amounts of information**

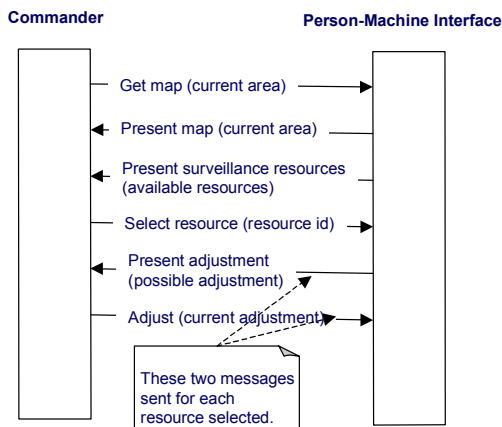


**Figure 8 Component diagram to show the abilities required to build DBA**



**Figure 9** Tree graph for the system “Build DBA”

From the tree graph you can see that the Commander depends on a Person-Machine Interface (PMI). Part of this interaction is to “Select and adjust surveillance resources”. This interaction is completed through a set of messages between the Commander and the PMI, which can be clarified in a UML Sequence diagram as shown in Figure 10.



**Figure 10** Sequence diagram for part of the interaction Commander/PMI

## 6 Conclusions

Originally a set of requirements was established with the hypothesis that a tailored version of the Unified Modeling Language (UML) should satisfy these requirements. The discussion has led to the conclusion that this was not quite possible, but that a pseudo code notation (subset of the Ada 95 programming language) should be added to give the formality needed together with understandability for end users. The

requirements stated for a Systems Engineering Modeling Language (SEML) together with the solution studied are:

1. The SEML shall include modeling of system components of categories Operator, Software and Hardware  
Extended application of the UML Component diagram allows modeling of the required component categories.
2. The SEML shall include modeling of system missions and abilities  
Extended application of the UML component diagram allows modeling of missions as objects with system abilities seen as actions within the mission objects.
3. The SEML shall support modeling of system structure (architecture)  
The Component diagram can be developed into “Tree graphs” to show dependencies on multiple system levels and thus give the necessary structural (architectural) overviews with the main relation between components in structure being “depends on”.
4. The SEML shall support modeling of system behavior  
System behavior can be modeled with UML State and Activity diagrams, but for end user readability a pseudo code notation is preferred (“Formalized English”).
5. The SEML shall be simple enough for practitioners to learn in a short time and its notation should not be too difficult to explain to end-users.  
Through using only a small subset of the most intuitive UML diagrams, combined with “Formalized English” (pseudo code) this requirement can be satisfied.
6. The SEML shall include sufficient formalism for the models produced to be possible to analyze automatically.  
The pseudo code notation satisfies this requirement.

The proposed tailoring of the UML may introduce some problems, which should however be possible to manage:

- The update of the Component diagram can be seen as a violation of the standard. Since the updates are useful for Systems Engineering purposes, they should however be seen as allowed tailoring of the standard, since the “spirit of the diagram” is retained.
- Software engineers may protest that inheritance is not discussed and that this is a core feature of Object Orientation. Inheritance however may not be the most important mechanism in Systems Engineering and it is still quite possible to model inheritance in pseudo code and in the diagrams recommended.

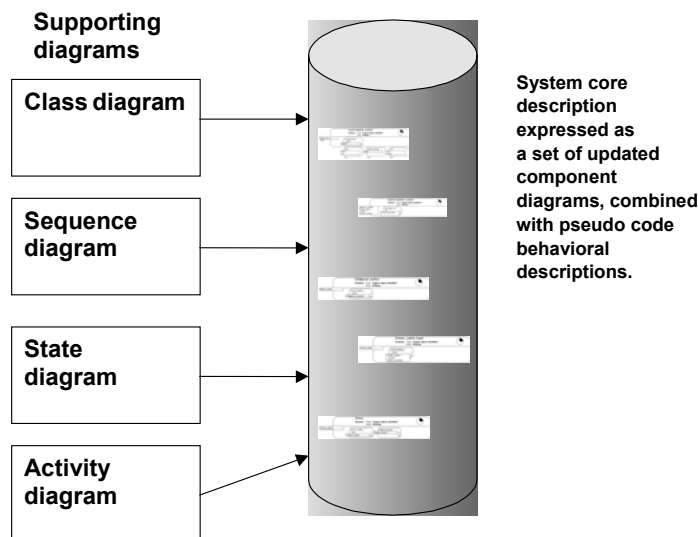


Figure 11 Summary of recommended UML diagrams

- End users will not read program code. Often true, but “Formalized English” will help designers to produce a consistent system and the resulting text will be simple to explain to end users.
- The UML already has the “Use cases”, why take the trouble to introduce “Mission objects” and “Abilities”? Use cases may be good enough for software work, but they only represent the interface between a software object and its user. That is not sufficient for Systems Engineering, which is why the “Mission objects” and “Abilities” are needed.

The result is that an SEML has been created, which allows you to build a formal, but still understandable core description of a system and to supplement this description with UML diagrams, as required by the system concerned, as outlined in Figure 11.

## 7 References

1. David Marca and Clement McGowan IDEF0/SADT™ Business Process and Enterprise Modeling, Eclectic Solutions
2. Grady Booch Software Engineering with Ada. Benjamin Cummings 1983
3. Jean-Pierre Rosen and HOOD User’s group. HOOD An Industrial Approach for Software Design 1997
4. James Rumbaugh and others. Object-oriented modeling and design. Prentice Hall 1991
5. The UML Notation guide. Published at <http://www.rational.com>
6. Martin Fowler & Kendall Scott. UML Distilled. Addison Wesley 1999
7. Craig Larman. Applying UML and Patterns. Prentice Hall 1998
8. Ivar Jacobson. Object-Oriented Software Engineering. Addison-Wesley 1992
9. Odel language description. Published at <http://toolforsystems.com>
10. Ingmar Ögren. On principles for Model-Based Systems Engineering. Systems Engineering Journal Nr. 1 2000
11. Tom DeMarco. Structured Analysis and System Specification. Yourdon Press 1979
12. Annotated Ada 95 Reference Manual, Version 6.0. Intermetrics Inc. 1995

## 8 The author

Ingmar Ögren was graduated with an M SC in Electronics from the Royal University of Technology in Stockholm in 1966. He then worked with the Swedish Defense Material Administration and various consulting companies until 1989 with systems engineering tasks in areas such as Communications, Aircraft and Command & Control.

He now chairs the board and is owning partner in two companies:

- Tofs which produces and markets the Tofs (Tool For Systems) software.

- Romet, which consults in the area of systems engineering methods with the method O4S (Objects For Systems) as its main product.

He is also a teacher of Systems and Software Engineering and has had several papers accepted at international conferences. Ingmar Ögren is a member of MOSIS (Modeling and Simulation in Sweden) and INCOSE (International Council Of Systems Engineering)

Further information about Ingmar Ögren can be found on the web page <http://www.toolforsystems.com>

