On Properties of Modeling Approaches

Rolv Bræk

Norwegian University of Science and Technology, NTNU, Trondheim, Norway,
rolv.braek@item.ntnu.no

Abstract. This paper argues for three fundamental properties that every modeling approach should posses. It presents a simple classification schema for models and proposes some comparison criteria. Finally it provides some reflection on properties currently in the CMA "parking lot".

Keywords: Modelling, Properties, Classification

1 Three Fundamental Properties

When comparing modelling approaches, it may sometimes be worth while to take a step back and consider a few fundamentals. Simply put, we make models primarily in order to develop better systems at lower cost. The question is what to model and how to model in order to achieve this simple goal.

I have found that models should balance three fundamental properties:

- Readability for human comprehension and communication.
- Analysability for reasoning and tool support.
- Realism for efficient implementation and documentation.

Possessing just one or two of these properties makes a modelling language less useful than a language that possesses all three. Being simple and intuitive for human understanding is fine, but if the gap to design and implementation is too big, its value is limited. Earlier data flow methods such as SADT suffered from this problem. Being formal and sound is also fine, but if it is too hard to read and understand by practitioners, or cannot be efficiently implemented, its value is again reduced. Programming languages, of course are realistic, but hard to read and understand, even for programmers themselves.

Readability for human comprehension and communication is probably the most important property. It is about more than just reading what is written down. It is about understanding the meaning and its implications. Readability therefore require that the meaning is clear and unambiguous. An alternative term for readability is *understandability* as proposed by Bran Selic in [1]. Building a shared understanding and agreement among stakeholders is a necessary precondition for quality. We must face the fact that our limited mental capacity and ability to communicate precisely is the main barrier to overcome in systems engineering. Models are our main tools for this and that is why readability is

so important. Somehow the models must help to reduce the mental burden by 1) decomposing into parts that can be described and understood as independently as possible and 2) expressing as explicitly as possible everything what is important to understand, but hard to overview and memorise.

Analysability for reasoning and tool support is necessary in order to ensure models that are complete and consistent, to compare systems, to validate interfaces, and to verify properties. For this purpose the language must have a semantic foundation suitable for analysis by humans as well as by tools. Human reasoning is important because one cannot completely rely on tools to understand the problems, to help avoid them and to correct them.

Realism for efficient implementation and documentation is sometimes overlooked by theoreticians, but essential for two main reasons; 1) that it shall be possible to use the models to derive efficient implementations and 2) that the models can serve to document the real system after it has been built. If this is not the case the models will be of less use and likely to erode and be thrown away once the system is realised. Model driven engineering has realism as a core property, but not always readability. We comment here that having a platform independent model does not necessarily mean that the model is readable.

Readability and analysability is not just a function of the languages being used, but also of the method guidelines and criteria applied when structuring models, including composition and decomposition criteria.

Any viable modeling approach must satisfy several additional properties such as modularity, scalability and maintainability, but the three properties: readability, analysability and realism may be seen as the raison d'etre for modelling and should therefore be central to any comparison of modelling approaches.

2 A simple classification of models

A simple and useful way to classify models is to distinguish between *design* models and *property* models in one dimension and *abstractions* in the other.

Design and property models. It is useful to distinguish between models that define the *design* of a system and models that define *properties* of the system design. Design models define the structure and behaviour of an abstract or concrete system (or system family) and can be seen as blueprints for building the system. Property models specify properties that the designed system is "measured" against. A performance model for instance is a property model, but not a design model. It is used for specification and analysis, but not as a blueprint for the system itself. Sequence diagrams are primarily used as property models to define interaction scenarios, but are normally not complete enough to serve as design models. Still they are very useful for documenting (properties of) a real system, and also for partial design synthesis and many other purposes. Communicating extended state machines on the other hand, are suitable as design models since they can define complete behaviours and also can be realised efficiently and automatically.

Abstractions. Some use "abstraction" in the meaning of aggregation or composition; i.e to represent a composite structure as a single entity. Other uses it in the opposite sense of concrete, to replace physical entities by abstract entities better suited for some purpose. For instance to represent functional behaviour by abstract state machines rather than statements in programming languages, or to use queueing models for performance rather than computational models. I propose to use the latter meaning here and to subdivide into three main abstractions: 1) Functionality, 2) Physical architecture and 3) Realisation.

In the end one will always need to develop detailed realisation models expressed using for instance, programming languages. Other models are introduced in order to improve on quality and cost compared to what can be achieved using programming languages (or HDLs) alone. The purpose of the physical architecture is to document the realisation in terms of hardware and software units. The purpose of functionality models is to define logical behaviour and information handling as precisely and completely as possible. Getting the functionality right, as experienced by the environment and users of a system, is the most pressing quality issue in most cases. Therefore, models of functionality are central to many modelling approaches. Functionality may be further refined into for instance, platform independent and platform specific models, but this is details.

The six categories defined by these two dimensions enable a meaningful and simple classification of models and modelling approaches that is useful for comparison while avoiding reference to time limited and project specific concepts like phases. It assumes that models are useful not just in a particular phase, but once defined, are useful throughout the remaining lifetime. Since a complete approach normally needs to cover all six categories, it also provides an indication of completeness.

3 On Comparison Criteria

3.1 Readability

Readability is hard to assess and quantify on any objective scale since it depends on individual background and preferences. Some people like text while others prefer graphics. Some are at ease with mathematical formula while others think in boxes and arrows. The foremost criterion is whether the approach is substantially better than programming languages (or HDLs) for comprehension and communication. So much better as to clearly be worth the modelling effort.

Programming languages are good at defining data types and action sequences (algorithms). Improvements must be sought where programming languages are not so good. In other words; to represent what is important, but implicit or invisible in programming languages. This is typically concepts from the real world such as time, concurrency, external events, sessions, data flows, part structures and state evolution. Lifting the abstraction level to concepts closer to the problem

domain normally helps to improve readability and to simplify communication with domain experts as well.

Due to its transient, dynamic and often endless nature, behaviour modelling is the main challenge. Behaviour is both hard to define and difficult to fully overview and understand. Lack of understanding and overview leads invariably to errors. Since the perceived system quality depends heavily on the system behaviour, it is important to avoid errors and get the behaviour right. Therefore, readability of behaviour models is a particularly important property of a modelling approach. When it comes to behaviour, readability is about how easy it is to overview all possible courses of actual behaviour. One should note here that action sequences as expressed in programming languages are good at specifying what a computer shall do, but not so good at defining all possible states and transitions that can be reached during behaviour execution.

Dijkstra said in his famous paper on structured programming [2] (roughly) that "one should let progress through the computation be mapped on progress through program text in the most straight forward manner". For abstract behaviour models this means to structure the models to reflect the behaviour execution in the most straight forward manner, since this helps to reduce the mental burden of mapping from the model to the meaning.

So, what characterises behaviour executions? This depends on the problem domain. Roughly one may distinguish between transformational and reactive behaviour [3]. Reactive behaviour is essentially characterised by execution of concurrent state transition sequences. Defining the concurrency and state space evolution as explicitly as practically possible helps towards readability of corresponding behaviour models. This points towards some kind of state transition model. But this may not be sufficient in itself. Petri nets for example, can express concurrency and state transitions, but the states are represented by token markings that sometimes only can be found by executing the net. Thus, the net can be very expressive, but hard to read. State machines provides explicit representation of states and transitions, but finding the states and transitions for optimal readability is often difficult.

In the web services community, state-less behaviour is considered a design goal because it reduces many problems associated with keeping sessions and session states. As long as the nature of the behaviour is session and state-less, this fine. But if there are inherent sessions and states that must be dealt with during execution, it does not help to hide them in the models (which is always possible) because this will reduce readability and increase the likelihood of making errors. (We experience problems of this kind in many web-sites.) When the behaviour execution has to deal with sessions and states, readability is increased by modelling these explicitly.

One way to assess readability of behaviour models in the criteria document could be to ask what characterises the application behaviour and to what degree and how Dijkstra's principle is applied. Is the behaviour reactive or transformational? Are there individual sessions to maintain or not? Are the session behaviour state-full or not? How is concurrency among sessions modelled? Another

important indicator for readability (and realism) is to what degree the models are used after the system has been implemented for documentation, maintenance and evolution purposes. Are models the preferred source of information, or not?

3.2 Analyzability

There is abundance of formal methods and tools specifically made for analysis. There are model checkers, theorem provers and SAT solvers, but they do not seem to be much used in main-stream systems development. One important reason is that they tend to use languages and paradigms that are unfamiliar to most systems developers. A second important reason is that the tools are not well integrated with the development tools normally used. Consequently there is additional cost involved and also the problem of keeping analysis models consistent with the development models. Important criteria related to analysability are:

- what properties can be determined.
- is the analysis directly based on the development models or is there a transformation involved; if so is it manual or automatic.
- is the analysis compositional in the sense that modules may be analysed separately and then combined without having to reanalyse the modules.
- are there size limitations to systems that can be analysed (scalability)
- is the approach also suited for human reasoning,
- does the approach provide rules and guidelines for use by designers to detect and remove problems?

3.3 Realism

Realism is not the same as being directly executable. It means that realisations can be related to the model in a relatively straight forward manner, and that the real system will behave as specified in the models (allowing for acceptable semantic differences). An important criterion for realism is whether the models serve as useful documentation for the real system or just as a temporary development step. Another important criterion is whether the approach allows several alternative realisations that can satisfy different non-functional requirements such as for instance performance and availability.

4 Some Reflections on the Parking Lot

Reusability Reuse calls for modules that can be encapsulated, defined, understood and analysed separately and then be composed in may different contexts. Hence, modularity is a precondition for reusability. In many cases there are additional cost associated with making units reusable, such as additional documentation and interface definitions, that effectively prevents making modules reusable. The cost of designing-for-reuse, is therefore a limiting factor. Also the cost of designing-with-reuse should be considerably

lower than the cost of designing from scratch so that there is a substantial value to be gained from reuse.
Important criteria for comparison can be:
- the additional cost of design-for-reuse.
- the saved cost of design-with-reuse compared to no reuse.
- the percentage of reused modules achieved in a typical design.

A general lesson from reuse research is that modules representing domain entities tend to be more stable and reusable than implementation level (software) modules. Hence, the abstraction and domain closeness is important for reuse. Reusability is related to readability in the sense that modules need to be readable in order to be reusable. Additionally they should be possible to understand and use safely without having to understand all inner detail. Does the approach tend to identify modules with interfaces that satisfy this property?

Inter-module dependency and interaction This criterion is related to both reusability and readability. Well defined modules with narrow interfaces help both understanding and reuse and possibly also analysis. A central question is: what is considered a module and what is the nature of its interfaces? Is it communication interfaces and if so, what kind of communication: method calls, synchronous interactions, asynchronous messaging? If not communication, what else?

Expressiveness Expressiveness can be related to Readability, but high expressiveness does not necessarily imply high readability. Quite the contrary, one may have very expressive statements that are extremely hard to read and understand. This kind of expressiveness is in direct conflict with readability, and not desirable. On the other hand it is also possible to achieve high expressiveness and readability at the same a time by means of techniques such as aggregation and encapsulation.

Usability (readability, understandability) I would suggest making *Readability* a criterion in its own right, since Usability seems to encompass a lot more.

5 Concluding remark

The purpose of this paper has been to give some inputs to the discussion and further elaboration of comparison criteria.

References

1. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software September/October 2003
2. Dijkstra, E.W.: Notes on Structured Programming. In Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press 1972.
3. Manna, Z, and Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems Specification. Springer Verlag 1992.