Vision Statement for Substrate 2025 Discuss

Dave Thomas (dave@bedarra.com)

Why a modern Substrate for Computational Science?

My opinions are based on experience building both academic *Smalltalk, Lisp and commercial environments OTI/IBM VisualAge Smalltalk and Java; Eclipse; Kx Analyst*. It would be arrogant to call these substrates, but they were what we had and used at the time. An environment consisted of an open collection of authoring tools; language tools; libraries (frameworks and components) which supported different domains, through multiple perspectives of the authors and other consumers. They were provisioned through versioning and deployment repositories to geographically distributed team members.

They took more time, did less than expected, and were challenging. They lacked sufficient active external contributors' viewpoints as well as their resources. In cases such as APL, Lisp, Smalltalk, Java - premature competition resulted in divergent environments which impeded adoption as well as investment in standards and more functional/performant products. More importantly it forced talented research groups to lock into specific environments making it all but impossible to establish deep collaborations.

Unfortunately, these environments are IMO still the most productive for building interesting creative tools. However, tomorrow's researchers and product developers see little value in stepping back into the 1980s-90s. Those who do often find it difficult to publish and share their work. Those building the newest collaborative creative tools see no path from the promising work in old environments to timely use of these ideas in tomorrow's product!

We MUST leave the 80s-90s time capsule to explore and build a better modern substrate with the necessary functionality and interop for the future. The design and development of next-generation creative collaborative tools necessitate a robust foundation that exceeds the capabilities of a small, isolated team. Individual efforts can highlight ideas but rarely scale beyond a few enthusiasts.

The substrate's sustainability is crucial. A disciplined engineering team must maintain the substrate to ensure its broad longer-term usefulness. One needs only look at Linux and Java to see that even these substantial ecosystems are themselves fragile. The only way I know to create complex software which is robust and maintainable is bottom up. "*Everything should be written top down, except the first time" Alan Perlis, who also inspired Alan Kay.*

What is the Substrate?

- Hopefully, this workshop will lead to a consensus on features that should be considered for the substrate. It must provide the key concepts and capabilities of the individuals and their artefacts from previous generations. I look forward hearing your vision. Here is my wish list. *I include in italics a pointer to work that some people may not be familiar with.*
- A multilanguage authoring environment with the features *Augment NLS, Smalltalk, Lively, Racket, Glamours Toolkit*, with metamodels.
    - The languages must provide proper pattern matching capabilities.
    - At least one of the languages needs to support the computational and data capabilities equivalent of k4 with a Turing complete equivalent of SQL or Datalog.
    - A small safe C– language for building robust efficient runtimes to build VMs and libraries and external interfaces.
    - First class model/design languages for describing system/architecture concepts - both data and relationships (ER, Graph) as well as state machines, decision tables and constraint/flow tables. *Simula and Beta tried to create a single language for modeling and coding.*

- High quality well designed and efficiently implemented libraries. Libraries should have just sufficient interfaces with well-defined semantics with examples! Libraries should where possible be components rather than frameworks. One should not require introspection of the source code to use a module, class, or function. *OO languages make it a badge of honor to implement the libraries in that language. This can have an impact on performance, functionality, and reuse of a proven library. Serialization needs in the VM not a library function as it is in Java.*

- There should be no event loop tyranny. There should be active objects (first actors) ideally with straightforward send-receive-reply of *Toth, QNX, V-Kernel and Harmony* RTOS. Every object should be capable of rendering itself concurrently with other objects – even the Amiga OS associated a process with every window. Both UX and applications need to be able to use actors for process structuring. UX needs them to support multimodal | multiuser interfaces. *Erlang OTP provides robust process structuring guidance, despite having hidden queues. Actra/Smalltalk incorporating Harmony to create an industrial prototype used for developing radar ESM algorithms.*

- OS and browser widgets contain bugs and limitations. Today's apps tie themselves in knots trying to multiplex interactions and rendering via the event loop! The substrate must provide fast multilingual multi-font authoring and editing tools with the expressive power of Tex/Latex and the usability of a good word processor. *Augment implemented an immutable journal, fine grained addressing using purple numbers; and scriptable links to support dynamic on the fly views. Tex provided algorithms for unbeatable text and visual presentation.*

- Text, 2D, 3D, Sound, Video, Ink need to be equals. Ideally the graphics and widgets should be uniformly vector based. There needs to 2D graphics at least as good as an interactive grammar of graphics. *Display Postscript – SunNeWs, Squeak Morphic demonstrate the viability of this approach, but current browser and OS platforms fail to provide vectors all the way down. The excellent book Grammar of Graphics is still the best specification of a graphing, plotting library. Processing also provides a nice environment for visual creators. Augment implemented an immutable journal, fine grained addressing using purple numbers; and scriptable links to support dynamic on the fly views.*

- Fine grained semantic versioning integrated into the substrate. It should be semantic based on program units not file based. One wants to avoid the hell of Git branches, merge conflicts, and rebasing. *ENVY/Developer explored this space in the late eighties; however, we didn't deal with shared globally persistent name management. Unison elegantly deals with name management using unique hashing. Pujil -uses the theory of patches to improve over Git.*

- The ability to quickly create new tools/substrate extensions without needing to understand the deep underlying machinery. *The best exemplar I am aware of is the Glamorous Toolkit GT which leverages the inspector abstraction to enable developers to quickly obtain visibility to an existing opaque entity or an external entity.*

- Today, we have loosely coupled collaborative development with a shared build environment, repository, messaging platform, and CRDT Lo-Fi editor. We still, however, edit our code on the client and execute large filed based parser, type checker; data and control flow analyzer; linter; on a client or server. Using server-based tools can enhance scalability and collaboration by supporting multi-developer systems. We have powerful, high-memory machines. Edits in this world would be transactions executed in the compile server. Robust multilanguage language servers can support faster cycle times for teams of developers. Optimizers could run continuously. *Energizer for C++ discussed these concepts in the early 1990s.*

- Programming in the Large – There are still no good tools for working with exceptionally large code bases. The Agile fan boys talk about refactoring; but in practice it is a major rewrite. These changes involve significant modifications to key code bases. They take way too long and often fail. Given a server-based environment such as envisioned above one should be able to separate the important activities of understanding the code base; planning the major changes; explore the impact of the changes and be able to apply them as transactions. Extensive runtime monitoring can complement the static semantics of the code base with runtime information which is now customary practice for cloud based micro services.

  Given a substrate with the above I think a small team should be able to build a Brett Victor Games Developer environment in months versus years.

Is this an engineering problem or a research problem?

It is clearly both! There are innovations in design, in algorithms, in implementation.

IBM and Sun plus others spent $2B to bring Java to where it is today.
Clearly not everything was publishable research but look at the number of papers in OOPSLA, ECOOP, LCSE … by Java and related environments.

Look at the companies trying to build new collaborate notebooks, data analysis tools etc. They could all focus on their core product if they first did not need to each build their own infrastructure – a collaborative spreadsheet; a Notion like Wiki; a linked note taking environment etc.

********** PAPER 15 **********
AUTHORS: David Thomas
TITLE: Vision Statement for Substrate 2025 at Programming 2025

++++++++++ REVIEW 1 (Gilad Bracha) +++++++++
I humbly suggest a name for the system outlined: Nirvana. It is a very tall order -even though I agree with most of it, I am skeptical of achieving it. I do want a live document system that can manage typeset like TeX, manage big data with the power of K/APL, support actors at least as well as Erlang, be as programmable and as live as Smalltalk, as easy to use as Logo, programmable all the way down etc etc. And of course it needs a well funded team.

Since I don't expect that to happen in the traditional way, I can only hope that AI will allow for something as useful and as pleasant (or more) with much less effort.

++++++++++ REVIEW 3 (Pavel Bažant) +++++++++
Dear Dave,

The mixed academic and commercial background you describe provides a unique and valuable perspective. Building environments to support multi-domain creative teamwork is personally interesting to me, still highly relevant, and not a solved problem.

You mention that premature competition impeded adoption and the formation of standards, and prevented deep collaborations between research groups. I'd love to hear more about such historical and social/political context. I wonder why Common Lisp didn't succeed then, given its standardisation.

Most reasons these systems didn't take over the world might be "subtle and hard to see" (Marcel Weiher, personal communication). I've tried to introduce mature but fringe technology (JetBrains MPS, Prolog) in a team setting and it fizzled. I cannot quite pinpoint why.

Commenting in-context below:

What is the Substrate? ... It must provide the key concepts and capabilities of the individuals and their artefacts from previous generations...
- Yes, we should study the past. But this is not so easy as many of the concepts can only be learnt by spending some time at emulated old systems.
- I like that you mention a wide array of interesting technologies, not just the usual culprits.
- As you mention later in the paper, part of the reason we don't have these nice things is missing infrastructure. Syncing, ubiquitous addressability. Standard ways to describe binary formats. A standard low level language to express pure computation. To build this requires political will.
- You mention a lot of DSLs. If we get meta-tooling right, DSLs should be easy to do.

High quality well designed and efficiently implemented libraries.
- I think we already do have a lot of libraries. If only we could trivially and safely call their functions from any language...

Event loop tyranny, Unison, Pijul.
- I am in favour of actors, too. You might find the Pony language interesting.
- Unison is great, but assigning a UUID to every element is also viable.
- Pijul deserves more attention.

The ability to quickly create new tools/substrate extensions without needing to understand the deep underlying machinery.
- I hate to say that, but AI might circumvent the need for a proper solution.

Today, we have loosely coupled collaborative development with a shared build environment, repository, messaging platform, and CRDT Lo-Fi editor. ...
- Can you explain that in more detail?

Programming in the Large...
- Important, but I am not qualified to comment on that

Is this an engineering problem or a research problem?
- Others might disagree, but I'd argue it is a _funding, design and engineering problem_. "Just" redesigning proven concepts to work in today's environment would already be terrific.

They could all focus on their core product if they first did not need to each build their own infrastructure...
- Yes that would help.

I've enjoyed reading the article.

++++++++++ REVIEW 4 (Steven Githens) +++++++++
What struck me the most about this was "the substrates sustainability is crucial", which I think is very important to keep in mind, especially in a workshop where everyone is keen and excited to "leave the 80s-90s time capsule" and create new things. In order for a substrate to actually be useful to those in the large, they need to know that the games they build on top of it will continue to work far in to the future. Also, has someone who is currently modernizing a substrate that was originally started in the 80's, I'm not sure it's always so important for us to be able to build a better, more modern thing, as it may be to sustain and modernize an already existing substrate.