

Development, management, and deployment of algorithms for synthetic biology

Anne Van Devender¹, Douglas Densmore²

¹ Washington and Lee University

² University of California, Berkeley

ABSTRACT

Synthetic biology is at the point where we can assemble complex systems from standard parts. This assembly needs to be done in an efficient way. There exists a number of algorithms which can aid in this process. We need a way to get these algorithms to the user easily. These algorithms also need to talk to databases and talk to DNA sequence manipulation software containing information on the basic parts. The paper will discuss how this integration is done in the Clotho infrastructure and discuss the success of algorithms implemented in this framework.

1. INTRODUCTION

Synthetic Biology is an emerging field that is centered around building devices from biological parts for both useful application and biological understanding. Although based in biology, the field is very much a *design science* that resembles aspects of engineering in the way systems are methodically constructed, yet retains biology's empirical aspects. As in the early days of electrical engineering, systems are becoming more complex to the point where creating designs by hand is unmanageable. Synthetic biologists need ways to automate this process to both save time and ensure reliable system design. These design environments not only need to be reliable, but also they should be unified.

This task is further complicated by the lack of standardization and unification within synthetic biology. There are no real standards within synthetic biology for building individual devices making the creation of large systems in the field difficult. The beneficial effect of standardization is that it would significantly increase economic productivity in the field thus permitting higher levels of activity and complexity in system design [10]. It would decrease repetition and increase system information. In this paper we will not attempt to address implementation of individual standards for tools in synthetic biology, but instead we will discuss ways of solving this problem through flexible tool design.

Without a standard to follow tools must be modular to accommodate not only the current variety of practices, but also future

change. With our tool, Clotho, we provide a modular system that allows for unification by using both object-oriented programming and platform-based design. The modularity of the system is designed using a core component that manages communication between all pieces of the tool. This core allows pieces to be added to the tool with little knowledge about the tool itself. Unification is a result of this modularity as pieces can be added easily allowing for not only rapid growth, but also ensuring communication is maintained between all pieces.

The focus of this paper will be automating design and increasing connectivity through the implementation of algorithms within a larger software tool using the Clotho Algorithm Manager. We will demonstrate how we use object-oriented programming in combination with ClothoConnections to achieve an environment that is both modular and highly usable.

1.1 Approach

Synthetic biology is a relatively new field, and the tools developed within the field should focus on what currently exists within the field rather than attempt to extend beyond its practicality. Many tools available are far more advanced than is applicable in synthetic biology as they depend on quantitative part information and interfaces that are usually not available and often make unrealistic assumptions about the part connections [10]. This renders these tools useless for practical applications.

In order to create tools for practical application, a bottom-up approach to design is necessary. Bottom-up design in this context refers to the idea of designing a core set of tools that begin with parts in existence to build larger systems. This approach opposes the top-down idea of conceptualizing a large-scale system and building parts to meet that use. The bottom-up approach is currently more appropriate for synthetic biology as it is able to grow, be added upon, as synthetic biology grows.

Our tool Clotho is designed using this bottom-up approach. It contains a core set of tools known as ClothoConnections that support a variety of functionality to provide for undemanding integration of new parts. These tools are the main source of this modularity within Clotho as they contain all of the major functions and allow a new connection to be added in only a few lines of code.

The focus for this paper will be a specific ClothoConnection, the ClothoAlgorithmManagerConnection.

1.2 Organization

Section 2 will discuss the related work in this area. Section 3 will describe Clotho's system architecture. Section 4 will describe how algorithms are integrated in the the Clotho design environment. Section 5 will describe the results of testing for both the

Algorithm Manager and the algorithms themselves. Section 6 will give our conclusion. Section 7 will describe our future work regarding additional algorithms and extension of database connectivity.

2. RELATED WORK

This section will discuss other design environment for synthetic biology.

BioJADE is one of the earliest design environments created for synthetic biology. BioJADE allows users to specify a system abstractly, producing a view similar to electrical circuits, alter that system, and simulate its behavior [6][5]. Upon completion of a design the user is able to package the part and store it in a database. The benefits of this tool are its database connectivity and simulation elements, but it presents a high-level view of systems as it allows construction of parts but prevents the user from altering any information associated with a specific part such as sequence data[10].

APE(A Plasmid Editor) allows users to interact with a DNA sequence including editing, highlighting restriction sites, and translation [4]. Unlike BioJade, this tool is highly practical, but lacks comprehensiveness as its functions are restricting to sequence manipulation. It has no database connectivity, nor does it have a concept of a biological part or system.

BioStudio is a new design environment as it is still currently being developed. Its focus is not gene design, but rather genome design at the physical level (as opposed to BioJade's logic level). BioStudio is a set of scripts for manipulation, which uses GBrowse, an open-source as its viewing editor (Sarah Richardson, personal communication). BioStudio's concept is larger than that of Clotho as its focus is genomes and not genes, it also lacks the comprehensive component as it specifically deals with sequence manipulation

Athena, like BioStudio, is a new software development whose design focus is integrating engineering concepts into synthetic biology[3]. It has a variety of functions including performing simulations, automatically deriving transcription rate expressions, and manipulating synthetic DNA sequences [3]. This tool is comprehensive, but lacks database connectivity for part import and export.

Clotho is designed to achieve both comprehensiveness and connectivity.

3. CLOTHO ARCHITECTURE

The Clotho system is developed around the key concept of "orthogonalization of concerns" as described by Platform-based design[8][2]. Specifically it separates communication, coordination, and computation. Therefore each aspect of the system is classified as to what type of operations it is involved with and the communication between components is explicitly separated from the computation of each component[9]. A diagram of this architecture can be seen in Figure 1. The coordination of the system is also removed from each individual component and maintained in a central location. This is the *ClothoCore*.

3.1 ClothoCore

The ClothoCore object maintains control over three types of system objects. *ClothoHubs*, *ClothoConnections*, and *ClothoAlgorithms*. Hubs are categorized as view type, connector type, function type, and interface type. There is one type of each of these hubs in the system. Connections belong to one hub each and belong to the hub corresponding to the type of connection they are derived from (view, connector, function, or interface). View connections should deal with the display of biological information. This information can be both graphical and/or text. Views may also present system information to the user. Connector type connections should con-

nect Clotho to external tools or data sources. Function connections should be processing engines for data. Interface connections should be points of interaction for the user to control the operation or settings assigned to Clotho. Interface connections can also manage libraries which Clotho uses as well.

The ClothoCore maintains a simple addressing scheme which allows it to find connections in a global context (across the whole system) as well as specific to each hub. A data object called ClothoData is used to process data and communicate between connections via hubs and the core. The ClothoCore can send ClothoData objects back and forth from one connection to another (point to point communication) or from one connection to many connections (broadcast communication). Clotho connections can be grouped as well into subgroups (e.g. those which process error messages or process dna data) which can be sent data specifically in a broadcast fashion.

/subsectionClothoData A ClothoData data object contains the following information: sender, recipient, use code (what this should be used for), operation type (which operation should this trigger), payload (the actual data), payload info (specific instructions for the payload), and also allows for miscellaneous information as well to be passed. Once a connection has packaged the data it wishes to send, it then contacts the core for the intended recipient(s) and the core takes care of the rest. Once a connection has been contacted by the core, it then can initiate a transaction back to the core in response to the sender of the data. This can occur as long as needed to finish the transaction.

Because of the modular way in which Clotho has been designed, a developer wishing to use Clotho need only create a connection derived from one of the 4 basic types of connections. Once the connection has been defined, it need only be instantiated in the Clotho main file and then call the required *activate* method (see Figure 2 for an example) and the ClothoCore takes care of the rest. Activation of a connection associates it with the core, a specific hub, provides it a global and hub address, and runs any needed start up routines.

```
ClothoConnection sequenceview = new  
ClothoSequenceViewConnection("SequenceView_  
Connection", "Sequence View", _core);  
sequenceview.activate();
```

Figure 2: Adding a ClothoConnection

The objective of this paper will be to describe one specific aspect of the Clotho architecture, the Algorithm Manager. A diagram of this section can be seen on the left hand side of Figure 1 enclosed in the dashed lines and will be discussed in depth in the next section.

4. ALGORITHM DEVELOPMENT ENVIRONMENT

The ClothoAlgorithmManagerConnection, like any connection in Clotho, is modular. This connection extends modularity further as within this connection a developer may also add another type of object, a ClothoAlgorithm. Like adding a connection to Clotho with the *activate()* method, a ClothoAlgorithm is just as easy for a developer to introduce into the Clotho system. This includes the development of the algorithm, the "wrapping" of the algorithm in a particular Java class, and instantiation of the algorithm in Clotho's main class. Figure 4 shows the code for adding it to the main class. This section will specifically discuss the development of the algorithm through the object-oriented programming concept of inheri-

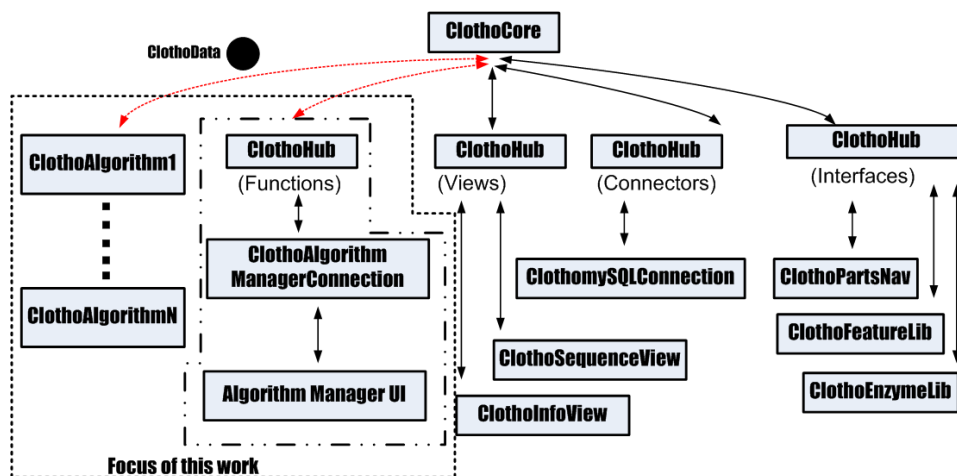


Figure 1: Clotho System Architecture

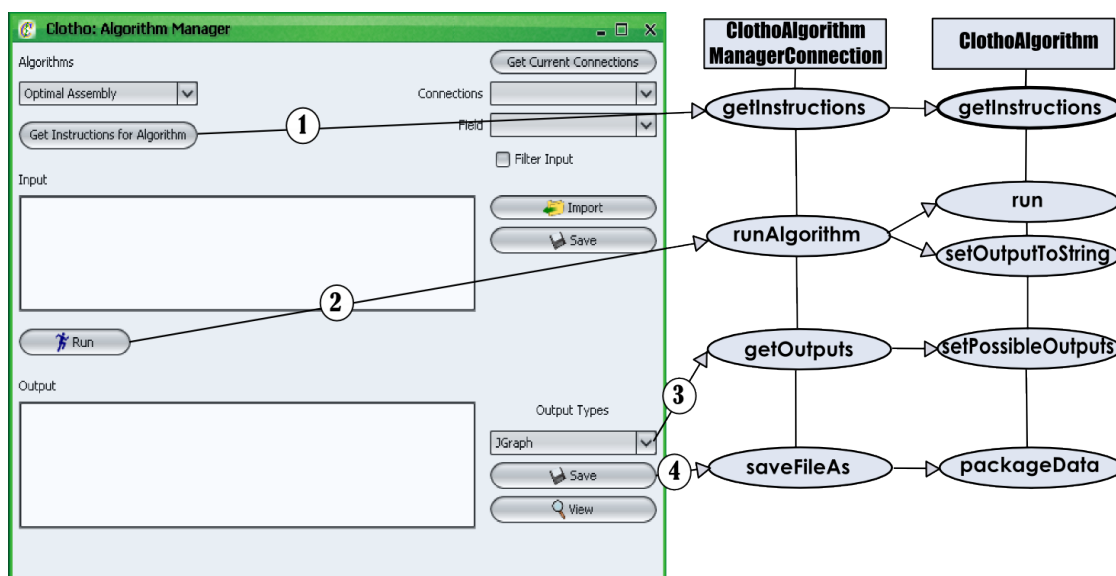


Figure 3: Algorithm Manager Screen Shot

tance[11].

4.1 Inheriting from the ClothoAlgorithm Class

An algorithm class must extend the ClothoAlgorithm abstract class and include its five abstract methods:

- `getInstructions`
- `getOutputToString`
- `packageData`
- `run`
- `setPossibleOutputs`

These methods are required as they are used in the ClothoAlgorithmManagerConnection to interact with the GUI. Below are sections devoted to each method.

4.1.1 `getInstructions`

The first method, `getInstructions`, must return a textual representation (String) with instructions for how to run the algorithm including information about formatting the input and what to expect for the result. The GUI will display this information to the user so that they know how to interact with each algorithm. While technically not required for correct operation, if this method is poorly implemented, the algorithm will be useless to users new to the algorithm.

4.1.2 `getOutputToString`

The next method, `getOutputToString`, must take the result from the algorithm and convert that to some textual (String) representation. Again, this text should be informational to the user because this information will be displayed in the GUI after running the algorithm. Much like `getInstructions`, this method is really for the users consumption and its quality will determine the algorithm's usefulness.

4.1.3 *packageData*

The *packageData* method also works with the output of the algorithm, but instead of a textual representation, this method should save the output to some specified file type. This method requires two parameters: a filename and a type of file. The filename will be selected by the user through a file chooser window, and the type of method will also be selected by the user through a menu of output types available in the GUI for the algorithm manager. This information will be passed to *packageData*, which will then save the file in the specified location and format.

4.1.4 *setPossibleOutputs*

The *setPossibleOutputs* method works with the *packageData* method to differentiate the types of files that should be created. This method takes a String array as its only parameter, which serves to list the possible outputs that algorithm can produce. First of all, this array will serve to populate a menu in the GUI of all possible outputs for the specified algorithm by calling the *ClothoAlgorithmManagerConnection*'s *getOutputs* method, which retrieves all outputs set for that algorithm. This same array will then be used by *packageData* to determine what type of file the user has chosen from the output menu in the GUI, and convert the data into that output's format.

4.1.5 *run*

Lastly, the *run* method will be used to actually run the algorithm by taking a String input as a parameter and creating some collection of output. This output can then be converted into the formats mentioned above using the *packageData* method. The collection of output will also be used by the *getOutputToString* method for display in the output area.

4.1.6 *Instantiating the Algorithm*

Once the algorithm has extended the required class, the final step for integrating an algorithm into Clotho is to instantiate a *ClothoAlgorithm* object within the Clotho main file and then call the *register* method. Registering an algorithm in the *ClothoCore* adds the algorithm to a global list of registered objects, which can then be accessed with the *ClothoAlgorithmManagerConnection*.

This object-oriented design not only provides for ease of integration for the developer, but it also becomes functional within the Algorithm Manager's architecture. Because all algorithms in Clotho must inherit the *ClothoAlgorithm* class, the *ClothoAlgorithmManagerConnection* can manage this interaction by keeping a single instance of the *ClothoAlgorithm* class. It can then use this instance for simple communication between the GUI and the selected algorithm.

4.2 GUI to Algorithm Communication

The Algorithm Manager tool (see Figure 3) within Clotho uses the *ClothoAlgorithm* class methods, both non-abstract and abstract, to create the interaction between the GUI and the *ClothoAlgorithmManagerConnection*. This interaction happens both generally and specifically for chosen algorithms. The first indication of this interaction is the dynamic population of the GUI's "Algorithms" menu with all *ClothoAlgorithm* objects that have been registered as described above.

Further interaction becomes specific to an algorithm as the user chooses one from the "Algorithms" menu. Once selected, the communication between the GUI and *ClothoAlgorithmManagerConnection* become algorithm-specific as a class variable called *currentAlgorithm* is set to the selected algorithm and all methods within the *ClothoAlgorithmManagerConnection* are run with that algorithm instance.

When the "Get Instructions for Algorithm" button is selected, the GUI will call the *ClothoAlgorithmManagerConnection* method *getInstructions*, which will in turn call the selected algorithm's *getInstructions* method as seen in Part One of Figure 3. The user would then follow those instructions to format input for running the chosen algorithm. After the input is formatted correctly, selecting the "Run" button within the GUI would call the *currentAlgorithm*'s *run* method and *getOutputToString* method. The *getOutputToString* method would then return a textual result to be displayed in the GUI's "Output" text box. This process is seen in Part 2 of Figure 3.

When interacting with the output, it would again be algorithm-specific. Like the "Algorithms" menu, the "Output Types" menu would be dynamically populated upon selection of an algorithm. The populating would be dependent upon the selected algorithm's *setPossibleOutputs* method to communicate all available file types, which can be seen in Part 3 of Figure 3. The user could then choose one of the output types from that algorithm as specified and click the "Save" button. This action would call *ClothoAlgorithmManagerConnection*'s *saveFileAs* method, which would then call the selected algorithm's *packageData* method with the selected output type and the chosen file location as parameters as seen in Part 4 of Figure 3. The "View" button could then retrieve that file location and open the appropriate application to view that type of output.

We have tested this functionality of instantiating the algorithm with developers and using the GUI to algorithm communication with users.

5. RESULTS

The success of this project is relevant to both users and developers. Users, specifically synthetic biologists, should be able to judge the tool's success based on its practical usability for actual experiments. Developers will be able to test the tool's modularity in attempting to add further algorithms and connections. The results of the Algorithm Manager can be looked at from these two perspectives.

5.1 User Results

The user results can be broken down into two types: tool usability and algorithm functionality.

5.1.1 Tool Usability

Tool usability is less concrete as it is defined subjectively by a user test group. The tool was tested by 11 researchers on UC Berkeley's International Genetically Engineered Machines (iGEM) team. Their experience ranges from high school students to graduate researchers. The Algorithm Manager was found to be intuitive with questions only be asked about specific algorithms' functionalities.

5.1.2 Algorithm Functionality

Algorithm usability is easier to define as efficiency is a definite measure of its success. Only one of the two algorithms was tested in practical application. This algorithm, "2ab Assembly Algorithm," takes a set of goal parts as input, and attempts to minimize the depth and number of construction intermediates required to build all goal parts in parallel. A goal parts can be defined as any combination of biological parts desired to be created.

This algorithm was used by the Anderson Lab at UC Berkeley to assemble 496 parts. Had no overlap been found, 5961 total parts would have to have been made. The program generated an assembly tree that required only 1283 total parts be built. That is a savings of 4678 parts.

This cost can also be evaluated monetarily as gene synthesis cost around \$1 per base pair of DNA with an average of 3000 base pairs

per part. The monetary cost would be decreased by 80% in using the algorithm versus not accounting for overlap.

Time is another important cost to consider. This assembly took just over 4 minutes to complete using the algorithm. If this same process were attempted by hand, it would take multiple people over 3 days to complete. Again, monetary cost would factor into this time in the lab. Overall, using the 2ab Assembly Algorithm within the software tool is significantly more efficient than performing this same task by hand.

5.2 Developer results

The developer results were tested by taking the “2abAssembly Algorithm” discussed above and implementing it into the Clotho Algorithm Manager in the method described. The developer was given information concerning the five methods required for implementation by the ClothoAlgorithm abstract class. He implemented these methods and actually found that the majority of these methods were already defined in his algorithm. Next, his algorithm was added into Clotho by adding the two lines of code into the Main method. This process only took around 30 minutes to complete and his algorithm was fully integrated into Clotho.

```
BaselineOptimalAssembly alg0 = new
BaselineOptimalAssembly("Optimal Assembly");
alg0.register(assemblyconnection);

TwoabAssembly alg1 = new TwoabAssembly
("2abAssembly");
alg1.register(assemblyconnection);
```

Figure 4: Instantiating an Algorithm in Clotho

The results for the tool itself, although limited, do provide some assessment of success of the software. The algorithm testing provided more conclusive and obvious results in demonstrating success. Overall, the tool was both efficient and easy to use for user and developer alike.

6. CONCLUSION

We have shown a way to provide synthetic biologists access to algorithms that is both usable and modular. Our Algorithm Manager provides accessibility for both the user and the developer, which was demonstrated by the integration of two assembly algorithms into our tool. In this growing field of synthetic biology, it will be important that algorithms be made accessible to the user rapidly and implemented in a user-friendly environment.

Moreover, we have demonstrated that not only is the tool beneficial, but the algorithms implemented within the tool offer drastic improvement over previous assembly methods. These algorithms will save synthetic biologists valuable time and resources, and through the Algorithm Manager they become feasible for practical application.

7. FUTURE WORK

We are currently working to add other assembly algorithms to the tool as well as to extend its database functionality.

7.1 Additional Algorithms

Christopher Batten, a computer engineering researcher at the University of California, Berkeley, wrote the algorithm “Baseline Optimal Assembly” that is one of the default algorithm within Clotho,

and he has further extended this algorithm to take common subparts into account during assembly[1]. This extension, “Optimal Assembly with Common Subparts,” builds goal parts by optimizing common subpart use.

For example, if the algorithm were to build the part “abcdab” with the common subpart “ab” it would produce a more optimal assembly set than building the part without common subparts. The algorithm’s run time would slightly increase with the existence of common subparts, but would still be a drastic improvement over attempting this same assembly manually. The result would outweigh the cost as it would save assembly steps and stages within the lab ultimately resulting in saved laboratory resources. For more information see Batten’s paper “Algorithms for Optimal Assembly.”

7.2 Extension of Database Connectivity

Beyond the implementation of algorithms to the tool, its database connectivity is being extended. Currently the ClothoMySQLConnection talks to a MySQL database. The user is able to navigate to a part within a connected database and export that part directly to the input area of the Algorithm Manager. The ClothoAlgorithmManagerConnection receives information not only about the field that is exported into the text area, but also about all database fields that are associated with that part. This exportation becomes a proof of concept for connecting the database to the algorithms.

7.2.1 Preventing Redundant Part Creation

This concept will be extended further for chosen algorithms. For example, when assembling a part it may be beneficial for an algorithm to know that the part already exists within the given database. If it exists, then any goal part containing this part will not build that part, but rather it will inform the user that it already exists within her repository. This information will save the user time in assembly as they will not waste resources building a part that already exists.

7.2.2 Adding Results of the Algorithm to a Database

After building a part, the user will also be able to access the database and add her new part. This will allow all users connecting to that database to know that the part has already been built. Information about the assembled part should be extended to include all associated part data as found through the database connection. The user would then have information about the part other than the input field, which would allow her to better understand the assembled part for practical laboratory use.

7.2.3 Assessing Difficulty

The implementation of these two extensions are differing in difficulty. Adding additional algorithm, as stated in the description, should be trivial. Increasing database connectivity, while already partially implemented, may prove more difficult for complete integration. In small scale use it is feasible, but on a larger scale external problems exist. These problems will arise from the lack of standardization concerning biological parts repositories in the field of synthetic biology.

There is no standard on how to store part information, so different databases may contain different fields which refer to the same data. This inconsistency would cause information about a part to be misinterpreted, decrease efficiency, and cause errors. Provisional BioBrick Language (PoBol)[7] is a step in the right direction, and could provide a standard that would ease database integration, but until synthetic biologists can agree on a standard, the idealistic view of database sharing is not realistic.

8. ACKNOWLEDGMENTS

The authors would like to thank Prof. J. Christopher Anderson, Christopher Batten, and Will DeLoache for their valuable input and contributions to this paper. This work was also made possible by the SUPERB program at UC Berkeley.

9. REFERENCES

- [1] C. Batten. Algorithms for optimal assembly. Published on his website at <http://www.mit.edu/~cbatten/work/ssbc04/optassembly-ssbc04.pdf>, July 2008.
- [2] L. Carloni, F. D. Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based design for embedded systems. In *The Embedded Systems Handbook*. CRC Press, 2005.
- [3] D. Chandran, F. Bergmann, and H. Sauro. Athena: Modular cad/cam software for synthetic biology. WWW page, 2008.
- [4] M. W. Davis. Ape: A plasmid editor. WWW page, 2008.
- [5] D. Densmore. Platform-based design of synthetic biological tools. WWW page, 2008.
- [6] J. A. Goler. Biojade. WWW page, 2008.
- [7] R. Grünberg and J. Morrison. Pabol. WWW page, 2008.
- [8] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. In *IEEE Transactions on Computer-Aided Design*, volume 19, December 2000.
- [9] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, February 2002.
- [10] S. C. Sleight, D. Chandran, F. T. Bergmann, L. P. Smith, M. Cowell, J. Morrison, R. G. Grünberg, J. C. Anderson, J. Cumbers, and H. M. Sauro. Standards and tools in synthetic biology. Received through personal communication with J. Christopher Anderson, July 2008.
- [11] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM.