Position Paper for SEMAT

Stephen J. Mellor

StephenMellor@StephenMellor.com

The Underlying Issues: Scale and Composability

We can cross a stream by felling a log across the banks, but construction of the world's longest suspension bridge, the Akashi-Kaikyō, required engineering. Though many definitions of engineering do not mention "scale" (or "software" for that matter), scale is surely inherent, otherwise we could simply hack down the log and cross the stream.

Most approaches to managing scale employ hierarchical decomposition, creating units based on implementation, managerial organization, work allocation or even just size. This manages scale for humans, but fails to generate units that can be composed into systems effectively. Moreover, decomposition provides no basis for finding semantic units. A button is not a necessary part of an elevator, nor an ATM inherent in banking, yet they are parts of their respective systems. Instead, we need an approach that partitions systems based on semantics.

In addition to breaking the system apart, we must also be able to compose units reliably into a system. Frameworks have been proposed as a solution to such reuse and composability, but each framework imposes a system architecture, and when several frameworks are used together they can suffer from what Garlan calls "architectural mismatch."

What this means is that the extensible kernel requires:
- An approach to system decomposition based on semantic units ("subject matters")
- An approach to combining subject matters independently of system architecture

Deriving and Capturing Subject-Matter Semantics

The content of each subject matter must be derived in some manner. This entails abstraction, a oft-mentioned topic that actually receives scant attention in most software engineering texts. As abstractions are formulated, criteria must be applied to the growing expression of the semantics of the subject matter to refine the abstractions.

Today, these abstractions are often captured as an implementation choice: an Account class, say, rather than the underlying meaning of an account. Instead, semantics should be captured in an executable formalism that can be subjected to formal analysis.

The formalism must allow each semantic element ("fact") to be captured independently from all others so that each can be separately established and verified. For example, normalization techniques provide a formalism for capturing the semantics of data, and, by organizing information into sets of tuples, it rests on a firm formal foundation. Similarly, behavior-over-time must be captured in terms of orthogonal states, allowing for reachability and decidability analysis, and each element of processing must have the same behavior each time it executes so it can be tested once and will henceforth always be correct.

The formalism must be executable to allow for early testing and feedback on requirements. With a firm formal foundation (such as sets, states and functions), where each element is independent and orthogonal from all the others, we can know they are correct and compose them reliably into an entire subject matter.

What this means is that the extensible kernel requires:

- Abstraction techniques, via example or “patterns”
- A set of criteria for evaluation abstractions
- A firm distinction between semantics and software
- A executable formalism for capturing semantics comprising composable elements

System Characteristics

Correct functionality is not enough; the system must also execute “efficiently.” A credit card application that bills the customer for every transaction, as opposed to batching transactions monthly, would have the correct functionality but inappropriate system characteristics.

System characteristics are often expressed in implementation terms (performance, memory, response time), but the term includes broader system measures such as mean-time-to failure. Unfortunately, each “vertical” community has its own vocabulary, so web-application designers use one vocabulary to describe these characteristics, while embedded engineers use another. There is a strong need for a taxonomy of terms to characterize the requirements on the manner in which the system provides its functionality.

Each subject matter within the system provides a set of services to its clients, and each has “efficiency” limits and characteristics. For example, a user interface subject matter may “think” in terms of events that cause it to refresh the screen, which would not be suitable for a client that requires periodic updates. There is a strong need for a taxonomy of terms to describe the implementation-independent performance characteristics of each subject matter.

What this means is that the extensible kernel requires:

- A taxonomy for describing required system characteristics
- A taxonomy for describing performance characteristics of each subject matter

System Architecture

An execution engine that can execute the formalism directly may not meet the efficiency requirements for a given system, and so may require transformation into an implementation. An *application-independent system architecture* is a statement of the conceptual entities that make up the implementation, and the rules whereby the system shall be constructed. It may contain entities such as Processor, Iterator, Thread, and so on. The application is mapped to these entities and executed directly or transformed into hardware or software that can.

In other words, we must treat the architecture of the system as a *separate subject matter*, and like any other subject matter it has certain performance characteristics. Here, the construction of a taxonomy to describe these characteristics is even more critical that for other subject matters. Just as Knuth described the order of sorting algorithms, we must be able to describe, in a taxonomic fashion, how a system architecture performs. This will allow software engineers to select an appropriate system architecture.

What this means is that the extensible kernel requires:

- An approach to describing system architecture independently of the application.
- A taxonomy describing characteristics of application-independent system architecture

Combining Subject Matters

To construct the system, the subject matters must be combined into a system. This involves making relationships between subject matters (a button push ⇔ elevator request, eg), applying the general rules housed in the architecture (each class ⇔ struct, eg) and specific allocations of elements to implementation units (each marked class ⇔ VHDL entity, eg).

These relations and rules (though not the specific instances, obviously) can be defined generally. That is, class ⇔ table and attribute ⇔ column are both links between metaclasses, while EntryToState ⇔ FunctionCall (which could be used to link a Button changing state to a callback to call a lift) is an example of a link between of a (meta)behavior and a (meta)class.

The number of rules is, at most, the square of the number of elements in the formalism (each type of element in one subject matter may be linked to each type of element in another).

What this means is that the extensible kernel requires:

- As small an executable formalism as possible
- A library of well-defined composable rules for linking subject matters

Skills and Work Partitioning

One sign of an emerging engineering discipline is specialization of skills. Today, we still promote programmers into analysts and analysts into managers. Partitioning a system—and the work involved to build it—into subject matters enables skills specialization. Building an application requires customer interaction and abstraction skills, while constructing a system architecture requires knowledge of system characteristics and implementation technologies. We need to identify the skills required to construct different kinds of parts of the system.

Successful big systems are few, though big systems can be composed from smaller units. Today, we often partition systems into units based on implementation, managerial organization, or work allocation, and then define in blood the interfaces between the units. (This approach is especially prevalent in embedded systems/SoC where hardware engineers are separated early from their software counterparts.) Unfortunately, when the teams come together, the interfaces rarely match. To address this problem, work must be partitioned along subject matter lines, and then combined according to a limited set of composable rules.

What this means is that the extensible kernel requires:

- A defined set of skills and examples of their usage based on type of subject matter
- Criteria for (self-)allocation of personnel based around subject matters

Discipline

An engineering discipline requires data to evaluate the process whereby systems were constructed and the success or failure of those systems. Today, we collect virtually none. What little data is collected cannot reliably be compared to that collected on other projects.

What this means is that the extensible kernel requires:

- A set of measures for process quality
- A set of measures for system/subject-matter quality

An extensible kernel will need more than this, but a focus on scale, composability, formalism, characterization, system architecture, skills and measurement is a good start!

Appendix

The following is a summary of the points made above, numbered for ease of reference.

What this means is that the extensible kernel requires:

1. An approach to system decomposition based on semantic units ("subject matters")
2. An approach to combining subject matters independently of system architecture
3. Abstraction techniques, via example or "patterns"
4. A set of criteria for evaluation abstractions
5. A firm distinction between semantics and software
6. A executable formalism for capturing semantics comprising composable elements
7. A taxonomy for describing required system characteristics
8. A taxonomy for describing performance characteristics of each subject matter
9. As small an executable formalism as possible
10. A library of well-defined composable rules for linking subject matters
11. A set of measures for process quality
12. A set of measures for system/subject-matter quality

Note that requirement 9 is a refinement of Requirement 6.