BRAHMS: Novel middleware for integrated systems computation

Ben Mitchinson[a], Tak-Shing Chan[a], Jon Chambers[a], Martin Pearson[b], Mark Humphries[a], Charles Fox[a], Kevin Gurney[a], Tony J. Prescott[a]

[a]*Adaptive Behaviour Research Group, Department Of Psychology, The University Of Sheffield, Sheffield, S10 2TN, UK.*
[b]*Bristol Robotics Laboratory, University West Of England, Bristol, BS16 1QD, UK.*

Abstract

Biological computational modellers are becoming increasingly interested in building large, eclectic models, including components on many different computational substrates, both biological and non-biological. At the same time, the rise of the philosophy of embodied modelling is generating a need to deploy biological models as controllers for robots in real-world environments. Finally, robotics engineers are beginning to find value in seconding biomimetic control strategies for use on practical robots. Together with the ubiquitous desire to make good on past software development effort, these trends are throwing up new challenges of intellectual and technological integration (for example across scales, across disciplines, and even across time)—challenges that are unmet by existing software frameworks. Here, we outline these challenges in detail, and go on to describe a newly developed software framework, BRAHMS, that meets them. BRAHMS is a tool for integrating computational process modules into a viable, computable system; its generality and flexibility facilitate integration across barriers, such as those described above, in a coherent and effective way. We go on to describe several cases where BRAHMS has been successfully deployed in practical situations. We also show excellent performance in comparison with a monolithic development approach. Additional benefits of developing in the framework include source code self-documentation, automatic coarse-grained parallelisation, cross-language integration, data logging, performance monitoring, and will include dynamic load-balancing and 'pause & continue' execution. BRAHMS is built on the nascent, and similarly general purpose, model markup language, SystemML. This will, in future, also facilitate repeatability and accountability (same answers ten years from now), transparent automatic software distribution, and interfacing with other SystemML tools.

Key words:
integration, framework, embodied modelling, biomimetic robotics

1. Introduction

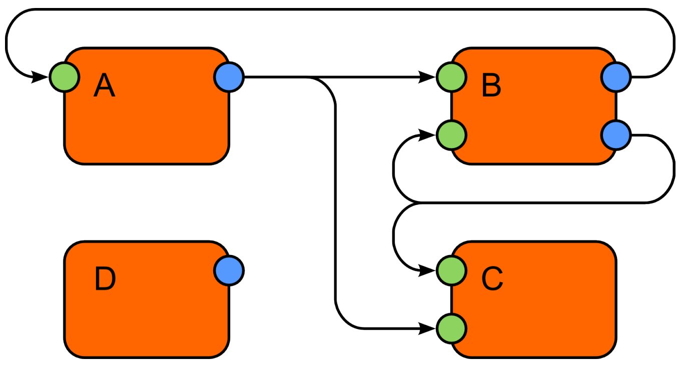

Figure 1: Example block diagram of an integrated dynamic system built from independent processes. Rectangles are processes, circles are input/output ports, and arrows are links between ports.

Moves are afoot in the world of computational biological modelling towards the construction of large, integrated, models [1, 2, 3, 4, 5, 6, 7, 8], in the spirit of Daniel Dennett's 'whole iguana' [9]. Such models will generally include components that represent disimilar biological operations (e.g. neural processing, muscle mechanics, pharmacological interactions) at different levels of abstraction (e.g. biophysical, point cells, cell populations, tissues, purely phenomenological, see for instance [13]). Implementations of such integrated models can be either 'modular' (each component stands independently and exposes an interface to allow its integration into a larger system) or 'monolithic' (where such interfaces are lacking or informal, such that reconfiguration of the larger system may require modification of components). Here, we highlight some advantages of modular implementation, and introduce a new proposed framework for such implementations. Whilst this framework has its roots in solving problems in the field of biological modelling, we emphasise that it is not limited by those origins, and we expect it to be of equal interest to researchers in other fields and sectors. The problems of integration, of course, become all the more visible when crossing disciplinary and organisational boundaries.

These developments have particular salience when we also consider the parallel move in biological modelling towards embodied models [16, 17], that is, deploying models as robot controllers. This approach is intended to discover the shortcom-

Email addresses: b.mitchinson@shef.ac.uk (Ben Mitchinson)

ings of models of biological systems by asking them to solve the same problems with which the organism itself is faced—*in vivo* modelling, as it were. The particular problems thrown up by this move are discussed further, below. This, in itself, is driving the engineering discipline of biomimetic robotics [10], whereby biological mechanisms are seconded to replace technological mechanisms in real-world robot control problems. The goal of this work is to produce information-processing systems that integrate conventional control systems with biological models, with algorithms at different levels of abstraction, and that maintain reliable performance in complex, changing, real-world environments. Whilst embodied modelling research can often be performed under laboratory conditions with generous resources, if biomimetic systems are to be deployed effectively in production robots they must be able to operate in severely resource-limited environments. The case studies, presented below, are taken from these two, related, frontiers of research. The first addresses all of these problems simultaneously—large-scale integration, reliable deployment in a complex environment, and strict resource limitations.

Modularity is considered a desirable general trait in the design of complex software [11], in large part owing to the principle of 'divide and conquer' [30], breaking down a problem into manageable, and largely independent, sub-problems. As integrated models become more complex, so researchers can expect to benefit increasingly from the software-engineering advantages of modular development—that is, developing collections of relatively independent 'processes', 'linked' together to form a 'system' that can be computed, Fig. 1. In addition, model components have become complex in themselves so that reimplementing a component from published equations is no longer trivial, driving a need to share component implementations if peers are easily to review the operation of, or include in their own integrated models, components published by others. Components embedded in a monolithic system cannot easily be extracted, and thus cannot easily be included in other systems. Furthermore, software development in the academic environment has needs that are particularly well-served by the modular approach. At base, our intent is dissemination of knowledge, so the only barriers to sharing software are technical. Physical researchers build models of physical systems, and since we all research in the same universe, these models are inherently reusable. Software is (for most academic disciplines) not the goal itself but a necessary (and laborious) step towards the goal, so that the 'not invented here' culture is ameliorated in comparison with some other arenas. Finally, directly representative models of the physical world are, by nature, dynamic system models, so the interface that each module requires is, essentially, identical.

We define a Modular Execution Framework (MEF) as any middleware that facilitates linkage of software components into a computable dynamic system—a well-known example is Simulink [12]. Given the varied levels of abstraction and the disimilar operations represented, it is not generally possible to compute an integrated model in a single computation *engine*—rather, the challenge is to link task-specific engines together [8]. A bespoke solution to this integration problem may serve for any single project but, as we will see, a general solution offers more for less effort, and the startup cost is considerably outweighed by the immediate benefits.

In Section 2, we run over the particular challenges and requirements for integrated computation, with emphasis on the academic research environment; we expect most of these issues to be familiar to workers in other environments, however. We go on in Section 3 to introduce our proposal, BRAHMS [14], beginning with an overview of its use. In Section 4, we contrast BRAHMS with existing and developing solutions and show it to be well positioned with respect to these. In Section 5 we report case studies from projects that are using BRAHMS in solving real-world research problems. In Section 6 we report, briefly, on performance and scaling. Section 7 provides a complete worked example of developing for BRAHMS. We report on project status in Section 8 and conclude in Section 9 that BRAHMS already offers a solution to most of the identified challenges and will, through planned developments, meet the remainder.

2. Challenges And Requirements

2.1. Varied Development

The primary challenge (as described above) is to integrate software processes, and the primary requirement is, therefore, to offer a middleware platform which will execute processes in concert. Inter-disciplinary integration may include processes developed in, on, or by different authors, centres, platforms, programming languages, human languages, programming styles and (importantly) at different times. Cross-industry, integration may also be required across problem domains and technical languages. Without direct communication and substantial refactoring, such disparate offerings will not generally be integrable. Software engineers meet such challenges by offering fixed, public, interfaces to develop against. In this context, an interface requires two facets: one between process and framework, the other between processes; these interfaces must be general (exclude no possibilities), static (backward compatibile), accessible and available in multiple programming languages on multiple platforms.

Since it is not immediately obvious, we choose to highlight in particular the importance of integration across time. Often, researchers wish to revisit their own, or other workers', investigations at a later date, and the first stage of this is confirming that the original results can be reproduced. Surprisingly often, in our experience, this is not possible, and the reasons are not always clear. An effective tool for integration through time must offer repeatability, such that 'old' code can still be executed, and will generate the same results that it did originally.

2.2. Varied Deployment

High-end multiprocessing hardware is increasingly becoming available to research labs, supporting a rapid growth in the development of 'large-scale' models [15] (models with many dynamic states). At the same time, increasing focus on 'embodied modelling' [16, 17] (deployment of behavioural models on robotic hardware) is generating use cases based on low-end hardware. These two trends push the computational envelope at opposite ends, and any solution must be deployable in all these environments. A researcher may develop initially on a desktop machine, for convenience; experimental work may involve large models or parameter spaces and, thus, high-end hardware; embodied models will eventually be deployed on robots. If there is an intellectual interest in mobile robotics *per se*, robotic deployments may be required to run entirely on low-end embedded hardware. Alternatively, if the interest is more in the capabilities of the model, such deployments may have to run across embodied systems and larger, off-robot, systems, with the two hardware components communicating over some network (e.g. Wireless Ethernet). The requirements are, that a researcher should only have to develop once for such varied deployment cases, and that the middleware should be able to take advantage of the resources of the currently available hardware without becoming unwieldy on low-end hardware.

2.3. Code Sharing

Computational researchers spend much time authoring software, and disappointingly often this work is repeated in other labs, by other researchers, or even by the same researchers, when documentation, compatible source-code and/or compatible binary code is, or becomes, unavailable. Anecdotal (and more concrete [5, 18]) evidence suggests that '...easier to rewrite it myself than try and obtain/understand/integrate their original code...' is a common story (this should not be taken to indicate that reimplementation is a trivial task—see the referenced works). Any solution should offer great potential for code sharing and reuse, which is to say more than that the code *could* be integrated—it must be *straightforward* to do so. This requires a (preferably, automatic) archiving/distribution mechanism, that shared code be in a form that is immediately usable (rather than having to be configured, have its dependencies met, and then be compiled with a particular tool, say), and that the solution encourages authors to document their work [19] (facilitating 'intellectual' integration).

In addition, 'background functionality' like parallelisation or data marshalling is neither trivial nor quick to author (and most researchers do not want to become software engineers), so sharing such functionality with all process developers is desirable. Therefore, as much functionality as possible should be subsumed into the middleware—'general' process code should be shared. We use the term 'supervisor' to refer to this shared code, which might, at minimum, be responsible for reading a system document, loading required processes, connecting them together, progressing them through time, collating results, and returning these to the caller.

2.4. Open Standards

There is more to working with integrated systems than their execution; other possibilities include a system design GUI and an archival/retrieval tool (see also Section 2.3). It is, thus, a requirement that the middleware should work with open and extensible data standards. Moreover, the needs of academic research are constantly changing and often cannot wait—the solution must be able to support these changing needs in a timely manner. In practice, therefore, it is a requirement that the solution be open source and extensible by anyone. It is also widely-held in the community that science should be available to everyone, which strongly favours the adoption of non-commercial solutions.

2.5. Adoption

If they are to adopt any proposal, potential users must perceive its advantages to outweigh its costs, both initially and in the long-term. The short-term requirement is met if the startup cost is sufficiently low—interfaces must be few, simple, and well documented; immediately available 'added value' will help to offset this cost. The long-term requirement is met if, in comparison with an equivalent bespoke monolithic design, overall performance does not suffer and per-process development effort is similar or less. Furthermore, an integration solution should by its nature be inclusive, so the solution must be available to all, no matter their access to resources; this observation echoes that above in favouring a non-commercial solution. Adoption will be the more willing the more freedom that is given to the developer to do things their way—this means making the interface available in multiple languages, on multiple platforms.

3. BRAHMS

BRAHMS represents part of our commitment to the philosophy and methodologies of neuroinformatics: developing general purpose tools that facilitate large-group working in neuroscience, and sharing and reusing resources. It is an MEF developed in-house during the course of a large-scale, multi-centre project (WhiskerBot [3]) which presented many of the challenges outlined in Section 2. This was a computational neuroscience project, but from the outset BRAHMS was required to integrate diverse processes (see Section 5). The primary design goals of BRAHMS are performance, flexibility and extensibility. BRAHMS is open source and licensed under the GNU General Public License. In this section, we provide a brief description – for much more detail and formal specifications, see the documentation [14].

3.1. Overview

BRAHMS operates on systems, Fig. 1, progressing them through time and generating output, Fig. 2, which comprises, in large part, logs of process output ports. BRAHMS represents systems using the file format SystemML [20], an open XML-based format for representing stateful dynamic systems. Processes can be developed by dropping state initialization and update code into one of the provided templates (using a programming language that suits the developer). Systems are built from these locally-developed processes, processes developed by other researchers, and processes provided with BRAHMS,

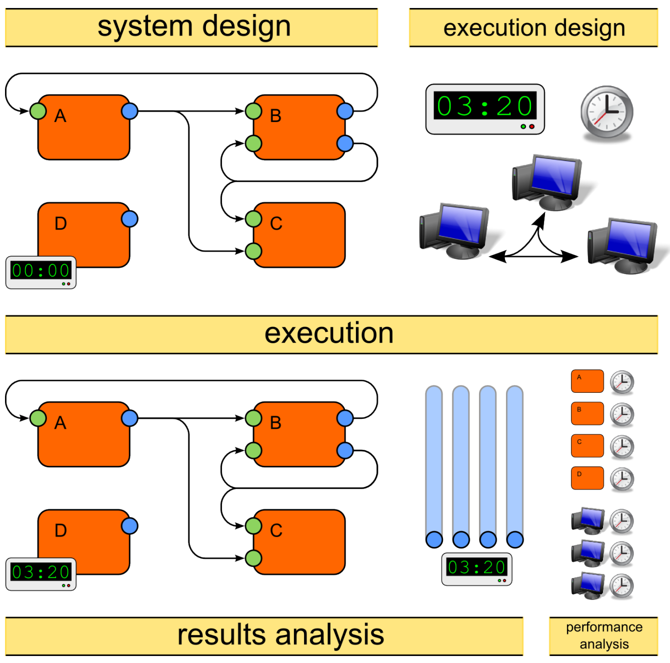

Figure 2: Typical BRAHMS workflow: system design (link processes into a system); execution design (run time, performance monitoring options, deployment model); execution (progress system, log output ports, monitor performance); results analysis (review new system state and output logs); performance analysis (review per-process and per-machine data).

in just a few lines of script. BRAHMS is invoked to execute these systems, taking advantage of parallel computing resources where available, and the results can easily be pulled into an analysis environment. BRAHMS is not tied to any particular interactive environment but, currently, comprehensive invocation bindings exist only for Matlab [12]. In time, the library of provided (and third party) processes will increase, additional supervisor functionality will accrue, and bindings will be developed for other environments (see Section 9.2 for future plans).

BRAHMS is implemented as an execution 'engine', which progresses SystemML documents through time, and a lightweight front-end for that engine that is invoked from the shell. Taken together, these comprise the 'supervisor' discussed in this document, and detail of the relationship between the two is beyond our scope, here. We note, however, that alternative front-ends may be authored in future, providing a graphical interface to BRAHMS, for example (we mention this, also, in section 9.2). In addition, the BRAHMS engine may be included as a component in third-party software systems.

3.2. *Systems and SystemML*

SystemML [20] defines a virtual namespace of specifications of components (processes, data containers, utilities). The namespace comprises public branches (which are shared, persistent, and can be augmented by any user subject to some constraints) and private, local, branches (which are unconstrained). A SystemML file contains a snapshot of a stateful dynamic system in time, Fig. 1 – that is, a collection of stateful processes (drawn from the namespace) and a collection of stateful links (containing data containers drawn from the namespace) connecting them together. Along with the SystemML infrastructure, which provides practical support (a server for public branches of the namespace and a software toolbox for interacting with the server and with SystemML files), these constitute the SystemML project. The first public namespace server is expected to be commissioned in the coming year (meanwhile, only local namespace branches are in use). The BRAHMS supervisor's main functionality is to load a SystemML system from file, instantiate it using the namespace, progress it through time, and finally, if required, store its final state to file.

Each process in a SystemML file is represented as a SystemML class name along with class-specific state data. The class name indexes the SystemML namespace to find a process specification, which details the process interface and the computation it performs. Attached to that specification may be implementations for different environments, e.g. BRAHMS. A process may publish outputs, each of which specifies a transport protocol (e.g. periodic transport at a specified sample rate) and data structure. The data structure comprises a SystemML class name and class-specific state data. The class name indexes the SystemML namespace to find a data container specification, which details the data interface and the data it contains (for instance, BRAHMS ships with a data class representing a multidimensional numeric array). As with processes, implementations of the data container for BRAHMS may be attached to the specification node. Each link in a SystemML file specifies a source output (implicitly, thus, a transport protocol and data structure), a destination input, and transport-protocol parameters (e.g. sample delay, for a periodic transport protocol).

The class-specific state data for a process or data container (termed 'StateML') is understood by design tools for and implementations of those classes (for instance, NeuroML [8] might be the StateML for some neural processes). These class-specific data are not used by a SystemML client such as BRAHMS so the set of systems that can be represented is extensible simply by the addition of new process or data classes to the namespace. Transport protocols are the responsibility of the middleware, however, so extending this set requires augmentation of the BRAHMS engine.

3.3. *Processing Model*

The BRAHMS processing model is deterministic, with asynchronous threads and processes synchronized by data transfers (there is no additional message passing at run-time, only the propagation of data through links). Data transfers can be unidirectional, and/or through arbitrarily long pipes, so physical processing can desynchronize significantly where permitted, allowing good advantage to be taken of available resources. Currently, all inter-process links are periodic (though data rates can vary, where the data container class being transported is of variable size). The only request made of processes by the supervisor at run-time is that they 'service' their input and output ports; that is, read and write their ports at a given time instant. This requires that any pre-requisite processing be completed, so the process is implicitly asked to compute a series of deterministic time intervals (often referred to as 'ticks' or 'steps', especially in periodic systems). Since BRAHMS supports mixed sample rates, these intervals may be of varying periods (if inputs or outputs are running at sample rates that are not multiples of one another).

BRAHMS does not attempt to provide run-time functionality – interaction with the system from the framework is limited to pause and cancellation. This is by design: such interaction is provided exclusively by component processes of the system. Of course, processes can provide any interaction desired, so this is not a limitation (rather it is part of the modular philosophy). As example, a GUI process was recently developed in our group that allows the user to inject signals into the system using the mouse at run-time. Conversely, multiple GUIs are under development that provide visual feedback of system operation (i.e. act as virtual instruments); one is included in the current public release, for testing.

Note that this absence of run-time functionality also leads to an important aspect of the BRAHMS approach to integration. Where data streams require transformation to be understood by source and destination processes, this transformation must be provided by an interposed process – BRAHMS itself offers no data transformation services. An example of such a transformation is the numeric stream resample process provided with BRAHMS. Note, however, that BRAHMS *is* responsible for data transport, so 'transformations' that are purely transport operations (such as transport delay) can be implemented by the middleware (or, if appropriate, by a process).

3.4. Interfaces

The core of BRAHMS is the supervisor-process interface—C was chosen as the language for this for its durability and interoperability with other languages. Language bindings are provided for C++, Matlab and Python, and additional bindings are expected to follow. The interface as a whole has been designed according to modern API design principles [21], for example with an eye to minimality, extensibility, and backward compatibility. It comprises four aspects, as follows.

First, processes offer an extensible events interface through which the supervisor can invoke process operations (such as initialize or service ports). The remaining three aspects are callback interfaces allowing processes to invoke supervisor functionality in the context of an event. They are: an implementation of parts of the W3C XML Document Object Model interface [22] (for manipulating StateML, amongst other things), the inter-process interface (for interacting with ports and, thus, the wider system), and a small interface of BRAHMS-specific functions (e.g. serving to send log messages or obtain random seeds).

In addition to the above, each data container class offers a class-specific interface to allow manipulation of its contents by processes. Inter-process data passing then proceeds as follows: the source process publishes a port during initialization, specifying the class and structure of data containers to pass over it, forming the source end of a link; at run-time, the source process writes data into a container using the class-specific interface, and passes the container into the link; the supervisor marshals the data across the link according to the transport protocol; the destination process pulls the container from the other end of the link and reads the contents using the class-specific interface.

One interesting aspect of the interface is the serialize/unserialize function, used to transfer data objects between machines during system connection. The same interface is offered by each process and utility, providing both for automatic run-time load-balancing (both within and between machines) and for a 'pause & continue' execution model, whereby the system can be paused, stored on disk, and reloaded and continued at a later date or on another machine, with or without modification whilst it is paused. BRAHMS does not currently offer a real-time operational mode, though the interface allows for this expansion in future.

3.5. Modules

Process classes are implemented in modules, in one of the languages for which bindings are provided. A process module implements responses to calls from the supervisor on the events interface. A simple process module might respond to only two events, first to publish an output, and second to pass data into it on each time step. Data classes are implemented in similar modules, but receive a different set of events. A simple data module might respond to two events, to log its current state for later retrieval, and to return its log to the supervisor. In addition, it will offer an interface to its content for use by processes, as described above.

Note that a process module might be a front-end or interface for an existing computational engine, rather than an implementation of a new algorithm. Depending on the architecture of the engine, this may be straightforward. Illustrating that this is possible, we note that the above is an adequate description of the component language bindings for Matlab and Python, which are each implemented as a BRAHMS process. Similarly, a BRAHMS process has been developed that provides an interface with the Webots [24] robotic simulator. We do not anticipate difficulties in adding lightweight front-ends to many bespoke computational engines, though the practical problems in linking to particular engines remain to be discovered.

3.6. Supervisor

The BRAHMS supervisor is authored (in C++) as a standalone executable, so it does not require a virtual machine or scripting engine and is therefore resource-light. It is invoked with an 'Execution File', which contains instructions on how to execute the system (see Fig. 2). It then reads the 'System File' (the system represented in SystemML), instantiates the system (by loading modules and passing them their state from the system file), connects processes together (creates ports and links), then supervises the progression of the system through time, managing the transport of data through the links. At a predetermined stop time (or following cancellation by the user or by a process), it collates logs of each output port for which logs were requested in a 'Report File', and terminates the system. If required, the complete state of the system can be written back to a new system file for later reinstantiation, but the primary desired result of an execution will usually be the port logs in the Report File.

Automatic parallelisation is provided at a coarse-grained (process) level. Finer-grained parallelisation can be implemented within processes, but it will generally be much easier to break processes up and let BRAHMS parallelise them (models that consist of collections of similar objects, like network models, lend themselves particularly to this technique). One operational mode of the supervisor, 'Solo', offers lightweight parallelisation in a shared-memory environment using multithreading. Another, 'Concerto', offers multiprocessing for parallelisation over a computing cluster, currently using either TCP/IP or MPI as the communications layer. Note, that Concerto performs an identical operation to Solo, it just distributes it across multiple processes.

The data transport provided by the supervisor might comprise sharing a memory pointer (Solo), or routing a container over ethernet (Concerto), or through a stateful system file to another user, on another platform, at another time. Such transport operations are achieved without onus on the process developer. Background functionality beyond parallelisation includes inter-process data compression (Concerto), time-windowed port logging, distributed-system performance monitoring, and a planned pause & continue execution model (whereby system snapshots at non-zero time can be coherently stored, exchanged, examined, modified, or sent for further execution).

3.7. Connectivity

Connectivity in BRAHMS is according to the SystemML standard; we describe this, briefly, here. Each process expresses two interfaces, input and output. Each interface consists of one or more 'sets', which are semantic groups of input and output ports, allowing them to be treated in groups depending on their significance. Most processes do not need this semantic grouping, and express only the 'default' set on each interface. Links between processes are specified in the SystemML document, so the 'structure' of the input interface (number and name of inputs) is known when the process is first called. However, the nature of each individual output is unknown until it is created by the process, so system initialisation is an iterative procedure, with each process (in general) being called repeatedly, with more and more of its inputs available at each call. On each call, the process may create any number of outputs, which the framework will then route to any processes that take them as inputs. Thus, the presence or absence, number, class, transport specification, structure, content, and name of ports on a process output interface may all depend on any aspect of the process input interface or of the process state data. Whilst many existing processes do not express such dependencies (or only relatively trivial dependencies), this model allows complete flexibility where required.

3.8. Accountability

One of the dimensions of Varied Development, above, is time. This means that we should be able to integrate, today, process code that was generated many years ago. However, computing environments change, and researchers forget. BRAHMS offers accountability; that is, everything that bears on the results produced from an execution (details of each loaded library module, external library, of the run-time environment, platform, operating system, etc.) is recorded in an 'execution report' in the Report File. If a repeat of the execution does not produce identical results, it is possible to identify why (thus, accountability favours repeatability by identifying sources of disagreement). Note that this is dependent on the deterministic processing model used in BRAHMS, which provides 100% reproducibility given the same initialisation.

3.9. Software Development Kit

A BRAHMS release includes template processes, process development tutorials, and example processes authored in all four currently supported languages (C, C++, Matlab, Python). Creating a new BRAHMS process involves only copying the template for the chosen programming language, and adding the content that performs the actual algorithm intended. Also included is the BRAHMS 'Standard Library', a collection of processes implementing simple operations (such as sum, product and resample) which are intended to be useful in production systems, whilst doubling as rich illustrative material. The Standard Library also includes the data container class that will be most useful, 'data/numeric', which is a container for an N-dimensional array of numeric data in a comprehensive (and extensible) range of fixed-bit-width element formats.

The Matlab invocation bindings included with BRAHMS provide interfaces to the Execution, System, and Report files, and allow invocation of the BRAHMS executable, such that BRAHMS appears and behaves like a Matlab toolbox. Note that this does not imply that BRAHMS is tied in to Matlab in any way; it is just that bindings have only been authored for this environment (you *can* author these files in a text editor, but we do not recommend this approach). These bindings allow the easy construction of systems from processes and links, and the design of the generic StateML used by the processes in the Standard Library (termed 'DataML'). BRAHMS does not know about process state specifics, in general, so additional tools are needed to design StateML for processes that do not use DataML—however, DataML can represent multi-dimensional arrays, associative arrays, and ordered sets, so it is expected to prove adequate for a large majority of developments (all existing BRAHMS processes use DataML as their StateML). Invocation Bindings for other interactive environments are expected to follow in future (Python and Octave are likely to be targeted early). A future GUI system design tool would provide equivalent functionality, and could be seen as another 'binding'.

A 272-page documentation set is available online [14], including a complete Reference Manual, and an extensive and developing User Guide. The User Guide includes walkthroughs and tutorials for developing systems and processes, as well as discussion around the integration problem, material detailing BRAHMS internals, and considerable and accreting support documentation.

4. Related Projects

Many integration efforts have already been made (or are underway) that meet some of the above challenges in more constrained domains—just in Computational Neuroscience, for instance, NSL [26], CATACOMB [27], MOOSE [31], and NEST [32], to name but a few. Our concern, here, is to solve these challenges in such a way that integration can take place more generally; therefore, we do not discuss these more targeted solutions. Nonetheless, we note that such bespoke solutions might be imported into BRAHMS as processes in a straightforward way, so that BRAHMS provides integration across these solutions.

We do not have the space, here, for a comprehensive review of more general frameworks, though we note that we have been unable to identify any alternative that takes the same inclusive, general, approach of BRAHMS. Below, we very briefly contrast BRAHMS with three approaches to integration that we consider to be of potential interest in our particular area (computational neuroscience). We also draw the reader's attention to the other frameworks discussed in this issue.

4.1. Simulink

Simulink [12] has a long history, and is a useful tool for learning about integrated systems. More recently, it offers multi-language support (Matlab, C, C++, Ada, Fortran) but as yet no standard support for parallelisation even within a shared-memory space. The data format is open (though not extensible, since Simulink is proprietary). Involving at least a Matlab installation and instantiation, the resource requirement is substantial (though we currently have no data on run-time performance). In the long-term, support may improve in the technical areas where Simulink does not meet the requirements, but it is likely to remain costly, closed source and with a large resource footprint. In summary, whilst Simulink is a mature and well-developed product, it fails to meet several of the challenges we outlined above.

4.2. IKAROS

IKAROS [28] (also, this issue) is a project of similar spirit to the BRAHMS project, but has rather different focus. Its positive points include the 'WebUI', which allows real-time monitoring of system state through a browser, good documentation, and a simple developer interface. It also provides some facilities that are outside of the scope discussed here. However, where BRAHMS aims to meet all of the challenges listed above, IKAROS attempts to solve a more constrained problem, albeit in a straightforward and effective way.

IKAROS is constrained to a single inter-process data-type (2D single-precision matrices), does not support dynamic creation and structuring of outputs based on connectivity, has more constrained support for process state, and the plugin architecture requires the whole system to be rebuilt from source when new modules are added. This last is a particular problem for accountability, since the code that executed to generate archived results is generally no longer available. There is no discussion of bindings for other languages. The IKAROS interface is authored in C++, which might become a problem in the future if the project intended to move towards dynamic loading or accountability. IKAROS also lacks some framework functionality available in BRAHMS, though such features can presumably, as for BRAHMS, be added without modifying the plugin codebase. In summary, we understand that IKAROS and BRAHMS are solutions to different, if not quite orthogonal, problems.

4.3. Large-Scale Modeling Program and MUSIC

The International Neuroinformatics Coordinating Facility (INCF) have recently launched a program to foster infrastructure for researchers working with large-scale neural models. Attendees of the first program workshop [8] noted the large and growing set of neuron simulators available, and agreed on the importance of interoperability and component reuse. Focus was on modularity, particularly the supervisor-process interface, process-process interface, and common file format. Consequently, run-time inter-process communication was also discussed as a necessary future development to integrate computation engines into systems. They also highlighted background functionality (node allocation, communications initialisation) and marshaling of extremely large data sets.

All of these issues are addressed by our proposal. Other interoperability concerns raised in the workshop report are not applicable to BRAHMS; e.g. no application scripting is required since BRAHMS is responsible for procedure. Within the program, a communications library called MUSIC [29] is under development. The MUSIC approach leaves everything but inter-process communication to the process, in stark contrast to BRAHMS, which aims to provide as much common functionality as possible. Each approach has its advantages—in particular, large and/or closed-source efforts may regard their work as more suited to a MUSIC interface than a BRAHMS front-end (though the latter is generally not onerous to construct, as briefly discussed). In contrast, BRAHMS allows extremely rapid development of powerful cross-platform engines, which MUSIC does not. We do not consider that BRAHMS and MUSIC are competitors, and expect to offer a BRAHMS-MUSIC interface in the near future.

5. Case Studies

5.1. WhiskerBot

The WhiskerBot robot is an embodied model of rat behaviour with an eclectic control model including biomimetic, bio-inspired, and classical control components. The hardware [3, 33] and software [3, 34] architecture of WhiskerBot are covered in much greater detail elsewhere. Here, we summarise the platform and computational model, describe how BRAHMS was used to meet the integration needs of the project, and discuss what we learned about integrating biological modelling and robot development. The particular needs included multiple deployments (high-performance computing and low-performance real-time computing), processes with different timebases, hardware interfacing, and easy reconfigurability. This last is likely to be required in many mixed-discipline research projects, where different experiments may need to be run using related hardware/software tools; we discuss the advantages of reconfigurability further, below.

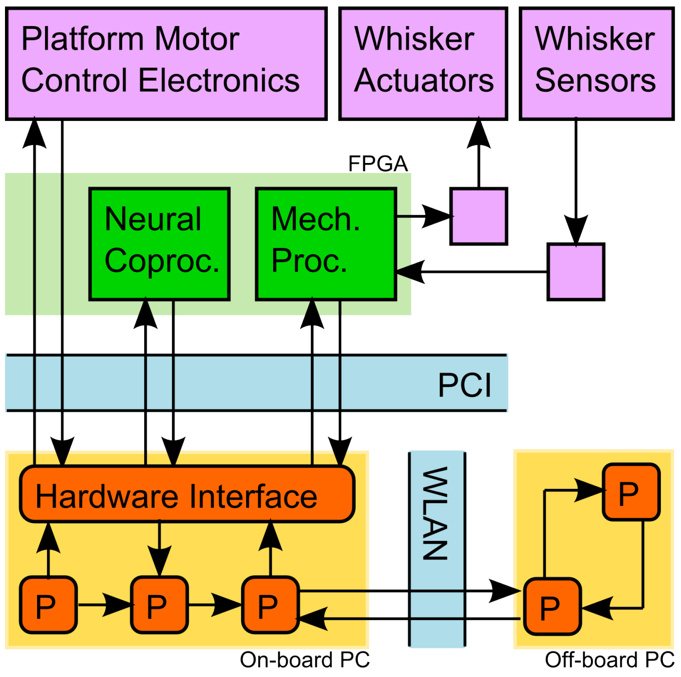

Figure 3: WhiskerBot/ScratchBot architecture summary. Hardware information-processing is above the PCI bus, with two processors implemented in FPGA (marked), and the remainder custom electronics and DSP. BRAHMS processes (rounded rectangles) are distributed on the on-board PC, with further processes optionally linked in on the off-board PC.

The hardware platform consists of a sensory head and an information-processing stack, mounted on a differential-drive robotic platform with associated control electronics along with a lead-acid battery and power switching equipment. The head is equipped with novel analog sensors emulating the capabilities of rat whiskers, each of which is driven to move in one plane with respect to the head by contraction of a shape metal alloy fiber, emulating muscular contraction. DSP-mediated pathways connect the head, upstream and downstream, with a Field-Programmable Gate Array (FPGA) bank (a programmable logic array). The FPGA interfaces, similarly, with the control electronics of the drive platform. On the upstream side, the FPGA attaches to the PCI bus of a single-board computer (Nallatech BenNuey PC104plus, Intel Celeron, 650MHz, FSB 133MHz, 512MB SDRAM). Finally, the PC is equipped with wireless/wired internet, connecting it at long latency to the laboratory LAN for off-board processing. That one aspect of this project was an investigation of eclectic hardware approaches to mobile information processing is reflected in this architecture.

In its onboard configuration, the computational model consists of the following. In hardware, a model of sensory signal generation in rat whisker sensors comprises a biomechanical model, which was laid on to the DSP, and a model of hundreds of primary afferent neurons (sensory cells), laid on to the FPGA. Also, a model of whisker motor pattern generation, implemented in hundreds of neurons, also laid on to the

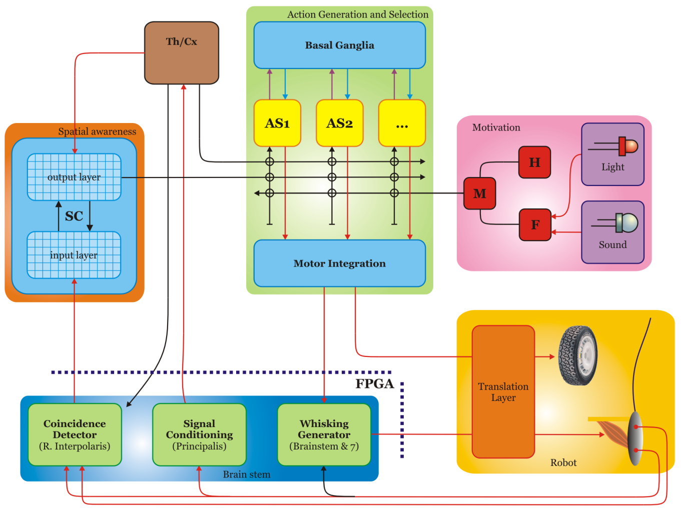

Figure 4: Summary diagram of the WhiskerBot computational model. Not all processes are shown, and the details are unimportant, but the figure illustrates the different types of processing being integrated. Clockwise from bottom-left: spiking neuron models of parts of brainstem (in FPGA), a part of the brain which is seen here as performing sensory signal conditioning; a rate-coded neural model of Superior Colliculus, a part of the brain concerned with spatial awareness; information-theoretic models of higher brain regions, including Thalamus and Cortex; arithmetic and neural model of action selection [16] in a part of the brain called Basal Ganglia; a sensory/arithmetic model of motivation; a geometric platform abstraction layer (modelling body, sensor and actuator mechanics).

FPGA. The FPGA exchanges data with the PCI bus of the PC at 2kHz. In the PC, fourteen BRAHMS processes are integrated to form the remainder of the control system, running variously at 200Hz or 2kHz. These include computational neuroscience models of various parts of the brain using various and distinct modelling approaches (see the caption of Figure 4 for a summary), a hardware-interface sub-system that uses the current environmental noise level to drive the animals motivation (loud noises scare it), and a platform abstraction layer that generates control signals for the mobile platform to implement the plans of the more abstract processing modules. Interfacing the software and hardware, an additional BRAHMS process encapsulates the upstream/downstream interface to the head and the FPGA neural coprocessor by interfacing with the PCI bus using the hardware manufacturer's API. Upstream and downstream translation processes stand between this hardware interface process and the computational processes, performing conversion between spike-timing and firing-rate encodings of neural signals, as well as upsampling and downsampling, as appropriate. The total data transfer rate in this configuration is between 1 and 2MiBs^{-1}, depending on data container content, in tens of separate inter-process links.

In its laboratory configuration, the model is similar, but the roles of the hardware modules are played by additional BRAHMS processes. The FPGA neural coprocessor and the two-part model of whisker sensory cells are replaced by software simulators, in place of the DSP/FPGA implementations. The sensory and motor equipment is simulated in a plane physics engine. This configuration allows substantial scaling up of all the computational processes, partly through deployment on much more powerful hardware (e.g. a processing cluster), but primarily because there is no longer a need to meet the real-time constraint of the send/receive cycle on the PCI bus. A third, supplementary, configuration, has the role of the FPGA neural coprocessor played by an emulator, which performs identically to the FPGA model at the binary bit level; this is used to confirm the relationship between the onboard and developmental configurations.

The three configurations described above serve different roles, and answer different research questions. The first, onboard, investigates the performance of the computational model embodied in a real-world sytem [16], but the experimental space is sorely limited by resource constraints. At the same time, it illustrates that BRAHMS can satisfy the real-time constraints of this system (5mS period PCI send/receive cycle) robustly. Note that the real-time constraint of this configuration is met through consistent iteration times, rather than through the provision of any real-time guarantee (as mentioned above, BRAHMS does not currently offer a real-time mode). The second configuration, in pure software, allows the free investigation of much larger-scale or higher-resolution models and the importance of scale, the latter being of particular interest when modelling neural systems since the amount of redundancy in neural processing systems is an open question. The third configuration allows easy in-depth investigation of the operation of the fixed-point hardware implementation in comparison with the original model, and thus offers confirmation that the model is rendered sensibly in the first configuration (debugging the operation of the hardware coprocessor *in situ* is, of course, laborious). In addition, it allows testing of scaling-up of the hardware model at low cost. Importantly, the BRAHMS approach to modelling and deployment allows all these configurations to exist alongside, without generating work for the developer—in all cases, the common modules use the same implementations, and are merely parameterised differently. This general result, of being able to maintain integration across widely different deployment cases, is a significant achievement that was not previously possible.

In addition to these 'complete robot' configurations, it is straightforward to wire up new arrangements of the system that target particular questions. For instance, a new biological result [35] recently indicated that rats use short-latency feedback to effect active control of the movements of their whiskers. We were able to quickly reconfigure the software architecture to implement the same strategy on the robot in a bench-test, performing stationary whisking against an obstacle, and observe the changes in the nature of the sensory signals collected with equivalent feedback in place. This sort of flexibility is a great advantage when engineering solutions in a cross-discipline context such as this, where the intellectual environment can change completely mid-project.

This approach to model development also makes it possible to make use of earlier effort in later work; that is, to integrate across time. WhiskerBot has now evolved into ScratchBot, an artefact of the ICEA project [4], and the computational model uses much of the same implementation as was used on the previous robot, allowing the development process to genuinely 'hit the ground running'. The mobile drive platform has a different structure, without nonholonomic constraints—this requires only updating the motor translation layer. The sensory hardware has been redesigned, based on lessons from the first edition, and now has eighteen whiskers driven by DC motors—this requires some updates to the hardware, and reprogramming the DSP, but otherwise only adds additional information, which the developer of the upper computational model can integrate into higher modules at leisure. Also, additional processing modules are being added to the upper computational model (hippocampal and more realistic cortical models)—this requires only implementing the new modules and wiring them in (modifying the script that stitches the processes together into a system). Since some of the new models are too computationally intensive to run on the onboard PC, we are moving towards using the WLAN/LAN connectivity of the robot to spread the BRAHMS implementation across the onboard and an offboard, more powerful, machine, using Concerto. This seamless expansion would not have been possible had we been working with a monolithic system, or with another, less flexible, framework.

In computational neuroscience, a model generally proves itself in the long term by performing comparably in different environments, rather than being tailored to a particular test case. A monolithic implementation of a robot control system such as that described, could not be tested in this way. However, the individual modules developed to implement biological or bio-inspired solutions in this project certainly can; either individual modules, or groups of modules, can be tested and/or examined by other researchers, with ease. Alternatively, they can be adopted for use as components of other projects, whether to further research into their own operation, or to play supporting roles in unrelated investigations. Parts of the WhiskerBot model have, for example, been seconded for use in an unrelated oculomotor model, described below. Both reproduction of results and adoption of components are supported just by the publication of the SystemML document describing the model. Module implementations can be obtained from the SystemML server (not yet commissioned), and the SystemML system described in the document can be broken apart by the user into valid component parts, if only parts of the system are to be reused.

5.2. Other Use Cases

Apart from the Scratchbot artefact, the ICEA project [4] includes several other lines of research. One, for instance, is a model of a rat performing a maze-navigation task. The task is to learn the spatial layout of the maze, to associate reward with combinations of location and stimulus (lights are lit in maze arms by the experimenter), and to make decisions based on this information as to which way to locomote. The spatial layout is learned in a model of Hippocampus, which is posited to maintain maps of the world. Reward is processed in models of Amygdala (dealing with reward itself) and Cortex (dealing with association and reward prediction). Decisions are made in Basal Ganglia [16]. Finally, the model is expressed in a simulated world, containing a simulated rat and maze, implemented in Webots [24].

BRAHMS is used to integrate the individual model components. In the latest approach to this task, a Hippocampal model, developed in Hungary in Python, is integrated with the Basal Ganglia model, developed in the UK in Matlab, with the Webots simulator (developed commercially, in C++), as well as with BRAHMS Standard Library components. The Cortex and Amygdala models remain, as yet, undeveloped, but the developer can be confident that their integration will be straightforward whatever form they take, as described previously. In this way, the modular approach breaks time dependencies as well as software dependencies.

Some implementations from the WhiskerBot model are currently in use as part of the large-scale oculomotor control model of the REVERB project [23]. This model deploys across a compute cluster and a separate, differently configured, robot-control machine. The robot itself is a model of the vertebrate eye, with 2 degrees of actuated freedom of look direction, and a camera as sensor; interaction with the robot consists of sending control signals to the actuators and collecting data from the camera. The design has the members of the cluster communicating over the Myrinet installed on the cluster, whilst the robot-control machine is linked in over the LAN; this implementation is not yet complete, but in the meantime the same system is being investigated in different configurations (on the developer's desktop, and on the cluster without using the Myrinet). Flexibility in deployment options is allowing this work to run at full-speed despite the deployment environment not yet being ready. Parallel computation is allowing the developer to rapidly obtain feedback on modifications to the model parameters, yet this model is not 'targeted' at parallel computation; rather, it can be deployed in parallel, when appropriate.

As the oculomotor project develops, the control model will choose the actions taken by a simulated mobile platform for the eye robot; since the robot itself cannot move, the world will be moved around it by rendering onto a hemispherical shell that surrounds the eye robot. This will require the integration of a bank of rendering servers, and the complete system will be required to integrate on a consistent time interval to keep up with the camera and projector refresh rates, as well as to provide timely control signals for the robot actuators. With much of the hard work of these integration problems performed by BRAHMS, the researchers are, rather, able to concentrate on how their models behave. With consistent inter-model interfaces, it is assured that the models they are working with now will fit together straightforwardly with the software that is developed to solve the rendering task.

Future uses for BRAHMS in our own group are lining up fast, as expected. A new, detailed, model of Striatum (part of the action selection system used in WhiskerBot) will use BRAHMS to integrate across distinct cell model populations with different complexities and different sample rates, and to parallelise the execution of the model. Another application will be to investigate cortical models developed in the Topographica modeling package with different control strategies for whiskers interacting with physical objects. BRAHMS will provide the software to stitch together the physical whisker sensors, the whisker actuators, and the Topographica models, in a closed control loop. BRAHMS has also been chosen as the integration platform for the large European project BIOTACT [25]. Taken together, these use cases illustrate reuse, various dimensions of integration, and varied substrates of deployment.

6. Performance

To measure overhead and scaling, we designed the following model. We compute a simple (and meaningless) operation across (double-precision scalar) elements on a rectangular grid of size P by E elements. Each of the P groups of E elements is assigned to a single process. Each process computes O elemental operations per element on each iteration (the elemental operation consists of a floating-point addition and division—care was taken in design to avoid floating-point exceptions). Thus, we can scale the number of processes (P), the working set size (PE), and the computational complexity of a process iteration (EO) independently. The result of each of the P processes is passed to the next process, forming a loop, such that P inter-process links are also performed.

We implemented the model twice—once in BRAHMS, and once as a monolithic executable (both in C++, with C-style inner-loop optimisations). The BRAHMS implementation was run multi-threaded (MT, one thread per process up to the default maximum of eight per core) and single-threaded (ST), for a total, along with the monolithic case (M), of three implementations. We used the same object code for the inner loop (O, E) in each case, and briefly reviewed the generated assembler for surprises (Windows only). We did not attempt to optimise the outer loops (P, N) in the monolithic implementation, but we are happy that the (very simple) implementation was not obviously wanting, and we note that ST and M asymptote to the same execution time as complexity increases.

For each of the two benchmarks described below, at each parameter point, we computed a large number of iterations (N) and measured the execution time of the slowest worker thread, T_b, using system-supplied performance counters. We repeated the whole of each benchmark several times (R), and took the minimum T_b across these repetitions at each parameter point, to filter out any noise arising from other demands on the test platforms. We define the effective elemental operation time, $T_e = T_b/PEON = T_c + T_o$, with T_c the time to compute the elemental operation and T_o the implementation-specific per-operation overhead. Thus, T_e is expected to asymptote to T_c (which is unknown) as process iteration complexity (EO) increases. T_c itself is, in general, dependent on the parameters, as the changing working-set size moves the effective workspace between CPU caches and main memory; here, we kept working-set size small enough so that T_c was constant throughout. We obtained a good estimate of T_c, then, based on the longest execution of the monolithic implementation, and used this as a baseline against which to compare the execution times returned by BRAHMS. Execution times returned by the monolithic implementation were consistently in line with expectations based on this estimate of T_c (i.e. no overhead and no multi-core speed-up). T_e/T_c, finally, is a measure of the overhead that is the cost of using an implementation, with unity indicating no cost. We also define the monolithic process iteration time, $T_p = T_e \times EO$, which is the time spent overall on each iteration of each process using the monolithic implementation.

The three implementations (MT, ST, M) were built and run on two software platforms: (A) Windows XP 32-bit SP2 and (B) Ubuntu 32-bit 8.10. Both were built and run on the same hardware platform: Intel Core2 Quad Q6600 2.40GHz 4GB PC2-6400. Performance was similar on the two platforms (though not identical, see Figures). Performance on several other hardware and software platforms (not reported) displayed no inconsistencies.

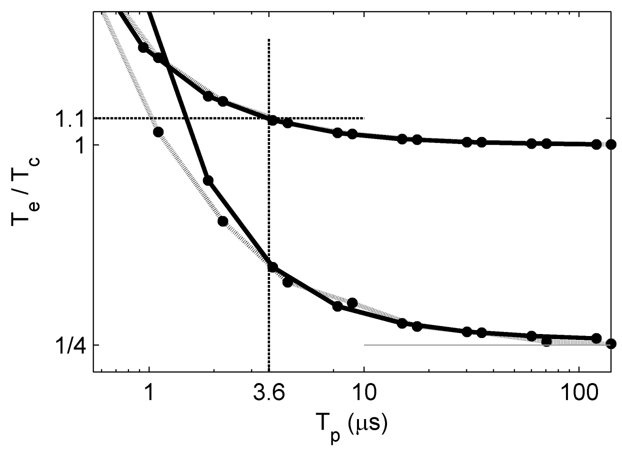

Figure 5: Overhead incurred by using BRAHMS against monolithic process iteration time (solid/dotted lines, platform A/B; ST and MT are both shown, MT is the lower pair). Overhead decreases as process iteration time increases, falling below 10% (ST) at an iteration time of around 3.6μs (marked with dashed lines). Fine solid line to right indicates ideal speed-up for 4 cores as overhead becomes negligible.

The first benchmark measured the overhead of one process and one link as the process iteration time, T_p, was varied. The parameter set had $P = 16$, $E = 32$, $N = 1000$ and O varying between 1 and 256 (for a working-set of a few kiB). The small working set size is necessary because with larger working sets, it is not possible to achieve a low enough T_p to detect the overhead. The results in Figure 5 indicate that the ST overhead becomes small (10%) with T_p around 3.6μs, suggesting an overhead of around 360ns per iteration for the process and link together. In the MT case, the overhead is larger, due to the cost of inter-thread synchronisation. However, very good use of the hardware is already achieved with process iteration times as low as 10μs. Platform A (Windows) appears to incur a slightly higher cost for inter-thread synchronisation, but the difference is already negligible at $T_p = 3.6\mu$s.

The second benchmark aimed to confirm that BRAHMS performance scales well. To this end, we chose a parameter point with a small per-process memory requirement ($E = 256$, $O = 256$, $N = 25$) and varied the process (and, therefore, link) count, P, between 1 and 1024. This gives a working-set size that varies between a kilobyte and a megabyte (overheads due to the framework not included). Note also that the thread count follows P to begin with, but is capped at 32 (see above). The

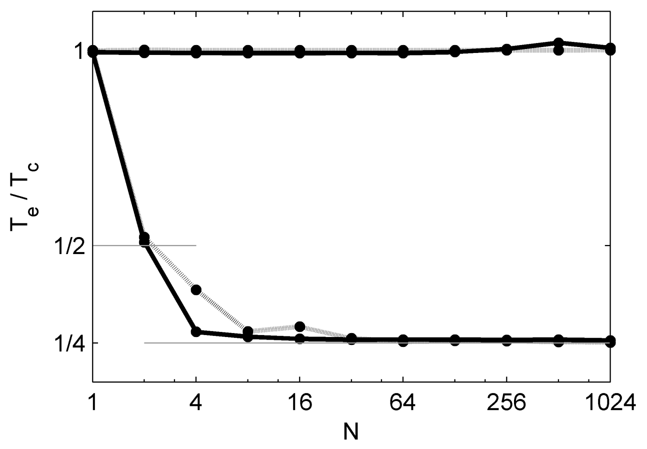

Figure 6: Overhead incurred by using BRAHMS against number of processes/links in the computed system (solid/dotted lines, platform A/B; ST and MT are both shown, MT is the lower pair). System scale has no discernible impact on performance. Fine solid lines to left indicates ideal speed-up for 2/4 threads (on 4 core hardware).

results of Figure 6 show that increasing P has no discernible impact on ST or MT performance on either platform.

Both above benchmarks were computed using `brahms_bench` which is supplied with BRAHMS 0.7.2 (the data published here were obtained using a pre-release version). The first benchmark is `overheadi`, the second is `scaling`. Direct measurements of T_o on each platform (known as `overhead` in `brahms_bench`) gave figures of around 230ns for a process and 130ns for a link (ST); the sum of these to 360ns is entirely consistent with the figure returned by the indirect overhead benchmark detailed above.

7. Example

To illustrate explicitly, we provide a complete example system, here. The example we choose is the classic Lotka-Volterra predator-prey model (sometimes known as ‘rabbits and foxes’), which is one of the tutorials that ships with BRAHMS (though in the shipped tutorial it is implemented in m-script; here, we implement it in C++). This system is not a good candidate for modularisation, since the computation of each module consists of one or two floating-point operations per process iteration, but it exemplifies the major features (taking state, taking inputs, providing outputs, linking a system).

7.1. Example Process

The ‘rabbits’ process was based on the ‘1199’ (the most recent C++ binding) template, with the changes detailed in Figure 7. We do not show the ‘foxes’ process, but it is similar.

The code begins with the class declaration, which declares the single function called by the framework and the state data used by the process, as well as the input and output ports. The remainder of the code is the implementation of the function, which handles (in this case) four events.

EVENT_STATE_SET is called once, asking the process to initialise its state (unserialize). In other words, to accept parameters. EVENT_INIT_CONNECT may be called multiple times, with more inputs available on each call. The details of handling this event vary with the nature of the process. This process creates its output on the first call, and validates its input on the last – this procedure suffices for many types of process. EVENT_INIT_POSTCONNECT is serviced only because timing data has been finalized at this point, and the process needs to calculate its sample period. EVENT_RUN_SERVICE is called multiple times during execution, with the clock set to the instant at which the framework requires the input and output interfaces to be serviced. Our process accepts only accepts regular ticks (because it sets F_NOT_RATE_CHANGER), so it can safely ignore the clock and simply step its dynamics on each call to this event. Reading input and writing output are usually exactly as simple as shown here.

7.2. Example System

The 'rabbits and foxes' system script creates an empty system, adds the two processes ('rabbits' and 'foxes'), links them together in a loop, then sends the system for execution (Figure 8). This script mostly uses the SystemML Matlab Bindings (sml_system), which are provided as part of SystemML (though they are currently shipped with BRAHMS, for convenience) to construct the system. It goes on to use the BRAHMS '995' (Matlab) Invocation Bindings (brahms_execution), to create, modify, and invoke an execution of that system. The output of the script is given in Figure 9.

7.3. Multi-processing

This system is already deployed in multiple (two) threads (9μs per iteration). However, since the computation in each thread is so tiny, it actually runs rather faster (1μs per iteration) if we disable multi-threading, since the overhead due to inter-thread signalling is avoided. Nonetheless, we can run in Concerto mode (multi-processing) by adding a single line to the system script telling the execution to launch two BRAHMS 'voices' on the local system using the sockets layer to communicate.

Separate multi-processing instances are called 'voices' to distinguish them from the 'processes' that are components of a system; running multiple voices on a single machine is a testing configuration, referred to as a 'babble'. The iteration time returned by this two-voice babble is around 60μs, showing up the additional overhead of serializing each data packet into and out of the sockets layer before passing it to the destination process. The results returned are, of course, identical. The execution runs for 1000 samples (10 seconds at 100Hz), so run phase completes in around 60ms.

Depending on the configuration of the platform, launching a Concerto execution on multiple nodes may be achieved in exactly the same way (one extra line in the system script) if that line specifies the IP addresses of the target machines. In another case, perhaps where the deployment hardware is asymmetric, it is straightforward to request through the bindings that a specific

```
#define COMPONENT_FLAGS (F_NOT_RATE_CHANGER)

...

class Rabbits : public Process {
  Symbol event(Event* event);

  DOUBLE a, b, r, T;
  numeric::Input input;
  numeric::Output output;
};


Symbol Rabbits::event(Event* event) {
  switch(event->type) {
    case EVENT_STATE_SET: {
      ...

      a = nodeState.getField("a").getDOUBLE();
      b = nodeState.getField("b").getDOUBLE();
      r = nodeState.getField("r").getDOUBLE();
      return C_OK;
    }

    case EVENT_INIT_CONNECT: {
      if (event->flags & F_FIRST_CALL) {
        output.create(hComponent);
        output.setName("out");
        output.setStructure(TYPE_REAL | TYPE_DOUBLE,
          Dims(1).cdims());
      }

      if (event->flags & F_LAST_CALL) {
        input.attach(hComponent, iif.getPort("in"));
        input.validateStructure(TYPE_REAL | TYPE_DOUBLE,
          Dims(1).cdims());
      }

      return C_OK;
    }

    case EVENT_INIT_POSTCONNECT: {
      T = sampleRateToPeriod(time->sampleRate);
      return C_OK;
    }

    case EVENT_RUN_SERVICE: {
      DOUBLE f = *((DOUBLE*) input.getContent());
      r += T * (a * r - b * r * f);
      output.setContent(&r);
      return C_OK;
    }
  }

  return S_NULL;
}
```

Figure 7: C++ source code for the 'rabbits' process. Some boilerplate code from the template is elided (marked '...').

```
% create empty system
sys = sml_system;

% add process
state = struct('a', a, 'b', b, 'r', r);
sys = sys.addprocess('rabbits', ...
  'dev/abrg/rabbits', fS, state);

% add process
state = struct('c', c, 'f', f);
sys = sys.addprocess('foxes', ...
  'dev/abrg/foxes', fS, state);

% link them
sys = sys.link('rabbits>out', 'foxes<in');
sys = sys.link('foxes>out', 'rabbits<in');

% execute
exe = brahms_execution;
exe.stop = 10; % run 10 secs
exe.all = true; % log everything
out = brahms(sys, exe); % execute

% plot
plot([out.rabbits.out; out.foxes.out]')
```

Figure 8: Complete M (Matlab) script for the 'rabbits and foxes' system. This script produces the plot in Figure 9.

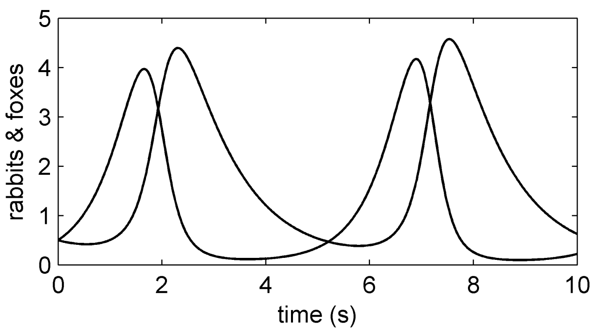

Figure 9: Output from the script of Figure 8, for the following parameters: $a = 2$, $b = 1.2$, $c = 0.9$, $r = 0.5$, $f = 0.5$, $fS = 100$.

```
% execute
exe = brahms_execution;
exe.addresses = {'192.168.1.10' '192.168.1.11'};
exe.launch = 'each ssh $(ADDR) brahms $(EXECFILE)
  --voice-$(VOICE) $(ARGS) $(LOG)';
...
```

Figure 10: Additions to the system script for a typical multi-processing invocation. The token 'each' indicates that the line should be run once for each voice. Multi-processing systems that handle the launching of multiple instances themselves (such as MPI), thus, do not need this prefix.

command be issued to start each voice. If desired, the execution can, of course, be performed without using the bindings (i.e. there is no need to have Matlab deployed where BRAHMS is running). Figure 10 shows the two lines that would typically be added to the system script to launch it on a typical processing cluster using `ssh`.

8. Status

BRAHMS has been public since April 2007; version 0.7.1, the latest bugfix release on the stable 0.7 branch, was released 5 November 2008. Funding for maintenance of this branch has already been secured until the end of the BIOTACT project at the end of 2011; we intend to apply for further funding in the meantime to support the development of planned features. Processes authored against the 0.7 branch will interoperate with future releases. At time of writing, a release candidate of version 0.7.2 is undergoing testing.

Solo is now fairly mature, having been performing well in a variety of environments for around two years, with only minor changes. Concerto is less mature and has beta status, but is considered sufficiently stable for deployment in the REVERB project. All releases are available for Windows 32-bit, GNU/Linux 32-bit (built on Ubuntu) and GNU/Linux 64-bit (built on Debian). We anticipate offering builds for other platforms in the near future.

Note that aperiodic links, pause & continue execution, real-time mode, execution reports, and the SystemML public namespace are items that have been discussed, but that are not yet implemented or fully implemented.

9. Conclusions And Future Works

9.1. Meeting The Challenges

BRAHMS solves the primary challenge of integration across Varied Development by specifying a common, flexible interface in multiple programming languages, against which new computational engines can be developed, and onto which existing computational engines can be imported. It meets the challenge of Varied Deployment through implementation as a lightweight standalone native executable and by allowing processes to be developed in similarly lightweight native code (if required for deployment on very limited systems, the supervisor can be built without some facets, e.g. the GUI and multiprocessing support). BRAHMS offers extensive (and accreting) background functionality in the supervisor, meeting one aspect of the challenge of Code Sharing (a BRAHMS 'hello world' process requires only one line of code inserted into the template, yet can distribute its complex processing across massively parallel resources).

BRAHMS supports the pragmatic challenges of sharing process code (by allowing the distribution of pre-compiled binaries rather than source code and by providing accountability) and goes some way to fostering documentation by defining and documenting a public process interface (development against a known interface self-documents to some extent, since a naïve reader knows at least some aspects of what a piece of code is intended to do). However, it does not directly address the challenge of sharing process code—see Section 9.2 for details of how this will be addressed by future developments. BRAHMS employs Open Standards throughout.

Adoption of the suite is well underway: the success of BRAHMS as the integration framework for the WhiskerBot project has led to its being chosen to play this role in three other well-funded research projects, involving varied use cases (5), and users have reported finding the workflow agreeable.

New processes benefit from being built into the BRAHMS framework by taking advantage of services provided by the system, and are constrained in their operation only by the requirements of the supervisor-process interface (i.e. a process is free to interact with the operating system, with hardware, with the user, as required). Integrating existing processes into BRAHMS can be achieved either by wrapping the existing processing engine in a lightweight BRAHMS process (contingent on cooperation and/or access to source code) or by meeting the communications interface or API of the existing software. Some discussed features are, as noted in Section 8, incomplete; however, users can begin using BRAHMS immediately and take advantage of feature additions as they become available.

BRAHMS has been shown to perform comparably with a monolithic implementation in terms of execution time, so long as the iteration time of the individual process is large enough. In addition, we have demonstrated that performance scales comfortably to over a thousand processes, running either sequentially or in separate threads, and we know of no reason why problems should occur with still larger counts.

The multi-configuration approach adopted for the WhiskerBot project is made possible by the use of BRAHMS as the integration tool. The first configuration is an example of an autonomous robot using biomimetic control techniques to solve practical problems (navigation, object localisation and discrimination, motivation, action selection) in a realistic environment. At the end of the WhiskerBot project, this robot was equipped to follow navigational plans, explore its environment, and locate objects, albeit in a fairly rudimentary way. However, it is also an example of the embodied modelling approach. The platform continues to develop (in the context of ICEA), using feedback from the performance of the embodied model to inform the development of the biological models using the laboratory configuration, which in turn is feeding back to generate more effective and more comprehensive biomimetic control strategies, as well as informing the evolution of the design of the robotic platform. We believe that this integrated development approach, where models are cycled through constrained and unconstrained environments, tested against biological data as well as against real-world control problems, will continue to prove its worth as a route to developing tomorrow's robotic controllers. We consider that BRAHMS is the only integration framework currently available that lends itself to supporting this cross-discipline approach to development.

9.2. Future Work

Above, we mentioned the SystemML file format, which is used by BRAHMS to represent systems. This open file format is a point of interface between BRAHMS and other tools. Beyond that, in future, SystemML will also offer an infrastructure for the publication of processes represented in such systems. Per-process data in the infrastructure will include specification of parameterisation and of input/output interfaces, as well as of the algorithm itself. The infrastructure will provide archiving, distribution, version control and automatic patching (without breaking accountability) of published processes. The interplay of this infrastructure with accountability will ease the identification and removal of software bugs, whilst guaranteeing backwards compatibility. This infrastructure will allow BRAHMS to meet the final challenge identified above, that of effective sharing of process code. The formal algorithm/implementation publication mechanism is also expected to contribute to the reduction of the practical problem we term the 'Tale of Two Models', that of algorithmic details becoming lost irrecoverably in undocumented optimised code.

Asked to execute a SystemML document that contains published processes that are not available locally, BRAHMS will be able to obtain implementations of the specified processes from a SystemML server and, thus, execute the model without manual intervention. Thus, processes published to the infrastructure will be immediately available to co-workers (the infrastructure is not intended to distribute models—other tools already exist to serve this end). In addition, where expert authors choose to make available processes of general interest, these can be used in any model.

We are committed to completing the maturation of Concerto, of the BRAHMS Standard Library and of the existing process language bindings. In addition, the SystemML infrastructure will be put in place in the coming year to support publications made by projects that have already adopted the suite. Contingent on funding, we intend to develop process bindings for other languages (e.g. Java and Octave), as well as invocation bindings to allow the use of BRAHMS from other interactive environments (e.g. Python and Octave).

One other likely development for the near future is a GUI system designer (operating within the SystemML space entirely, this is strictly-speaking independent of BRAHMS). Incorporating the BRAHMS engine into this software will create a design-and-execute environment similar to that provided by, for example, Simulink [12]. Developments that are under discussion for future development are the inclusion of services providing similar functionality to the IKAROS WebUI, and the addition of support for real-time processing. As the core technologies stabilise, we hope that BRAHMS will become increasingly community-driven; as a highly modular project, it is well suited to distributed development.

10. Acknowledgments

The authors gratefully acknowledge the patience and contributions of the other members of the Adaptive Behaviour Research Group and of the ICEA project in software testing, in particular Martin Pearson, the main end user during early development, and Alex Cope, the main end user during development of Concerto. This work was partially supported by European 6th Framework Grant IST 027819 ICEA and EPSRC Research Grant EP/C516303/1.

References

[1] P. F. Dominey and M. A. Arbib, “A cortico-subcortical model for generation of spatially accurate sequential saccades”, *Cereb Cortex*, 2:153-175, 1992.

[2] J. W. Brown, D. Bullock and S. Grossberg, “How laminar frontal cortex and basal ganglia circuits interact to control planned and reactive saccades”, *Neural Networks*, 17:471-510, 2004.

[3] M. J. Pearson, A. G. Pipe, C. Melhuish, B. Mitchinson and T. J. Prescott, “Whiskerbot: A Robotic Active Touch System Modeled on the Rat Whisker Sensory System”, *Adaptive Behavior*, 15:223-240, 2007.

[4] *ICEA*, European Union Framework 6 IST-027819, http://www.iceaproject.eu.

[5] J. M. Chambers, *Deciding where to look: A study of action selection in the oculomotor system*, PhD Thesis, The University Of Sheffield, 2007.

[6] J. G. Fleischer, J. A. Gally, G. M. Edelman and J. L. Krichmar, “Retrospective and prospective responses arising in a modeled hippocampus during maze navigation by a brain-based device”, *Proc Natl Acad Sci U S A*, 104:3556-3561, 2007.

[7] B. Girard, D. Filliat, J. Meyer, A. Berthoz and A. Guillot, “Integration of Navigation and Action Selection Functionalities in a Computational Model of Cortico-Basal-Ganglia-Thalamo-Cortical Loops”, *Adaptive Behaviour*, 13(2):115-130, 2005.

[8] M. Djurfeldt and A. Lansner, *Proceedings of 1st INCF Workshop on Large-scale Modeling of the Nervous System*, Stockholm, Sweden, 2006, *in* Nature Precedings, doi: 10.1038/npre.2007.262.1

[9] D. C. Dennett, “Why not the whole iguana?”, *Behavioral and Brain Sciences*, 1:103–104, 1978.

[10] M. O. Franz, H. A. Mallot, “Biomimetic Robot Navigation”, *Robotics and Autonomous Systems*, 30:133–153, 2000.

[11] D. L. Parnas, “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, 15(12):1053-1058, 1972.

[12] Mathworks, *Matlab & Simulink*, http://www.mathworks.com.

[13] K. Gurney, T. J. Prescott, J. R. Wickens and P. Redgrave, “Computational models of the basal ganglia: from robots to membranes”, *Trends in Neuroscience*, 27(8):453-459, 2004, doi: 10.1016/j.tins.2004.06.003

[14] B. Mitchinson and T.-S. Chan, *BRAHMS*, http://brahms.sourceforge.net.

[15] M. Djurfeldt, Ö. Ekeberg and A. Lansner, “Large-scale modeling – a tool for conquering the complexity of the brain”, *Frontiers in Neuroinformatics*, 2:1, 2008, doi: 10.3389/neuro.11.001.2008

[16] T. J. Prescott, F. M. Montes González, K. Gurney, M. D. Humphries and P. Redgrave, “A robot model of the basal ganglia: Behavior and intrinsic processing”, *Neural Networks*, 19(1):31-61, 2006.

[17] B. Webb, *Biorobotics*, AAAI Press, 2001.

[18] R. Chavarriaga, T. Strösslin, D. Sheynikhovich and W. Gerstner, “A Computational model of parallel navigation systems in rodents”, *Neuroinformatics*, 3(3):223-242, 2005.

[19] D. L. Parnas, “Software Aging”, *in 16th international conference on Software engineering*, Sorrento, Italy, 2004, 279-287.

[20] B. Mitchinson and J. Chambers, *SystemML*, http://sourceforge.net/projects/systemml.

[21] J. Bloch, *How to Design a Good API and Why it Matters*, Javapolis, Antwerp, Belgium, December 12–16, 2005.

[22] Le Hors A. *et al.* (ed.), *W3C Document Object Model Core*, http://www.w3.org/TR/DOM-Level-3-Core/core.html.

[23] *REVERB*, EPSRC Research Grant EP/C516303/1, http://reverb.abrg.group.shef.ac.uk.

[24] O. Michel, “Webots^{TM}: Professional Mobile Robot Simulation”, *International Journal of Advanced Robotic Systems*, 1(1):39-42, 2004.

[25] *BIOTACT*, European Union Framework 7 ICT-215910, http://www.biotact.org.

[26] A. Weitzenfeld, M. A. Arbib and A. Alexander, *The Neural Simulation Language: A System for Brain Modeling*, MIT Press, 2002.

[27] F. Howell, R. Cannon, N. Goddard, H. Bringmann, P. Rogister and H. Cornelis, “Linking computational neuroscience simulation tools : a pragmatic approach to component-based development”, *Neurocomputing*, 52-54:289-294, 2003.

[28] C. Balkenius *et al.*, *IKAROS*, http://www.ikaros-project.org.

[29] Ö. Ekeberg, M. Djurfeldt, *MUSIC: Multi-Simulation Coordinator, Request For Comments*, http://www.incf.org, 2008.

[30] R. Mall, *Fundamentals of Software Engineering*, Prentice-Hall, 2004.

[31] U. S. Bhalla *et al.*, *Multi-scale Object-oriented Simulation Evironment*, http://moose.sourceforge.net.

[32] M-O. Gewaltig, M. Diesmann *NEST (Neural Simulation Tool)*, Scholarpedia 2(4):1430, 2007.

[33] M. J. Pearson, A. G. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche, “Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach”, *IEEE Transactions on Neural Networks*, 18(5):1472-1487, 2007.

[34] B. Mitchinson, M. Pearson, C. Melhuish, T. J. Prescott, “A Model of Sensorimotor Coordination in the Rat Whisker System”, *From Animals to Animats 9: Proceedings of the Ninth International Conference on Simulation of Adaptive Behaviour*, Lecture Notes in Computer Science 4095:77-88, Springer-Verlag: Berlin, 2007.

[35] B. Mitchinson, C. J. Martin, R. A. Grant, T. J. Prescott, “Feedback control in active sensing: rat exploratory whisking is modulated by environmental contact”, *Proceedings of the Royal Society B*, 274(1613):1035-1041, 2007.