Knowledge Representation in Synthetic Biology

James Alastair McLaughlin

Submitted for the degree of Doctor of Philosophy in the School of Computing, Newcastle University

September 2019

Supervised by Professor Anil Wipat & Dr Dana Ofiteru

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or other qualification at Newcastle University or any other institution.

James Alastair McLaughlin

September 2019

Publications arising from this thesis

- **McLaughlin**, **J. A.**, Myers, C. J., Zundel, Z., Wilkinson, N., Atallah, C., and Wipat, A. “sboljs: Bringing the Synthetic Biology Open Language to the Web browser”. In: *ACS synthetic biology* 8.1 (2018), pp. 191–193.
- **McLaughlin**, **J. A.**, Myers, C. J., Zundel, Z., Mısırlı, G., Zhang, M., Ofiteru, I. D., Goñi Moreno, A., and Wipat, A. “SynBioHub: A Standards-Enabled Design Repository for Synthetic Biology”. In: *ACS synthetic biology* 7.2 (2018), pp. 682–688.
- **McLaughlin**, **J. A.**, Pocock, M., Mısırlı, G., Madsen, C., and Wipat, A. “VisBOL: web-based tools for synthetic biology design visualization”. In: *ACS synthetic biology* 5.8 (2016), pp. 874–876.
- Madsen, C., **McLaughlin**, **J. A.**, Mısırlı, G., Pocock, M., Flanagan, K., Hallinan, J., and Wipat, A. “The SBOL Stack: a platform for storing, publishing, and sharing synthetic biology designs”. In: *ACS synthetic biology* 5.6 (2016), pp. 487–497.
- Cox III, R. S., **McLaughlin**, **J. A.**, Grünberg, R., Beal, J., Wipat, A., and Sauro, H. M. “A Visual Language for Protein Design”. In: *ACS synthetic biology* 6.7 (2017), pp. 1120–1123.
- Zhang, M., **McLaughlin**, **J. A.**, Wipat, A., and Myers, C. J. “SBOLDesigner 2: an intuitive tool for structural genetic design”. In: *ACS synthetic biology* 6.7 (2017), pp. 1150–1160.
- Cox, R. S., Madsen, C., **McLaughlin**, **J. A.**, Nguyen, T., Roehner, N., Bartley, B., Bhatia, S., Bissell, M., Clancy, K., Gorochowski, T., Grünberg, R., Luna, A., Le Novere, N., Pocock, M., Sauro, H., Sexton, J. T., Stan, G.-B., Tabor, J. J., Voigt, C. A., Zundel, Z., Myers, C., Beal, J., and Wipat, A. “Synthetic biology open language visual (SBOL visual) version 2.0”. In: *Journal of Integrative Bioinformatics* 15.1 (2018).
- Brown, B., Atallah, C., **McLaughlin**, **J. A.**, Misirli, G., Goñi-Moreno, Á., Roehner, N., Skelton, D. J., Bartley, B., Beal, J., Poh, C. L., et al. “Capturing Multicellular System Designs Using the Synthetic Biology Open Language (SBOL)”. in: *bioRxiv* (2018), p. 463844.
- Quinn, J. Y., Cox III, R. S., Adler, A., Beal, J., Bhatia, S., Cai, Y., Chen, J., Clancy, K., Galdzicki, M., Hillson, N. J., Novere, N. L., Maheshwari, A. J., **McLaughlin**, **J. A.**, Myers, C. J., Umesh, P., Pocock, M., Rodriguez, C., Soldatova, L., Stan, G.-B. V.,

Swainston, N., Wipat, A., and Sauro, H. M. “SBOL visual: a graphical language for genetic designs”. In: *PLoS biology* 13.12 (2015), e1002310.

- Cox III, R. S., Madsen, C., **McLaughlin**, **J. A.**, Nguyen, T., Roehner, N., Bartley, B., Beal, J., Bissell, M., Choi, K., Clancy, K., Grünberg, R., Macklin, C., Mısırlı, G., Oberortner, E., Pocock, M., Samineni, M., Zhang, M., Zhang, Z., Zundel, Z., Gennari, J. H., Myers, C., Sauro, H., and Wipat, A. “Synthetic biology open language (SBOL) version 2.2. 0”. In: *Journal of Integrative Bioinformatics* 15.1 (2018).

- Mısırlı, G., Nguyen, T., **McLaughlin**, **J. A.**, Vaidyanathan, P., Jones, T., Densmore, D., Myers, C. J., and Wipat, A. “A computational workflow for the automated generation of models of genetic designs”. In: *ACS synthetic biology* 7.2 (2018), pp. 682–688.

- Mısırlı, G., Nguyen, T., **McLaughlin**, **J. A.**, Myers, C., and Wipat, A. “Standard Enabled Model Generator for Genetic Circuit Design”. In: *9th International Workshop on Bio-Design Automation.* 2017, p. 42. URL: `http://www.iwbdaconf.org/2017/docs/IWBDA_2017_Proceedings.pdf`.

- Beal, J., Cox, R. S., Grünberg, R., **McLaughlin**, **J. A.**, Nguyen, T., Bartley, B., Bissell, M., Choi, K., Clancy, K., Macklin, C., Madsen, C., Mısırlı, G., Oberortner, E., Pocock, M., Roehner, N., Samineni, M., Zhang, M., Zhang, Z., Zundel, Z., Gennari, J. H., Myers, C., Sauro, H., and Wipat, A. “Synthetic Biology Open Language (SBOL) Version 2.1. 0”. In: *Journal of Integrative Bioinformatics* 13.3 (2017), pp. 30–132.

- Misirli, G., Taylor, R., Goñi-Moreno, A., **Mclaughlin**, **J. A.**, Myers, C. J., Gennari, J., Lord, P., and Wipat, A. “SBOL-OWL: An ontological approach for formal and semantic representation of synthetic biology information”. In: *ACS synthetic biology* (2019).

- Roehner, N., Bartley, B., Beal, J., **McLaughlin**, **J.**, Pocock, M., Zhang, M., Zundel, Z., and Myers, C. J. “Specifying Combinatorial Designs with the Synthetic Biology Open Language (SBOL)”. in: *ACS Synthetic Biology* (2019).

- Mısırlı, G., Hallinan, J., Pocock, M., Lord, P., **McLaughlin**, **J. A.**, Sauro, H., and Wipat, A. “Data integration and mining for synthetic biology design”. In: *ACS synthetic biology* 5.10 (2016), pp. 1086–1097.

- Myers, C. J., Beal, J., Gorochowski, T. E., Kuwahara, H., Madsen, C., **McLaughlin**, **J. A.**, Mısırlı, G., Nguyen, T., Oberortner, E., Samineni, M., Wipat, A., Zhang, M., and Zundel, Z. “A standard-enabled workflow for synthetic biology”. In: *Biochemical Society Transactions* 45.3 (2017), pp. 793–803.

- Mısırlı, G., Madsen, C., Murieta, I. S. de, Bultelle, M., Flanagan, K., Pocock, M., Hallinan, J., **McLaughlin**, **J. A.**, Clark-Casey, J., Lyne, M., Micklem, G., Stan, G.-B., Kitney, R., and Wipat, A. “Constructing synthetic biology workflows in the cloud”. In: *Engineering Biology* 1.1 (2017), pp. 61–65.

- Beal, J., Nguyen, T., Gorochowski, T. E., Goñi-Moreno, A., Scott-Brown, J., **McLaughlin**, **J. A.**, Madsen, C., Aleritsch, B., Bartley, B., Bhakta, S., et al. "Communicating Structure and Function in Synthetic Biology Diagrams". In: *ACS synthetic biology* (2019).

Abstract

Synthetic biology, or SynBio, is a relatively new and exciting field concerning the formalisation of genetic engineering into a design, build, test, learn lifecycle common to other engineering disciplines. This lifecycle can be used to systematically develop biological systems, such as synthetic genetic circuits — where transcriptional machinery is repurposed to construct familiar electronic circuit concepts such as logic gates — and other engineered devices such as biosensors or drug production factories.

Synthetic biological systems are typically designed by repurposing existing natural and synthetic biological parts. This design process is made possible by knowledge about part structure and function, which can be experimentally derived or predicted using bioinformatics methodologies. However, the process of gathering such knowledge is arduous, as it is often computationally intractable, distributed across multiple disparate databases with semantic and syntactic heterogeneity, or even not recorded at all.

The research question motivating this work is how the machine-tractability of knowledge can be improved in order to make the synthetic biology design process more efficient. There are both short-term and long-term approaches. The short-term approach is to improve the ease of access and machine-tractability of existing knowledge relevant to SynBio design. The long-term approach is to establish the software and data infrastructure necessary to enable knowledge about future designs to be documented in a standardized manner.

This work investigates both approaches with research into data standards, significantly furthering the development of the Synthetic Biology Open Language (SBOL) to improve the machine-tractability of design knowledge; the research and development of novel technology for data integration to make existing information easier to access; conversion of an existing dataset, the iGEM Registry, into an enriched SBOL representation; the development of SynBioHub, a repository for the sharing and dissemination of future SynBio designs; and SynBioCAD, a visual tool enabling synthetic biologists to capture their designs using data standards.

Acknowledgements

In no particular order, I would like to thank FUJIFILM DioSynth Biotechnologies and the SAgE faculty for funding this project; my supervisors Anil Wipat and Dana Ofiteru; the SBOL community, especially Chris Myers; and everyone in our team at Newcastle University past and present, especially Göksel Mısırlı and Ángel Goñi Moreno.

This work is dedicated to my family, without whom it would not have been possible. I would especially like to thank my parents Andy & Tricia McLaughlin for putting up with me all of this time; my wife Heather McLaughlin for her patience during my late-night writing sessions; and my brother Nick Smith for all of his encouragement over the years.

I would also like to thank my friends Joseph Mullen, Keith Flanagan, James Skelton, and Christian Atallah for all of the helpful discussions, advice, and support.

Attribution

The work produced as part of this thesis is open source, and has been developed as such in collaboration with the synthetic biology community. Therefore, much of the research presented has additional contributors beside myself. While co-authorship is made explicit in publications, it is not as clear in thesis format. To avoid any ambiguity, a list of projects with a description of my own contribution and external contributions follows. The involvement of my supervisors, Anil Wipat and Irina Dana Ofiteru, is implicit.

- SBOL is a community standard, and was established long before I began my research. I have contributed significantly to the standard by serving as an elected editor for two years, attending numerous workshops, participating in discussions both online and offline, contributing to publications, and submitting SBOL enhancement proposals (SEPs). The SEPs to which I have contributed are included in Appendix A, each including their respective authorship information.

- sboljs: The first version was developed by myself. Later versions have code contributions from Chris Myers, Zach Zundel, Dany Fu, and Nathan Wilkinson (University of Utah).

- sbolgraph: The first version was developed by myself. Later versions have code contributions from Christian Atallah (Newcastle University).

- pysbolgraph: Originally ported from sbolgraph as a joint effort with James Scott-Brown (University of Oxford). Later versions have code contributions from Christian Atallah, Bradley Brown, and Lewis Grozinger (Newcastle University).

- iGEM to SBOL conversion: The first version was developed by myself, then furthered to be a complete conversion with help from Chris Myers (University of Utah) during his sabbatical at Newcastle University.

- SBOL Stack: Based on an idea by Matthew Pocock. Developed in collaboration with Curtis Madsen, with contributions from Goksel Misirli, Matthew Pocock, Keith Flanagan, and Jennifer Hallinan. The code for its server API and client libraries were written by myself.

- SynBioHub: Original version developed by myself in collaboration with the SBOL Stack authors, and based on a user interface design by Antarctic Design. Later versions have contributions from Chris Myers, Zach Zundel, Oliver Flatt, and Michael Zhang (University of Utah); and Christian Atallah (Newcastle

University). The Java client code was developed by myself, and later integrated into libSBOLj. The Web of Registries service was developed by Zach Zundel at the University of Utah. SynBioHub Lab is being developed in collaboration with Christian Atallah.

- VisBOL: First version developed by myself, originally as my dissertation project for my undergraduate degree. Development continued into my Ph.D., and it now has contributions from Chris Myers, Zach Zundel, Dany Fu, Nathan Wilkinson, and James Scholz (University of Utah); and Arezoo Sadeghi (Boston University).

- ldf-facade, dnarichment, and SynBioCAD were developed by myself and had no external contributors at the time this thesis was written.

Source code

Chapter 3: Machine-tractability in the design process

sboljs — `https://github.com/SynBioDex/sboljs`
sbolgraph — `https://github.com/udp/sbolgraph`
pysbolgraph — `https://github.com/udp/pysbolgraph`
SBOL Stack — `https://github.com/ICO2S/sbolstack`

Chapter 4: Data harmonization using Linked Data Fragments (LDF)

ldf-facade — `https://github.com/BioEnrichment/ldf-facade`
distrowatch-ldf — `https://github.com/udp/distrowatch-ldf`
jbei-ice-js — `https://github.com/udp/jbei-ice-js` jbei-ice-ldf — `https://github.com/udp/jbei-ice-ldf`

Chapter 5: Conversion and Enrichment of the iGEM Registry

igem2sbol — `https://github.com/udp/igem2sbol`
dnarichment — `https://github.com/udp/dnarichment`
enrichment2 — `https://github.com/udp/enrichment2`

Chapter 6: SynBioHub: a standards-enabled design repository for synthetic biology

SynBioHub — `https://github.com/synbiohub/synbiohub`
SynBioHub Lab — `https://github.com/synbiohub/synbiohub-lab`

Chapter 7: SynBioCAD: a standards-enabled design tool for synthetic biology

SynBioCAD – `https://github.com/SynBioCAD/synbiocad`

Other code developed for this work

bioterms — `https://github.com/udp/bioterms`
sequence-formatter — `https://github.com/udp/sequence-formatter`

fmaprefix – `https://github.com/udp/fmaprefix`
sbolmeta – `https://github.com/udp/sbolmeta`
rdf-serializer-xml – `https://github.com/ICO2S/rdf-serializer-xml`
jfw – `https://github.com/SynBioCAD/jfw`

Contents

1 Introduction **17**
- 1.1 Motivation for this work . . . 17
- 1.2 Aims & Objectives . . . 18
- 1.3 Contribution of this thesis . . . 19
- 1.4 Thesis structure . . . 20

2 Background **22**
- 2.1 Synthetic biology . . . 22
  - 2.1.1 Genetic circuits . . . 23
  - 2.1.2 An engineering lifecycle . . . 24
- 2.2 Data standards . . . 28
  - 2.2.1 The Resource Description Framework (RDF) . . . 29
  - 2.2.2 The Synthetic Biology Open Language (SBOL) . . . 32
- 2.3 Data integration . . . 42
  - 2.3.1 Relational databases and their limitations . . . 43
  - 2.3.2 RDF triplestores . . . 44
  - 2.3.3 Linked Data . . . 45
  - 2.3.4 Query federation . . . 46
  - 2.3.5 Linked data fragments . . . 46
- 2.4 Sharing and dissemination . . . 48
  - 2.4.1 iGEM Parts Registry . . . 49
  - 2.4.2 Addgene . . . 51
  - 2.4.3 JBEI-ICE . . . 53
- 2.5 Standards-enabled tooling . . . 56
  - 2.5.1 SBOL Visual . . . 56
  - 2.5.2 Tooling for SBOL Visual . . . 57
- 2.6 Conclusion . . . 60

I Knowledge representation **61**

3 Machine-tractability in the design process **62**
- 3.1 Introduction . . . 62
- 3.2 Challenges for adoption of SBOL . . . 63
  - 3.2.1 Limitation in scope . . . 63
  - 3.2.2 Lack of data . . . 63
  - 3.2.3 Lack of tooling . . . 65

3.2.4 Portability 65
3.2.5 Complexity 65
3.3 Improving portability of SBOL 66
3.3.1 sboljs: SBOL on the Web 67
3.3.2 sbolgraph 69
3.4 SBOL Stack 76
3.5 Enhancement proposals for SBOL 77
3.5.1 Integration of provenance 77
3.5.2 Alignment with external ontologies 78
3.5.3 Removal of unnecessary specification of serialization 79
3.5.4 Addressing structural/functional dichotomy 79
3.5.5 More intuitive nomenclature 82
3.6 Discussion & Conclusion 82
3.6.1 Libraries for SBOL 84
3.6.2 SBOL Stack 85
3.6.3 SBOL3 proposals 85
3.6.4 Future work 86

II Data Integration 90

4 Data harmonization using Linked Data Fragments (LDF) 91
4.1 Introduction 91
4.2 Linked Data Fragments for non-RDF data sources 92
4.3 Modelling non-RDF services using triple patterns 93
4.3.1 Time complexity 95
4.4 ldf-facade: An intelligent server for LDF 95
4.4.1 Tracking state 98
4.5 Example usage: RDFizing DistroWatch 99
4.6 SynBio applications 100
4.6.1 JBEI-ICE RDF using ldf-facade 100
4.7 Discussion & Conclusion 107
4.7.1 ldf-facade 110
4.7.2 Federated design knowledge using SBOL Stack and LDF 110
4.7.3 Future work 111

5 Conversion and Enrichment of the iGEM Registry 117
5.1 Introduction 117
5.2 Modelling the iGEM database as SBOL 118
5.2.1 Categories 123
5.2.2 De-flattening 123
5.3 iGEM-SBOL graph queries 124
5.3.1 Use of transcription factor binding sites in the iGEM Registry . . 124
5.3.2 Re-use of parts in the iGEM Registry 125
5.4 Enrichment 128
5.4.1 Building a knowledge base 129
5.4.2 RDF representation 132

5.4.3 Enrichment tool 132
5.5 Discussion & Conclusion 139
5.5.1 Future work 139

III Sharing and Dissemination 143

6 SynBioHub: a standards-enabled design repository for synthetic biology 144
6.1 Introduction 144
6.2 Architecture of SynBioHub 145
6.2.1 SynBioHub vocabulary 147
6.2.2 Users and graphs 149
6.3 Interfaces 149
6.3.1 Web interface 150
6.3.2 Programmatic API 150
6.4 Web of Registries 150
6.5 SynBioHub Lab 151
6.5.1 Whole-lifecycle knowledge management 151
6.5.2 Nomenclature 152
6.5.3 Architectural changes 152
6.6 Discussion & Conclusion 153
6.6.1 The FAIRdom principles 154
6.6.2 Dereferenceable URIs 155
6.6.3 Future work 155

7 SynBioCAD: a standards-enabled design tool for synthetic biology 167
7.1 Introduction 167
7.2 Layouts for genetic circuits 170
7.2.1 Specifying layouts 170
7.2.2 Layout to SBOL references 171
7.2.3 Creating layouts for SBOL data 173
7.2.4 Configurating layouts 175
7.2.5 JSON representation 177
7.3 Interactive genetic circuit visualizations for the Web 177
7.3.1 Rendering layouts with Scalable Vector Graphics (SVG) 177
7.3.2 Interactivity 178
7.4 The SynBioCAD application 182
7.5 Discussion & Conclusion 187
7.5.1 Future work 187
7.5.2 Conclusion 188

Conclusions 190

8 Discussion & Conclusion 190
8.1 The groundwork: Machine-tractable design 190
8.2 Aim 1: Access to existing knowledge 192

8.3 Aim 2: Accessibility of future knowledge 194
8.3.1 Conclusion 195

Appendix A SEPs 207

Appendix B Examples of valid RDF/XML incompatible with libSBOLj 243

Abbreviations

API — Application Programming Interface

CAD — Computer Aided Design

CDS — Coding Sequence

FAIR, FAIRdom — Findability, Accessibility, Interoperability, and Reproducibility

GO, GO term — Gene Ontology, Gene Ontology term.

HMM — Hidden Markov Model

HTTP — Hypertext Transport Protocol

JSON — JavaScript Object Notation

LDF — Linked Data Fragments

ORF — Open Reading Frame

PWM — Position Weight Matrix

RDF — The Resource Description Framework

RDF/XML — A serialization of the Resource Description Framework (RDF) using the eXtended Markup Language (XML)

SBO, SBO term — Systems Biology Ontology, Systems Biology Ontology term

SBOL — The Synthetic Biology Open Language

SBOL Visual, SBOLv — The Synthetic Biology Open Language Visual

SEP — SBOL Enhancement Proposal

SO, SO term — Sequence Ontology, Sequence Ontology term

SPARQL — SPARQL Protocol and RDF Query Language

TF — Transcription Factor

TFBS — Transcription Factor Binding Site

Turtle — Terse RDF Triple Language

URI — Uniform Resource Identifier

URL — Uniform Resource Locator

XML — eXtended Markup Language

iGEM — The Internationally Genetically Engineered Machine

1. Introduction

Synthetic biology (SynBio) is to genetic engineering what software engineering is to computer programming, applying principles such as modularisation, standardisation, and a design-build-test-learn lifecycle to the design of biological systems just as software engineering does to the design of computer programs [1]. The ultimate objective of SynBio is also the same as software engineering: to turn a largely ad-hoc, unpredictable process into a rigorous engineering pipeline.

Quantifiable and reproducible development of novel biological systems has the potential to revolutionise everything from pharmaceuticals [2] to space exploration [3]. SynBio is expected to become a £62bn market by 2020 [4], and is also a highly active area of academic research. In the UK, synthetic biology has been made a BBSRC priority due to its potential to "solve a number of major global challenges in fields including health and wellbeing, energy, food security and the environment" [5].

Clearly, the potential of synthetic biology is widely acknowledged. However, there is still much work to be done in order to accomplish the end-goal of a truly reproducible, deterministic pipeline for the development of biological sytems. Every stage of the engineering lifecycle must be meticulously optimised. Recent innovations in targeted gene editing technologies such as CRISPR-Cas9 and fast, affordable sequencing have dramatically improved efficiency in the build stage [6]. However, the *design* stage is still predominantly a manual process, requiring the designer to gather knowledge by trawling through multiple disparate sources of information, many of which are computationally intractable and therefore cannot be queried systematically.

1.1 Motivation for this work

The research question motivating this work is how the machine-tractability of knowledge can be improved in order to make the synthetic biology design process more efficient.

While there have been significant optimizations in many stages of the synthetic biology lifecycle, the *design* stage has comparably been neglected. Designing a synthetic biological system requires gathering large amounts of domain-specific information about what kind of parts to use and how those parts might behave. Sometimes such information is available in databases, but often with many differing representations. Sometimes the information is hidden away in forms that are not easily computationally tractable, such as free text in publications. Worst of all, often the necessary information has never been recorded and only exists as the domain knowledge or intuition of human experts.

The issue of improving access to existing knowledge can be addressed using a range of computational techniques collectively known as *data integration* [7]. Many of these techniques, such as data harmonization, where the semantics of heterogeneous datasets are adapted to enable them to complement each other; data warehousing, where data from multiple disparate datasources are combined into one larger dataset; and query federation, where multiple disparate datasets are queried dynamically at the time of access; are already established in bioinformatics. However, knowledge about *engineered parts* is different from knowledge about naturally occurring genomes, and requires its own repositories and data representations.

Accessing *existing* knowledge is only a short-term solution to a much wider problem: that data about synthetic biological parts is routinely not recorded, or is recorded in a form that is not easily computationally tractable. Solving this problem for the long-term requires a paradigm shift in the tooling that people use to design synthetic biological systems. Instead of using bespoke data formats and private databases, software must be built on open standards and public repositories, so that the design information is made readily accessible both to the software and users today, and those in the future.

Therefore, this work has two major research themes. The first theme, with the motivation of making SynBio design easier in the short-term, is to explore how access to existing knowledge useful for SynBio design can be improved using data standardization and integration methodologies. The second theme, with the motivation of making design easier for future synthetic biologists, is to explore how software and data infrastructure can be adapted to ensure that future synthetic biological parts can be documented in a well-defined and computationally tractable form.

1.2 Aims & Objectives

The two main research aims of this work are:

1. To explore how access to **existing knowledge** can be improved for the synthetic biology design process
2. To propose data standards and software infrastructure to make **future knowledge** about designs more accessible

These aims were broken down into the following objectives:

- Research and develop technologies to capture SynBio design in a machine-tractable representation
- Research and develop strategies for data integration applicable to synthetic biology datasets
- Investigate approaches to the enrichment of existing knowledge about biological parts
- Research and develop tooling for the sharing and dissemination of synthetic biological designs

1.3 Contribution of this thesis

Development of data standards

Much of this work concerns contributions to the synthetic biology community data standard, the Synthetic Biology Open Language (SBOL), which is shown to serve as a foundation both for short-term data harmonization and as a standardized, machine-tractable data model upon which future SynBio tooling can be built.

This work provides numerous significant contributions to the SBOL standard, including *sbolgraph* (section 3.3), a design pattern for creating SBOL-enabled tooling with complete demonstrative implementations in multiple programming languages[1,2]; and a series of SBOL Enhancement Proposals (SEPs) suggesting improvements to the core standard (attached in Appendix A). The SEPs culminate in a proposed specification for the next major iteration of the standard (SBOL 3.0) described in chapter 3.

Data integration methodology and applications

Biodesign is typically dependent on a huge amount of data. Unfortunately, these data are often spread across hundreds of different databases with differing syntax and semantics. The process of discovering information about parts to use in a SynBio design can therefore be extremely tedious and error-prone, requiring significant manual effort.

One of the benefits of the adoption of data standards such as SBOL is that it is now possible to apply *data integration* techniques to improve access to synthetic biology knowledge bases. The research in chapter 4 explores how recent innovations in data integration can be applied to SynBio, and also how their fundamentals can be extended with the development of ldf-facade[3], a novel data integration framework for the dynamic conversion of legacy datasets.

Also on the theme of data integration, this work explores the application of data standards to existing datasets to facilitate improved tractability and interoperability with a complete conversion of the iGEM Registry of Standard Biological Parts to SBOL2, described in chapter 5.

Finally, the SBOL2 data standard can represent much more about a biological part than just DNA, such as information about gene products and their interactions. However, existing datasets, including the iGEM Registry, do not provide such information. The Enrichment[4] system proposed in chapter 5 shows how it may be possible to systematically apply bioinformatics tooling to "fill in the blanks" by augmenting designs with additional data.

A publishing workflow

A crucial concept in engineering — particularly software engineering — is the re-use of smaller components to build larger designs. This process is still difficult in synthetic

[1]`https://github.com/udp/sbolgraph`
[2]`https://github.com/udp/pysbolgraph`
[3]`https://github.com/BioEnrichment/ldf-facade`
[4]`https://github.com/udp/dnarichment`

biology for various reasons, not least of which is the unpredictable nature of biological components. Another reason is that designs are not being communicated effectively. Publications often omit crucial information necessary to reproduce designs [8], and the large part repositories that do exist, such as the iGEM Registry, suffer from issues arising from their lack of a standardised, machine-tractable data model (section 2.4.1).

SynBioHub[5] (chapter 6) is an open-source repository developed primarily as part of this work to facilitate the dissemination of synthetic biological designs. Just as software engineering has code repositories such as GitHub and Bitbucket, SynBioHub aims to help synthetic biologists work together more effectively by making it easy to share design information in a standardised manner. Since its publication, SynBioHub has been adopted by projects such as the NSF Living Computing Project (LCP) [9] and the DARPA Synergistic Discovery & Design Project (SD2) [10].

In addition to providing a repository, this work also includes the development of *SynBioCAD*[6] (chapter 7), a user-facing computer aided design (CAD) tool which enables users to communicate their designs with both a standardised SBOL Visual representation and an SBOL data backend. Together, SynBioCAD and SynBioHub provide the groundwork to enable users to document their designs using data standards and publish them for sharing and dissemination.

1.4 Thesis structure

This thesis consists of a background chapter (pp. 22-61), followed by five research chapters (pp. 62-189) and discussion and conclusions (pp. 190-195). The research chapters are as follows:

I Knowledge representation (pp. 62-90)

- Chapter 3 *Machine-tractability in the design process* (pp. 62-90) concerns furtherance of the development of the Synthetic Biology Open Language (SBOL), a standard for machine-tractable synthetic biology design representation.

II Data Integration (pp. 91-143)

- Chapter 4 *Data harmonization using Linked Data Fragments (LDF)* (pp. 91-116) explores how the recent innovation in the RDF community of Linked Data Fragments (LDF) can be repurposed for the harmonization of non-RDF resources to RDF, and how it can be applied to SBOL data.

- Chapter 5 *Conversion and Enrichment of the iGEM Registry* (pp. 117–1) explores how the SBOL standard can be applied to an existing dataset, the iGEM Registry of Standard Biological Parts, and how the wider scope of the SBOL2 data model can be used to enrich existing parts with additional knowledge.

[5]`https://wiki.synbiohub.org`
[6]`https://biocad.io`

III Sharing and Dissemination (pp. 144-189)

- Chapter 6 *SynBioHub: a standards-enabled design repository for synthetic biology* (pp. 144-166) concerns the development of SynBioHub, a repository for the sharing and dissemination of synthetic biology designs built on the SBOL standard/
- Chapter 7 *SynBioCAD: a standards-enabled design tool for synthetic biology* (pp. 167-189) concerns the development of SynBioCAD, a Web-based CAD tool for visualizing and editing SBOL designs.

2. Background

Humans have been engineering biology for thousands of years. Early techniques such as selective breeding and plant crossing are responsible for many of the species with which we are familiar today, such as nearly all livestock and agricultural plant species [11] [12].

Today, techniques based on molecular cloning allow us to be much more specific about the features that we do or do not change. Instead of making millions of unpredictable changes across a genome as with selective breeding, we can make just one or several specific *targeted* changes — a process known as *genetic engineering*.

Genetic engineering has become much more powerful in recent years. With technologies such as CRISPR-Cas9 and fast, affordable sequencing, we can now develop engineered biological systems of unprecedented scale and complexity [6]. This rapid increase in capability has parallels with innovations in computing in the late 20th century. Primitive, single-tasking computers became gigahertz microcomputers in a matter of decades, and electronics hobbyists gave rise to the brand new discipline of software engineering [13]. It is thought that genetic engineering is on the cusp of a similar revolution, and its answer to software engineering is *synthetic biology (SynBio)*.

2.1 Synthetic biology

In the same way that software development had to adopt engineering principles to transition from small-scale applications to critical worldwide infrastructure, it is generally accepted that the development of biological systems must adopt a similar approach to "scaling up" to unlock its full potential [14]. This approach, termed *synthetic biology*, includes principles [15] such as:

1. **Modularisation:** the decomposition of a large system into distinct modules which can be composed in different ways to build different systems. Modularisation is a common task in software engineering, where large codebases can be decomposed into smaller, re-usable libraries [16]. In a synthetic biology context, an example of modularisation would be the isolation of a promoter from a natural system so that it can be re-purposed to drive transcription in an engineered system.

2. **Standardization:** agreement on a common way of performing operations to enable interoperability [17]. In software engineering, examples of standards include file formats, so that if one software package saves a file, another can load it; and the use of standard protocols so that one piece of software can

communicate with another. An example of a synthetic biology application of standardisation is the BioBrick standard, which allows DNA to be easily digested and re-assembled in a well-defined manner.

3. **An engineering lifecycle:** the application of a pipeline of specify, design, build, test, learn. In other engineering disciplines, the development process is split into a progression of clearly defined stages [18]. Together, these stages create a lifecycle, where information learnt from one iteration can feed into the design process of the next.

2.1.1 Genetic circuits

The building blocks of a biological system — organic molecules — are also their currency, in the same way that electrons are the currency of electronic circuits. Biological systems can detect, modify, and produce molecules such as antibiotics [19], therapeutic proteins [20], and functional DNA.

One of the definitive tools in the synthetic biology arsenal is the *genetic circuit*: a logical system built from organic molecules. The implementation of genetic circuits depends on a set of natural biological processes [21]:

- Promoters are regions of DNA which initiate the transcription of downstream *coding sequences* (CDSs) to molecules of messenger RNA (mRNA), when recognised by an enzyme known as RNA polymerase
- mRNA molecules are translated into amino acid sequences by ribosomes. In prokaryotic systems, ribosomes are recruited by a ribosome binding site (RBS), a region of the mRNA between the promoter and the CDS.
- A string of amino acids (primary structure) may then fold into further structures to form a protein.
- Some proteins, known as *transcription factors* (TFs), have specific domains that enable binding to a region of DNA. Sometimes, this binding happens in the region of a promoter, and can either activate or inhibit transcription.
- Sites around a promoter where such binding can happen are known as *operators* or, more specifically, *transcription factor binding sites* (TFBSs).

Genetic circuits can be either natural, e.g. the *lac* operon for conditional activation of lactose metabolism found in many bacteria; or synthetic, e.g. engineered biological designs such as the repressilator [22] and the genetic toggle switch [23].

The *lac* operon

The *lac* operon, first documented in *E. coli* by Jacob & Monod in 1961 [24], is a widely studied example of a natural genetic circuit. The *lac* operon encodes β-galactosidase, an enzyme which can metabolise lactose. β-galactosidase is normally expressed at low levels, unless specific conditions are met: glucose must be absent AND lactose must be present.

A synthetic biologist might describe the *lac* operon as a genetic circuit device comprising several sub-systems (Fig. 2.1):

- A glucose sensor: the catabolite activator protein (CAP) binds to cyclic AMP (cAMP), a small molecule synthesized in the absence of glucose. CAP only binds to its target DNA site when in a complex with cAMP. Therefore, the transcription factor binding site for CAP-cAMP can be used as an operator to detect glucose.
- A lactose sensor: lacI is constitutively expressed, and binds to an operator site of the promoter pLac to inhibit transcription. When allolactose binds to lacI, it can no longer bind to the lacI binding site. Therefore, the transcription factor binding site for lacI can be used as an operator to detect lactose.
- A glucose and lactose sensitive promoter: pLac contains the aforementioned glucose sensor and the lactose sensor operator sites, and therefore is controlled by both the presence of lactose and the absence of glucose.

The Elowitz repressilator

Synthetic genetic circuits are typically constructed by repurposing components from natural systems. For example, lacI and pLac from the *lac* operon are well-documented as a transcription factor-promoter pair where the transcription factor represses the promoter. It is possible to extract the DNA sequence for pLac and the lacI CDS and engineer them into a different system.

Elowitz et al. (2000) pioneered the genetic “repressilator” circuit using three such pairs of transcription factors and promoters: lacI and pLac, tetR and pTet, and λ cI. In the repressilator, each transcription factor represses the promoter driving expression of the next in order to create an oscillating negative feedback loop (Fig. 2.2) [22].

The lacI/tetR toggle switch

The lacI/tetR toggle switch published by Gardner et al. in 2000 [23] is another well-known example of a synthetic genetic circuit, using lacI/pLac and tetR/pTet to create a “switch” that can be flipped on or off, therefore effectively storing 1 bit of information biologically. The switch can be toggled from an “on” state to an “off” state by adding aTc, which prevents the inhibition of pTet and allows tetR to be expressed, which represses pLac to maintain an “off” state until IPTG is added to flip the switch back to “on” (Fig. 2.3).

2.1.2 An engineering lifecycle

One of the core principles of synthetic biology is the application of a design, build, test, learn lifecycle (Fig. 2.4) common to other engineering disciplines [18]. While there are varying definitions of this lifecycle, it is generally described as comprising four stages:

- The *design stage* takes a *specification* - for example, “I want a system to detect the presence of IPTG” - and produces a *design*, for example, “I will use pLac to drive GFP expression and clone it into *E. coli*”

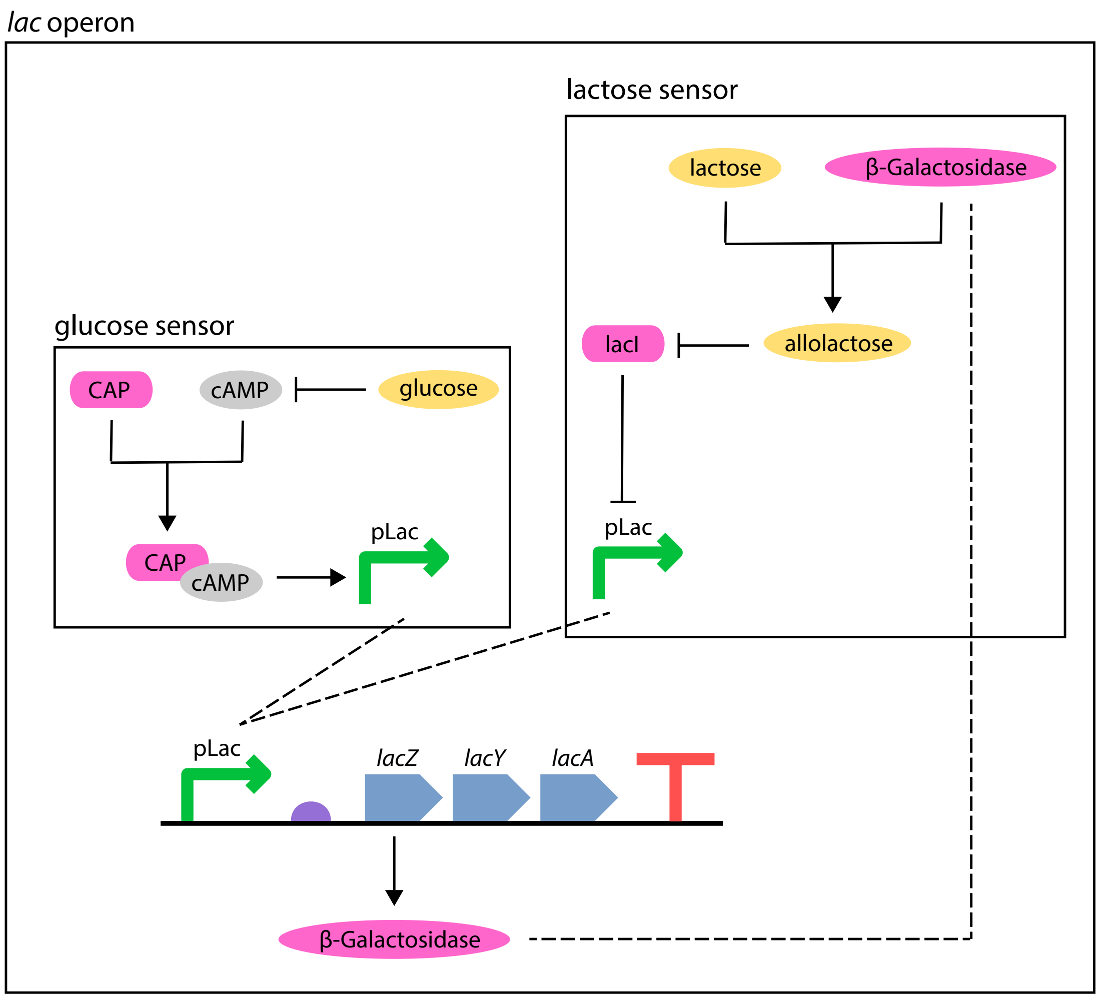


Figure 2.1: *The* lac *operon depicted as a device comprising a glucose sensor and a lactose sensor. Dashed lines indicate equivalence, arrows indicate production or induction, and flat-headed arrows indicate repression. The green "bent arrow" glyph is the SBOL Visual symbol for a promoter, and the blue pentagon glyph is the SBOL Visual symbol for a coding site (CDS).*

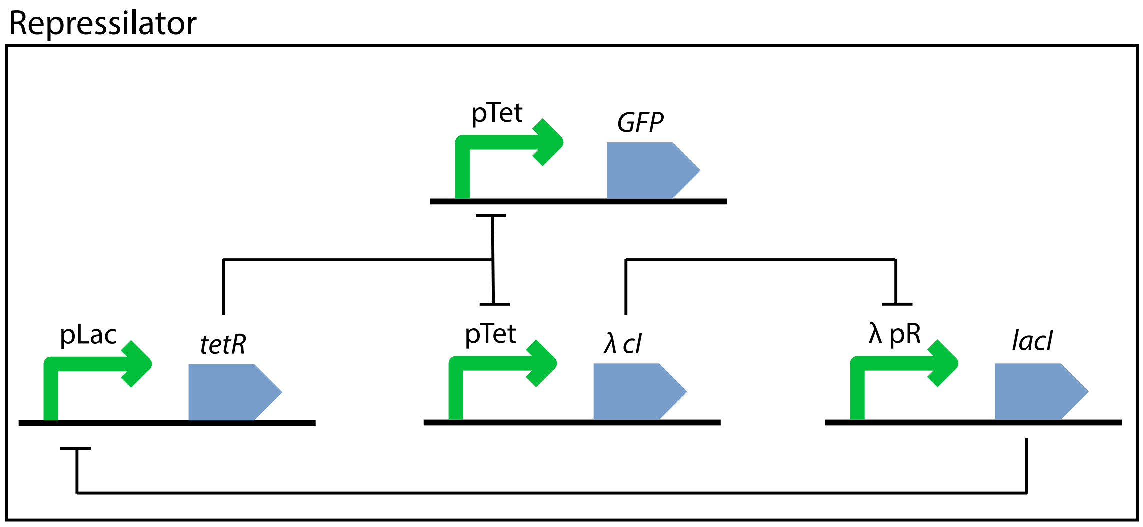


Figure 2.2: *The Elowitz repressilator circuit depicted using SBOL Visual. Each transcription factor (tetR, λ cI, and lacI) represses the promoter driving expression of the next in order to create an oscillating negative feedback loop.*

- The *implementation stage* or *build stage* takes a *design* and produces an *implementation*. In this case, we could construct the DNA and use a cloning vector to insert it into competent cells.
- The *testing stage* takes an *implementation* and produces test results. For example, using a plate reader to measure GFP fluorescence.
- The *learn stage* interprets test results to inform the design stage. Did the design work? If not, why not? How can test results be used to improve the design?

Optimisation of any stage of this lifecycle can improve turnaround for synthetic biologists. In the testing stage, for example, we now have fast and affordable sequencing technology which allows rapid validation of constructs. In the implementation stage, we now have cutting edge targeted gene editing technologies such as CRISPR-Cas9 [25]. However, the design stage — the focus of this work — has arguably been neglected in comparison to other stages of the lifecycle. SynBio design is still largely a cumbersome, manual process which requires significant domain knowledge and human intervention.

One of the problems with SynBio design is that unlike, for example, electronic circuit engineering, there is no consistently accurate way to predict the behaviour of a biological system. Modelling and simulation in SynBio are continually advancing, but they cannot yet provide the same reliability for testing designs prior to construction that is possible in other engineering disciplines [26]; while the designer of an electronic circuit can rapidly build and test prototypes using CAD software before committing to building anything physical, the designer of a biological system is still bound by the turnaround time of lab experiments. Improving the accuracy of computational models to the point where they can be used in place of experiments would make biodesign less time-consuming, and as such has been the focus of many recent efforts to improve the efficiency of the design stage, such as the SBML modelling language [27] and the BioModels repository [28].

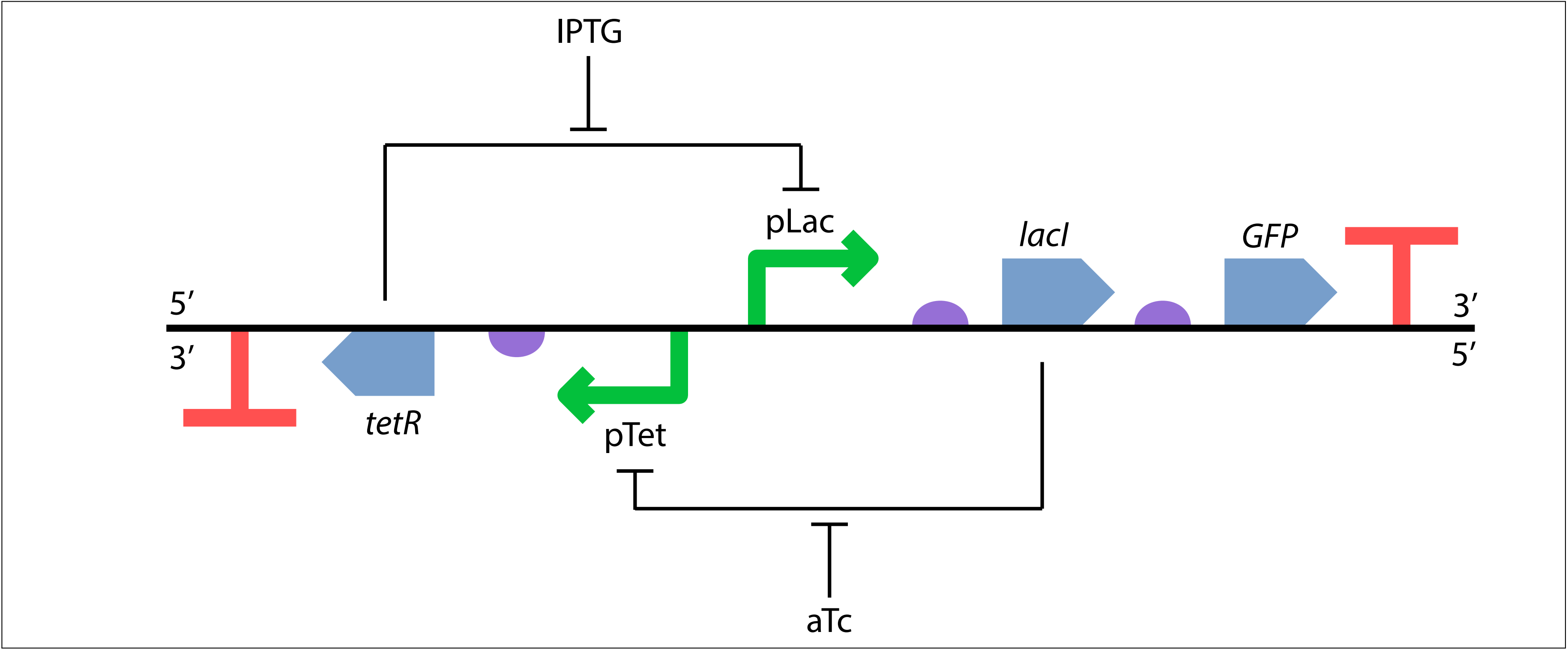


Figure 2.3: *A graphical depiction of the lacI/tetR genetic toggle switch described by Gardner et al., 2000* [23]. *The lacI transcription factor inhibits pTet in the absence of aTc, and the tetR transcription factor represses pLac in the absence of IPTG. The switch can be toggled from an "on" state to an "off" state by adding aTc, which prevents the inhibition of pTet and allows tetR to be expressed, which represses pLac to maintain an "off" state until IPTG is added to flip the switch back to "on".*

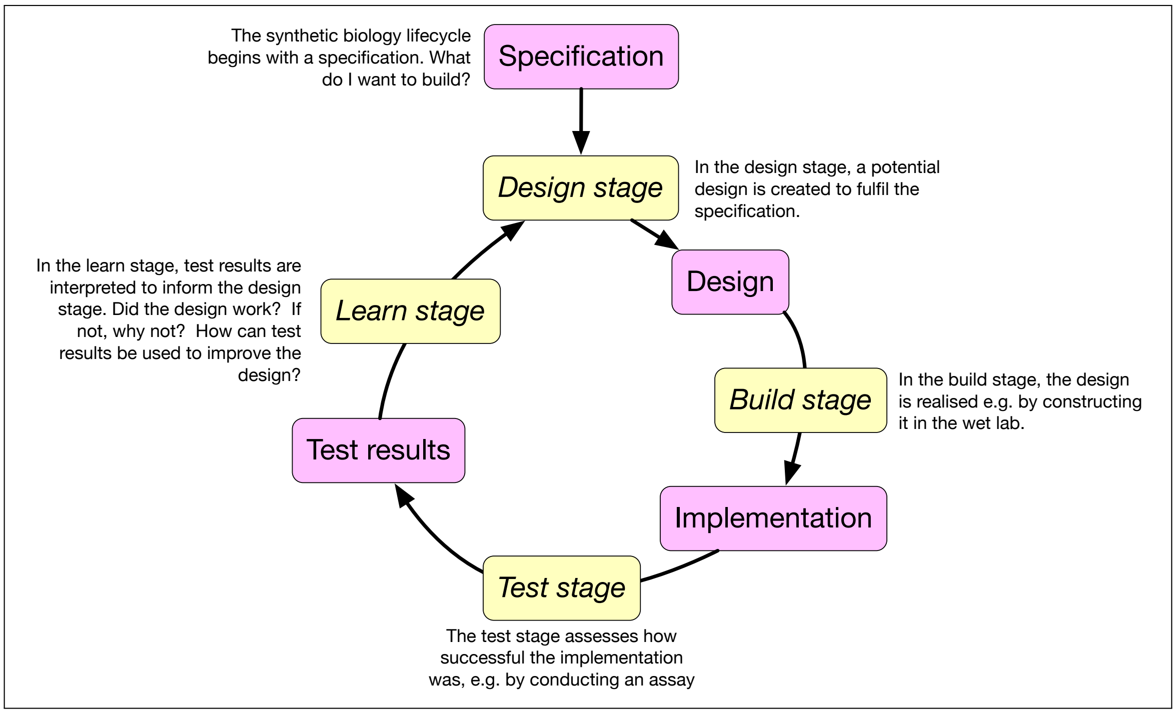

The synthetic biology lifecycle begins with a specification. What do I want to build?
Specification
Design stage
In the design stage, a potential design is created to fulfil the specification.
In the learn stage, test results are interpreted to inform the design stage. Did the design work? If not, why not? How can test results be used to improve the design?
Learn stage
Design
Build stage
In the build stage, the design is realised e.g. by constructing it in the wet lab.
Test results
Implementation
Test stage
The test stage assesses how successful the implementation was, e.g. by conducting an assay

Figure 2.4: *The Synthetic Biology lifecycle consists of four stages: design, build, test, and learn. The design stage produces a design from a specification; the build stage produces an implementation from a design; the test stage produces test results from an implementation; and the learn stage completes the lifecycle by using the test results to inform the design stage.*

Another active area of research in design optimisation is biodesign automation (BDA): the process of automatically producing designs given a specification. The annual International Workshop on Bio-Design Automation (IWBDA) [29] has been running since 2009, and regularly brings together researchers worldwide to work on BDA challenges. A notable recent BDA innovation is Cello [30], a tool which allows system descriptions written in the hardware description language Verilog (IEEE standard 1364) to be translated into genetic circuits when given a library of parts and constraints.

2.2 Data standards

There are many attributes that can be used to describe a biological design. For example, the Gardner et al. toggle switch paper [23] contains all of the following in the form of free-text and diagrams:

- *Structure* — What features does the design have, and in what order?
- *Hierarchy* — Is the design made up of smaller parts? If so, what are they?
- *Intended function* — What is the design for and how is it expected to work?
- *Sequence data* — What are the sequences (e.g. DNA or protein) that make up the design, if any?

```
>BBa_I13522 Part-only sequence (937 bp)
tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcactactagagaaagaggagaaatactagat
gcgtaaaggagaagaacttttcactggagttgtcccaattcttgttgaattagatggtgatgttaatgggcacaaattttct
gtcagtggagagggtgaaggtgatgcaacatacggaaaacttacccttaaatttatttgcactactggaaaactacctgttc
catggccaacacttgtcactactttcggttatggtgttcaatgctttgcgagatacccagatcatatgaaacagcatgactt
tttcaagagtgccatgcccgaaggttatgtacaggaaagaactatatttttcaaagatgacgggaactacaagacacgtgct
gaagtcaagtttgaaggtgatacccttgttaatagaatcgagttaaaaggtattgattttaaagaagatggaaacattcttg
gacacaaattggaatacaactataactcacacaatgtatacatcatggcagacaaacaaaagaatggaatcaaagttaactt
caaaattagacacaacattgaagatggaagcgttcaactagcagaccattatcaacaaaatactccaattggcgatggccct
gtccttttaccagacaaccattacctgtccacacaatctgccctttcgaaagatcccaacgaaaagagagaccacatggtcc
ttcttgagtttgtaacagctgctgggattacacatggcatggatgaactatacaaataataatactagagccaggcatcaaa
taaaacgaaaggctcagtcgaaagactgggcctttcgttttatctgttgtttgtcggtgaacgctctctactagagtcacac
tggctcaccttcgggtgggcctttctgcgtttata
```

Figure 2.5: *FASTA files consist of a header, beginning with a > character and followed by a free-text description; and a sequence which spans the subsequent line or lines. Multiple FASTA entries can be combined into the same file by simple concatenation, though tool support for multiple sequences is not universal.*

- *Characterisation data* - How does the design function in a particular host?
- *Metadata* — What is the design called? Does it have a description?
- *Provenance* — Who made the design, why did they make it, what did they make it from, and when did they make it?

Unfortunately, the *computational* representation of a design usually consists of a very small subset of these attributes [8]. For example, a design could be represented in FASTA [31] format, containing only sequence data and minimal metadata (Fig. 2.5). GenBank [32][33] format is a slight improvement, as it has sequence annotations which capture structure (Fig. 2.6), but crucial information such as provenance is still omitted. Furthermore, neither FASTA or GenBank allow for the specification of an abstract design *without* sequences, which is necessary in synthetic biology to represent the conception of a design (e.g. "a genetic toggle switch") prior to its realisation for a specific host (e.g. "a LacI/TetR genetic toggle switch optimised for *E. coli*").

The reason for these limitations is a common theme in this work. Standards such as FASTA and GenBank were created for the annotation of naturally occurring systems, not engineered ones. While some of the information — such as sequence information — is common to both natural and engineered parts, it is clear that a different data standard is required to truly capture the attributes of a synthetic design. The replacement of these archaic, impenetrable text-based file formats with modern, machine-tractable data standards could make the design process amenable to modelling and automation, and make knowledge about existing and future designs more discoverable and easier to document in a standardized manner.

2.2.1 The Resource Description Framework (RDF)

Though biology still makes extensive use of text-based file formats such as FASTA and GenBank, the rest of computing has largely moved on. Text-based formats are often both syntactically and semantically ambiguous, and the implementation of bespoke parsers and formatters for such formats is error-prone.

```
LOCUS       BBa_I13522               937 bp    DNA     linear   UNK 29-May-2019
DEFINITION  pTet GFP
ACCESSION   BBa_I13522
VERSION     BBa_I13522.1
FEATURES             Location/Qualifiers
     promoter        1..54
     RBS             63..74
     CDS             81..800
     terminator      809..888
     terminator      897..937
ORIGIN
        1 tccctatcag tgatagagat tgacatccct atcagtgata gagatactga gcactactag
       61 agaaagagga gaaatactag atgcgtaaag gagaagaact tttcactgga gttgtcccaa
      121 ttcttgttga attagatggt gatgttaatg ggcacaaatt ttctgtcagt ggagagggtg
      181 aaggtgatgc aacatacgga aaacttaccc ttaaatttat ttgcactact ggaaaactac
      241 ctgttccatg gccaacactt gtcactactt tcggttatgg tgttcaatgc tttgcgagat
      301 acccagatca tatgaaacag catgactttt tcaagagtgc catgcccgaa ggttatgtac
      361 aggaaagaac tatatttttc aaagatgacg ggaactacaa gacacgtgct gaagtcaagt
      421 ttgaaggtga tacccttgtt aatagaatcg agttaaaagg tattgatttt aaagaagatg
      481 gaaacattct tggacacaaa ttggaataca actataactc acacaatgta tacatcatgg
      541 cagacaaaca aaagaatgga atcaaagtta acttcaaaat tagacacaac attgaagatg
      601 gaagcgttca actagcagac cattatcaac aaaatactcc aattggcgat ggccctgtcc
      661 ttttaccaga caaccattac ctgtccacac aatctgccct ttcgaaagat cccaacgaaa
      721 agagagacca catggtcctt cttgagtttg taacagctgc tgggattaca catggcatgg
      781 atgaactata caaataataa tactagagcc aggcatcaaa taaaacgaaa ggctcagtcg
      841 aaagactggg cctttcgttt tatctgttgt ttgtcggtga acgctctcta ctagagtcac
      901 actggctcac cttcgggtgg gcctttctgc gtttata
//
```

Figure 2.6: *Unlike FASTA, the GenBank format allows the promoter, RBS, CDS, and terminator regions of the DNA to be annotated as features.*

There are a range of modern solutions for general data representation, with varying degrees of semantic expressiveness and machine-tractability. On a basic level, the issue of ambiguous syntax can be solved by using a format such as JSON [34] or XML [35], and various standards such as JSON Schema [36] and XML Schema [37] exist to add semantics to such formats while retaining the traditional "tree-based" model of a document containing data fields. An example of the application of such an approach to a biological dataset is the UniProt XML API [38]: a record for a protein can be retrieved as an XML document which can be interpreted by any XML parser, and its semantics validated using the UniProt XML schema. While this approach is more formalised than the ad-hoc text-based formats that came before, it is still a reductive knowledge representation. A UniProt protein is not just a document; it is a grouping of knowledge about a particular resource which is part of a much larger web of information.

The Resource Description Framework (RDF) [39] (Fig. Fig. 2.7) is a simple model formalized by the World Wide Web Consortium (W3C) to describe named properties and their values. The core principles of RDF are straightforward:

- In order to describe a resource — which could be anything from a book to a bacterium — it first needs to be assigned an identifier. The Web already has a system for identifiers called URIs, which are a superset of URLs used by browsers.
- To assign a property to a resource, the property is also given an identifier URI. There are many published collections of properties, known as *vocabularies*.
- Finally, the property also needs a value object. This object could be a string (i.e.

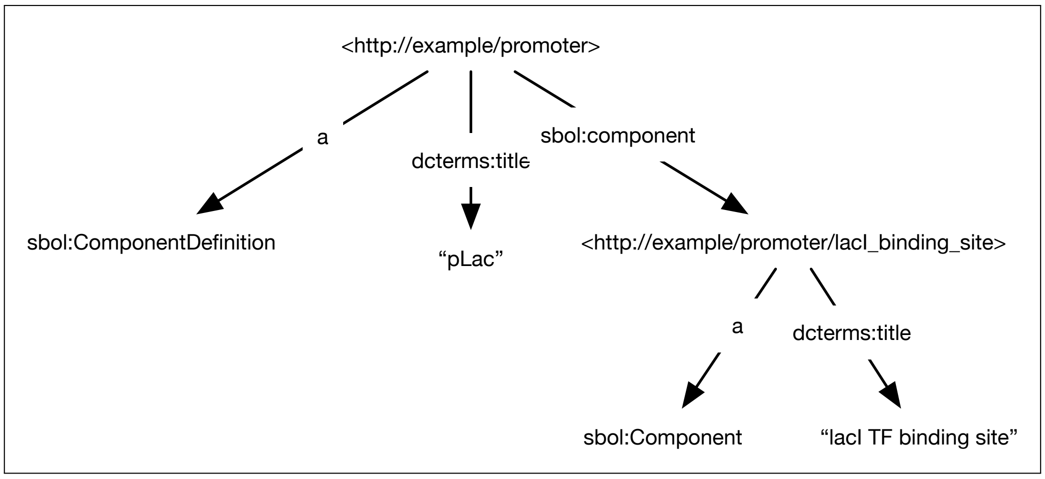


Figure 2.7: *An example of an RDF graph capturing knowledge about the pLac promoter. Each edge in the graph is an RDF predicate connecting a subject to an object. The subject, predicate, and object together forms a triple, for example `<http://example/promoter> dcterms:title "pLac"`.*

some text), a number, or even the identifier of another resource. This triad of subject, predicate, object is known as a *triple*.

- As the subject and the object of a triple may both themselves be the URIs of resources, it is possible to link two resources together. This networking of resources by triples forms a directed graph.

One of the key features of an RDF graph-based knowledge representation is that both the nodes and the edges have an identity. As these identities can be any valid URI, it is helpful if providers of RDF data agree on common vocabularies so that knowledge from different datasets is semantically compatible. For example, the Dublin Core Metadata Initiative specifies that the URI `http://purl.org/dc/terms/title` (usually abbreviated to the prefixed from `dcterms:title`) can be used to specify "a name given to the resource" [40]. Regardless of whether the resource is a book in a library, a film, or a biological part, if it has an edge with the URI `dcterms:title`, one navigating the graph knows that the edge points to its name.

The ability to use any URI as the identity of an edge also allows for highly complex and specific vocabularies, known as *ontologies* [41]. Such vocabularies are particularly useful in a biological dataset, where in addition to having typical properties such as a title and description, a resource might also have properties such as "molecular structure" or "melting point". The use of ontologies is well-established in biology with projects such as the Gene Ontology (GO) [42] and the Sequence Ontology (SO) [43].

The suitability of ontology-backed RDF graph representations for biological data is widely acknowledged in the bioinformatics community, with some of the oldest and largest biological datasets such as UniProt and PubChem publishing official RDF versions [44] [45]. The European Bioinformatics Institute (EBI) recently launched a unified RDF platform integrating datasets such as BioModels, ChEMBL, and Ensembl, citing the need to "better serve complex research questions across resources" [46].

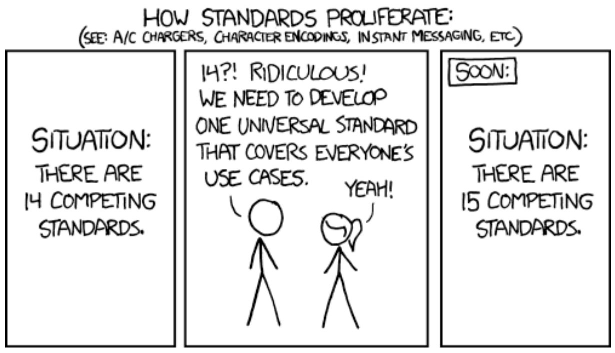

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 2.8: *Cartoon by Randall Monroe of XKCD fame, common to every set of slides at any conference concerning standards development. CC-BY-NC 2.5* `https://creativecommons.org/licenses/by-nc/2.5/`

2.2.2 The Synthetic Biology Open Language (SBOL)

The Synthetic Biology Open Language (SBOL) is an RDF vocabulary specifically for the for the representation of synthetic biology designs [47]. Like FASTA and GenBank, it can be used to describe sequences and their features. However, SBOL can also capture essential *design* information, including compositional hierarchy, intended function, provenance, and experimental data.

In order to avoid the situation captured brilliantly by cartoonist Randall Monroe (Fig. 2.8), SBOL does not attempt to re-invent the wheel, instead re-using existing standards where possible. SBOL uses existing ontologies such as dcterms for metadata, the sequence ontology for functional assignments, and Prov-O for provenance (Fig. 2.9).

The first version of SBOL ("SBOL1") was described in its 2011 BBF RFC as enabling "the electronic exchange of information describing DNA components used in synthetic biology" [52]. SBOL1 provides a straightforward progression from non-standardized, text-based formats such as GenBank, with the notable addition of compositional hierarchy. SBOL1 has a simple set of four data objects (or "classes") that can be used to capture data about a part:

- The `DnaComponent` object is used to represent a biological part. It has a name, description, a type taken from the Sequence Ontology (for example, `SO:0000167` for "promoter"), an optional associated `DnaSequence` object, zero or more associated `SequenceAnnotation` objects.
- The `DnaSequence` object is used to specify a DNA sequence. It has a `nucleotides` property which stores the sequence as a string, much as a FASTA file would.
- The `SequenceAnnotation` object is used to annotate a region of a `DnaSequence`. It specifies the start and end position of the region in base pairs; whether the annotation is on the forward or reverse strand; and, optionally, links to another `DnaComponent` to specify that the annotation is composition of another part.

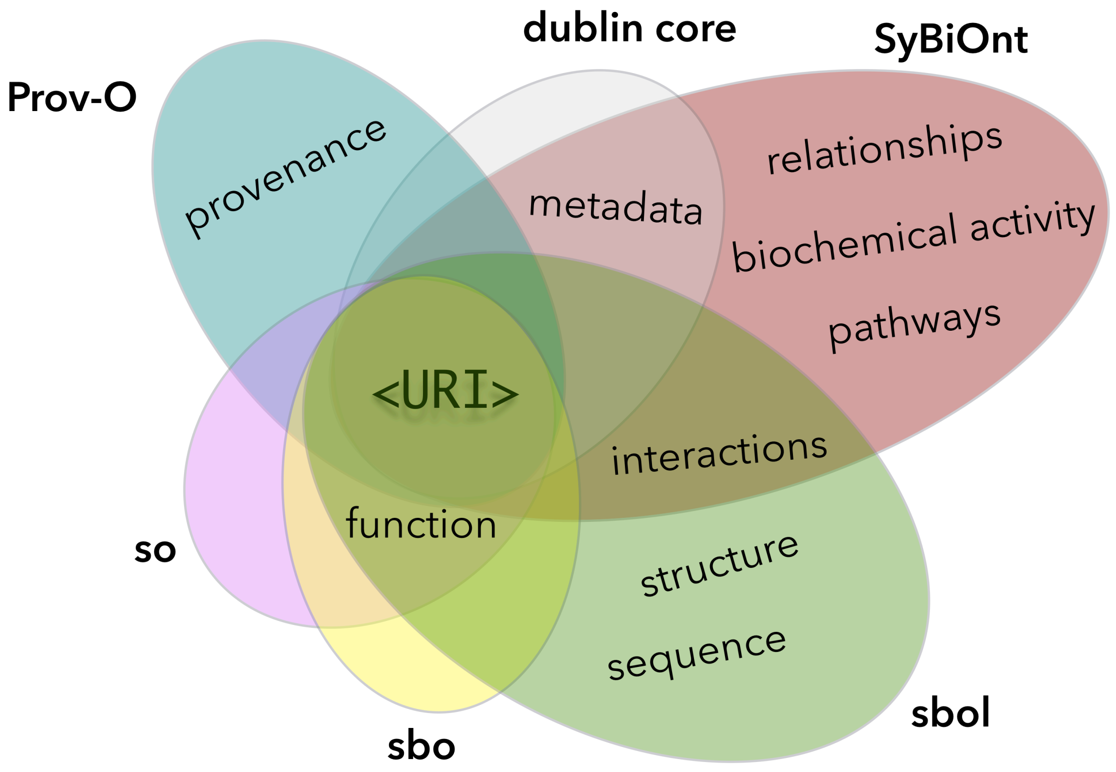


Figure 2.9: *Examples of ontologies that can be used to describe biological parts. The part is represented by a URI (center). SBOL can be used to describe the structure and sequence of a part. The Sequence Ontology (SO)* [43]*, and Systems Biology Ontology (SBO)* [48]*, can be used to annotate the part with functional assignments. The Provenance Ontology (Prov-O)* [49] *can be used to describe "who, what, how, when" the part description was produced. Dublin Core* [50] *annotations are used to add metadata, such as a name and description. Finally, the SyBiOnt ontology* [51] *enables enriched annotations of biochemical activity, pathways, and further information about interactions.*

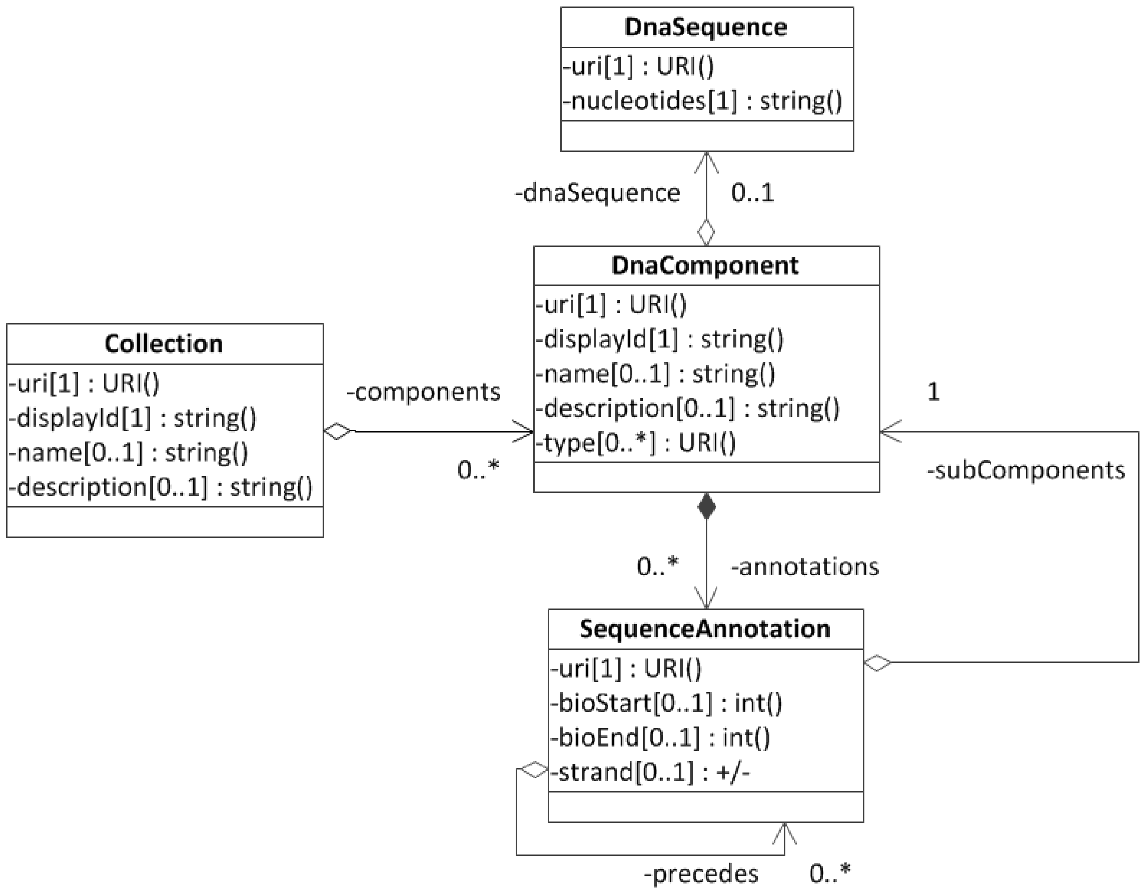


Figure 2.10: *A UML diagram of the SBOL1 data model, showing the relationship between its four classes:* `DnaComponent`*,* `DnaSequence`*,* `SequenceAnnotation`*, and* `Collection`*. Source: BBF RFC 84:* Synthetic Biology Open Language (SBOL) Version 1.0.0 *(Galdziki et al.)*

	FASTA	GenBank	SBOL1	SBOL2
Structure	✗	✓	✓	✓
Hierarchy	✗	✗	✓	✓
Intended function	✗	✗	✗	✓
Sequence data	✓	✓	✓	✓
Characterisation data	✗	✗	✗	✓
Metadata	✓	✓	✓	✓
Provenance	✗	✗	✗	✓

Table 2.1: *Assessment of the ability of different data standards to represent the list of design attributes described in section 2.2.*

- The `Collection` object allows multiple `DnaComponent` objects to be grouped together. For example, a library of promoters could have a corresponding `Collection`.

The use of SBOL1 has a number of advantages over text-based formats. SBOL1 has a well-specified RDF/XML representation, which can be interpreted by any RDF or XML software library — solving the potential ambiguities of parsing tabulated formats such as GenBank. SBOL1 also mandates the use of well-defined terms from the Sequence Ontology to specify the type of components instead of ambiguous free text labels. Most importantly, SBOL1 allows a `DnaComponent` to be composed of multiple, smaller `DnaComponent` definitions, enabling the fundamental synthetic biology concept of composition.

Version 2.0 of SBOL ("SBOL2") was released in 2015. In SBOL2, the concept of a `DnaComponent` was replaced with a more generalised `ComponentDefinition`, which means that in addition to DNA, SBOL2 can represent any possible component of a biological system including proteins, RNAs, and small molecules (Fig. 2.12).

SBOL2 also introduced `ModuleDefinition`, an object intended to represent more abstract, "functional" relationships. While, like `ComponentDefinition`, a `ModuleDefinition` can contain instantiations of components, it cannot have a sequence. A `ModuleDefinition` can specify *interactions* between its component instances, such as a CDS encoding a protein or two proteins forming a complex [1].

Because the SBOL specification is built on RDF, all SBOL data is implicitly also an RDF graph. This fact that SBOL can be viewed as a knowledge graph is not immediately obvious from either the SBOL specification or from the software implementation of SBOL in libSBOLj. This is likely because the recommended serialization of SBOL is a specific subset of RDF/XML, a method of serializing RDF triples using XML syntax [53]. While SBOL can be deserialized relatively trivially using an RDF/XML parser, its serialization is slightly more complicated; RDF/XML has both a nested and flat form, and the SBOL

[1]The issue of when to use a `ComponentDefinition` and when to use a `ModuleDefinition` remains a point of contention within the SBOL community. Proposals to resolve this by merging the two into a unified object are detailed later in chapter 3

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix igem: <http://parts.igem.org/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix sbol: <http://sbols.org/v2#> .
@base <http://parts.igem.org/Part:BBa_I13522> .

<http://parts.igem.org/Part:BBa_I13522> a <http://sbols.org/v2#ComponentDefinition> ;
  dcterms:title "BBa_I13522" ;
  dcterms:description "pTet GFP" ;
  dcterms:created "2005-06-30T11:00:00Z" ;
  dcterms:modified "2015-08-31T04:07:35Z" ;
  igem:partStatus "Released HQ 2013" ;
  igem:discontinued "false" ;
  igem:dominant "true" ;
  igem:sampleStatus "In stock" ;
  dc:creator "jkm" ;
  sbol:type <http://www.biopax.org/release/biopax-level3.owl#DnaRegion> ;
  sbol:role <http://identifiers.org/so/SO:0000804> ;
  sbol:sequence <#sequence> ;
  sbol:sequenceAnnotation <#promoterRegion>, <#rbsRegion>, <#cdsRegion>, <#terminator1Region>, <#terminator2Region> ;
  sbol:component <#promoter>, <#rbs>, <#cds>, <#terminator1>, <#terminator2> .

<#promoter> a sbol:Component;  sbol:definition <http://parts.igem.org/Part:BBa_R0040>;  sbol:access sbol:public .
<#rbs>  a sbol:Component;  sbol:definition <http://parts.igem.org/Part:BBa_E0040>;  sbol:access sbol:public .
<#cds>  a sbol:Component;  sbol:definition <http://parts.igem.org/Part:BBa_B0034>;  sbol:access sbol:public .
<#terminator1> a sbol:Component;  sbol:definition <http://parts.igem.org/Part:BBa_B0012>;  sbol:access sbol:public .
<#terminator2> a sbol:Component;  sbol:definition <http://parts.igem.org/Part:BBa_B0010>;  sbol:access sbol:public .

<#promoterRegion> a sbol:SequenceAnnotation;    sbol:location <#promoterLocation>; sbol:component <#promoter> .
<#rbsRegion>  a sbol:SequenceAnnotation;    sbol:location <#rbsLocation>;  sbol:component <#rbs> .
<#cdsRegion>  a sbol:SequenceAnnotation;    sbol:location <#cdsLocation>;  sbol:component <#cds> .
<#terminator1Region> a sbol:SequenceAnnotation;    sbol:location <#terminator1Location>; sbol:component <#terminator1> .
<#terminator2Region> a sbol:SequenceAnnotation;    sbol:location <#terminator2Location>; sbol:component <#terminator2> .

<#promoterLocation> a sbol:Range; sbol:start "1";     sbol:end "54";    sbol:orientation sbol:inline .
<#rbsLocation>  a sbol:Range; sbol:start "63";    sbol:end "74";    sbol:orientation sbol:inline .
<#cdsLocation>  a sbol:Range; sbol:start "81";    sbol:end "800";   sbol:orientation sbol:inline .
<#terminator1Location> a sbol:Range; sbol:start "809";   sbol:end "888";   sbol:orientation sbol:inline .
<#terminator2Location> a sbol:Range; sbol:start "897";   sbol:end "937";   sbol:orientation sbol:inline .

<#sequence> a sbol:Sequence ;
  sbol:encoding <http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html> ;
  sbol:elements """tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcactactagagaaagaggagaaatactagatgcgtaaaggagaagaactttt
cactggagttgtcccaattcttgttgaattagatggtgatgttaatgggcacaaattttctgtcagtggagagggtgaaggtgatgcaacatacggaaaacttacccttaaatttatttgca
ctactggaaaactacctgttccatggccaacacttgtcactactttcggttatggtgttcaatgctttgcgagatacccagatcatatgaaacagcatgactttttcaagagtgccatgccc
gaaggttatgtacaggaaagaactatatttttcaaagatgacgggaactacaagacacgtgctgaagtcaagtttgaaggtgatacccttgttaatagaatcgagttaaaaggtattgattt
taaagaagatggaaacattcttggacacaaattggaatacaactataactcacacaatgtatacatcatggcagacaaacaaaagaatggaatcaaagttaacttcaaaattagacacaaca
ttgaagatggaagcgttcaactagcagaccattatcaacaaaatactccaattggcgatggccctgtccttttaccagacaaccattacctgtccacacaatctgccctttcgaaagatccc
aacgaaaagagagaccacatggtccttcttgagtttgtaacagctgctgggattacacatggcatggatgaactatacaaataataatactagagccaggcatcaaataaaacgaaaggctc
agtcgaaagactgggcctttcgttttatctgttgtttgtcggtgaacgctctctactagagtcacactggctcaccttcgggtgggcctttctgcgtttata""" .
```

Figure 2.11: *The iGEM part BBa_I13522 represented in SBOL2 using Turtle serialization. The features are not only annotated but linked to other parts using* `sbol:definition`*, which enables composition. Additional namespaces (in this case,* `igem:`*) can be used to add information which is not part of the core SBOL data model. Features introduced in SBOL2 such as the ability to represent proteins and interactions are not used in this example because the iGEM part does not provide such information, but it would be possible to add them to provide a richer view of the part.*

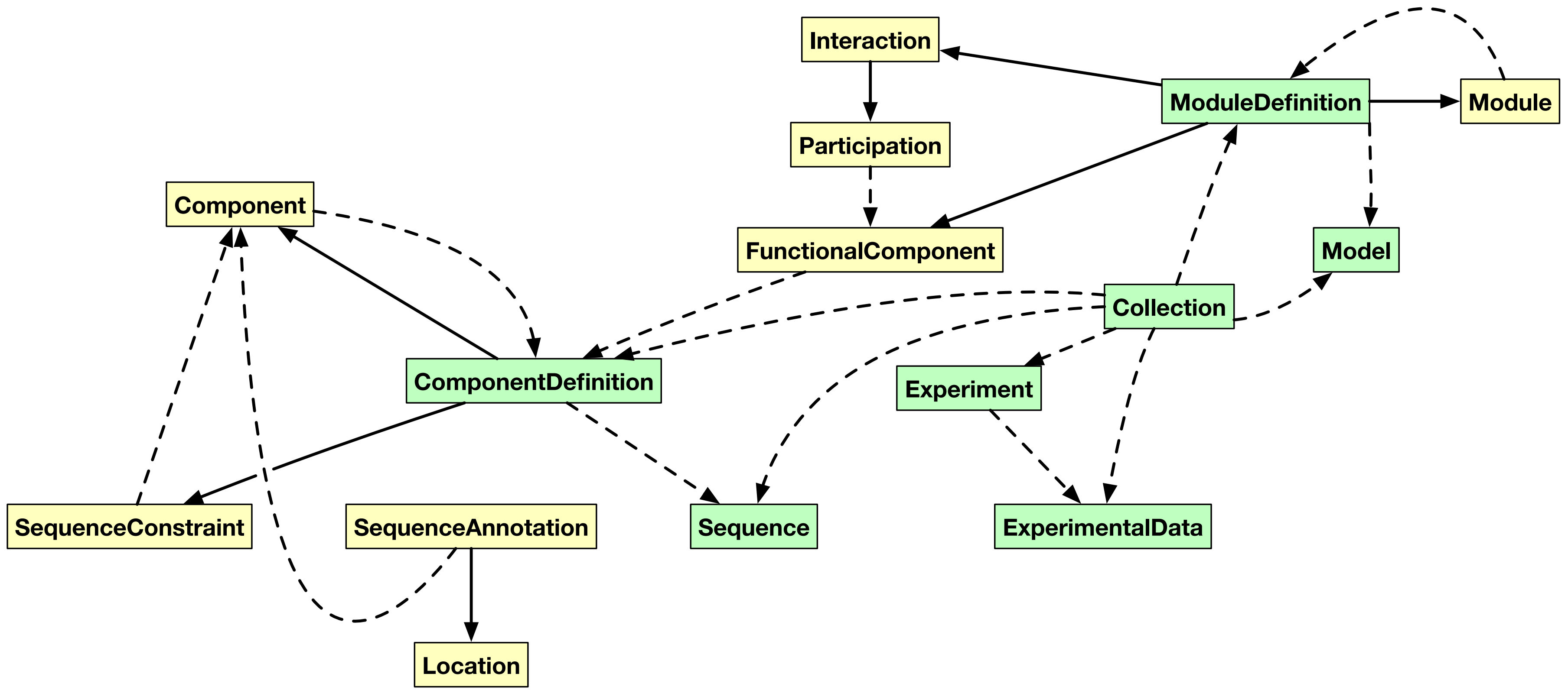


Figure 2.12: *The Synthetic Biology Open Language (SBOL) version 2 data model. Top-level objects are shown in green. Compared to version 1,* `DnaComponent` *has been generalized into* `ComponentDefinition` *to allow the representation of non-DNA molecules; the parallel* `ModuleDefinition` *data model has been added to introduced to allow the representation of functional relationships; and, recently, support for experimental data has been added.*

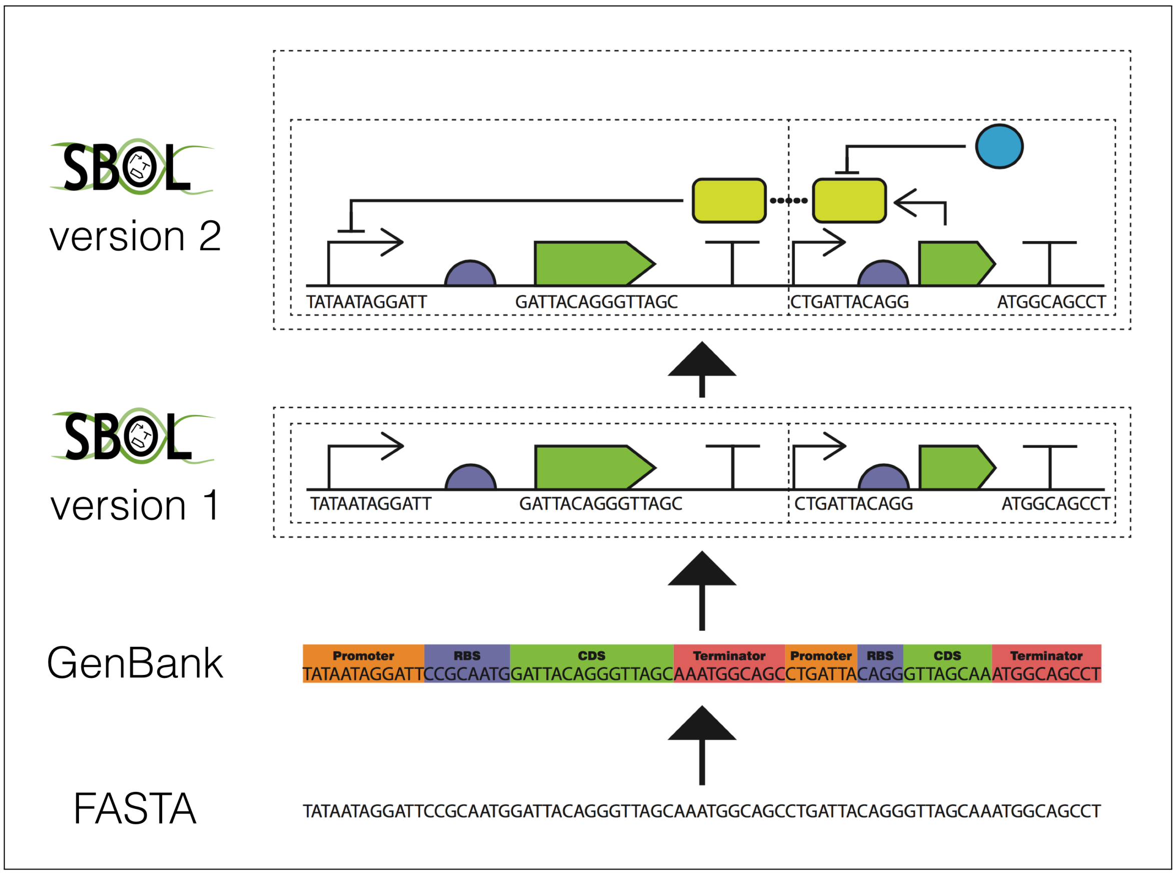
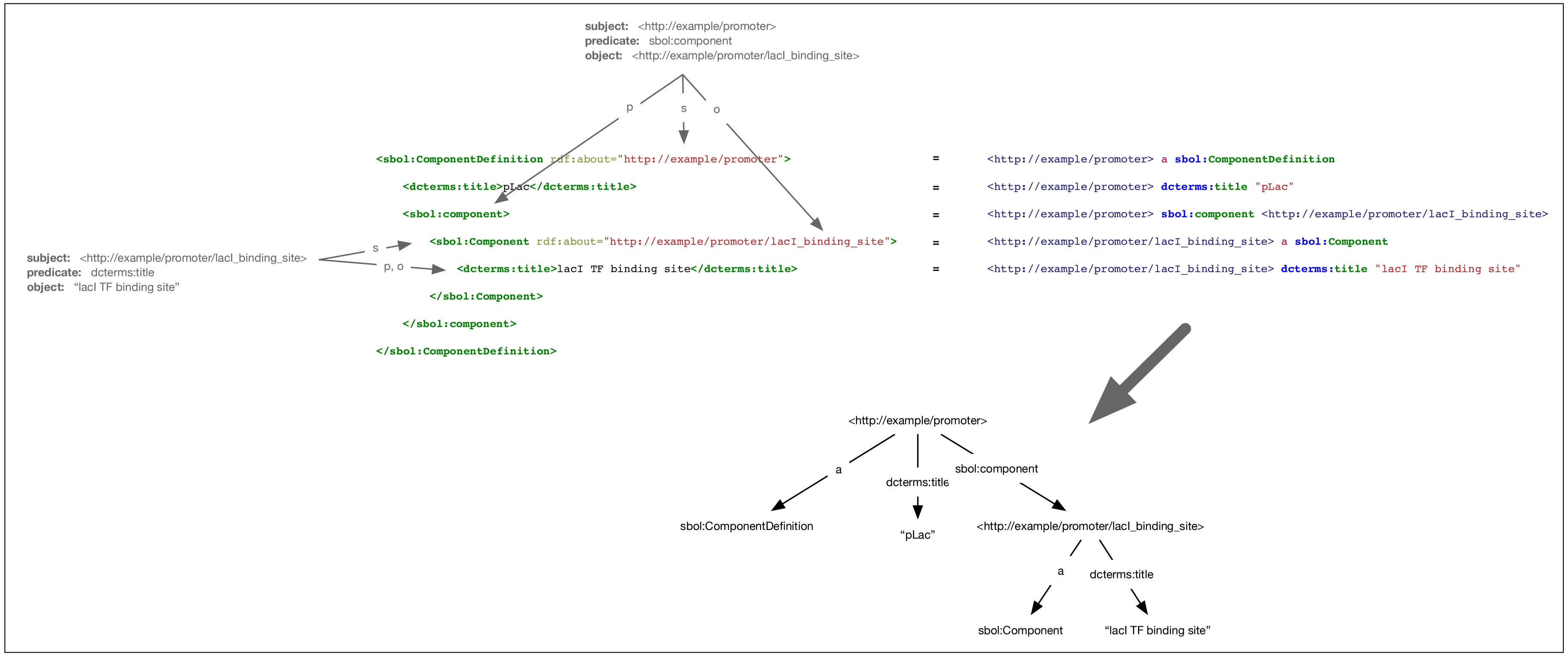


Figure 2.13: *A comparison of FASTA, GenBank, SBOL1, and SBOL2. FASTA represents only a sequence. GenBank represents a sequence with annotations. SBOL version 1 adds hierarchy and the functional assignment using ontology terms. SBOL version 2 adds interactions. Figure credit: The SBOL community*

specification requires a specific hybrid of the two despite their equivalence as specified by the RDF/XML specification (Fig. 2.15).

SBOL is slowly gaining traction in the synthetic biology community. It is now recommended by the ACS Synthetic Biology journal [54], and supported as a data exchange format by some existing tools such as Benchling [55]. However, there remain few examples of SBOL "in the wild", likely due to a number of issues discussed later in this work.



```
<sbol:ComponentDefinition rdf:about="http://example/promoter">
    <dcterms:title>pLac</dcterms:title>
    <sbol:component>
        <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
            <dcterms:title>lacI TF binding site</dcterms:title>
        </sbol:Component>
    </sbol:component>
</sbol:ComponentDefinition>
```

```
= <http://example/promoter> a sbol:ComponentDefinition
= <http://example/promoter> dcterms:title "pLac"
= <http://example/promoter> sbol:component <http://example/promoter/lacI_binding_site>
= <http://example/promoter/lacI_binding_site> a sbol:Component
= <http://example/promoter/lacI_binding_site> dcterms:title "lacI TF binding site"
```

subject: <http://example/promoter>
predicate: sbol:component
object: <http://example/promoter/lacI_binding_site>
p
s
o
subject: <http://example/promoter/lacI_binding_site>
predicate: dcterms:title
object: "lacI TF binding site"
s
p, o
<http://example/promoter>
a
dcterms:title
sbol:component
sbol:ComponentDefinition
"pLac"
<http://example/promoter/lacI_binding_site>
a
dcterms:title
sbol:Component
"lacI TF binding site"

Figure 2.14: *The same fragment of SBOL data represented in RDF/XML, RDF triples, and as a graph. The fact that the RDF/XML corresponds to an RDF graph is not immediately obvious if unfamiliar with RDF concepts, so it is unsurprising that existing implementations interpret SBOL as structured data using an XML parser rather than taking advantage of its graph representation.*

RDF+XML as required by SBOL specification

```
<sbol:ComponentDefinition rdf:about="http://example/promoter">
    <dcterms:title>pLac</dcterms:title>
    <sbol:component>
        <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
            <sbol:definition rdf:resource="http://example/lacI_binding_site" />
            <dcterms:title>lacI binding site subcomponent</dcterms:title>
        </sbol:Component>
    </sbol:component>
</sbol:ComponentDefinition>
<sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
    <dcterms:title>lacI binding site</dcterms:title>
</sbol:ComponentDefinition>
```

Equivalent RDF+XML rejected by SBOL specification

```
<sbol:ComponentDefinition rdf:about="http://example/promoter">
    <dcterms:title>pLac</dcterms:title>
    <sbol:component rdf:resource="http://example/promoter/lacI_binding_site"/>
</sbol:ComponentDefinition>
<sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
    <sbol:definition rdf:resource="http://example/lacI_binding_site"/>
    <dcterms:title>lacI binding site subcomponent</dcterms:title>
</sbol:Component>
<sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
    <dcterms:title>lacI binding site</dcterms:title>
</sbol:ComponentDefinition>
```

```
<sbol:ComponentDefinition rdf:about="http://example/promoter">
    <dcterms:title>pLac</dcterms:title>
    <sbol:component>
        <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
            <sbol:definition rdf:resource="http://example/lacI_binding_site"/>
            <dcterms:title>lacI binding site subcomponent</dcterms:title>
        </sbol:Component>
    </sbol:component>
</sbol:ComponentDefinition>
<sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
    <dcterms:title>lacI binding site</dcterms:title>
</sbol:ComponentDefinition>
```

```
<sbol:ComponentDefinition rdf:about="http://example/promoter">
    <dcterms:title>pLac</dcterms:title>
    <sbol:component>
        <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
            <sbol:definition>
                <sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
                    <dcterms:title>lacI binding site</dcterms:title>
                </sbol:ComponentDefinition>
            </sbol:definition>
            <dcterms:title>lacI binding site subcomponent</dcterms:title>
        </sbol:Component>
    </sbol:component>
</sbol:ComponentDefinition>
```

Figure 2.15: *While SBOL is serialized using syntax defined by the RDF/XML specification, it requires a highly specific subset where definitions of certain classes are nested while others are not. In RDF/XML, all four of these examples are different representations of identical data; but in SBOL, only the left example is considered valid.*

2.3 Data integration

“There are very few molecular operations that you understand in the way that you understand a wrench or a screwdriver or a transistor”

— *Rob Carlson, Biodesic* [56]

One of the challenges in optimising the design stage is that the necessary information to design a biological system is often not easily accessible. This issue of access to existing knowledge is an established problem in computer science. The field of *data integration* is defined as a “set of techniques that enable building systems geared for flexible sharing and integration of data across multiple autonomous data providers” [7, p. 1]. Essentially, how can value be extracted from a set of datasources that are not necessarily aligned in location, syntax, or semantics?

In synthetic biology, these datasources can be divided into two categories: those that provide traditional bioinformatics knowledge (i.e. information about natural organisms), and those that provide synthetic biology knowledge (i.e. information about engineered parts). While there have been numerous efforts to integrate bioinformatics knowledge [57], data integration efforts for knowledge about synthetic biology parts have been few and far between.

SynBioMine [58] is an instance of the InterMine [59] data warehousing software specifically for synthetic biology, but mostly integrates data from a number of prokaryotic genomes in the *Bacillus* and *Escherichia* families, rather than integrating information about engineered parts. While it does contain information about parts, this comes exclusively from the SynBIS database [60].

SynBIS, or “the Synthetic Biology Information System”, was developed at the Centre for Synthetic Biology and Innovation (CSynBI) at Imperial College to “support its multistage characterisation pipeline of biological parts and disseminate its results” [61]. SynBIS indeed provides access to a database of parts available in SBOL format, each of which is associated with extensive characterisation data. The data themselves are useful, but it is not clear that SynBIS can provide a general solution for integrating data in the synthetic biology community. While the data provided by SynBIS are open, they appear to come exclusively from the CSynBI characterisation pipeline described in the SynBIS paper: there are no means for a user outside of the project to submit new information. Furthermore, the code behind the SynBIS software has not been released, which means that it would not be possible for a user to start an independent instance.

More generally, data integration encompasses a broad range of techniques. Harmonization using data standards, as described in section 2.2, is one such technique, used to overcome the challenge of semantic and syntactic heterogeneity. Another challenge in data integration is the aggregation of data from distributed datasources, a problem linked to the fundamentals of how data is stored and made available for querying.

2.3.1 Relational databases and their limitations

In 1970, Edgar F. Codd published the classic paper *A Relational Model of Data for Large Shared Data Banks* [62]. At the time, data was typically formatted as structured files, which meant that the application had to be aware of the internal representation of the data in order to process it. To address this problem, Codd defined a relational view of data derived from mathematical relation theory.

Most of the popular databases — such as MySQL and PostgreSQL — are derived from Codd's work, and are termed *relational databases*. In relational databases, data are stored in tables with rows and columns, where each row represents a record, and each column represents a property of that record. A simple biological example could be a table of proteins, with each row representing a protein, and columns for its identifier, name, and amino acid sequence.

In relational databases, multiple tables are often linked together. For example, a table of protein domains could also be defined, with columns for the identifier of the protein, an identifier for the domain, and a link to a term from the gene ontology. The occurrence of the same protein identifiers across both tables creates a one-to-many relationship between proteins and domains, allowing questions such as "which domains does protein X have" to be answered with a simple SQL query such as:

```
SELECT  domain.term
    WHERE   protein.name  =  "X"
    AND     protein.id = domain.protein_id
```

In this example, it may also be useful to link a domain to multiple proteins in order to answer questions such as "which proteins have domain Y" — creating a many-to-many relationship, rather than a one-to-many. This is also achievable in a relational database by removing the protein identifier from the domain table (so that each domain is no longer linked directly to a protein), and creating a third "protein has domain" association table with two columns: one for the identifier of the protein, and one for the identifier of the domain. We can still answer "which domains does protein X have" by retrieving the rows corresponding to the protein from the "protein has domain" table and looking up the resulting domain identifiers in the domains table, or:

```
SELECT domain.term
    WHERE   protein.name = "X"
    AND protein_has_domain.protein_id = protein.id
    AND protein_has_domain.domain_id = domain.id
```

Additionally, we can also now answer "which proteins have domain Y" using the same approach with domain identifiers and the protein table, or:

```
SELECT protein.name
    WHERE protein_has_domain.protein_id = protein.id
    AND domain.name = "Y"
    AND protein_has_domain.domain = domain.id
```

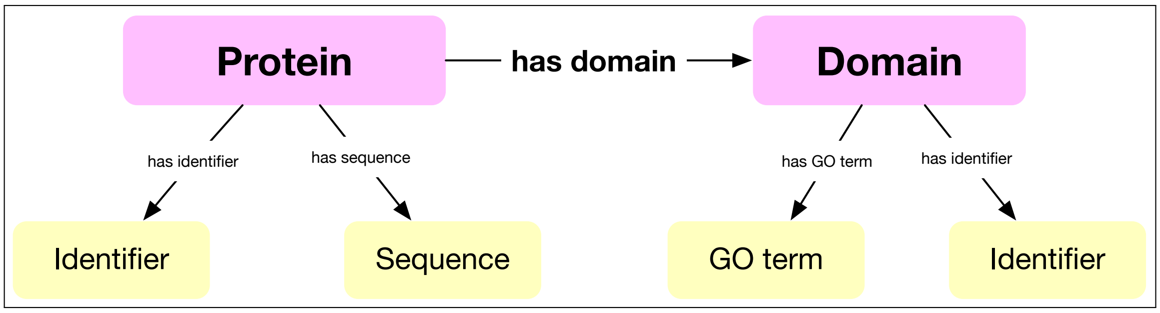


Figure 2.16: *An example data model for a set of proteins and their domains, modelled as a graph. Unlike with a relational database representation, no additional work beyond creating the edges is necessary to enable queryability in any direction.*

Using a relational database in this manner is adequate *providing the requirements do not frequently change.* Adding further columns would be considered a change in the database schema, which may require updating queries, and creating procedures for database migration. Adding relationships between tables potentially requires creating entire new association tables just to represent the relationship. While a relational database is perfect to store, for example, customer names and addresses, there are entire vocabularies such as the Sequence Ontology [43] containing thousands of attributes one might use to describe a biological entity, each of which would require a linker table.

2.3.2 RDF triplestores

The aforementioned example was a set of proteins, each of which has some attributes; a set of domains, each of which has some attributes; and a relationship between the proteins and their domains. Using the Resource Description Framework (RDF) previously described in section 2.2.1, this problem can be modelled trivially as a *graph* with nodes and edges (Fig. 2.16).

As described in section 2.2.1, the RDF concept of named properties makes it particularly well suited to biology. The same resource can be described with both simple properties such as “name” and “description”, and highly domain-specific properties such as “molecular structure” and “melting point”. These properties form edges in the graph, and thus can be trivially navigated in either direction without the need to create linker tables or join queries as one would in a relational database. Also unlike a relational database, the specific properties to be used in the graph need not be defined at the time when the database is created. Properties from any vocabulary can be inserted, removed, and queried dynamically.

The *SPARQL Protocol and RDF Query Language* (SPARQL) was standardised in 2008 [63] to provide a syntax to express queries over an RDF graph. At first glance, SPARQL queries look similar to SQL queries for relational databases. However, an RDF dataset has no tables or columns to select over. Instead, a SPARQL query is composed of *triple patterns* in which any or none of the subject, predicate, and object of a triple can be specified. Evaluating the SPARQL query returns triples from the database that match the pattern. Multiple triple patterns can be combined to create a query that navigates the graph on multiple layers. The example queries of “which domains does protein X have" and "which proteins have domain Y" in SPARQL would be:

```
SELECT ?domain WHERE {
    <protein> has_domain ?domain
}
```

and:

```
SELECT ?domain WHERE {
    <X> has_domain ?domain
}
```

No special linker tables are necessary to perform either of these queries, as all RDF relations can be queried in either direction. Databases built for storing and querying RDF are known as *triplestores*. Just as there exist many different relational database systems, there are a variety of different triplestores available, such as RDF4J (formerly Sesame [64]), Virtuoso [65], Apache Jena [66], and BlazeGraph [67].

2.3.3 Linked Data

In 2001, Berners-Lee et al. published an article in the Scientific American describing "a new form of Web content that is meaningful to computers" [68]. The article begins with a short section of speculative fiction describing a highly connected future where machine agents can communicate directly with each other to automate tasks through a global data sharing system termed the *Semantic Web*. It then details how the Semantic Web might be implemented: through the use of RDF backed by ontologies.

Berners-Lee later published a design note describing "Linked Data", a set of recommendations for how to publish data on the Web [69]: essentially, to identify and describe things with HTTP URIs, and to make links to other resources. The difference between the three related concepts of RDF, the Semantic Web, and Linked Data can be confusing, but can be summarised as:

- The Resource Description Framework (RDF) is a simple model for describing resources using triples consisting of a subject, predicate, and object. RDF can be used without any obligation to also adopt Linked Data or the Semantic Web.
- Linked Data is a set of principles encouraging the availability, machine-tractability, standardization, and integration of data. Linked Data encourages, but does not require, the use of RDF.
- The Semantic Web is the end goal of the original Berners-Lee publication in 2001: an integrated, machine-tractable Web of Linked Data.

Whether or not the Semantic Web has been, or can be, globally successful as described in its initial paper is a contentious issue. However, its success is largely irrelevant to the use of RDF and Linked Data technology for specific domains such as biology. With or without a true Semantic Web, RDF can be used as a powerful tool for data representation and integration.

2.3.4 Query federation

While the use of RDF can address the intractability of data by making its relationships and properties easier to navigate, there is still a logistical problem. Biological data is large, and relevant knowledge is likely to be spread across multiple databases in different locations.

There simplest approach to address this problem is *data warehousing*, where the data are physically copied to create one larger dataset in a single location. There are a number of problems with this approach. The most obvious is scalability: without seriously limiting scope, there is simply too much biological data to gather it all into one place, even with modern computing hardware. Other problems include synchronisation — in that once the data has been copied, if they are updated at source the changes will not be mirrored in the copy — and that it may not be possible to download entire datasets for warehousing due to technical or licensing restrictions.

An alternative approach is *query federation*, also known as *virtual integration*. In a federated approach, the data remain in their original location and are accessed as needed at query time [7, p. 9]. Query federation is both scalable, as a query can be federated to many different datasets without the need for extensive computational resources to warehouse their data at a single location; and can access the current, live version of each dataset rather than relying on an ahead-of-time copy as with data warehousing. However, these advantages come at the cost of significantly increased complexity, as the query client must have the ability to access multiple resources and collate the results.

Query federation is supported in SPARQL by manually specifying which endpoint to use for each section of the query through the `SERVICE` clause. For example, a SPARQL query to retrieve the chemical reactions from ChEBI associated with the ribC enzyme `P54575` from UniProt would be:

```
PREFIX up: <http://purl.uniprot.org/core/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rhea: <http://rdf.rhea-db.org/>

SELECT ?equation WHERE {

  SERVICE <https://sparql.uniprot.org/sparql> {
    <http://purl.uniprot.org/uniprot/P54575> up:annotation ?anno .
    ?anno up:catalyticActivity ?activity .
    ?activity up:catalyzedReaction ?reaction .
  }

  SERVICE <https://sparql.rhea-db.org/sparql> {
    ?reaction rhea:equation ?equation .
  }
}
```

2.3.5 Linked data fragments

The standard method for RDF triplestores to provide querying capability is through the SPARQL protocol [70]. The triplestore accepts HTTP requests containing SPARQL queries and responds with the results. This approach makes querying triplestores very easy for clients: even highly complex SPARQL queries which require significant server-side computation to evaluate require nothing from the client but submitting the query and waiting for a response. For servers, the demands are much higher, as they are responsible for all of the complexity of query processing.

Linked Data Fragments (LDF) are an emerging publishing method for sources of RDF [71] where the complexity is moved from the server to the client. Instead of providing access to a server-side querying engine as with a SPARQL endpoint, an LDF server provides a much simpler API by which partial triples (triple patterns) can be matched. For example, given a partial triple consisting only of a bound subject, an LDF server can respond with a set of triples where the missing predicate and object are "filled in" from the source dataset.

Complex queries can still be performed using LDF servers, but the responsibility of processing the SPARQL query and breaking it down into triple patterns is shifted from the server to the client. This approach provides true *query federation*, as the SPARQL query can be digested into a series of pattern matches over *multiple* LDF servers. Explicit federation by use of `SERVICE` clauses is no longer necessary; instead, queries can implicitly use multiple datasources as each triple pattern can be evaluated by multiple servers in parallel. For example, with LDF, the query from section 2.3.4 could become simply:

```
PREFIX up: <http://purl.uniprot.org/core/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rhea: <http://rdf.rhea-db.org/>

SELECT ?equation WHERE {
  <http://purl.uniprot.org/uniprot/P54575> up:annotation ?anno .
  ?anno up:catalyticActivity ?activity .
  ?activity up:catalyzedReaction ?reaction .
  ?reaction rhea:equation ?equation .
}
```

The LDF client then:

1. Slices up the query into the four triple patterns, and requests the number of expected results for each pattern from each server. For example, a UniProt LDF server might respond that it would have millions of results for `?anno up:catalyticActivity ?activity`, but only eight results for `<http://purl.uniprot.org/uniprot/P54575> up:annotation ?anno`.

2. Re-orders the patterns so that the pattern with the least triples (i.e. the most selective) is requested first.

3. Uses the results from the first pattern to fill in the blanks in other patterns, therefore making them more selective before querying again. For example, `?anno up:catalyticActivity ?activity` could become `<http://purl.uniprot.org/uniprot/P54575#SIPCAD628AC67FF287E> up:catalyticActivity ?activity` for each annotation if `<http://purl.uniprot.org/uniprot/P54575> up:annotation ?anno` has already been evaluated.

This movement of complexity from the server to the client not only enables federation, but also has advantages server-side. LDF servers are, by nature, simple to implement, and do not require the extensive server resources that server-side SPARQL processing typically demands. Reducing server-side complexity by providing only LDF has been shown to have the potential to improve availability of datasets [72].

2.4 Sharing and dissemination

The application of data standards and data integration technology has the potential to address the first aim of this work by improving access to existing knowledge about biological parts. However, there remains the problem of how to deal with *future* knowledge. How can information about newly engineered biological parts be stored in a form which is easily accessible by the designers and design tools of tomorrow?

Repositories of biological sequences have existed ever since computers powerful enough to build them were developed. One of the oldest, GenBank, contains around 200 million mostly DNA sequences as of 2019 [73]. Other repositories contain data about proteins, such as UniProt, which maintains access to around 150 million amino acid sequences [74]. There are also numerous bespoke repositories for specific organisms, such as SubtiWiki [75] for *Bacillus subtilis* and ECMDB [76] for *Escherichia coli*.

However, like the data representations discussed in section 2.2, these repositories were developed to catalogue *naturally occurring sequences*, rather than engineered biological “parts” intended for re-use [32] [77]. While there are clearly similarities, in that an engineered part still usually has annotated sequence information, there are many details such as compositional hierarchy and provenance that do not exist for natural parts and are therefore not possible to store in such databases.

Recently, various repositories have been created specifically to address the problem of storing information about engineered designs, such as the iGEM Parts Registry [78], JBEI-ICE [79], and the Virtual Parts Repository [80]. There are also repositories for specific engineering purposes, such as the Addgene [81] repository for storing plasmids. The ability of these repositories to provide a general solution to the problem of publishing engineered designs in SynBio was assessed using the following criteria (Table 2.2):

- **Public read-write** — can any user both browse the repository and upload new designs?
- **Standardization** — is the data representation used by the repository a standardized format with a clearly defined specification?
- **Whole-design representation** — does the repository have the facility to represent an entire design, including, for example, gene products and their interactions, or is it limited to a subset of the biology such as a DNA sequences?
- **Machine-tractability** — are the constituent parts of a design available to computational queries?
- **Reproducibility** — can an independent instance of the repository be created by other users?

	iGEM	Addgene	JBEI-ICE	VPR
Public read-write	✗	✗	✓	✗
Standardization	✗	✗	✓	✗
Whole-design representation	✗	✗	✗	✓
Programmatic access	✗	✗	✓	✓
Machine-tractability	✗	✗	✗	✓
Reproducibility	✗	✗	✓	✗

Table 2.2: *An assessment of the abilities of different design repositores*

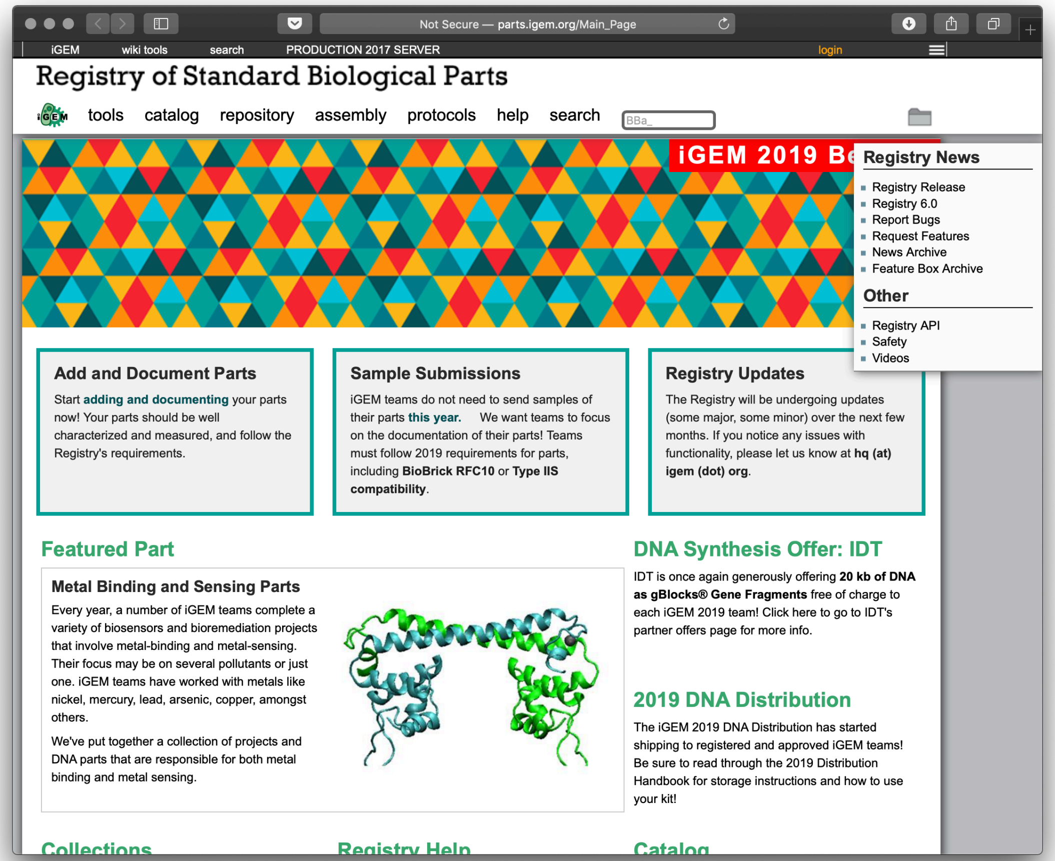
2.4.1 iGEM Parts Registry

The Internationally Genetically Engineered Machine (iGEM) competition is an annual event in which teams of students worldwide compete to create novel, engineered biological systems [82]. Teams participating in the iGEM competition receive distribution plates containing DNA samples of a wide variety of different parts or "BioBricks" which can be repurposed to create new parts, providing an excellent example of how the engineering concepts of modularity and re-use can be applied in practice to synthetic biology. Parts created by teams are submitted both as *in vitro* samples to iGEM Headquarters, and *in silico* as DNA sequences to the iGEM Registry of Standard Biological Parts (Fig. 2.17). The Registry, established alongside iGEM in 2003, currently provides access to over 20,000 documented parts specifically for the purpose of composing engineered biological systems, with their corresponding annotated DNA sequences and a wealth of experience information from iGEM participants [78].

Public read-write: It is necessary to have an iGEM account to submit to or edit the registry, which currently requires either participating in iGEM or explicitly requesting an account.

Standardization: The internal representation used by the repository is not standardized or documented. Although iGEM parts can be downloaded in FASTA or GenBank format, much of the information contained in the database is lost in conversion. While a conversion of the iGEM dataset to SBOL format was developed as part of this work (chapter 5), the SBOL version remains derivative of the original data representation, and therefore the registry itself does not benefit from the wider scope provided by SBOL. The Knowledgebase of Standard Biological Parts (SBPkb) attempted to address this issue in 2011 by converting the iGEM Registry to SBOL, therefore enabling the application of SPARQL queries [83]. However, this was at the time of SBOL1, and no server software was provided.

Whole-design representation: The Registry is strongly linked to its *in vitro* counterpart (physical BioBricks), and therefore is DNA-centric, and does not include the other layers of biology that can be represented by SBOL2. For example, even when a part is explicitly marked as coding for a protein, the sequence of the protein is not



Registry of Standard Biological Parts

Add and Document Parts

Start **adding and documenting** your parts now! Your parts should be well characterized and measured, and follow the Registry's requirements.

Sample Submissions

iGEM teams do not need to send samples of their parts **this year.** We want teams to focus on the documentation of their parts! Teams must follow 2019 requirements for parts, including **BioBrick RFC10** or **Type IIS compatibility**.

Registry Updates

The Registry will be undergoing updates (some major, some minor) over the next few months. If you notice any issues with functionality, please let us know at **hq (at) igem (dot) org**.

Featured Part

Metal Binding and Sensing Parts

Every year, a number of iGEM teams complete a variety of biosensors and bioremediation projects that involve metal-binding and metal-sensing. Their focus may be on several pollutants or just one. iGEM teams have worked with metals like nickel, mercury, lead, arsenic, copper, amongst others.

We've put together a collection of projects and DNA parts that are responsible for both metal binding and metal sensing.

DNA Synthesis Offer: IDT

IDT is once again generously offering **20 kb of DNA as gBlocks® Gene Fragments** free of charge to each iGEM 2019 team! Click here to go to IDT's partner offers page for more info.

2019 DNA Distribution

The iGEM 2019 DNA Distribution has started shipping to registered and approved iGEM teams! Be sure to read through the 2019 Distribution Handbook for storage instructions and how to use your kit!

Figure 2.17: *Homepage of the iGEM Registry, a repository providing access to over 20,000 documented parts specifically for the purpose of composing engineered biological systems.*

provided, making it difficult to cross-reference iGEM parts with databases of protein information or use protein-oriented bioinformatics tools. This is essentially the same problem with reductive file formats such as GenBank and FASTA. While nearly all BioBricks were originally derived from naturally occurring DNA, they are often heavily modified, and have associated provenance information along with characterisation data for many different organisms. iGEM has much of this information in free-text in its wiki, but not yet in a machine-tractable data format.

Programmatic access: The iGEM registry has only a limited API [2]. Parts can be retrieved as XML, but there is no facility to perform search queries.

Machine-tractability: Without standardized data format or an API to query the Registry, it is difficult to access knowledge about parts in anything other than free-text.

Reproducibility: While the data provided by the Registry is publicly accessible, its source code is not. It is not possible to setup an independent instance of the Registry.

2.4.2 Addgene

Addgene provides a repository for archiving and distributing plasmids [81], containing information about over 31,000 plasmids in 2014 [84]. Like iGEM, the Addgene curators store physical samples *in vitro* to match the *in silico* data records.

Public read-write: Anyone can deposit plasmids to Addgene. Submission is a physical process for an actual quantity of *in vitro* plasmid DNA. Though Addgene does also store annotated sequences, it is not a repository for conceptual designs without a corresponding physical representation. The question of public database access is therefore not clearly applicable: while Addgene is an open repository, writing to its database depends on the submission of a physical sample.

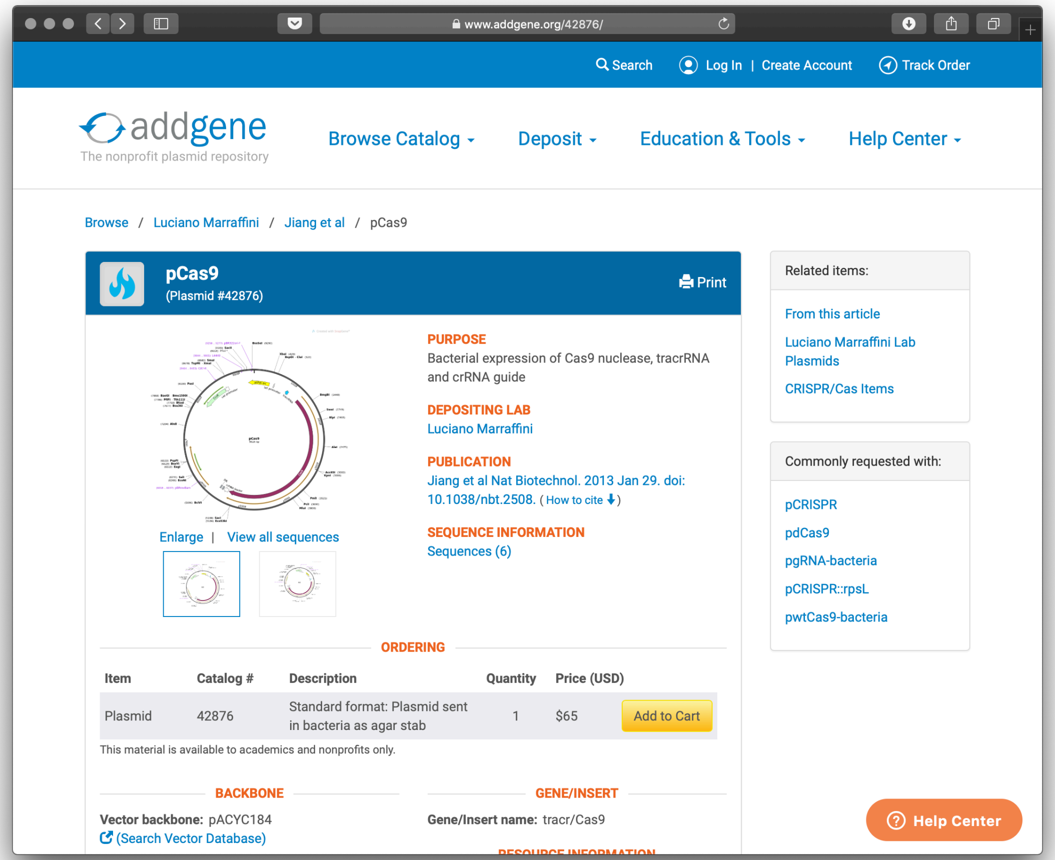
Standardization: Addgene provides sequence data in FASTA format, and many plasmids also have an associated supplemental GenBank file. Any other information about the plasmid is only visible through its Web page and is not captured in any downloadable format.

Whole-design representation: As with iGEM, the data representation of Addgene is strongly linked to physical DNA samples and comes with all of the associated problems. It is not possible to document anything other than an annotated plasmid.

Programmatic access: Addgene does not appear to have any kind of API.

Machine-tractability: The lack of API and the representation of parts only as Web pages makes Addgene highly computationally intractable.

[2] `http://parts.igem.org/Registry_API`



www.addgene.org/42876/

Search | Log In | Create Account | Track Order

addgene
The nonprofit plasmid repository

Browse Catalog | Deposit | Education & Tools | Help Center

Browse / Luciano Marraffini / Jiang et al / pCas9

pCas9
(Plasmid #42876)

Print

Enlarge | View all sequences

PURPOSE
Bacterial expression of Cas9 nuclease, tracrRNA and crRNA guide

DEPOSITING LAB
Luciano Marraffini

PUBLICATION
Jiang et al Nat Biotechnol. 2013 Jan 29. doi: 10.1038/nbt.2508. (How to cite)

SEQUENCE INFORMATION
Sequences (6)

Related items:
From this article
Luciano Marraffini Lab Plasmids
CRISPR/Cas Items

Commonly requested with:
pCRISPR
pdCas9
pgRNA-bacteria
pCRISPR::rpsL
pwtCas9-bacteria

ORDERING

Item	Catalog #	Description	Quantity	Price (USD)	
Plasmid	42876	Standard format: Plasmid sent in bacteria as agar stab	1	$65	Add to Cart

This material is available to academics and nonprofits only.

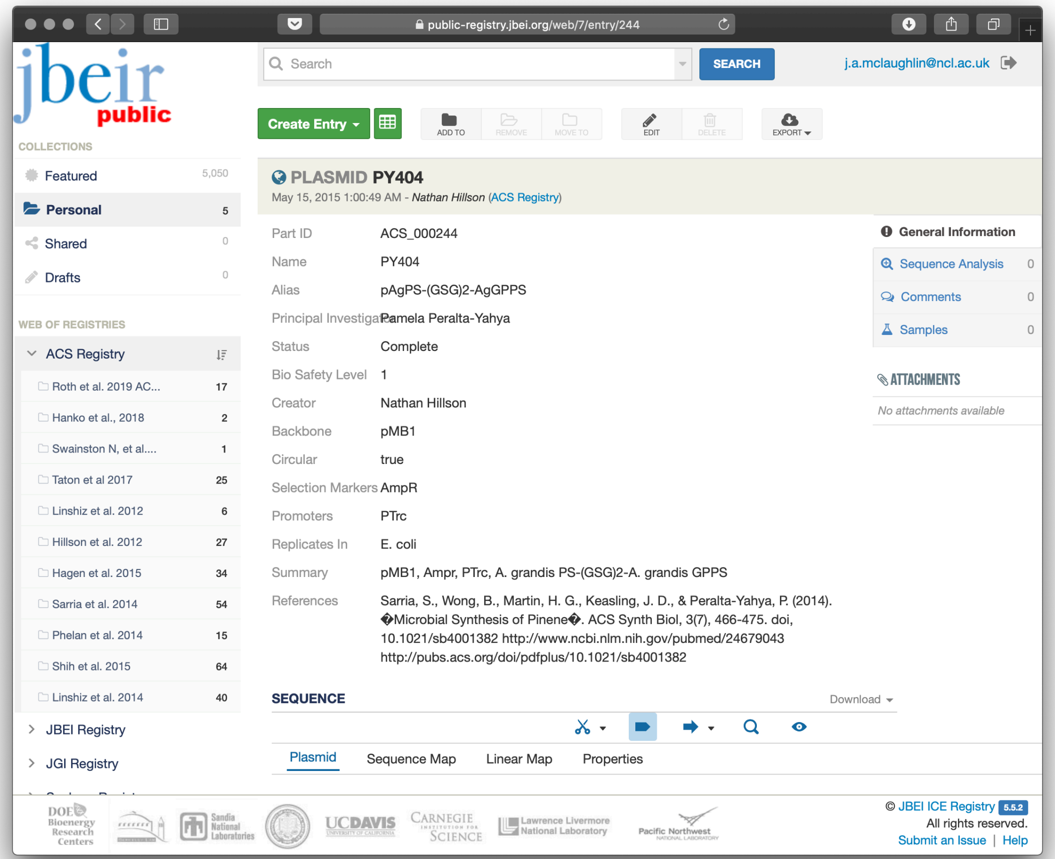
BACKBONE
Vector backbone: pACYC184
(Search Vector Database)

GENE/INSERT
Gene/Insert name: tracr/Cas9

RESOURCE INFORMATION

Help Center

Figure 2.18: *A plasmid in Addgene, a repository for archiving and distributing plasmids. Addgene contained information about over 31,000 plasmids in 2014* [84].



PLASMID PY404
May 15, 2015 1:00:49 AM - *Nathan Hillson* (ACS Registry)

Part ID	ACS_000244
Name	PY404
Alias	pAgPS-(GSG)2-AgGPPS
Principal Investigator	Pamela Peralta-Yahya
Status	Complete
Bio Safety Level	1
Creator	Nathan Hillson
Backbone	pMB1
Circular	true
Selection Markers	AmpR
Promoters	PTrc
Replicates In	E. coli
Summary	pMB1, Ampr, PTrc, A. grandis PS-(GSG)2-A. grandis GPPS
References	Sarria, S., Wong, B., Martin, H. G., Keasling, J. D., & Peralta-Yahya, P. (2014). �Microbial Synthesis of Pinene�. ACS Synth Biol, 3(7), 466-475. doi, 10.1021/sb4001382 http://www.ncbi.nlm.nih.gov/pubmed/24679043 http://pubs.acs.org/doi/pdfplus/10.1021/sb4001382

Figure 2.19: *Part view in JBEI-ICE, a repository for storing information about biological parts.*

Reproducibility: It is not possible to setup an independent instance of Addgene.

2.4.3 JBEI-ICE

The Joint BioEnergy Institute Inventory of Composable Elements (JBEI-ICE; referred to henceforth as simply “ICE”) was published in 2012, described as an “open source registry platform for managing information about biological parts” [79]. The motivation given in the ICE paper is reminiscent of the overarching research aims of this work: “there is a noticeable lack of a coherent suite of synthetic biology software tools that work together, which means that a biological engineer has sequence information and other information scattered in different documents and software packages”.

The ICE software is both open-source and permissively (BSD) licensed, enabling it to be used by both academia and industry. ICE also introduces the concept of a Web of Registries, whereby multiple instances can be connected together to share data. In addition to a user-friendly Web interface, ICE also provides an API to enable programmatic access, making it suitable for integration into a wider ecosystem of tooling.

Public read-write: Anyone can register an ICE account in order to both browse and submit parts.

Standardization: ICE supports the submission of parts in various file formats, including SBOL1 and, recently, SBOL2.

Whole-design representation: As ICE supports the submission of SBOL2 files, it technically provides support for storing whole-design knowledge. However, it lacks one of the key benefits of SBOL: computational tractability. Once SBOL has been uploaded to ICE, it can be queried based only on the fields duplicated into the ICE data model, which are a small subset of the richness provided by SBOL, as described in chapter 3. Any SBOL features developed in the future will be opaque to ICE unless ICE extends its own classes, which effectively means ICE operates two data models which are not synchronised.

Programmatic access: While ICE does provide an API, the API described in the documentation does not match the currently accessible version as of July 2019[3].

Machine-tractability: The underlying data model of ICE is a Java Hibernate interface. While this data model is not standardised, ICE has code to convert SBOL to the data objects used by the Hibernate interface and vice-versa.

Reproducibility: ICE is open-source, and anyone can start their own ICE instance.

Virtual Parts Repository

There are many repositories such as BioModels [28] which catalogue models for predicting the behaviour of biological parts. Since structural and sequence information about a part is very different from a predictive model, these repositories are not described here. The exception is the Virtual Parts Repository (VPR), a Newcastle University effort to host a set of composable, modular models for synthetic biology [85] (Fig. 2.20). Unlike other model repositories, the VPR also provides SBOL files providing the structural information associated with its models.

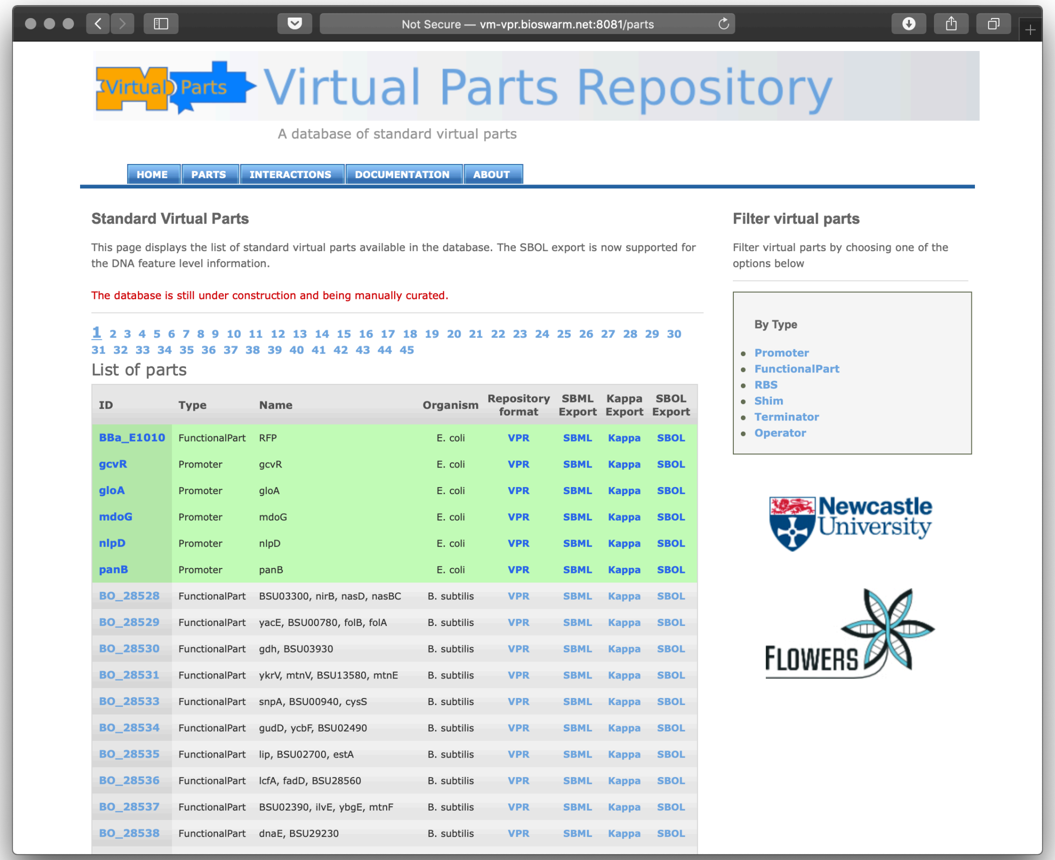
The VPR currently provides access to several thousand *B. subtilis* parts, each including both sequence information in SBOL1 and models available to download in various modelling languages.

Public read-write: While the VPR is an open repository, it is manually curated [80] and does not allow user submission of parts.

Standardization: The VPR provides multiple export options for each part: a VPR-specific XML format, SBOL1, and SBML [27] or Kappa [86] for models. SBOL1, SBML, and Kappa are all standardized formats.

Whole-design representation: The VPR provides a broad view of parts by combining a structural description with functional models. While upgrading its SBOL

[3]`https://github.com/JBEI/ice/issues/91`



Standard Virtual Parts

This page displays the list of standard virtual parts available in the database. The SBOL export is now supported for the DNA feature level information.

The database is still under construction and being manually curated.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45

List of parts

ID	Type	Name	Organism	Repository format	SBML Export	Kappa Export	SBOL Export
BBa_E1010	FunctionalPart	RFP	E. coli	VPR	SBML	Kappa	SBOL
gcvR	Promoter	gcvR	E. coli	VPR	SBML	Kappa	SBOL
gloA	Promoter	gloA	E. coli	VPR	SBML	Kappa	SBOL
mdoG	Promoter	mdoG	E. coli	VPR	SBML	Kappa	SBOL
nlpD	Promoter	nlpD	E. coli	VPR	SBML	Kappa	SBOL
panB	Promoter	panB	E. coli	VPR	SBML	Kappa	SBOL
BO_28528	FunctionalPart	BSU03300, nirB, nasD, nasBC	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28529	FunctionalPart	yacE, BSU00780, folB, folA	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28530	FunctionalPart	gdh, BSU03930	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28531	FunctionalPart	ykrV, mtnV, BSU13580, mtnE	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28533	FunctionalPart	snpA, BSU00940, cysS	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28534	FunctionalPart	gudD, ycbF, BSU02490	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28535	FunctionalPart	lip, BSU02700, estA	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28536	FunctionalPart	lcfA, fadD, BSU28560	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28537	FunctionalPart	BSU02390, ilvE, ybgE, mtnF	B. subtilis	VPR	SBML	Kappa	SBOL
BO_28538	FunctionalPart	dnaE, BSU29230	B. subtilis	VPR	SBML	Kappa	SBOL

Filter virtual parts

Filter virtual parts by choosing one of the options below

By Type

- Promoter
- FunctionalPart
- RBS
- Shim
- Terminator
- Operator

Figure 2.20: *Part list in the Virtual Parts Repository (VPR), a repository for both sequence information and models for biological parts.*

version to SBOL2 would enable both structure and function to be captured in the same data model, the VPR already provides whole-design representation with the use of multiple standards.

Programmatic access: The VPR has both a Web service interface and a Java API.

Machine-tractability: The API has endpoints such as `/parts/FunctionalPart/has_function/GO_0000155` which are akin to querying using RDF triple patterns. While not a complete SPARQL endpoint, the provision of this functionality is a significant improvement over the free-text searches provided by other repositories.

Reproducibility: Hosting an independent instance of the VPR is not currently supported.

2.5 Standards-enabled tooling

The Microsoft Word `docx` file format might be one of the most popular data standards in the world, but there are very few authors of Word documents who know how it is specified. Microsoft Word is the *tooling*: the software that provides an abstraction layer for the user to access the data standard without knowledge of its internals.

If SBOL is the equivalent of the `docx` format for synthetic biology, it does not yet have its corresponding equivalent to Microsoft Word. SBOL has a powerful and well-specified data model, but in order to actually *create* SBOL it it is usually necessary to write programs using an SBOL software library such as libSBOLj. Essentially, biologists are expected, in addition to their biology domain knowledge, to have programming skills in order to document a biological design.

The SBOL community has a complementary standard termed *SBOL Visual* [87], which is a separate effort to formalise the way that designs are already communicated in biology: through visual depictions. The SBOL data and SBOL Visual standards can be connected in two directions: visualisation tools such as dnaplotlib [88] allow a visual depiction to be created from SBOL data, and CAD tools such as SBOLDesigner [89] allow SBOL data to be created using SBOL Visual. Connecting the two standards in this manner creates a visual abstraction layer to enable the use of SBOL without interacting with the underlying data.

2.5.1 SBOL Visual

Using SBOL as a standard for design information enables knowledge that was previously heterogeneous to share a uniform representation. However, SBOL is a standard to make design information tractable to machines, rather than humans.

In publications about engineered biological systems, visual depictions are often used to convey structure and/or function. Many different kinds of visual depictions are used, depending on what properties about a design are important to convey (Fig. 2.21). At the lowest structural level, for example, a 3D rendering of a protein might be a useful depiction. At the highest functional level, a graph describing how a system works might be more appropriate.

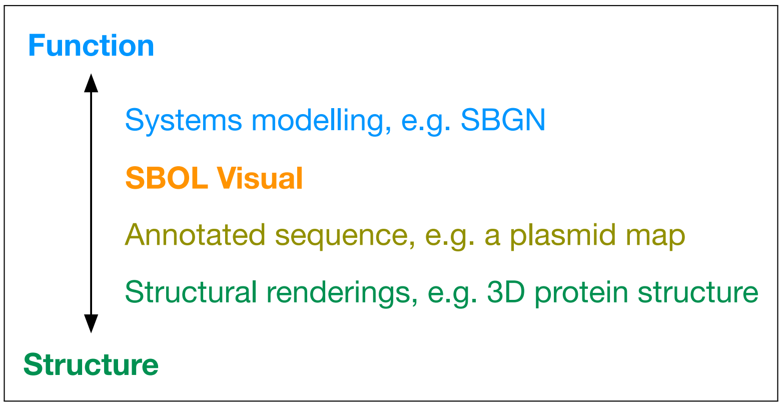

Function
Systems modelling, e.g. SBGN
SBOL Visual
Annotated sequence, e.g. a plasmid map
Structural renderings, e.g. 3D protein structure
Structure

Figure 2.21: *An overview of how different types of visual depiction compare with regards to structural and functional coverage.*

In synthetic biology, the necessary visual concepts to express e.g. the design of a genetic circuit often include both structure and function. Consequently, a graphical notation has evolved over time to convey the features of genetic circuit designs. This notation typically consists of glyphs to represent the function of regions such as promoters, ribosome binding sites, coding sequences, and terminators, alongside arcs to denote processes such as genetic production and protein-DNA binding.

The notation has typically been informally specified and varies widely among synthetic biologists; for example, while some use a "hairpin loop" symbol to represent a terminator [90], others might instead draw a "T" shape [21] or a circle [91].

Although variation among visual representation is less of a problem for human communication, the continued focus of synthetic biology on developing a well-defined, standardised, engineering approach requires that designs must be unambiguous at every stage of the design-build-test lifecycle. This formalisation is particularly important as designs grow in scale and complexity, where modules may become part of a larger system and consistency across visual representation is expected.

The SBOL Visual standard [87] was published in 2015 to address this need. SBOL Visual defines a set of glyphs to represent common features such as promoters, CDSs, and terminators (Fig. 2.22) along with a set of recommendations about how they are drawn.

2.5.2 Tooling for SBOL Visual

SBOL provides a data standard for the representation of biological designs, and SBOL Visual provides a visual standard. In order to make the two standards work together — i.e., produce an SBOL Visual diagram from SBOL data — visualisation software is required.

There are many tools available that can produce SBOL Visual diagrams, such as Pigeoncad [92] and TinkerCell [93]. However, very few of these tools can actually produce SBOL Visual *from SBOL data*. Pigeoncad, for example, has its own text-based

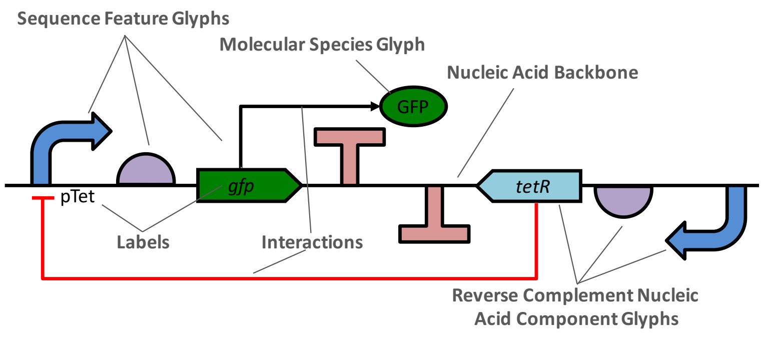
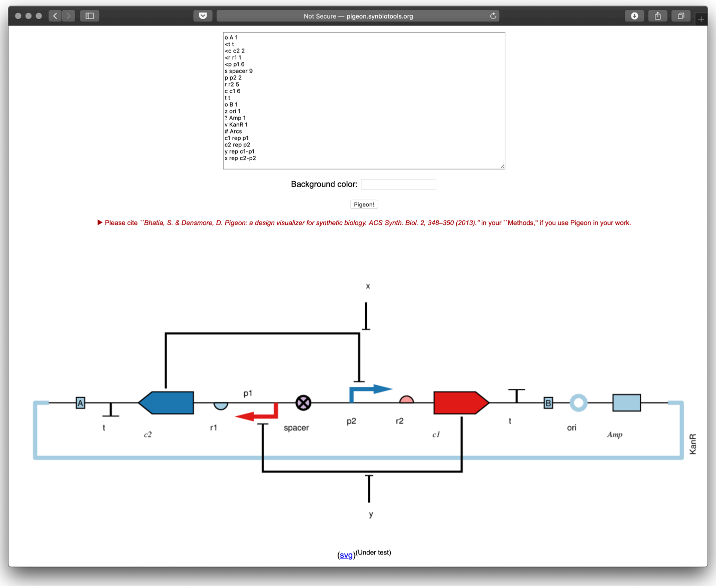


Figure 2.22: *An example of an SBOL Visual depiction. Image source: The SBOL Visual specification* `https://github.com/SynBioDex/SBOL-visual`

"Pigeon" syntax to specify what to draw (Fig. 2.23). TinkerCell predates SBOL and has its own internal representation of designs.



```
o A 1
<t t
<c c2 2
<r r1 1
<p p1 6
s spacer 9
p p2 2
r r2 5
c c1 6
t t
o B 1
z ori 1
? Amp 1
v KanR 1
# Arcs
c1 rep p1
c2 rep p2
y rep c1-p1
x rep c2-p2
```

Background color:

Pigeon!

▶ Please cite ``*Bhatia, S. & Densmore, D. Pigeon: a design visualizer for synthetic biology. ACS Synth. Biol. 2, 348–350 (2013).*" in your ``Methods," if you use Pigeon in your work.

Figure 2.23: *A screenshot of the Pigeoncad design visualizer for SBOL Visual. While Pigeon can produce SBOL Visual diagrams, it does so using its own notation (top) rather than from the SBOL data model.*

There are currently three tools that support both the SBOL Visual and SBOL Data standards: VisBOL [94], partly developed as part of this work; SBOLDesigner [89]; and dnaplotlib [88]. Unfortunately, all of these tools are restricted predominantly to the features of SBOL1. Neither SBOLDesigner nor dnaplotlib yet have support for rendering the functional aspects of SBOL such as `ModuleDefinition` and `Interaction`, and VisBOL has only recently added very primitive support. SBOLDesigner is also currently the only tool with both SBOL Visual and SBOL Data support to provide editing capability, i.e. the ability to interact with SBOL Visual glyphs in order to create or modify underlying SBOL data (Fig. 2.24).

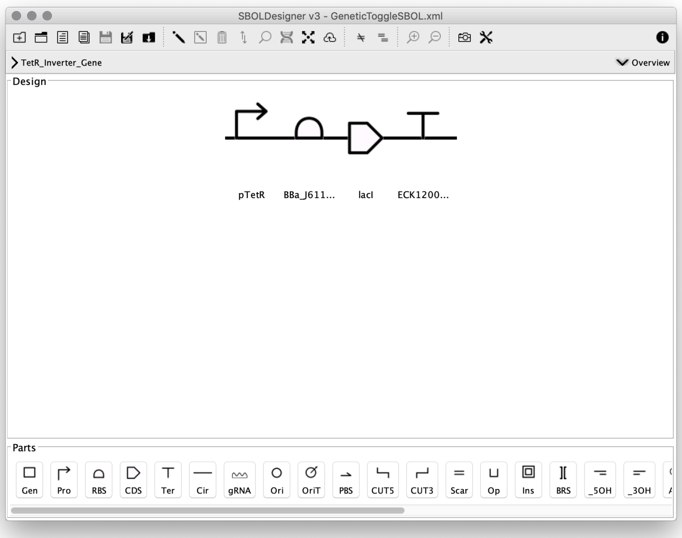


Figure 2.24: *A screenshot of SBOLDesigner, the only tool with both SBOL Visual and SBOL Data support that provides editing capabilities. While SBOLDesigner technically has support for SBOL2, it ignores the concepts of modules and interactions, essentially limiting its functionality to the features of SBOL1.*

2.6 Conclusion

This chapter provided an introduction to the synthetic biology lifecycle, and how the Synthetic Biology Open Language (SBOL) can be used to capture synthetic biology designs using the Resource Description Framework (RDF). RDF and SBOL are the fundamental underpinning technologies of this work. The related concept of data integration was also described, first in terms of existing work in the context of synthetic biology, then more generally as as a computing concept with an overview of established work and recent innovations from the Semantic Web community which provide the context for chapter 4. Section 2.4 presented a review of data repositories for sharing engineered biological parts. Finally, section 2.5 provided an overview of the SBOL Visual standard as a tool to bridge the gap between the data standard and its potential users.

Part I

Knowledge representation

3. Machine-tractability in the design process

3.1 Introduction

“ As the complexity of synthetic biology projects grow, it will be critical to standardize the exchange of designs and data ”

— *Christopher Voigt, Editor-in-Chief, ACS Synthetic Biology* [54]

One of the fundamental problems in optimising the synthetic biology design process is that the designs themselves are not captured in a computationally tractable form. Much of the design process is often performed on a whiteboard, in proprietary software, or in a word processor.

The lack of a standardised, machine-tractable representation for designs causes two major problems. First, the issue of knowledge discoverability: information about existing parts is difficult to find, making the design process more time-consuming; and any new parts created by the design process will, in turn, be difficult to find for use in future designs. Second, without a machine-tractable standard it is not possible to model the design process in software to provide design automation or the automated enrichment of design knowledge.

As described in section 2.2, the Synthetic Biology Open Language (SBOL) is a data standard specifically for the representation of designs in SynBio. When this work began, version 2.0 of the SBOL specification had just been released, adding the ability to describe molecule types other than DNA — for instance, proteins and RNAs — along with functional information such as molecular interactions. The most recent version, SBOL 2.3, is capable of capturing an even richer set of design information including aspects such as provenance and experimental data.

However, SBOL is not yet commonly used as part of the synthetic biology design process. This research chapter explores some of the issues with SBOL which may be responsible for inhibiting its adoption, and proposes new software libraries to make SBOL easier for developers to use along with a set of proposals to simplify the standard.

The research goals of this chapter are to investigate (a) to how SBOL can be made more accessible to potential users, and (b) how complexity in the SBOL data model can be reduced while preserving expressiveness.

Attribution: sboljs and sbolgraph were originally developed as part of this work. The current version of sboljs now has code contributions from Chris Myers, Zach Zundel, Dany Fu, and Nathan Wilkinson (University of Utah). The current version of sbolgraph has code contributions from Christian Atallah (Newcastle University). The SBOL Stack was developed in collaboration with Curtis Madsen based on an idea by Matthew Pocock, with contributions from Goksel Misirli, Matthew Pocock, Keith Flanagan (Newcastle University) and Jennifer Hallinan (Macquarie University). The code for its server API and client libraries was written as part of this work.

3.2 Challenges for adoption of SBOL

Capturing the design process using SBOL could significantly improve its computational tractability. With SBOL, an entire design can be captured using a standardized, fully navigable RDF data model, including many aspects which would previously only have been captured in free-text if they were documented at all. However, despite the continually increasing power of SBOL, its adoption by the community has been slow. It can be suggested that this is caused by several interconnected issues (Fig. 3.1). First, the bootstrapping problem, or *lack of data*: without existing data being available in SBOL, there is little motivation to use SBOL as there is no knowledge ecosystem with which to integrate. This problem is likely exacerbated by two other problems: first, the *limited scope* of SBOL means that it can not necessarily be used to represent all real-world use cases. Secondly, the *lack of tooling* for SBOL means it is difficult to get data into the SBOL format because there is insufficient software support. In turn, the lack of tooling is likely caused by the fact that SBOL is not always easily accessible to software developers due to the limitations of SBOL software libraries (*limited portability*), which is likely a result of the fact that the SBOL data model is difficult to implement (*complexity*).

3.2.1 Limitation in scope

As discussed in section 2.2, the universe of possible designs that can be captured using SBOL has grown from only DNA components in SBOL 1.0 to a much broader set of biological entities and relationships. However, the scope of the data model is still not sufficient to capture all of the information used in the design process previously described in section 2.2, such as characterisation data and provenance. This limitation in scope limits the modelling power of SBOL, restricting the ability for data to be captured.

In particular, the lack of a means to describe the provenance of designs makes the standardized description of iterations of the synthetic biology lifecycle impossible. Section 3.5 includes proposals to address this need by expanding the SBOL data model to recommend terms from the provenance ontology, PROV-O [49].

3.2.2 Lack of data

One of the most important reasons to adopt a standard is to fall in line with existing use of the same standard to enable interoperability. The majority of mobile phones charge

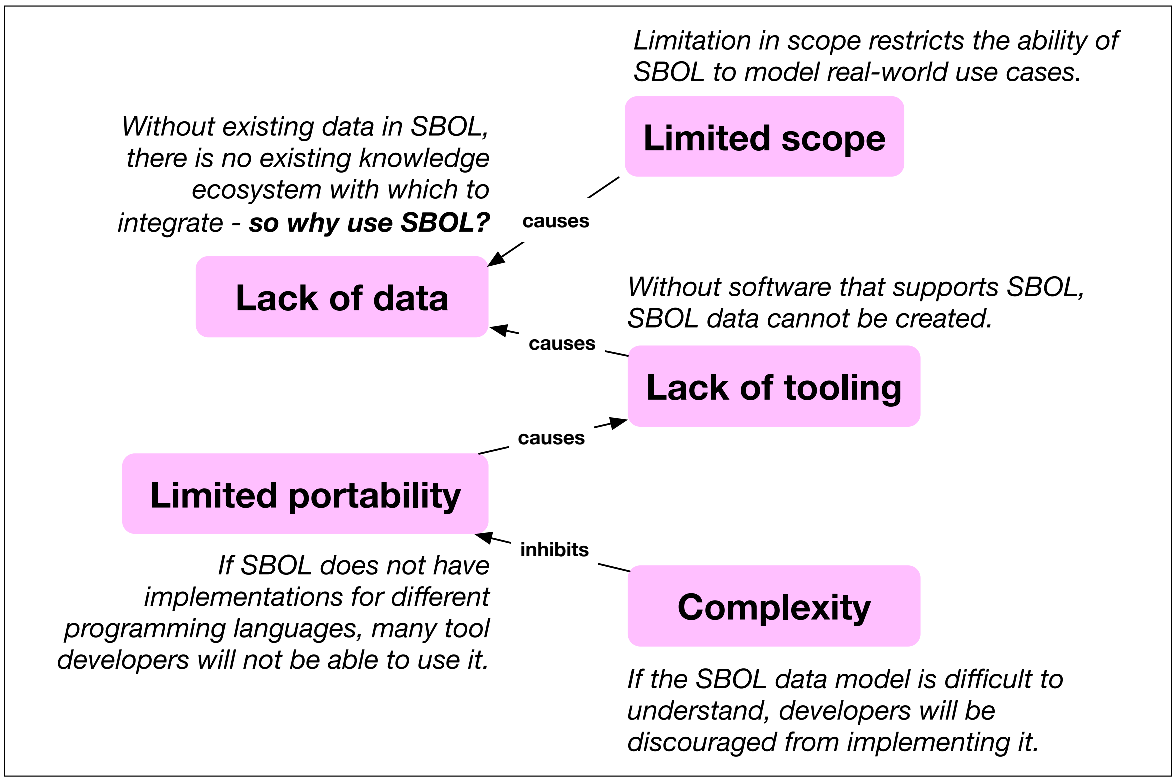

Limitation in scope restricts the ability of SBOL to model real-world use cases.
Limited scope
Without existing data in SBOL, there is no existing knowledge ecosystem with which to integrate - **so why use SBOL?**
causes
Lack of data
Without software that supports SBOL, SBOL data cannot be created.
causes
Lack of tooling
causes
Limited portability
inhibits
If SBOL does not have implementations for different programming languages, many tool developers will not be able to use it.
Complexity
If the SBOL data model is difficult to understand, developers will be discouraged from implementing it.

Figure 3.1: *While the Synthetic Biology Open Language (SBOL) greatly facilitates the representation of designs, its adoption has been slow. It can be suggested that this is caused by several interconnected issues: the inability of SBOL to model real-world use cases, or its* limited scope*; the bootstrapping problem, or* lack of data*; insufficient software support, or* lack of tooling*, limitations of SBOL software libraries, or* limited portability*; and implementation difficulty, or* complexity.

with Micro-USB not necessarily because it is the best type of USB connector, but because the majority of mobile phones charge with Micro-USB. Using the same standard is not only the path of least resistance, but means that chargers are largely interchangeable between different handsets.

To apply this analogy to the problem at hand, using SBOL as a design representation would be an obvious choice if the majority of *existing* parts were already represented in SBOL and could be used directly in new designs. A bootstrapping approach to solving this problem by converting existing datasets to SBOL to create an initial knowledge base of parts is explored in chapter 5 of this work.

3.2.3 Lack of tooling

If SBOL is to be adopted by the synthetic biology community, it has to be easy for the community to create SBOL data. As synthetic biology is fundamentally an interdisciplinary field, knowledge of computer science cannot be a prerequisite. While there have been numerous attempts to create graphical tools for creating SBOL, many of the features of the SBOL2 data model — such as the definition of modules and interactions — remain inaccessible without writing code.

Chapter 7 of this work is dedicated to this problem, describing the SBOL Visual standard for graphical communication of designs, and the development of a new CAD tool for SBOL with support for the SBOL2 data model.

3.2.4 Portability

To encourage developers of tools to adopt SBOL and thus make SBOL accessible to users, it has to be as easy as possible to build SBOL into tools. SBOL is implemented in the form of software libraries, which can then be used to add SBOL support to both new and existing software tools. When SBOL 2.0 was released, the only complete implementation was libSBOLj [95], which is exclusively for the Java programming language. In the 2019 Stack Overflow Developer survey, only 41.1% of users reported using Java, behind both Python at 41.7% and JavaScript at 67.8% [96]. Restricting the use of SBOL to Java, therefore, excludes the majority of developers and impedes the adoption of SBOL by software.

Chapter 3 provides solutions to this problem in the form of new SBOL libraries for JavaScript and Python, and a new, RDF graph-based design pattern for the development of such libraries to make it easier to add SBOL support to other languages in the future.

3.2.5 Complexity

Potential users of SBOL have often complained that the standard is overcomplicated and difficult to understand. While some degree of complexity is understandable given the intricacies of biological systems, it can be argued that SBOL compounds this problem by making its data model more complicated than necessary. Engineering biology is challenging enough without also having to learn the highly technical rules of a data standard which is equally as concerned about syntax and semantics as it is about representing biological concepts.

Reducing complexity in the SBOL standard wherever possible is essential to lower the barrier to entry and to encourage adoption by developers. Section 3.5 includes proposals for the next iteration of SBOL with the goal of making SBOL version 3 easier to understand, while retaining its ability to represent complex biological systems.

3.3 Improving portability of SBOL

SBOL is not a "tool" that can be used directly by users, but a specification that describes a particular data representation for design information. Any software that follows the specification by using this representation is deemed "SBOL-compliant", and can benefit from interoperability with all other SBOL-compliant software.

In order to make use of the specification from software, it is first necessary to implement a programmatic realisation of the SBOL data model. Typically, this takes the form of an object-oriented class hierarchy which mirrors the concepts described by the specification, coupled with the functionality to read and write SBOL from its RDF/XML serialization format. As it would be unreasonable to expect every software developer intending to add SBOL support to a tool to interpret the specification and develop a new implementation of SBOL, this functionality is encapsulated by open-source SBOL software libraries which can be used instead of working directly with the data model.

Prior to this work, the only complete implementation of SBOL2 was libSBOLj, restricting access to the SBOL standard to software written in Java. Developers using the two most popular programming languages, JavaScript and Python [96], were unable to add SBOL support to tooling. Making SBOL more portable by creating new libraries was therefore a logical step towards encouraging the adoption of SBOL by developers, and ultimately users of the tools they create. Today, there are five complete implementations of SBOL:

- libSBOLj[1] (Java)
- libSBOL [97] (C++ and Python)
- sboljs[2] (JavaScript)
- sbolgraph[3,4] (TypeScript, JavaScript, and Python)

Two of these libraries were developed as part of this work: *sboljs*, which enables SBOL to be used by JavaScript applications both server-side and in the context of a Web browser; and *sbolgraph*, a new design pattern for SBOL libraries where SBOL data is navigated as an RDF knowledge graph, with implementations in TypeScript and Python.

[1]`https://github.com/SynBioDex/libSBOLj`
[2]`https://github.com/SynBioDex/sboljs`
[3]`https://github.com/udp/sbolgraph`
[4]`https://github.com/udp/pysbolgraph`

3.3.1 sboljs: SBOL on the Web

The first target for improving portability of SBOL was to implement an SBOL library in JavaScript, enabling SBOL to be used from the context of a Web browser. There were several motivating factors for this choice:

- JavaScript has consistently been identified as the most commonly used programming language for the last seven years [96]
- The BioJS project [98] maintains a registry of over 200 libraries for working with biological data in JavaScript. Adding SBOL support to this software ecosystem could encourage developers already using JavaScript for biological applications to adopt the SBOL standard.
- There are already Web applications that support SBOL (such as VisBOL [94] and the SBOL Validator [99]), but do so using libSBOLj via additional server-side software. Native support for SBOL in the browser would mean these applications could run entirely in the browser, meaning server resources and Internet connectivity of the client would not be necessary.

The development of a new library for SBOL also presented an opportunity to explore a new design pattern. Despite the fact that SBOL is an RDF standard, none of the SBOL libraries have taken advantage of the fact that libraries for RDF are already well-established, tested, and standardised in all common programming languages. libSBOLj, for example, parses the RDF/XML serialization of SBOL using an XML parser in order to populate a hierarchy of Java classes, rather than using an RDF library. This method is both unnecessary, in that there already many libraries available to parse RDF/XML, and error-prone: for anything other than the specific subset of RDF/XML described in the serialization examples of the SBOL specification, the behaviour of libSBOLj is unpredictable (Appendix B).

It seems obvious that given SBOL is built on RDF, using an RDF library would be an easy way to implement an SBOL library. However, the RDF nature of SBOL is very unclear from the specification, which confusingly (and incorrectly, possibly by conflating RDF with RDF/XML) describes RDF as a serialization format [100, p. 16] rather than as an abstract model for capturing information. The specification is punctuated by examples of XML source which create the false impression that in order to implement an SBOL library, one must be able to read a specialized XML format. In reality, RDF/XML serialization already has a W3C specification [53] with many existing implementations, such as RDFLib [101] for Python and Apache Commons RDF [102] for Java — any of which could be used to read and write SBOL.

sboljs (lowercase as is convention among JavaScript libraries) is a complete implementation of SBOL2 for JavaScript, developed as part of this work. Unlike existing SBOL libraries, sboljs makes use of the fact that SBOL is an RDF data model, building on the rdf-ext [103] RDF library for deserialization.

Loading SBOL (deserialization)

As in libSBOLj, sboljs defines a set of classes with member variables mirroring the specification of classes in the data model. The difference is how these classes are populated. Rather than the libSBOLj approach of navigating an SBOL document as an XML tree, sboljs loads SBOL as an RDF graph, then queries the graph using triple patterns. For example, the process of loading `ComponentDefinition` objects is:

- Query the graph using the triple pattern `? a sbol:ComponentDefinition`. For each resulting triple, create an instance of the sboljs `ComponentDefinition` class with the subject URI.
- For each sboljs `ComponentDefinition` instance, match each of its properties using triple patterns and add the values to the class instance. For example, if the URI of the `ComponentDefinition` is `http://sbolstandard.org/lacI`, a triple pattern to retrieve its `dcterms:title` property would be `<http://sbolstandard.org/lac> dcterms:title ?`.
- If the instance references another instance by URI (e.g. a `ComponentDefinition` referencing a `Sequence` via the `sbol:sequence` property, store the URI in the instance and add it to a list of unresolved URIs

At the end of the loading stage, sboljs iterates through all loaded class instances and calls their `link` method, which replaces URI references to actual references to other class instances and removes them from the list of unresolved URIs. The separation of the link stage from the load stage means that the order of loading does not affect the ability of one class to reference another, and the list of unresolved URIs can be used to determine which resources are referenced by the SBOL but not immediately available (i.e. may need to be retrieved).

Saving SBOL (serialization)

While SBOL can be deserialized relatively trivially using an RDF/XML parser, its serialization is slightly more complicated. RDF/XML has both a nested and flat form, and as described in section 2.2, the SBOL specification requires a specific hybrid of the two despite their equivalence as specified by the RDF/XML specification. It is impossible to format data in this hybrid manner unless the code is aware of which objects SBOL deems as "top-level" and therefore should not be nested.

This is a significant problem, as it means that the output of established RDF tooling may not be loadable by SBOL libraries such as libSBOLj that treat SBOL as a specific XML data model instead of a serialization of RDF/XML. It would be much simpler and more interoperable if SBOL libraries considered any equivalent RDF/XML to be equivalent SBOL. However, due to the requirements imposed in the SBOL specification and their enforcement through implementations such as the manual XML-based parser in libSBOLj, it is still necessary to implement SBOL-specific serialization when developing an SBOL library.

This requirement is addressed in sboljs in the same manner as libSBOLj: by serializing SBOL manually using an XML serialization library, with specific serialization functions for each type of SBOL object. For example, the process of serializing `ComponentDefinition` objects is:

- For each sboljs `ComponentDefinition` instance, create an XML `sbol:ComponentDefinition` node
- For each property in the `ComponentDefinition`, add a child node with the property value
- For each property in the `ComponentDefinition` that references another object, add a child node with the URI of the object if the object is a top-level, or nest the node of the object otherwise

While this approach produces a serialization that is compliant with the SBOL specification, it means that each SBOL property exists in three places in sboljs: in the class definition, in the deserialization code, and in the serialization code. A more elegant and generally applicable approach to generating a specific hybrid of nested/flattened RDF/XML was later implemented in the sbolgraph library (section 3.3.2).

Usage

sboljs provides a straightforward JavaScript API which closely follows the SBOL data model (Fig. 3.2), with some notable structural differences from libSBOLj:

- sboljs allows oobjects which are not defined by the specification to be "top-levels" to be constructed and then later added to the document, whereas libSBOLj only allows top-levels to be constructed in the context of their parent.
- There is no concept of a "default URI prefix" in sboljs. The construction of URIs is entirely up to the user.

sboljs is available as a module on the npm package manager[5], and can be used both in the Web browser and via the node.js runtime (Fig. 3.3).

3.3.2 sbolgraph

As SBOL is built on RDF, it follows that standard graph operations should be applicable to navigate the data model. However, none of the current implementations of SBOL take advantage of this. All of the existing SBOL libraries — libSBOLj, and even sboljs and libSBOL which deserialize SBOL using RDF libraries — eventually store SBOL in a rigid in-memory object model, making the application of graph queries impossible. The existence of such an object model is evident in their APIs, in that most properties can only be followed in one direction. For example, in libSBOLj:

- It is possible to retrieve the definition of a component, but not all components with a specific definition (e.g. list all instances of pLac)
- It is possible to retrieve the roles of a component, but not all components with a specific role (e.g. list all promoters)

[5]`https://www.npmjs.com/package/sboljs`

```
/* prefix string for example purposes
 */
const prefix = 'http://sbolstandard.org/example/'

/* create a new empty SBOL document
 */
const doc = new SBOLDocument()

/* create component definitions for our promoter, rbs, coding site, and terminator
 */
const promoterDef = doc.componentDefinition(prefix + 'promoter')
const rbsDef = doc.componentDefinition(prefix + 'rbs')
const cdsDef = doc.componentDefinition(prefix + 'cds')
const terminatorDef = doc.componentDefinition(prefix + 'terminator')

/* add relevant role URIs to the component definitions
 */
promoterDef.addRole(SBOLDocument.terms.promoter)
rbsDef.addRole(SBOLDocument.terms.ribosomeBindingSite)
cdsDef.addRole(SBOLDocument.terms.cds)
terminatorDef.addRole(SBOLDocument.terms.terminator)

/* create an example component definition that will contain all of the components
 * we just created.
 */
const componentDefinition = doc.componentDefinition(prefix + 'exampleComponentDefinition')

/* we have component definitions for the various components.  now we need to
 * create components to instantiate them.
 */
const promoter = doc.component(componentDefinition.uri + '/promoter')
const rbs = doc.component(componentDefinition.uri + '/rbs')
const cds = doc.component(componentDefinition.uri + '/cds')
const terminator = doc.component(componentDefinition.uri + '/terminator')

/* hook up our new component instances with their definitions
 */
promoter.definition = promoterDef
rbs.definition = rbsDef
cds.definition = cdsDef
terminator.definition = terminatorDef

/* add them to the example component definition
 */
componentDefinition.addComponent(promoter)
componentDefinition.addComponent(rbs)
componentDefinition.addComponent(cds)
componentDefinition.addComponent(terminator)

/* serialize the newly created document as RDF/XML and print it to the console
 */
console.log(doc.serializeXML())
```

Figure 3.2: *Example code for sboljs that creates a transcriptional unit in SBOL2. Unlike libSBOLj, sboljs does not have the concept of a "default URI prefix", so the URI for each object is constructed manually. First, four `ComponentDefinition` objects are created using the `componentDefinition` function of `SBOLDocument`, and assigned the roles of promoter, RBS, CDS, and terminator. They are then added to a parent `ComponentDefinition` using `Component` objects created using the `component` function of `SBOLDocument`. Finally, the document is serialized as XML and printed to the console.*

```
$ npm install sboljs
+ sboljs@2.2.2
added 171 packages from 376 contributors and audited 131
packages in 5.423s

$ node
> let SBOLDocument = require('sboljs')
undefined
> let doc = new SBOLDocument()
undefined
> let sequence = doc.sequence()
undefined
> sequence.elements = 'aattggcc'
'aattggcc'
> doc.serializeXML()
'<?xml version="1.0" encoding="UTF-8"?>\n<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dcterms="http://purl.org/dc/terms/" xmlns:prov="http://
www.w3.org/ns/prov#" xmlns:sbol="http://sbols.org/v2#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#dateTime/"
xmlns:om="http://www.ontology-of-units-of-measure.org/
resource/om-2/">\n  <sbol:Sequence rdf:about="">\n
<sbol:elements>aattggcc</sbol:elements>\n    <sbol:encoding
rdf:resource=""/>\n  </sbol:Sequence>\n</rdf:RDF>'
>
```

Figure 3.3: *Example of the installation and usage of sboljs outside of the browser via the node.js REPL. A sequence is added to a document, and then the document is serialized as XML.*

- It is possible to retrieve the participants of an interaction, but not all interactions with a specific participant (e.g. list all interactions which reference IPTG)

It is possible that taking advantage of the RDF nature of SBOL could improve the tractability of existing SBOL knowledge by making SBOL software libraries more powerful. This was explored in this work with the development of *sbolgraph*[6], a new design pattern for SBOL libraries where the SBOL API is simply a view over an underlying RDF graph. Libraries developed in the style of sbolgraph have some significant advantages over their predecessors:

- Dramatically reduced code size, as there is no serialization or deserialization logic and each SBOL property only exists in one place in the library.
- Bi-directional properties. All properties are automatically navigable in both directions without any additional work to maintain references.

sbolgraph provides an API very similar to existing SBOL libraries, even though the underlying implementation is very different. There is no class structure to populate; all of the SBOL data remains in the graph, and every property access in sbolgraph results in the evaluation of a triple pattern (Fig. 3.4). Abstraction is achieved by using getter and setter functions, where the getter performs one or more triple pattern matches and the setter performs pattern deletions and/or triple insertions. For example, the following code fragment:

```
for(let sequence of componentDefinition.sequences) {
 console.log(sequence.elements)
}
```

is valid in both sboljs and sbolgraph. In sboljs, there is nothing happening behind the scenes: `sequences` is an array property of the `ComponentDefinition` class, and `elements` is a string property of the `Sequence` class. In sbolgraph, however:

- The `ComponentDefinition` class has no member variables other than a URI, and a reference to an RDF graph
- The `sequences` property of `ComponentDefinition` is actually a getter function which uses the RDF graph to match the triple pattern `<uri> sbol:sequence ?`, where the URI is the URI of the `ComponentDefinition`
- Each resulting triple contains a URI, which is used to instantiate a `Sequence` object. Like `ComponentDefinition`, `Sequence` only has two member variables: the URI, and a reference to the RDF graph. The list of newly instantiated `Sequence` objects is returned from the `sequences` getter function.
- The `elements` property of `Sequence` is actually a getter function which uses the RDF graph to match the triple pattern `<uri> sbol:elements ?`. The object of the resulting triple is returned as a string from the `elements` getter.

```
class ComponentDefinition {

 Graph graph;   // Reference to the graph
 URI uri;       // The URI of this object to use for lookups

 getter name() {
  return this.graph.match(this.uri, dcterms:title, ?).object
 }

 getter instances() {
  return this.graph.match(?, sbol:component, this.uri).subjects
 }
}
```

Figure 3.4: *Pseudo-code showing how classes for SBOL types are created in sbolgraph, using the ComponentDefinition to sub-component relation (`sbol:component`) as an example. In other libraries such as libSBOLj and sboljs, the ComponentDefinition class stores an array of sub-components. In sbolgraph, the ComponentDefinition class stores nothing, instead simply retrieving the set of sub-components from the RDF graph on demand. This approach also means the relation can trivially be followed in the other direction, retrieving the list of instances of a given ComponentDefinition. In libSBOLj, implementing such functionality would require either maintaining an additional array, or iterating through the entire document.*

This implementation means that sbolgraph effectively has no serialization or deserialization step. SBOL is simply loaded as an RDF graph using existing RDF libraries, and classes are defined to provide views over the graph by evaluating triple patterns of the form $sp?$ (retrieve a specific property for a subject), or $?po$ (retrieve the subject with a specific value for a property). The SBOL data remain in their RDF representation, and are never copied to a separate class structure as with other SBOL libraries.

The RDF graph used by sbolgraph was modified to maintain hash table indexes both for subject to predicate to object, and for object to predicate to subject[7]. Therefore, the time complexity for a triple pattern evaluation is the time complexity of at most three hash table lookups, with an average time complexity of $O(1)$ — no worse than the time complexity of accessing a member variable, but with the added advantage of navigation in both directions without maintaining duplicate state.

Serialization

As described in section 2.2, SBOL serialization requires a specific, nested form of RDF/XML. sboljs addressed this problem by manually serializing SBOL with an XML serialization library and functions defined specifically for each type in the data model. This manual approach is both error-prone, and requires updating the serialization code whenever the SBOL data model changes. In sbolgraph, a more generalised approach was taken:

[6] `https://github.com/udp/sbolgraph`
[7] `https://github.com/udp/rdf-graph-array`

- Define the set of predicates which cause nesting. For example, the `sbol:component` predicate causes nesting of a sub-component and should be in the set, but the `sbol:definition` predicate does not nest the referenced `ComponentDefinition` and therefore should not.

- For each distinct subject `s` in the graph, create an XML node: either an `rdf:Description` if there is no triple matching `<subject> a ?`, or a node of type `o` where `<subject> a <o>`.

- Iterate through all the triples `?s ?p ?o` in the graph. If `p` is an ownership relation, add the XML node corresponding to `o` and as a child of the XML node corresponding to `s`. Otherwise, add the XML node representing `p o` to the XML node corresponding to `s`.

Minimal knowledge of the SBOL data model — a set of ownership predicates — is required for this approach, which is a significant improvement over both libSBOLj and sboljs where the serialization logic had to manually specify every property in the data model. The lack of SBOL-specific code for both serialization and deserialization means that each SBOL property occurs in only one place in the library source code, meaning that it is trivial to add and remove properties to sbolgraph as the standard evolves.

Usage

The usage of sbolgraph is mostly the same as that of sboljs. However, sbolgraph takes advantage of the underlying graph representation to trivially implement functions which are not easy to implement in other SBOL libraries without having to maintain additional state or traverse the entire document tree. For example:

- The `getConstraintsWithThisSubject` and `getConstraintsWithThisObject` functions use the triple patterns `? sbol:subject <uri>` and `? sbol:object <uri>` to find any sequence constraints that refer to a sub-component

- The `getInstancesOfComponent` function uses the triple pattern `? sbol:definition <uri>` to retrieve any instances anywhere in the graph of a given component

Unlike sboljs, the JavaScript version of sbolgraph is implemented in TypeScript [104], an extension to JavaScript which adds strict type annotations. As TypeScript compiles to JavaScript, it is still fully compatible with JavaScript code. Alternatively, code to interoperate with sbolgraph can be written in TypeScript to enable compile-time type-checking when working with SBOL objects.

In addition to the JavaScript implementation, sbolgraph also has an implementation in Python[8]. The Python implementation is a direct port (Fig. 3.5), using the rdflib [101] library in place of rdf-ext.

[8]`https://github.com/udp/pysbolgraph`

```
export default class S2Sequence extends S2Identified {

    constructor(graph:SBOL2Graph, uri:string) {
        super(graph, uri)
    }

    get encoding():string|undefined {
        return this.getUriProperty(Predicates.SBOL2.encoding)
    }

    set encoding(encoding:string|undefined) {
        this.setUriProperty(Predicates.SBOL2.encoding, encoding)
    }

    get elements():string|undefined {
        return this.getStringProperty(Predicates.SBOL2.elements)
    }

    set elements(elements:string|undefined) {
        this.setStringProperty(Predicates.SBOL2.elements, elements)
    }
}
```

```
class S2Sequence(S2Identified):
    def __init__(self, g, uri):
        super(S2Sequence, self).__init__(g, uri)

    @property
    def encoding(self):
        return self.get_uri_property(SBOL2.encoding)

    @encoding.setter
    def encoding(self, encoding):
        self.set_uri_property(SBOL2.encoding, encoding)

    @property
    def elements(self):
        return self.get_string_property(SBOL2.elements)

    @elements.setter
    def elements(self, elements):
        self.set_string_property(SBOL2.elements, elements)
```

Figure 3.5: *Example code from sbolgraph (left) and pysbolgraph (right). Apart from syntactical differences, the code is essentially the same.*

3.4 SBOL Stack

SBOL provides a standardized, machine-tractable data standard for synthetic biology interchangeable between software written in multiple programming languages with the aid of software libraries. What remains unspecified is where SBOL data should be stored. A rich RDF description of a design is of little benefit if it is stored as a file on a harddisk without any connection to other resources or means for the application of queries. Linked Data that is not "linked" is simply "data".

In the RDF world, the standard approach for making data available is to store it in a triplestore and provide access via a SPARQL [63] endpoint. The use of triplestores and SPARQL queries with SBOL has been explored once before by Galdzicki et al. in 2011 [83]. However, their attempt was with SBOL1 and has not been further developed. It also stopped short of providing a general solution for deploying an SBOL-RDF database, instead only suggesting how existing triplestores could be used directly with the SBOL1 data model.

In order to revive this concept and update it for the SBOL2 data model, the SBOL Stack[9] repository was developed, partly as a result of this work. The SBOL Stack is a database for SBOL2 using an RDF triplestore as the backend, which later became the database component of SynBioHub (chapter 6). Its software components comprise:

- A node.js server developed as part of this work which abstracts over the RDF triplestore providing an SBOL-specific API using sboljs. The API can be used to perform tasks such as uploading, downloading, and searching for SBOL data. Each instance of the API can also be connected to other instances, enabling queries to be automatically federated across multiple SBOL-RDF datasets.

- Client libraries for Java and JavaScript developed as part of this work which can be used to programatically connect to the API from software. These libraries are intended to be used in conjunction with an SBOL library such as libSBOLj, sboljs, or sbolgraph.

As demonstrated by the implementation of sbolgraph in section 3.3.2, viewing SBOL as an RDF graph rather than as a "document" has significant advantages in making the data model easier to navigate. The SBOL Stack applies the same principle, but on a much larger scale. When uploaded to the triplestore, the entire concept of an SBOL "document" is dissolved: the triples in the XML file become triples in the triplestore in the same space as any other triples, and can be queried using the powerful mechanism of SPARQL queries.

The API endpoints provided by the SBOL Stack server are implemented as wrappers over federated SPARQL queries with a collation step. For example, the endpoint `/component/count` to count all of the `ComponentDefinition` objects executes the query `SELECT count(?component) WHERE { ?component a sbol:ComponentDefinition }` for its local triplestore, and also executes the same endpoint on any other SBOL Stack instances listed as federation endpoints. The results are then collated by adding together the counts.

[9]`https://github.com/ICO2S/sbolstack`

One of the most important functions of the SBOL Stack compared to using a triplestore directly is the ability to automatically download SBOL documents. As SBOL documents, when uploaded, become nothing but additional triples in the triplestore, retrieving documents back again is a more involved process than simply re-downloading a file. As SBOL has unlimited potential recursion depth, it is not possible to retrieve a complete "document" in a single SPARQL query. The SBOL Stack accomplishes this using a succession of CONSTRUCT queries:

1. Initialize an empty RDF graph, and a set for unresolved URIs initially containing the URI of the SBOL object that has been requested to download
2. For each unresolved URI, retrieve all properties using the SPARQL query `CONSTRUCT { <uri> ?p ?o } WHERE { <uri> ?p ?o }` and add them to the RDF graph
3. Add any objects referenced by SBOL predicates to the set of unresolved URIs, then repeat

The resulting RDF is then loaded using sboljs and re-serialized to produce the specific nested form of RDF/XML required by the SBOL specification, as described in section 2.2.

3.5 Enhancement proposals for SBOL

One of the advantages of SBOL being an open standard is that it is always subject to change. Anyone can write a proposal to improve SBOL, which is then put to the community for a vote. This section describes a number of such proposals, most of which have a corresponding SBOL Enhancement Proposal (SEP) included in Appendix A.

With the exception of section 3.5.1 *Integration of provenance*, the overarching motivation for all of these proposals is to *reduce complexity* in the SBOL data model. While addressing portability as discussed previously is critical to make it possible for developers to add SBOL support to tooling, it is equally important to ensure that the complexity of the standard is at a functional minimum to make the standard as accessible as possible to potential users.

3.5.1 Integration of provenance

One of the fundamental principles of synthetic biology is the application of a "design-build-test-learn" lifecycle. However, the scope of SBOL historically has been narrowly focused on the specification of designs, with little consideration for the description of provenance (i.e. the "who, what, why, and when" of the design process), which is essential to place the design within the broader context of a lifecycle iteration.

The provenance ontology, PROV-O [49], provides a set of terms for formally describing provenance information in RDF such as `prov:Activity`, `prov:Agent`, and predicates such as `prov:wasInformedBy` and `prov:endedAtTime`. These terms map naturally to the synthetic biology domain.

For example, a `prov:Activity` could be used to describe an assembly method in the build stage, or an *in silico* activity such as codon optimisation in the design stage.

While it has always theoretically been possible to use PROV-O with SBOL because of the inherent interoperability of RDF data models, the specification has so far lacked any formal recommendation, thus restricting the ability of SBOL to capture standardised provenance information. A proposal to integrate a recommendation for the use of PROV-O into the SBOL data model is described in SEP 009, *SBOL Provenance* (Appendix A).

3.5.2 Alignment with external ontologies

Despite the fact that SBOL is built on RDF, it is described in the specification in terms of classes with properties as one might implement in an object-oriented programming language. For example, page 18 of the SBOL 2.3 specification [105, p. 18] states:

> *the `Identified` class includes the following properties: `identity`, `persistentIdentity`, `version`, `wasDerivedFroms`, `name`, `description`, and `annotations`*

For a prospective implementer of SBOL, this is an extremely confusing statement for several reasons:

- What the specification refers to as an "identity" property is actually simply the URI of an object; there is nothing called "identity" in SBOL RDF/XML.
- What the specification refers to as "wasDerivedFroms" is actually the `prov:wasDerivedFrom` predicate defined by the PROV-O ontology [49]. Nowhere in the PROV-O specification or SBOL RDF/XML is it referred to using the plural "wasDerivedFroms"; only in the SBOL specification document.
- What the specification refers to as a "name" property is actually the `dcterms:title` predicate from the Dublin Core ontology [40]. Nowhere in the Dublin Core ontology specification or SBOL RDF/XML is it referred to as "name".
- What the specification refers to as "annotations" simply does not exist at all. It actually refers to the existence of any other RDF triples associated with the URI of an SBOL object.

The specification later clarifies [105, p. 20]:

> *Note that several of the properties are not in the `sbol` namespace, but are mapped to standardized terms defined elsewhere:*
>
> - *`identity` is serialized as `rdf:about`*
> - *`wasDerivedFroms` are serialized as `prov:wasDerivedFrom`*
> - *`name` is serialized as `dcterms:title`*

- `description` *is serialized as* `dcterms:description`

This re-definition and subsequent re-mapping of the names of properties adds a layer of confusion, and is unnecessary. RDF is specifically designed for data modelling, and the terms used by SBOL are already defined as part of well-established RDF vocabularies and ontologies. If SBOL simply stated that it "uses the wasDerivedFrom property from the PROV-O ontology", it would not need to define a new name or a mapping for "serialization".

The way the specification currently describes properties implies that SBOL is not an abstract data model, but an object-oriented class hierarchy to be implemented in programming languages. This may be true if SBOL is used only as a file format and exclusively loaded and saved using a library such as libSBOLj, which does use the renamed property names such as "wasDerivedFroms" in its classes. However, those who wish to implement new SBOL libraries or use SBOL directly as an RDF data model — either through RDF libraries as in this chapter, or using RDF technology such as SPARQL queries as will be discussed in depth in chapter 4 — will quickly discover that there is a disparity between what properties are called by the specification, and what they actually look like in the data.

A recommendation to reword the specification to use the correct name for all properties is formalised in SEP 032, *Do not rename ontologicially defined predicates* (Appendix A).

3.5.3 Removal of unnecessary specification of serialization

The serialization examples throughout the specification are redundant, as RDF/XML serialization is thoroughly explained by the RDF/XML specification [53] and has been widely implemented by many software libraries, many of which significantly predate SBOL. By including examples of XML in the description of each SBOL class, the SBOL specification creates the false impression that in order to implement an SBOL library, one must be able to read and write a specialized XML format, which is likely discouraging to developers interested in working with the SBOL data model.

While some examples of what RDF/XML serialization looks like might be helpful, the specification should make it absolutely clear that it is not defining a bespoke XML serialization format for SBOL, and that there are many existing RDF libraries which can work with the data model. This would encourage the use of standard RDF tooling and therefore make it easier for developers to adopt SBOL in tooling.

3.5.4 Addressing structural/functional dichotomy

When SBOL2 introduced interactions, they were added as part of a parallel data model that, despite being part of the same specification, largely sits separately from the core model shared with SBOL1.x. This dichotomy between structure and function in SBOL can cause additional hierarchy to be imposed on the specification of a design where they would naturally exist in the same scope.

Consider a simple example of modelling an auto-regulatory transcriptional unit. It is first necessary to create a `ComponentDefinition` to represent the unit as a whole, and a `ComponentDefinition` for each of the constituent promoter, RBS, CDS, and terminator parts. These parts can then be added to the parent `ComponentDefinition` using Component instantiations. Sequences can be added to each of the parts using `Sequence` objects (Fig. 3.6 A)

However, the auto-regulatory interaction cannot simply be added to the unit. First, it is necessary to create a `ModuleDefinition` which, like the `ComponentDefinition`, describes the unit — but this time, from a "functional" perspective. The unit must then be instantiated into the `ModuleDefinition` using a `FunctionalComponent`. However, it is still not possible to document the interaction, as the promoter and the CDS are contained within the `ComponentDefinition` and are not "exposed" to the level of the `ModuleDefinition`. Therefore, it is necessary to create two `FunctionalComponent` objects at the level of the `ModuleDefinition` — one for the promoter, and one for the CDS — and then two corresponding `MapsTo` relations. Finally, an Interaction can be created in the `ModuleDefinition` to indicate that the CDS has an indirect regulatory effect on the promoter.

The modelled part now has two simultaneous descriptions (Fig. 3.6 B). One uses the component data model to describe the composition of the part, its sequence, and its rudimentary function (via SO terms), and the other uses the module hierarchy to describe how the parts functionally relate to each other. This situation is problematic for a number of reasons:

1. The separation of concerns is a data model detail, and is not scientifically relevant from the perspective of a synthetic biologist modelling a system. The composition of sequences and specification of functional relationships are part of the same design. Therefore, a user should not be expected to be aware of the split in the data model, and consequently every tool with SBOL support will have to hide it through abstraction.

2. Such abstraction is extremely difficult, because the two data models both require hierarchical composition that does not have to be synchronised. For example, from a visual perspective, the obvious way to draw the above part is as a single backbone with four glyphs and an arc connecting the CDS to the promoter. However, the information about the interaction is in module-space, while the composition is in component-space. Rendering these as a single depiction would require a non-trivial "merge" operation to compose the two hierarchies into one.

3. By definition, the result of such a merge operation cannot be represented in SBOL, as the SBOL data model forbids it. Therefore, tooling will have to define a new data model in which the merged design can be represented, along with conversions from SBOL to and from this data model.

This problem can be addressed by unifying the two data models to provide a single, highly descriptive view of a design that can incorporate representations of both structure and function (Fig. 3.6 C, Fig. 3.7). A recommendation for this unification is included in SEP 025, *Merge ComponentDefinition and ModuleDefinition* (Appendix A).

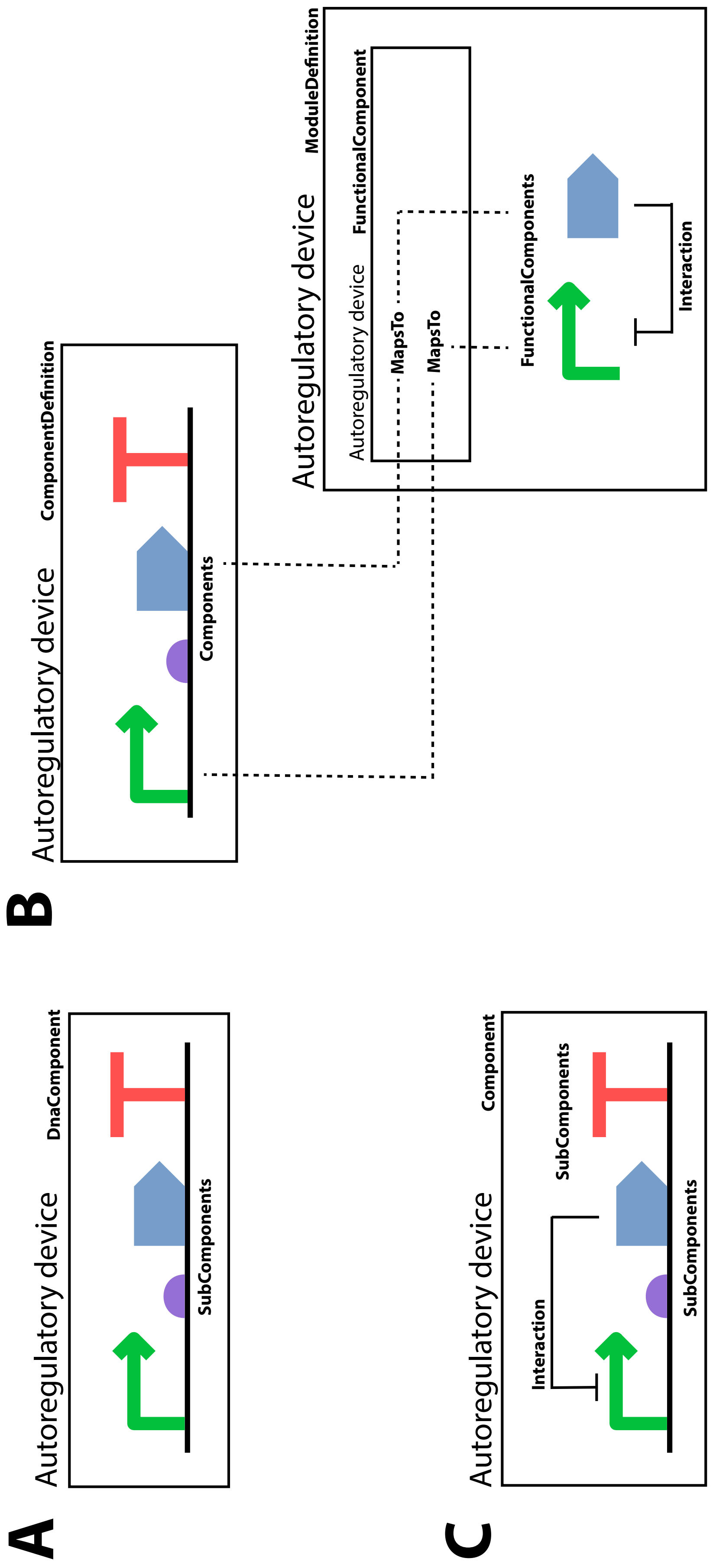


Figure 3.6: *An auto-regulatory transcriptional unit represented in SBOL1 (A), SBOL2 (B), and in a hypothetical SBOL3 with the accepted proposal of merging ComponentDefinition and ModuleDefinition (C).*

3.5.5 More intuitive nomenclature

In SBOL1, the composition relationship between a `DnaComponent` and its sub-components was represented using the `subComponent` predicate. In SBOL2, this predicate was replaced with a "pointer" object in order to allow additional information about the composition to be included, such as the `access` and `mapsTo` properties which determine how the sub-component can be referred to by other components.

While with the SBOL1 convention the name for such a pointer object would have been `SubComponent`, the decision made for SBOL2 was to instead rename `DnaComponent` to `ComponentDefinition`, and call the sub-component pointer object `Component`. This nomenclature has proven to be problematic for several reasons:

- The nomenclature does not match its usage. The `Component` class does not describe a component; it describes the composition relationship between a component and a sub-component.
- In SBOL discussions, what is referred to verbally as a "component" is actually the `ComponentDefinition` object, and what is referred to verbally as a "sub-component" is actually the `Component` object.
- It does not match the convention used in programming languages. For example, the Java class for a file is called `File`, not `FileDefinition`.

SEP 015 describes a proposal to return to the SBOL1 format: `ComponentDefinition` becomes simply `Component`, and `Component` becomes `SubComponent`. Coupled with the removal of `ModuleDefinition` and `Module` in SEP 025, `Component`, and `SubComponent` become the two central classes to the SBOL 3 data model (Fig. 3.7).

3.6 Discussion & Conclusion

A standard computational representation for synthetic biology designs is essential to ensure machine-tractability of past, present, and future design knowledge. It is clear that existing biological file formats such as FASTA and GenBank — while ubiquitous — do not transfer well to synthetic biology, as important information such as the specification of non-DNA parts, hierarchical composition, interactions, and provenance are not captured. SBOL provides a comprehensive solution to this problem. However, the adoption of SBOL has been slow, likely because of some of the issues identified in section 2.2 with a lack of data; lack of tooling; lack of portability; and an over-complicated data model.

The work described in this chapter focused on the issues of portability and over-complication, with the research goals of investigating (a) to how SBOL can be made more accessible to potential users, and (b) how complexity in the SBOL data model can be reduced while preserving expressiveness. Goal (a) was realised through the development of new libraries to allow SBOL to be used from different

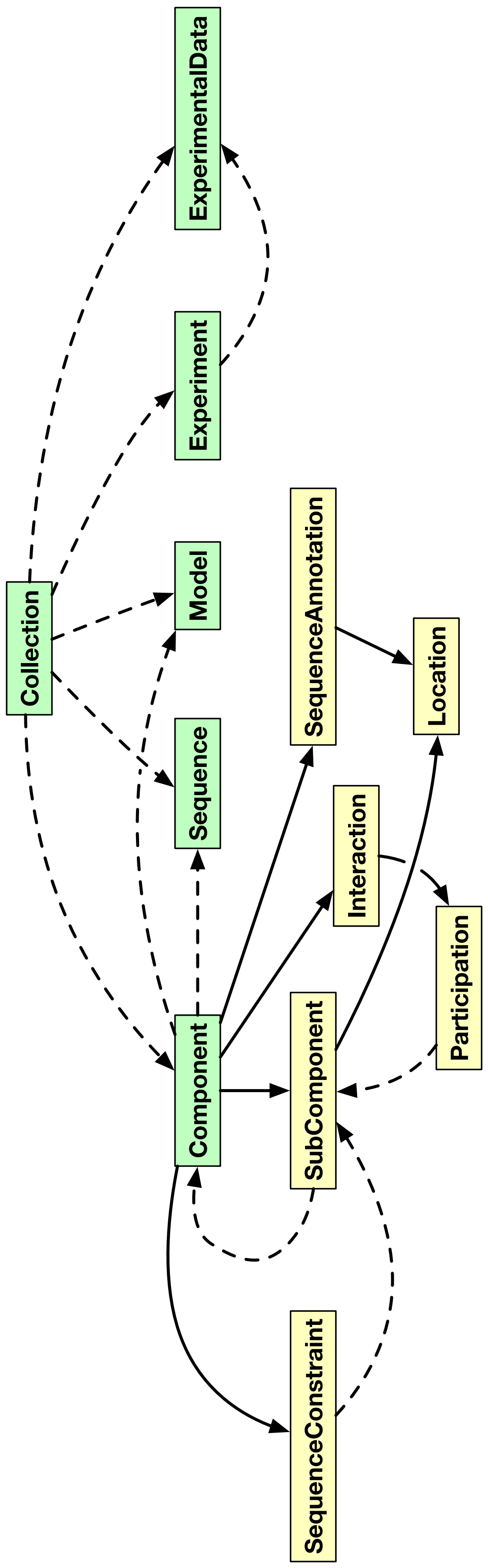


Figure 3.7: *The proposed data model for SBOL3. In contrast to the SBOL2 data model shown earlier in Fig. 2.12,* `ModuleDefinition` *and* `ComponentDefinition` *have been merged into one class (SEP 025) and renamed to Component (SEP 015).*

programming languages, and a database for storing SBOL in a form amenable to existing RDF tooling and SPARQL queries. Goal (b) was realised through a set of proposals to radically simplify the specification in SBOL version 3.

3.6.1 Libraries for SBOL

To encourage developers to use SBOL, it is critical that the portability issue is solved by implementing comprehensive library support. Prior to this work, the only viable option was libSBOLj. Even aside from being restricted to use in Java software, libSBOLj has many issues:

- Its implementation is unnecessarily complex, using manual XML parsing and serialization where the use of existing RDF libraries would be much simpler and less error-prone
- It has inconsistent support for RDF/XML standard; there are many examples of RDF/XML that are not correctly parsed by libSBOLj (Appendix B)
- Its representation of SBOL as an object-oriented class hierarchy is reductive and ignores the potential of SBOL as a graph-based data format. One of the main benefits of SBOL adoption is improved machine-tractability of design information, but — as demonstrated by sbolgraph — the tractability of the data model when accessed via libSBOLj is far more limited than necessary.

The alternative approach proposed in this chapter of using RDF libraries to work with SBOL has clear advantages in simplicity and consistency when compared to manually parsing XML as with libSBOLj. The libSBOL [97] library for C++ and Python, the development of which was contemporary with sboljs and sbolgraph, also uses an RDF-based approach. libSBOLj is intended to make it easier for developers to use SBOL. However, with the aforementioned issues — and the fact that an SBOL library can demonstratably be much smaller while implementing equivalent functionality — it could be argued that libSBOLj can now be replaced, either by either creating a Java SWIG [106] wrapper for libSBOL (as has already been done for Python), or by implementing a new library using the sbolgraph design pattern and an existing Java RDF library. If the proposals for SBOL3 are accepted, they could serve as the catalyst for this change; implementing major changes in libSBOLj with its extensive and complex serialization code would be much more complicated than changing, for example, the structure of facade classes in sbolgraph.

Nevertheless, there are potential caveats to the graph-based approach. For example, memory efficiency: while the memory usage of an RDF graph is highly dependent on the runtime environment and implementation of the RDF library, the data structures necessary for graph lookups are more complex than simple object properties. This is unlikely to be a problem for SBOL designs on the scale of those described herein; for example, the *Gardner et al.* toggle switch example from the SBOL2 specification weighs in at only 507 triples. However, future synthetic biology designs such as those at the genome scale may require a different approach, e.g. using an external triplestore as the data store rather than an in-memory graph.

In general, the fact that new libraries can be created and can work with the exact same data even when they have differing implementations is one of the strengths of SBOL being a well-defined standard. The situation today, with five complete implementations of SBOL across four different languages, is a significant improvement and will enable SBOL to be integrated into software where it previously would not have been possible.

3.6.2 SBOL Stack

If the sbolgraph library enables navigating SBOL as a graph on a small-scale design level, the SBOL Stack enables the same approach for large SBOL datasets. What the SBOL Stack actually provides is minimal: an API to facilitate querying the SBOL data model, and the functionality to download SBOL documents. The true value comes from the underlying triplestore, which is directly applicable to SBOL as a result of the RDF foundation of the data model.

When SBOL is stored in a triplestore, there is no longer the concept of a "document". Rather than a file format, SBOL becomes a vocabulary for Linked Data design knowledge. This representation is highly machine-tractable. SPARQL queries can be used to both navigate and integrate information about parts and designs, a concept which is explored in-depth in chapter 4.

The SBOL Stack contributes to the goal of making SBOL more accessible by demonstrating how SBOL can be used on a large scale with existing Semantic Web tooling, rather than as a file format using bespoke SBOL libraries. This concept is taken further in chapter 6 with the development of SynBioHub, a complete user-facing design repository for synthetic biology built on the same underlying triplestore technology as the SBOL Stack.

3.6.3 SBOL3 proposals

It is unsurprising that a standard developed by an international community has accumulated peculiarities and overcomplexity over time. It is important that the curators of the specification consistently ensure that it remains as concise and intuitive as possible. Fortunately, SBOL is still in its infancy. There is not yet a large existing user base to consider, or vast SBOL datasets with which backwards compatibility is essential. Any significant changes to the data model need to take place now, as it can only become more difficult in the future if the standard becomes successful.

The proposals made in this chapter for SBOL3 address the research goal of reducing the complexity of SBOL without reducing its expressiveness. In particular, addressing the structural/functional dichotomy in the data model by merging ComponentDefinition and ModuleDefinition is a significant simplification. For example, the toggle switch example from section 2.1 can be reduced from a complex hierarchy of FunctionalComponent and MapsTo relations (Fig. 3.8) to a simple component object containing sub-components and interactions (Fig. 3.9). If accepted, this proposal will both simplify the description of simple parts, and make it easier to scale SBOL to larger designs by reducing the necessity of MapsTo relations. Additionally, because MapsTo relations are still permitted, the SBOL2 approach is still equally valid, affected only by a change in nomenclature from modules to components.

3.6.4 Future work

Validation

While both sboljs and sbolgraph allow the serialization, deseralization, and programmatic access of the SBOL data model, they do not enforce the validation rules from the specification. Many of these validation rules are essentially validation rules for RDF/XML syntax which are already performed by the RDF library, such as `sbol-10201` which mandates that the "identity property of an Identified object is REQUIRED and must contain a URI". Others are unenforceable, for example `sbol-10202` "The identity property of an Identified object MUST be globally unique", which is presumably based on the assumption that SBOL only ever exists in an XML file and not in an RDF graph or triplestore where it is impossible to have multiple nodes with the same URI.

Considering all of the SBOL libraries apart from libSBOLj are now built on RDF libraries rather than XML parsing, it will likely be necessary to rethink many of these validation rules and work out which would still be applicable when the majority of SBOL does not ever pass through an XML parser. Validation can then be re-implemented in the new RDF-based SBOL libraries.

One possible way to implement validation rules would be as SPARQL queries. For example, noncompliance with `sbol-10512` "The sequences property of a ComponentDefinition is OPTIONAL and MAY contain a set of URIs" — which would be re-worded on acceptance of SEP 032 as "The sequence property of ComponentDefinition is OPTIONAL and MAY be used to refer to one or more Sequence objects" — could be detected using the SPARQL query:

```
SELECT ?cd WHERE {
    ?cd a sbol:ComponentDefinition ;
        sbol:sequence ?sequence .

    FILTER NOT EXISTS {
        ?sequence a sbol:Sequence
    }
}
```

Using SPARQL queries to implement validation rules would enable entire triplestores of SBOL data to be validated at once, which could be particularly useful considering the recent rise of triplestore-backed repositories for large SBOL datasets driven by adoption of SynBioHub (chapter 6).

Library generation

Porting sbolgraph from TypeScript to Python was not a difficult task, and could easily be repeated for any language with an RDF library. However, each new library for SBOL

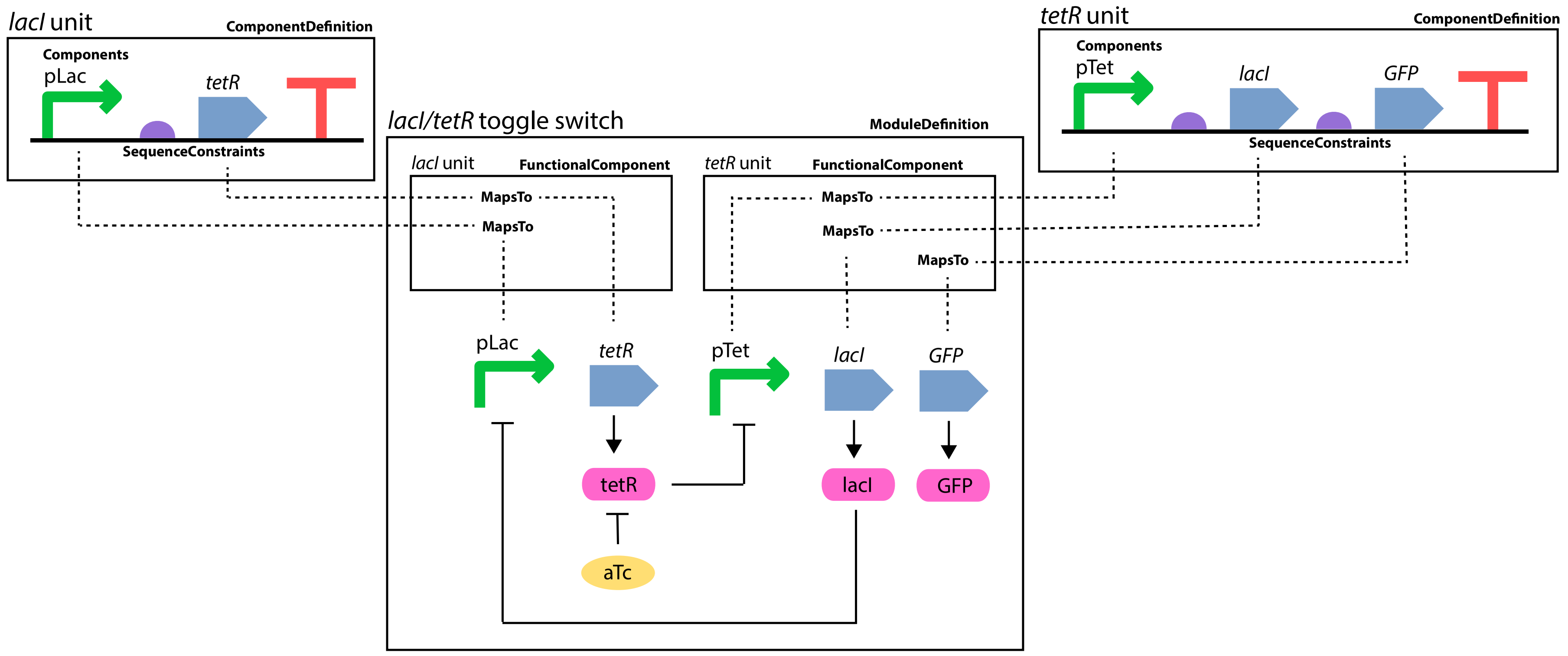


Figure 3.8: *The Gardner* et al. *toggle switch from section 2.1 represented using SBOL2 classes. In SBOL2, interactions can only exist in modules, which are a separate hierarchy from components. Describing the toggle switch therefore requires the instantiation of both a ModuleDefinition for the interactions and ComponentDefinitions for the transcriptional units, connected using MapsTo relations in the FunctionalComponent instantiations.*

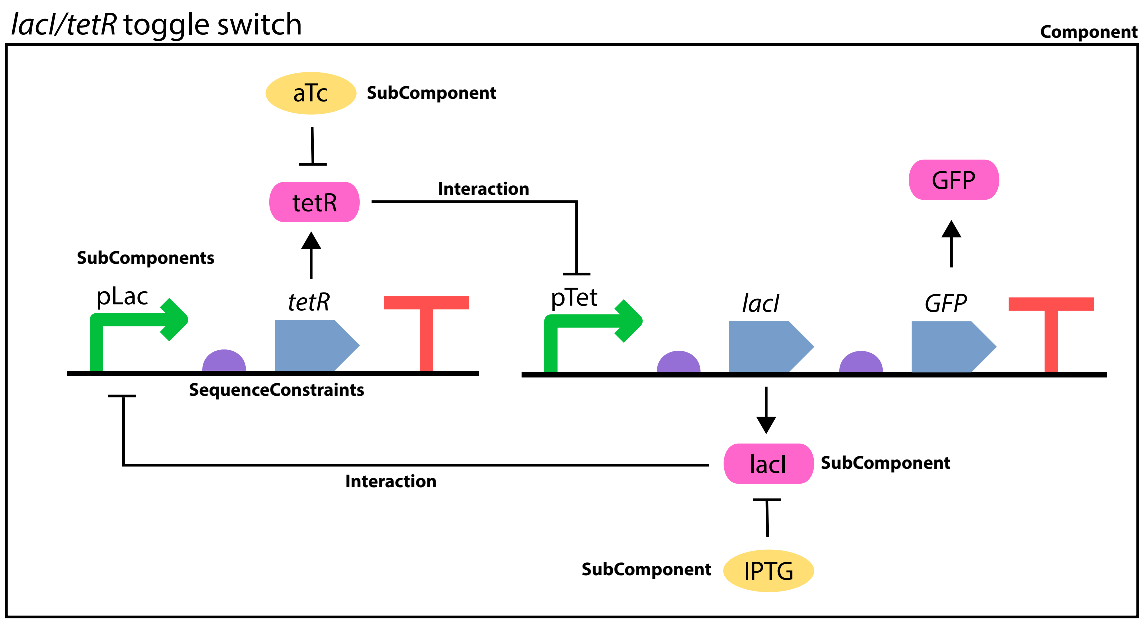


Figure 3.9: *The Gardner* et al. *toggle switch from section 2.1 represented using a hypothetical SBOL3 data model in which SEP 015 "Simplification of SBOL class names" and SEP 025 "Merge ComponentDefinition and ModuleDefinition" are accepted. In contrast to the SBOL2 example in Fig. 3.8, both the interactions and the structural components are located within a single Component.*

brings a new codebase which must be maintained to ensure synchronisation with the specification.

SBOL-OWL [107] is a recent effort to develop an ontology for SBOL, describing the data model in a machine-tractable format rather than a free-text specification. It may be possible to combine the minimalistic approach for library implementation pioneered by sbolgraph with the formal specification of SBOL-OWL by iterating the SBOL-OWL specification of the data model model and automatically generating libraries for different languages.

Abstraction

One of the major areas for future work in making SBOL more accessible is abstraction. While the software libraries described in this chapter bring the possibility of implementing SBOL to developers using different programming languages, they are directly exposing the underlying data model. The recent work of Bartley et al. [97] in the development of libSBOL demonstrates the alternative approach of hiding the underlying data model from developers, allowing SBOL to retain its complexity while becoming easier for developers to implement.

Having said this, it is important that abstraction is not seen as an alternative to addressing the underlying complexity of the data model. Any such complexity must be either justifiable or removed, as even with abstraction a complex data model complicates the development of libraries and discourages involvement with the standard. For example, abstracting over the split between the structural and functional hierarchies by providing a unified view would be pointless if there is no clear advantage to having the split in the data model. The standard and its software infrastructure are developed together, and future developers should never be resigned to the assumption that the data model cannot be changed.

SBOL is already itself an abstraction over the highly complex knowledge associated with bio-engineering. If it becomes necessary to build further abstraction layers to make SBOL usable, SBOL has introduced further complexity into an already challenging domain.

Conclusion

The need for a machine-tractable representation of design information, and its realisation in the form of the SBOL standard, were established prior to this work. However, SBOL is not yet commonly used as part of the design process. This chapter explored some of the interconnected reasons why this might be the case: lack of data, lack of tooling, non-portability, and complexity.

Solutions to the issue of non-portability were proposed in the form of three new software libraries for SBOL: sboljs, sbolgraph, and pysbolgraph. These libraries will make it easier for tool developers to implement SBOL support, which in turn may result in more SBOL data becoming available. Additionally, the greatly simplified RDF-centred model for building an SBOL library pioneered by sbolgraph may make it easier to implement SBOL support in further programming languages.

Finally, the issue of complexity was addressed with a series of proposals aimed at simplifying the SBOL data model for its version 3. Hopefully, the contributions to the SBOL standard and its associated software infrastructure described in this chapter will help to accelerate the adoption of SBOL by the wider community.

Part II

Data Integration

4. Data harmonization using Linked Data Fragments (LDF)

4.1 Introduction

Just as one developing an electronic circuit uses datasheets describing the properties of each electronic component, a synthetic biologist uses a combination of many different datasets that describe the predicted or previously observed behaviour of biological parts. As described in section 2.3, the machine-tractability of this existing knowledge is essential to optimizing the synthetic biology design stage. However, APIs for programmatic access are often missing or have low availability, and non-standardized data representations are often used which hinders interoperability.

Despite these issues, the value of the data themselves — and of the databases for preserving them — cannot be understated. This disparity of valuable data contained by less-than-perfect software is captured in a Hacker News comment[1] by a developer from the Internet Archive:

> “Archive.org itself had tons of kluges and several crude bits of code to keep it going but the aim was the keep the data secure and it did that. Someone . . . likened it to a ship traveling through time. Several repairs with limited resources have permanently scarred the ship but the cargo is safe and pristine. When it finally arrives, the ship itself will be dismantled or might just crumble but the cargo will be there for the future.”
>
> — *Noufal Ibrahim, Developer, Internet Archive*

The concept of Linked Data described in section 2.3 seems an ideal solution to provide a unified view of existing design knowledge. A distributed, ontology-backed RDF graph of knowledge about parts for synthetic biology would both significantly optimise information gathering in the design stage, and make it easier to computationally mine data for design automation. The SBOL standard for the representation of design knowledge is built using RDF, and RDF already has tooling for

[1]`https://news.ycombinator.com/item?id=18116365`

data integration such as federated SPARQL queries [63] and Linked Data Fragments [71].

Unfortunately, the application of Linked Data to SynBio is limited by a problem as old as the Semantic Web itself: much data is not yet represented in RDF. Converting design knowledge to RDF ahead of time to enable data integration would negate both of the significant benefits of query federation compared to data warehousing: that federation accesses the current, live version of a dataset, and does not require extensive computational resources.

The hypothesis of this research chapter is that it may be possible to provide an RDF view over a non-RDF dataset *at the time of query execution* by coupling the recent innovation of client-side querying pioneered by Linked Data Fragments with a custom server providing dynamic RDF conversion.

Attribution: It is important to clarify that Linked Data Fragments (LDF) and the LDF client which allows the execution of SPARQL queries over LDF servers were **not** developed as part of this work. The results of this work are (a) a novel method for modelling non-RDF datasets as RDF, and (b) ldf-facade, a new server compatible with the existing ldf-client which implements this method.

4.2 Linked Data Fragments for non-RDF data sources

As described in section 2.3, Linked Data Fragments (LDF) are a recent innovation where the complexity of executing a SPARQL graph query is moved from the server-side to the client-side [72]. In order to achieve this, the LDF client breaks down the SPARQL query into a series of triple patterns, where each pattern consists of an optional subject, predicate, and object. Unlike a SPARQL endpoint which must respond to any possible SPARQL query and process the query on the server-side, an LDF server only has one job: given a triple pattern, it must return the set of all triples which match that pattern.

Processing SPARQL queries can be highly computationally expensive. LDF is designed to reduce the load on servers by moving complexity to the client, giving the server less to do and therefore allowing it to respond to more requests. To this end, there are several implementations of LDF servers available, all of which are designed to enable exposing an existing RDF datasource (such as a SPARQL endpoint or data dump) as LDF.

The hypothesis of this chapter is that LDF can be repurposed to provide RDF views over non-RDF datasources at the time of query execution. Although performing a SPARQL query over a dataset that is not currently represented in RDF would be a daunting task, matching a triple pattern — the only task of an LDF server — is less so. For example, evaluating the triple pattern `<subject> ?p ?o`, or $?po$, simply means “retrieve all properties of `<subject>`”, a capability which any database or Web service should be capable of.

Testing this hypothesis requires research in two areas. The first is theoretical: given a non-RDF service such as an API, how can it be modelled in terms of the triples it could potentially provide? The second is more practical: though LDF servers are simpler than SPARQL endpoints, they are still expected to handle tasks which could reasonably be expected of a provider of a large dataset, such as the pagination of results. Implementing such functionality over a dynamic conversion from another datasource will require a layer of abstraction to ensure that upstream and downstream state remain consistent.

4.3 Modelling non-RDF services using triple patterns

While the majority of Web services and APIs do not use an RDF representation, it may still be possible to model them in terms of the subject-predicate-object model required by RDF. For example, if a Web service has the functionality to return all of the properties for a given resource identifier, it can be thought of in RDF terminology as having the ability to return all of the properties associated with a given subject, or, more formally, to evaluate the triple pattern $s??$.

Web services are usually much more restrictive than an indexing RDF triplestore, because triplestores are typically optimised for all possible triple pattern permutations. Given the option of the subject, predicate, and object of the pattern being either bound or unbound, there are $2^3 = 8$ possible patterns. While some of these patterns are commonly supported by APIs, such as $s??$ (list all properties given a subject), others such as $?po$ (list all subjects with a given value for a property) are less so. Thus, the method for evaluating a service for potential wrapping into LDF consists of answering two questions:

1. Which triple patterns can an API evaluate? For example, if all properties of a subject can be retrieved (as with most APIs), $s??$ can be evaluated. If the API provides “advanced search” functionality, it can often be used to implement complex triple patterns such as $?po$ by adding criteria to the search.
2. For the triple patterns that the service API is not able to respond to, how can we *generalise* or *specialize* results from other API calls to achieve the desired patterns?

Even with a restricted set of possible patterns, it is possible to derive an evaluator for one pattern from an evaluator for another using two operations:

1. **Generalization:** Make a bound part of the pattern s, p, or o work as an unbound $?$ by iterating all of its possible values. For example, the pattern $???$ can be answered when only $s??$ is available by iterating all possible subjects and filling in the s.
2. **Specialization:** Make an unbound part of the pattern $?$ work as a bound s, p, or o by filtering the results. For example, the pattern $sp?$ can be evaluated when only $s??$ is available by filtering the results to match only the specified predicate.

For example, consider a service that provides the functionality to retrieve the properties of a record given its identifier. This functionality can satisfy a $s??$ pattern, because given a subject (the identifier), it can return all of the triples with that subject (the properties of the record). However, it is too general to satisfy a $sp?$ pattern, because it does not select for a specific property. The specialization operation would solve this by first evaluating $s??$ to retrieve all of the properties, and then filtering out any properties where the predicate does not match the triple pattern.

Generalization works in the opposite direction, by generalising a bound part of the triple pattern to unbound. If the service only satisfies $s??$ by allowing all properties of a

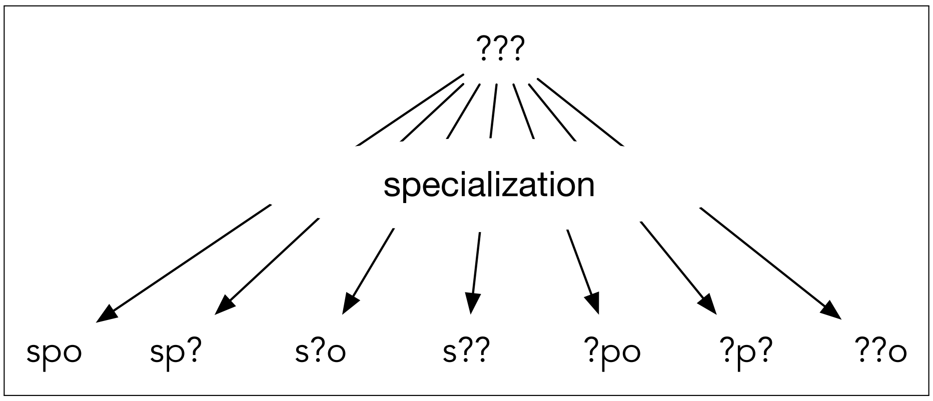


Figure 4.1: *If the pattern $???$ can be evaluated, all of the other pattern permutations can be evaluated by iterating and filtering the resulting triples in $O(|T|)$ where T is the set of all iterable triples.*

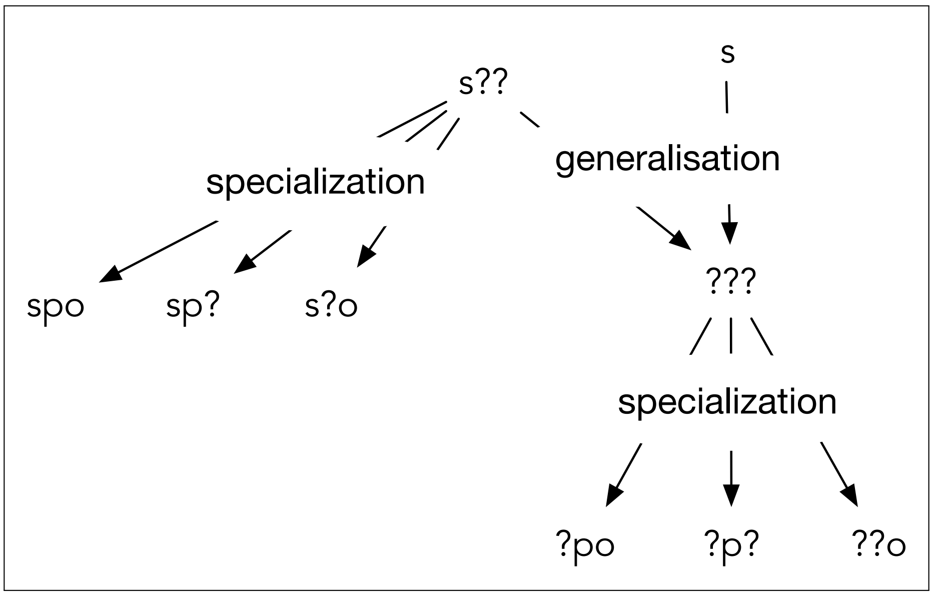


Figure 4.2: *If the pattern $s??$ can be evaluated and it is also possible to retrieve a list of subjects (represented here as s), it is possible to derive an evaluator for $???$ by iterating each subject to bind as the subject of successive $s??$ queries.*

specific object to be retrieved, it is not possible to retrieve all properties of all objects, or $???$. However, if it is possible to list all of the available subjects (e.g. a list of all record identifiers), $s??$ can be generalized to $???$ by using each identifier in the list, in turn, to fill in the subject of the triple pattern.

4.3.1 Time complexity

Where T is the set of all iterable triples, the result set R of a "match" — i.e. a filtered set of triples based on a subject/predicate/object pattern — will always be a subset $R \subseteq T$.

In practice, results are not directly accessible as a complete set, but instead iterable through an $iterate$ function that returns at least one triple t where $t \in R$. The time complexity of retrieving the entire result set is therefore $O(O(iterate) \cdot |R|)$. Where $iterate$ is $O(1)$, this becomes $O(O(1) \cdot |R|) = O(|R|)$, i.e. linear time depending on the size of the result set.

In the case of specialization, the iterate function is called repeatedly until a matching triple is found. This makes specialization a linear time operation: while the process of checking whether a triple matches is a simple $O(1)$ equality test, a matching triple may not be found until the end of the result set, making the worst case for specialization of a result set R $O(|R|)$.

Fig. 4.1 shows specialization of a $???$ iterator to find matches for any possible pattern. This is equivalent to traversing and comparing unsorted triples one at a time. While this approach allows any triple pattern to be matched, in the case of the $???$ pattern $|R| = |T|$, so the complexity to find a single matching triple in a set T is a linear $O(|T|)$.

If a constant time $s??$ (get all properties for a subject) iterator is provided as in Fig. 4.2, it can be specialized to evaluate spo, $sp?$, and $s?o$. This reduces their time complexity to $O(|R|)$, where R is the result set for $s??$ (i.e., $|R|$ is the number of triples with the specified subject). As $R \subseteq T$, $|R| \leq |T|$, so specializing $s??$ performs equal to or better than specializing $???$.

Unlike specialization, generalization is usually a constant time operation. For example, if a constant time $s??$ (get all properties for a subject) iterator is provided and it is also possible to enumerate all subjects in constant time, $s??$ can be generalized to provide a constant time iterator for $???$.

4.4 ldf-facade: An intelligent server for LDF

In order to test the idea of modelling non-RDF services as RDF, the aforementioned concepts are implemented in *ldf-facade* [2], an "intelligent" LDF server which allows logic to be used to dynamically respond to triple patterns. While existing LDF servers are typically used to expose a datasource with *less restrictive* querying capabilities (e.g. a SPARQL endpoint) as a service with *more restrictive* querying capabilities (triple pattern fragments), ldf-facade is designed to expose a datasource with *more restrictive* querying capabilities — such as a HTTP API — as triple pattern fragments, which are often *less restrictive* than the API.

[2] `https://github.com/udp/ldf-facade`

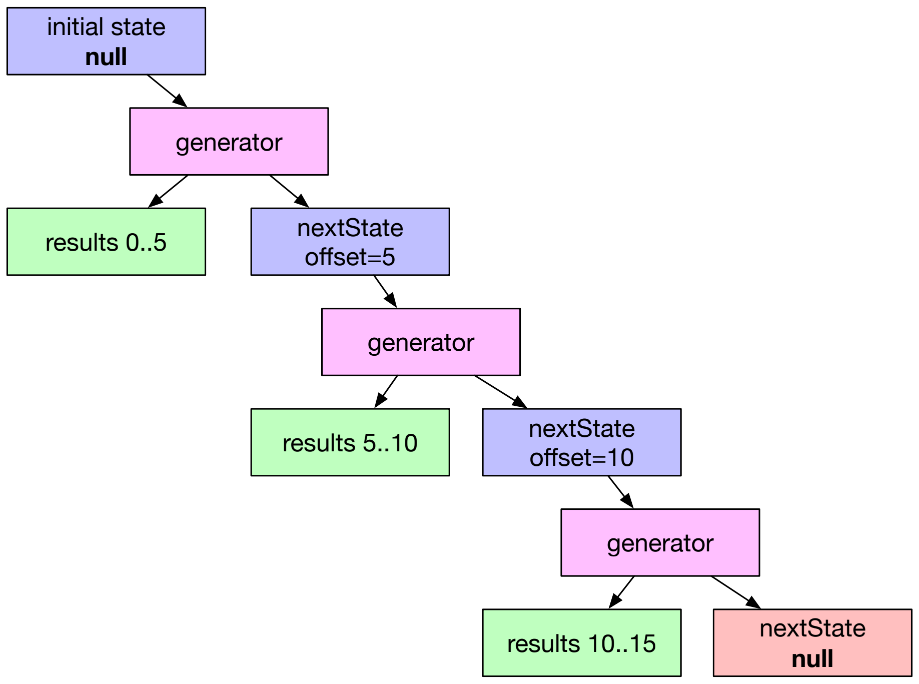


Figure 4.3: *An example of how a generator works in ldf-facade. The first time the generator runs, there is no initial state. It then emits the initial set of results (for example, the first 5), and the state parameter that can be used to retrieve the next set (for example, the next results from offset 5). The next time the generator runs, it receives offset 5, emits the results from 5 to 10, then returns the next state as the next results from offset 10. A consumer of the generator can "rewind" the result set at any time by passing a previous state.*

In response to a triple pattern, an LDF server must respond with an initial set of result triples; the total number of expected results; and the "controls": URLs which can be used to retrieve the previous or next page in the result set. In the case of a SPARQL query, the existing LDF client uses this functionality by first retrieving the first pages for each triple pattern in order to assess the total number of expected results. The pattern with the lowest number of expected results (i.e., highest selectivity) is retrieved first to make the query is cheap as possible.

The ldf-facade server implements this functionality using a system of composable generators, where a generator has the form `state? -> { results, nextState? }`. To iterate the entire result set of a generator, the consumer first calls the generator with no state to retrieve any initial results. If the generator returns a "next state", it can be called again with that state to retrieve the next results, until no next state is returned. The actual value of the state is opaque and entirely up to the generator; in the case of a generator that retrieves data from an upstream API it might be a database-specific offset or identifier.

Generators are expected to be deterministic, in that if called multiple times with the same state, the callback should return the same results each time. This allows the caller to "rewind" the result set by calling the generator with a previous state (Fig. 4.3). The ability to navigate result sets in both directions is necessary to comply with the specification of LDF, which expects servers to provide `hydra:prev` and `hydra:next` properties along with each page of results to supply URLs for the previous and next page.

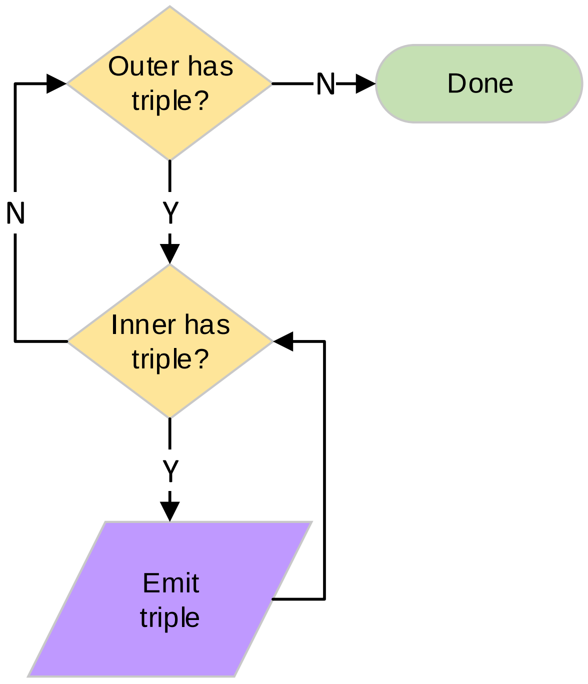


Figure 4.4: *The join operation defined for ldf-facade can compose two generators. For each triple emitted by the outer generator, the inner generator is called and its results, if any, are emitted. Join operations are used for both generalization — where the outer is the enumerator for subject/predicate/object and the inner is the generator to be generalized — and for specialization, where the outer is the generator to be specialized and the inner is the filter function.*

```
generator ???() {
 while(subject = s.iterate()):
  while(predicate, object = s??.iterate(subject)):
   yield triple(subject, predicate, object)
}

generator ?po(predicate, object) {
 while(triple = ???.iterate()):
  if triple.predicate == predicate and triple.object == object:
   yield triple
}
```

Figure 4.5: *Pseudo-code implementation of the model depicted in 4.2. The $???$ generator calls the s generator to retrieve the next subject, and then uses the subject as the parameter to the $s??$ generator to retrieve any matching triples. The $?po$ generator is then implemented using the $???$ generator. Note that the implementation of the $?po$ generator is inefficient here, as in the worst case it will make many comparisons against triples that do not match.*

Composition of generators is achieved using a join operation. Joining two generators returns a new composite generator which passes its input to a first generator (the "outer"); the output of the outer to another generator (the "inner"); then emits triples only if emitted by the inner (Fig. 4.4). Composite generators maintain a composite state containing both the outer and inner state to retain the same deterministic attributes as their composed generators.

The server allows generators to be registered to respond to certain available triple patterns. For example, if a non-RDF API was formalised as capable of responding to a $s??$ pattern (retrieve all details about a subject), a generator for $s??$ can be registered in the server with a generator that uses the subject URI to identify a resource, makes a call to the API, translates the result to RDF triples, and returns the triple result set.

In response to a request by the LDF client for a specific pattern, the library selects the closest generator and matches it to the query using generalisation and/or specialisation techniques. Both generalisation and specification are implemented as generators using join for composition. For example, when the closest available generator is $s??$ but the requested triple pattern is $???$, a generator which enumerates all subjects can be joined upstream of the generator for $s??$ to generalise it to $???$. Likewise, if the closest available generator is $???$ but the requested triple pattern is $s??$, a specializing generator which filters based on s can be added downstream of the $???$ generator to specialize it to $s??$.

4.4.1 Tracking state

One of the most important abstractions provided by ldf-facade is the mapping between upstream state and downstream state. As an analogy, imagine the facade service as the reader of a book in an unknown language. A translation service is attempting to a fool a third party — the client — into believing that they instead have an English copy of the book. The client can ask the facade service to read a page from the book, to turn to the next page, or to turn to any of the pages previously read.

If the client asks for the facade service to read a page of the book, the translation service translates a page into English and reads it back to the client. Sometimes the

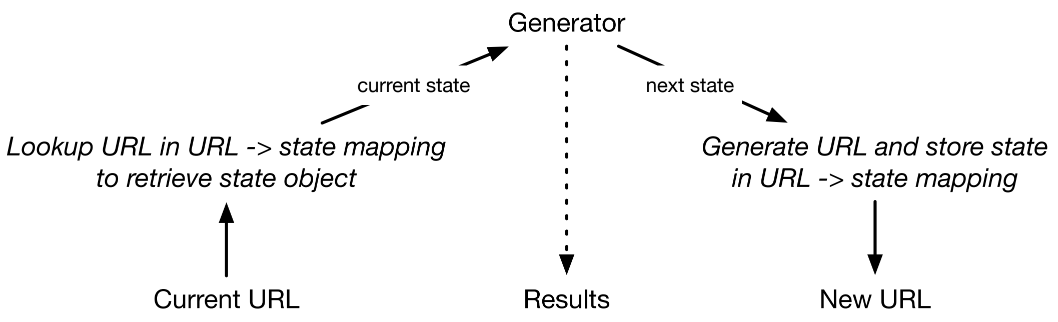

Generator
current state
next state
Lookup URL in URL -> state mapping to retrieve state object
Generate URL and store state in URL -> state mapping
Current URL
Results
New URL

Figure 4.6: *Process of performing an iteration. Each URI pointed to by* `hydra:next` *or* `hydra:previous` *predicates maps to a token known by the service. The service can then resolve the token to an opaque state object to pass to the iterator, which produces a) zero or more result triples and b) a new state object, which is then mapped to a newly generated token used as part of the next URI. Crucially, all iterators are deterministic: if the same state object is later passed again, the same results will be returned.*

English translation of the page might be longer than the original version, in which case the facade service stops early and tells the client that they have finished the page. Sometimes the English translation might be shorter than the original version, in which case the facade service turns over to the next page and reads part of it without telling the client. This results in the client having a different set of page numbers than the facade server. In order to satisfy the requirement that the client can ask for any of the previous pages and maintain the illusion, the facade service needs to remember the mapping between page numbers used by the client and page numbers in the original text.

In the case of ldf-facade, the original book could be the answer to a query from an upstream datasource. Where specialization is applied to the query results — i.e., upstream returned results other than those the client requested — the pages of the resulting triples may no longer map directly to the pages of the source result set. Page numbers are a simple example of the kind of state that may need to be maintained by the facade service. In practice, the information needed to retrieve part of a result set may be far more complicated, particularly where multiple sources of data are being combined e.g. in pattern generalization. ldf-facade is responsible for keeping track of which generator requires which state to function, and mapping states to unique URLs to provide the illusion of a contiguous paginated dataset to the client.

4.5 Example usage: RDFizing DistroWatch

As an initial proof-of-concept for the application of ldf-facade as a general tool for RDF in the computer science domain rather than specifically for biology, the DistroWatch [108] database of details about open-source operating systems was converted to an RDF datasource. DistroWatch serves as a good example as it is a large, continually updated database that is not represented as RDF.

As described at the beginning of this section, the method for producing an LDF view over a non-RDF datasource requires answering two questions: what kind of triple patterns the service is able to respond to, and how those triple patterns can be generalized or specialized to achieve others that are unavailable.

The DistroWatch website has an advanced search form (Fig. 4.7) which allows `?po` to be evaluated (i.e. when a predicate and object have been specified and we want to list all subjects that match). The search form can also be submitted without any criteria, enabling a list of all subjects to be obtained. Given a distribution, we can retrieve its details by visiting its detail page (Fig. 4.8), which allows `s??` to be evaluated. One possible way to accomplish the eight triple patterns, therefore, would be:

- **`s??`**: provided by the distribution detail page
- **`?po`**: provided by the Advanced Search form
- **`spo`**: evaluate `s??`, and specialize to filter by `p` and `o`
- **`sp?`**: evaluate `s??`, and specialize to filter by `p`
- **`s?o`**: evaluate `s??`, and specialize to filter by `o`
- **`?p?`**: use list of subjects to generalize `s??` to `???`, then specialize to filter by `p`
- **`??o`**: use list of subjects to generalize `s??` to `???`, then specialize to filter by `o`
- **`???`**: use list of subjects to generalize `s??`

In ldf-facade, implementing this functionality is as simple as registering the two patterns *s??* and *?po* with functions that query DistroWatch and return a set of triples. The generalisation and specialization operations then happen automatically, meaning that SPARQL queries can be executed using an LDF client (Figs. 4.9, 4.10).

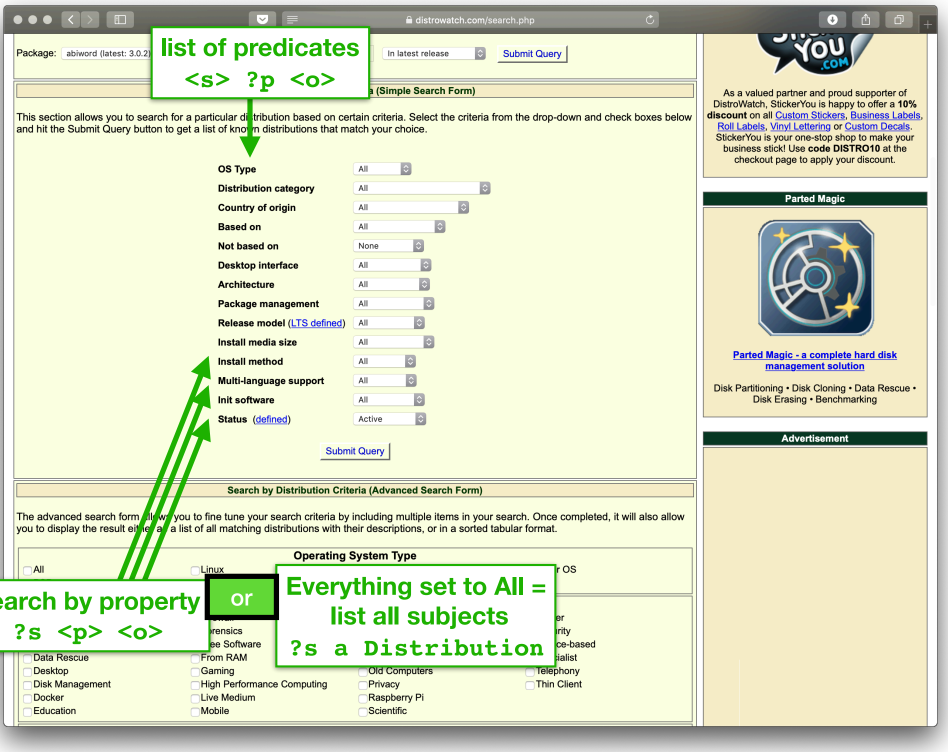
4.6 SynBio applications

As discussed in depth in chapter 3, one of the most useful attributes of SBOL is that it is an RDF data model, composed of RDF triples. Consequently, it is possible to store SBOL RDF in an RDF triplestore, allowing SBOL knowledge to be navigated using graph queries. The ldf-facade server described in this chapter provides the means to provide RDF views over non-RDF datasources. Would it be possible to leverage ldf-facade to create a virtual SBOL repository by creating an SBOL-RDF view over a non-RDF dataset?

4.6.1 JBEI-ICE RDF using ldf-facade

The JBEI-ICE repository reviewed in section 2.4 is a good test case for the possibility of dynamic conversion to SBOL-RDF. While it both has an API and provides access to parts as SBOL, it does not function as an RDF triplestore. The SBOL representation in ICE is limited to the description of parts using `ComponentDefinition`. Other data which could be represented in SBOL instead has a bespoke data representation, e.g. ICE defines the concept of "folders" instead of using the SBOL concept of collections.

The ICE API is a typical HTTP API using JSON to encode response data. For example, calling the endpoint `/rest/collections/FEATURED/folders` on the ACS Synthetic Biology ICE repository returns a JSON array of folders of the form:



distrowatch.com/search.php

Package: abiword (latest: 3.0.2) In latest release Submit Query

a (Simple Search Form)

This section allows you to search for a particular di tribution based on certain criteria. Select the criteria from the drop-down and check boxes below and hit the Submit Query button to get a list of kno n distributions that match your choice.

OS Type	All
Distribution category	All
Country of origin	All
Based on	All
Not based on	None
Desktop interface	All
Architecture	All
Package management	All
Release model (LTS defined)	All
Install media size	All
Install method	All
Multi-language support	All
Init software	All
Status (defined)	Active

Submit Query

Search by Distribution Criteria (Advanced Search Form)

The advanced search form llows you to fine tune your search criteria by including multiple items in your search. Once completed, it will also allow you to display the result ei e a a list of all matching distributions with their descriptions, or in a sorted tabular format.

Operating System Type

☐ All	☐ Linux		r OS

			er
	rensics		rity
	ee Software		ce-based
☐ Data Rescue	☐ From RAM		cialist
☐ Desktop	☐ Gaming	☐ Old Computers	☐ Telephony
☐ Disk Management	☐ High Performance Computing	☐ Privacy	☐ Thin Client
☐ Docker	☐ Live Medium	☐ Raspberry Pi	
☐ Education	☐ Mobile	☐ Scientific	

As a valued partner and proud supporter of DistroWatch, StickerYou is happy to offer a **10% discount** on all Custom Stickers, Business Labels, Roll Labels, Vinyl Lettering or Custom Decals. StickerYou is your one-stop shop to make your business stick! Use **code DISTRO10** at the checkout page to apply your discount.

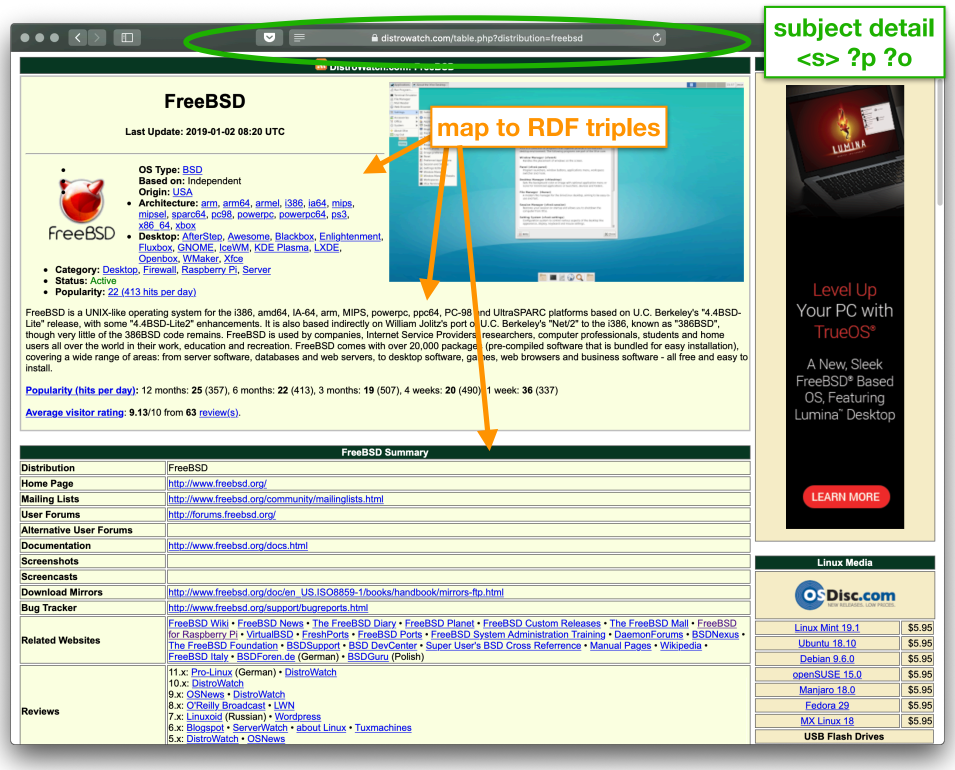
Parted Magic

Parted Magic - a complete hard disk management solution

Disk Partitioning • Disk Cloning • Data Rescue • Disk Erasing • Benchmarking

Advertisement

Figure 4.7: *A screenshot of the "Advanced Search" form of the DistroWatch website, annotated to show how it can be used to implement the evaluation of RDF triple patterns. The search criteria names form the list of predicates, and each criteria item that can be specified effectively allows binding a predicate and an object in a search, satisfying the case where* p *and* o *are bound but* s *is not. Searching without specifying any criteria acts as a list for all subjects.*


subject detail
<s> ?p ?o
map to RDF triples
distrowatch.com/table.php?distribution=freebsd
FreeBSD
Level Up Your PC with TrueOS®
A New, Sleek FreeBSD® Based OS, Featuring Lumina™ Desktop
LEARN MORE
Linux Media
OSDisc.com
USB Flash Drives

FreeBSD

Last Update: 2019-01-02 08:20 UTC

- **OS Type:** BSD
- **Based on:** Independent
- **Origin:** USA
- **Architecture:** arm, arm64, armel, i386, ia64, mips, mipsel, sparc64, pc98, powerpc, powerpc64, ps3, x86_64, xbox
- **Desktop:** AfterStep, Awesome, Blackbox, Enlightenment, Fluxbox, GNOME, IceWM, KDE Plasma, LXDE, Openbox, WMaker, Xfce
- **Category:** Desktop, Firewall, Raspberry Pi, Server
- **Status:** Active
- **Popularity:** 22 (413 hits per day)

FreeBSD is a UNIX-like operating system for the i386, amd64, IA-64, arm, MIPS, powerpc, ppc64, PC-98 and UltraSPARC platforms based on U.C. Berkeley's "4.4BSD-Lite" release, with some "4.4BSD-Lite2" enhancements. It is also based indirectly on William Jolitz's port of U.C. Berkeley's "Net/2" to the i386, known as "386BSD", though very little of the 386BSD code remains. FreeBSD is used by companies, Internet Service Providers, researchers, computer professionals, students and home users all over the world in their work, education and recreation. FreeBSD comes with over 20,000 packages (pre-compiled software that is bundled for easy installation), covering a wide range of areas: from server software, databases and web servers, to desktop software, games, web browsers and business software - all free and easy to install.

Popularity (hits per day): 12 months: **25** (357), 6 months: **22** (413), 3 months: **19** (507), 4 weeks: **20** (490), 1 week: **36** (337)

Average visitor rating: **9.13**/10 from **63** review(s).

FreeBSD Summary	
Distribution	FreeBSD
Home Page	http://www.freebsd.org/
Mailing Lists	http://www.freebsd.org/community/mailinglists.html
User Forums	http://forums.freebsd.org/
Alternative User Forums	
Documentation	http://www.freebsd.org/docs.html
Screenshots	
Screencasts	
Download Mirrors	http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/mirrors-ftp.html
Bug Tracker	http://www.freebsd.org/support/bugreports.html
Related Websites	FreeBSD Wiki • FreeBSD News • The FreeBSD Diary • FreeBSD Planet • FreeBSD Custom Releases • The FreeBSD Mall • FreeBSD for Raspberry Pi • VirtualBSD • FreshPorts • FreeBSD Ports • FreeBSD System Administration Training • DaemonForums • BSDNexus • The FreeBSD Foundation • BSDSupport • BSD DevCenter • Super User's BSD Cross Referrence • Manual Pages • Wikipedia • FreeBSD Italy • BSDForen.de (German) • BSDGuru (Polish)
Reviews	11.x: Pro-Linux (German) • DistroWatch 10.x: DistroWatch 9.x: OSNews • DistroWatch 8.x: O'Reilly Broadcast • LWN 7.x: Linuxoid (Russian) • Wordpress 6.x: Blogspot • ServerWatch • about Linux • Tuxmachines 5.x: DistroWatch • OSNews

Linux Media	
Linux Mint 19.1	$5.95
Ubuntu 18.10	$5.95
Debian 9.6.0	$5.95
openSUSE 15.0	$5.95
Manjaro 18.0	$5.95
Fedora 29	$5.95
MX Linux 18	$5.95

Figure 4.8: *A screenshot of the distribution detail page of the DistroWatch website, annotated to show how it can be used to implement the evaluation of RDF triple patterns. The distribution detail page of DistroWatch effectively lists all properties of a given subject, satisfying the case where s is bound but p and o are not.*

```
SELECT * WHERE {
    <http://distrowatch.com/ubuntu> ?p ?o .
}
```

(a)

```
[
    {"?p":"http://distrowatch.com/ostype","?o":"\"Linux\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Beginners\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Desktop\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Server\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Live Medium\""},
    {"?p":"http://distrowatch.com/origin","?o":"\"Isle of Man\""},
    {"?p":"http://distrowatch.com/basedon","?o":"\"Debian\""},
    {"?p":"http://distrowatch.com/desktop","?o":"\"GNOME\""},
    {"?p":"http://distrowatch.com/desktop","?o":"\"Unity\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"armhf\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"i686\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"powerpc\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"ppc64el\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"s390x\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"x86_64\""},
    {"?p":"http://distrowatch.com/package","?o":"\"DEB\""},
    {"?p":"http://distrowatch.com/rolling","?o":"\"Rolling\""},
    {"?p":"http://distrowatch.com/isosize","?o":"\" \""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"A\""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"1\""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"1\""},
    {"?p":"http://distrowatch.com/language","?o":"\"Yes\""},
    {"?p":"http://distrowatch.com/defaultinit","?o":"\"systemd\""},
    {"?p":"http://distrowatch.com/status","?o":"\"Active\""}
]
```

(b)

Figure 4.9: *A simple SPARQL query (4.9a) with the pattern s?? applied to the non-RDF DistroWatch dataset using ldf-facade, and its results (4.9b).*

```
SELECT * WHERE {
    <http://distrowatch.com/ubuntu> ?p ?o .
}
```

(a)

```
[
    {"?p":"http://distrowatch.com/ostype","?o":"\"Linux\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Beginners\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Desktop\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Server\""},
    {"?p":"http://distrowatch.com/category","?o":"\"Live Medium\""},
    {"?p":"http://distrowatch.com/origin","?o":"\"Isle of Man\""},
    {"?p":"http://distrowatch.com/basedon","?o":"\"Debian\""},
    {"?p":"http://distrowatch.com/desktop","?o":"\"GNOME\""},
    {"?p":"http://distrowatch.com/desktop","?o":"\"Unity\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"armhf\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"i686\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"powerpc\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"ppc64el\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"s390x\""},
    {"?p":"http://distrowatch.com/architecture","?o":"\"x86_64\""},
    {"?p":"http://distrowatch.com/package","?o":"\"DEB\""},
    {"?p":"http://distrowatch.com/rolling","?o":"\"Rolling\""},
    {"?p":"http://distrowatch.com/isosize","?o":"\" \""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"A\""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"1\""},
    {"?p":"http://distrowatch.com/netinstall","?o":"\"1\""},
    {"?p":"http://distrowatch.com/language","?o":"\"Yes\""},
    {"?p":"http://distrowatch.com/defaultinit","?o":"\"systemd\""},
    {"?p":"http://distrowatch.com/status","?o":"\"Active\""}
]
```

(b)

Figure 4.10: *A SPARQL query (a) requesting the intersection of two ?po patterns applied to the non-RDF dataset using ldf-facade, and its results (b).*

```
{
    "folderName": "Hillson et al. 2012",
    "count": 27,
    "propagatePermission": true,
    "type": "PUBLIC",
    "publicReadAccess": false,
    "canEdit": false,
    "id": 1,
    "creationTime": 1453777111302,
    "entries": []
}
```

Confusingly, the `entries` array is empty even if the folder has entries. The entries of a folder can be retrieved separately using the `entries` endpoint. For example, the endpoint `/rest/folders/4/entries` returns a JSON array of entries of the form:

```
{
    "ice": { ... },
    "id": 139,
    "type": "PLASMID",
    "parentIDs": [ 138 ],
    "linkedPartIDs": [],
    "index": 0,
    "ownerId": 0,
    "creatorId": 0,
    "status": "Complete",
    "shortDescription": "Promoter1 with BCD21-gfp",
    "creationTime": 1416293126696,
    "modificationTime": 0,
    "bioSafetyLevel": 1,
    "principalInvestigatorId": 0,
    "basePairCount": 0,
    "featureCount": 0,
    "viewCount": 742,
    "hasAttachment": false,
    "hasSample": false,
    "hasSequence": true,
    "hasOriginalSequence": true,
    "canEdit": false,
    "accessPermissions": [],
    "publicRead": false,
    "partId": "ACS_000139",
    "recordId": "f34a697b-724a-44e6-9112-d1c580f31dcd",
    "name": "pProm1_BCD21_Yeast"
}
```

It is possible to retrieve SBOL2 for a given entry by passing its identifier to `sequence` endpoints. For example, the URL `/rest/file/139/sequence/sbol2` returns SBOL2 RDF+XML corresponding to the part with identifier 139 (Fig. 4.11).

While a complete effort to expose specifically JBEI-ICE as a virtual RDF resource would be outside of the scope of this chapter, the hierarchy of folders containing SBOL parts was exposed as RDF using ldf-facade as a proof of concept [3]. The three relevant services were modelled as such:

- `/rest/collections/FEATURED/folders` satisfies both enumeration of possible values of s and of triples matching $s??$, where s is a virtual `sbol:Collection` corresponding to an ICE folder.

[3]`https://github.com/udp/jbei-ice-ldf`

```
<http://acs-registry.jbei.org/entry/ACS_000139/1> a sbol:ComponentDefinition ;
    dcterms:description "Promoter1 with BCD21-gfp" ;
    dcterms:title "pProm1_BCD21_Yeast" ;
    sbol:displayId "ACS_000139" ;
    sbol:persistentIdentity <http://acs-registry.jbei.org/entry/ACS_000139> ;
    sbol:sequence <...> ;
    sbol:type <http://www.biopax.org/release/biopax-level3.owl#DnaRegion> ;
    sbol:version "1" ;
    ice:bioSafetyLevel "1" ;
    ice:creationTime "2014-11-17 22:45:26.696" ;
    ice:creator "Chao Shih" ;
    ice:creatorEmail "ccshih@lbl.gov" ;
    ice:id "139.0" ;
    ice:longDescriptionType "text" ;
    ice:modificationTime "2016-05-12 11:01:36.291" ;
    ice:owner "Chao Shih" ;
    ice:ownerEmail "ccshih@lbl.gov" ;
    ice:principalInvestigator "Nathan Hillson" ;
    ice:recordId "f34a697b-724a-44e6-9112-d1c580f31dcd" ;
    ice:recordType "plasmid" ;
    ice:references "Shih, S. C., Goyal, G., Kim, P. W., ..." ;
    ice:selectionMarker "AMP and TRP" ;
    ice:shortDescription "Promoter1 with BCD21-gfp" ;
    ice:status "Complete" ;
    ice:versionId "f34a697b-724a-44e6-9112-d1c580f31dcd" ;
    ice:visibility "9" .

<...> a sbol:Sequence ;
    sbol:displayId "sequence_bff82c7554b50ed1450c11c178309dcd846d9110" ;
    sbol:elements "tgtctgt..." ;
    sbol:encoding <http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html> ;
    sbol:version "1" .
```

Figure 4.11: *A part in SBOL representation retrieved from JBEI-ICE, converted to Turtle syntax and abbreviated for readability.*

- `/rest/folders/<id>/entries` satisfies evaluation of $sp?$ where s is a virtual `sbol:Collection` corresponding to an ICE folder and p is `sbol:member`
- `/rest/file/<id>/sequence/sbol2` satisfies evaluation of $s??$ where s is an `sbol:ComponentDefinition`

For simplicity, the virtual RDF objects are assigned the URLs `<iceURL>/collections/<collectionID>` for ICE collections (e.g. "FEATURED"); `<iceURL>/folders/<folderID>` for folders; and `<iceURL>/parts/<partID>` for parts. As in this case the API does not provide any obvious URLs for any of the above, the mapping of ICE identifiers to URIs is up to the facade server. Functions are defined to provide conversion (by simple string manipulation) from URIs contained by incoming triple patterns to JBEI-ICE identifiers and vice versa. The logic of the $s??$ generator, therefore, becomes:

- Identify whether the URI refers to a collection, folder, or part and extract the substring which contains its JBEI-ICE identifier
- If the URI refers to a collection or folder, retrieve its JSON description from the ICE HTTP endpoint and generate a set of triples describing an `sbol:Collection`
- If the URI refers to a part, retrieve the SBOL2 from ICE, update its URI to point to the URI of the virtual RDF object, and relay it to the client as a set of triples

This minimal functionality immediately enables the application of simple SPARQL queries using the LDF client (Figs. 4.12, 4.13).

4.7 Discussion & Conclusion

While it would be much easier for the Semantic Web community if everything was RDF, making "RDFizers" and harmonization techniques such as those described in this chapter unnecessary, it is a reality that there are many different databases, APIs, and paradigms for data representation. It would be unrealistic to expect all data infrastructure to agree on RDF and SPARQL as a final solution.

The hypothesis of this chapter was that Linked Data Fragments (LDF) can be repurposed to dynamically convert non-RDF data to RDF at query-time. This possibility was both explored theoretically, and implemented in a new server for LDF termed ldf-facade. ldf-facade was shown to be applicable as a general tool outside of any specific domain with the successful execution of SPARQL queries over DistroWatch, a non-RDF database of open-source operating systems. Its application to SynBio was then explored by demonstrating how SPARQL queries can be executed over the JBEI-ICE design repository.

```
SELECT * WHERE {
    <https://acs-registry.jbei.org/folders/4> ?p ?o .
}
```

(a)

```
[
    {"?p":"http://www.w3.org/1999/02/22-rdf-syntax-ns#type","?o":"http://sbols.org/v2#Collection"},
    {"?p":"http://purl.org/dc/terms/title","?o":"\"Shih et al. 2015\""},
    {"?p":"http://ice.jbei.org#id","?o":"\"4\"^^http://www.w3.org/2001/XMLSchema#integer"},
    {"?p":"http://ice.jbei.org#entryCount","?o":"\"0\"^^http://www.w3.org/2001/XMLSchema#integer"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/139"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/138"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/137"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/136"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/135"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/134"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/133"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/132"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/131"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/130"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/129"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/128"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/127"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/126"},
    {"?p":"http://sbols.org/v2#member","?o":"https://acs-registry.jbei.org/parts/125"}
]
```

(b)

Figure 4.12: *A simple SPARQL query (a) executed over JBEI-ICE using the ldf-facade SBOL conversion layer, and the corresponding results (b). The query retrieves all properties associated with the folder ID 4, which are returned in the form of an SBOL* `Collection`*.*

```
PREFIX sbol: <http://sbols.org/v2#>
SELECT ?member ?na WHERE {
    <https://acs-registry.jbei.org/folders/4> sbol:member ?member .
    ?member sbol:sequence ?sequence .
    ?sequence sbol:elements ?na .
} LIMIT 10
```

(a)

```
[
    {"?member":"https://acs-registry.jbei.org/parts/139","?na":"\"tgtctgtaagcggatg...\""},
    {"?member":"https://acs-registry.jbei.org/parts/137","?na":"\"tgtctgtaagcggatg...\""},
    {"?member":"https://acs-registry.jbei.org/parts/135","?na":"\"tgtctgtaagcggatg...\""},
    {"?member":"https://acs-registry.jbei.org/parts/133","?na":"\"tgtctgtaagcggatg...\""},
    {"?member":"https://acs-registry.jbei.org/parts/131","?na":"\"tcagataaaatatttc...\""},
    {"?member":"https://acs-registry.jbei.org/parts/129","?na":"\"tcagataaaatatttc...\""},
    {"?member":"https://acs-registry.jbei.org/parts/127","?na":"\"tcagataaaatatttc...\""},
    {"?member":"https://acs-registry.jbei.org/parts/125","?na":"\"tcagataaaatatttct...\""}
]
```

(b)

Figure 4.13: *A SPARQL query (a) to select the sequences of entries in a JBEI-ICE folder using the ldf-facade SBOL conversion layer, and the abbreviated corresponding results (b). This query follows two layers of indirection using a virtual SBOL data model: one from the collection to the member, and another from the member to its elements.*

4.7.1 ldf-facade

By simplifying server-side logic, the prior work of LDF has provided a unique opportunity to reconsider where and when RDF data is generated. The ldf-facade server described in this chapter takes advantage of this opportunity by enabling RDF triples to be generated dynamically in response to a query, which allows "intelligent" functionality such as retrieving data from an upstream dataset, such as JBEI-ICE, and converting it to an RDF representation, such as SBOL.

ldf-facade serves as a proof of concept for the ability to use LDF to develop RDF abstractions over non-RDF datasets. However, the suitability of this approach as a general mediator layer to RDFize existing datasets remains limited. The examples described in this work required significant database-specific "glue" code to be written, which is time-consuming and potentially brittle if the upstream API changes. Additionally, they are dependent on the ability of the upstream API to respond to RDF-like patterns. If a pattern cannot be satisfied by an API call, it may result in a prohibitively expensive linear specialization of a much larger result set, or it may even not be possible to provide an answer at all.

While these considerations may make ldf-facade difficult to apply in practice today, the concept of dynamic RDFization using triple pattern generalisation and specialisation may be applicable to different systems in the future, particularly if RDFization is a two-way conversation in which the upstream API intentionally facilitates RDF interoperability.

4.7.2 Federated design knowledge using SBOL Stack and LDF

There are two approaches to capturing a design in SBOL. One is to effectively perform a mini-warehousing operation and copy information about all of the consistuent parts into the same place, creating a "monolithic" SBOL resource that describes everything about the design. An alternative "modular" approach is to only include immediately relevant design information, providing references to any composed parts rather than including their details.

The monolithic approach of attempting to capture all possible knowledge about a design in one place is a difficult because the scope is unlimited. If the sequence of an existing protein is to be included in the SBOL description of a design that incorporates that protein, should all of the information about its domains also be included? It also results in redundant copying of data. If the protein information was taken from a UniProt record, what if that record is later updated?

The alternative, modular approach has been difficult for practical reasons. Software and libraries for SBOL are only concerned with SBOL, so a link to, for example, UniProt is not dereferenceable to anything meaningful. While the examples of SBOL in the SBOL2 specification use URIs such as `http://www.partsregistry.org/BBa_J61101` to refer to parts from the iGEM Registry and `http://identifiers.org/uniprot/P42212` for UniProt [109, B.2.2], these URIs cannot be dereferenced to obtain actual knowledge about parts by any of the SBOL libraries.

An RDF view of SBOL, whether obtained natively by using the SBOL Stack or through conversions using technology such as ldf-facade as described in this chapter, can solve these problems by making SBOL knowledge part of a wider ecosystem of Linked Data. Viewed as a single "file", a design described in SBOL without documenting all of its constituent parts may appear to be missing data. Viewed as Linked Data, the SBOL can be seen as an RDF resource which *references* other existing resources. The federated querying enabled by LDF then allows the design to be navigated seamlessly, despite its composition from parts which are not located in the same place.

Consider the toggle switch example from the SBOL specification [109, B.2.2] (Fig. 4.14). Currently, its references to iGEM and UniProt are effectively intractable: its iGEM references are not resolveable URLs, and its UniProt references are identifiers.org URIs which cannot be interpreted by the SBOL libraries. However, by storing the example in the SBOL Stack and using the LDF client, it is possible to aggregate more comprehensive knowledge by executing integrative SPARQL queries across multiple datasets (Fig. 4.14)[4] With ldf-facade, the scope of data integration for design knowledge potentially grows to include *any* dataset, not just those already represented in RDF such as UniProt.

4.7.3 Future work

RDFizing more datasets

The obvious future work from this chapter is to make more datasets available as RDF. The two examples using DistroWatch and JBEI-ICE prove that the concepts behind ldf-facade can be realised to provide RDF access to non-RDF datasets, but are by no means comprehensive implementations intended for real-world usage.

Not all services will be amenable to the modelling in terms of triple patterns. However, considering each service as a "black box" which must be modelled as-is without any potential for modification may not be the only approach. Could RDFization become a two-way process in which the implementors of APIs, even without directly implementing RDF, can accommodate RDFization by making their APIs more "graph-friendly"?

Such an initiative could take the form of a set of best-practice recommendations derived from the formalism described in this chapter. For example, a recommendation to provide both $sp?$ and $?po$ could be "if a property of a resource can be retrieved, it should also be possible to retrieve a list of all resources with that property".

Intelligent LDF servers

The APIs shown here were intentionally chosen as they are HTTP Web APIs, and therefore distant from RDF representation. They serve as an extreme example of what might be possible to implement as part of LDF server logic, but there are many other possible applications for "intelligent" LDF servers. For example:

[4]In theory, this should be possible without any modification of the SBOL data. Unfortunately, the URIs used for UniProt are provided by identifiers.org, and the identifiers.org SPARQL endpoint [110] was unavailable at the time of writing; and RDF URIs for the iGEM Registry were only recently established as part of this work (chapter 5). Consequently, the UniProt URIs were manually changed to `purl.uniprot.org` equivalents and the iGEM URIs to `synbiohub.org` for example purposes.

```xml
<sbol:ModuleDefinition rdf:about="http://sbolstandard.org/example/tetr_inverter">
    <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter"/>
    <sbol:displayId>tetr_inverter</sbol:displayId>
    <sbol:role rdf:resource="http://parts.igem.org/cgi/partsdb/pgroup.cgi?pgroup=inverter"/>
    <sbol:functionalComponent>
        <sbol:FunctionalComponent rdf:about="http://sbolstandard.org/example/tetr_inverter/promoter">
            <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter/promoter"/>
            <sbol:displayId>promoter</sbol:displayId>
            <sbol:definition rdf:resource="http://www.partsregistry.org/BBa_R0040"/>
            <sbol:access rdf:resource="http://sbols.org/v2#public"/>
            <sbol:direction rdf:resource="http://sbols.org/v2#inout"/>
        </sbol:FunctionalComponent>
    </sbol:functionalComponent>
    <sbol:functionalComponent>
        <sbol:FunctionalComponent rdf:about="http://sbolstandard.org/example/tetr_inverter/TF">
            <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter/TF"/>
            <sbol:displayId>TF</sbol:displayId>
            <sbol:definition rdf:resource="http://identifiers.org/uniprot/Q6QR72"/>
            <sbol:access rdf:resource="http://sbols.org/v2#public"/>
            <sbol:direction rdf:resource="http://sbols.org/v2#inout"/>
        </sbol:FunctionalComponent>
    </sbol:functionalComponent>
    <sbol:interaction>
        <sbol:Interaction rdf:about="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI">
            <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI"/>
            <sbol:displayId>LacI_pLacI</sbol:displayId>
            <sbol:type rdf:resource="http://identifiers.org/biomodels.sbo/SBO:0000169"/>
            <sbol:participation>
                <sbol:Participation rdf:about="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI/Q6QR72">
                    <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI/Q6QR72"/>
                    <sbol:displayId>Q6QR72</sbol:displayId>
                    <sbol:role rdf:resource="http://identifiers.org/biomodels.sbo/SBO:0000020"/>
                    <sbol:participant rdf:resource="http://sbolstandard.org/example/tetr_inverter/TF"/>
                </sbol:Participation>
            </sbol:participation>
            <sbol:participation>
                <sbol:Participation rdf:about="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI/BBa_R0040">
                    <sbol:persistentIdentity rdf:resource="http://sbolstandard.org/example/tetr_inverter/LacI_pLacI/BBa_R0040"/>
                    <sbol:displayId>BBa_R0040</sbol:displayId>
                    <sbol:role rdf:resource="http://identifiers.org/biomodels.sbo/SBO:0000598"/>
                    <sbol:participant rdf:resource="http://sbolstandard.org/example/tetr_inverter/promoter"/>
                </sbol:Participation>
            </sbol:participation>
        </sbol:Interaction>
    </sbol:interaction>
</sbol:ModuleDefinition>
```

Figure 4.14: *An excerpt from the toggle switch example in the SBOL specification* [109, B.2.2] *showing references to UniProt and the iGEM Registry. While these references are not navigable by SBOL libraries, they provide useful connections to an integrative SPARQL query.*

```
CONSTRUCT {
    ?cd a sbol:ComponentDefinition ;
        sbol:type <http://www.biopax.org/release/biopax-level3.owl#Protein> ;
        sbol:role ?role ;
        sbol:sequence ?seq .
    ?seq a sbol:Sequence ;
        sbol:elements ?elements .
} WHERE {
    <http://sbolstandard.org/example/tetr_inverter> sbol:functionalComponent ?fc .
    ?fc sbol:definition ?cd .
    ?cd a up:Protein ;
        up:sequence ?seq ;
        up:classifiedWith ?role .
    ?seq rdf:value ?elements .
}
```

(a)

```
<http://purl.uniprot.org/uniprot/Q6QR72> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://sbols.org/v2#ComponentDefinition> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#type> <http://www.biopax.org/release/biopax-level3.owl#Protein> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.obolibrary.org/obo/GO_0003677> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.obolibrary.org/obo/GO_0045892> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.obolibrary.org/obo/GO_0046677> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.uniprot.org/keywords/195> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.uniprot.org/keywords/238> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#role> <http://purl.uniprot.org/keywords/805> .
<http://purl.uniprot.org/uniprot/Q6QR72> <http://sbols.org/v2#sequence> <http://purl.uniprot.org/isoforms/Q6QR72-1> .
<http://purl.uniprot.org/isoforms/Q6QR72-1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://sbols.org/v2#Sequence> .
<http://purl.uniprot.org/isoforms/Q6QR72-1> <http://sbols.org/v2#elements> "MMSRLDKSKVINSALELLNEVGIEGLKTRKLAQKLGVEQPTLYWHV..." .
```

(b)

Figure 4.15: *A federated SPARQL CONSTRUCT query (4.15a) to dynamically construct an SBOL* `ComponentDefinition` *using information obtained by cross-referencing an SBOL design with UniProt, and its abbreviated resulting SBOL data in N-Triples format (4.15b).*

- Mapping one set of ontology terms to another. An LDF server could expose an existing RDF dataset, but filter specific terms and return an alternative set.
- Providing functions instead of data. For example, the Virtuoso [65] triplestore defines built-in functions in the form of special predicates, such as `?string bif:contains ?substring`. An LDF server could provide functionality in the same manner by matching $?p?$ patterns.

The idea of providing functions through triple patterns could be particularly useful in bioinformatics, where there are already large RDF datasets available. For example, a server could conceivably respond to a `?protein inFamily ?family` triple pattern using Pfam [111] and encoding results as RDF, all at query time. Entire workflows which combine both RDF data resources and the execution of tooling could potentially be represented using SPARQL queries.

Caching of upstream data

The main caveat of using ldf-facade is that every request to the server concerns only one triple pattern, which means that every triple in the response must match that pattern. This means that even if the server has more information, it cannot be delivered if it concerns a different subject until the next request arrives. For example, retrieving the following SBOL object:

```
<sbol:interaction>
    <sbol:Interaction rdf:about="http://example/degradation">
        <sbol:type rdf:resource="http://identifiers.org/biomodels.sbo/SBO:0000179"/>
        <sbol:participation>
            <sbol:Participation rdf:about="http://example/degradation/TetR">
                <sbol:role rdf:resource="http://identifiers.org/biomodels.sbo/SBO:0000010"/>
                <sbol:participant rdf:resource="http://example/TetR"/>
            </sbol:Participation>
        </sbol:participation>
    </sbol:Interaction>
</sbol:interaction>
```

would require the evaluation of two triple patterns, because of the intermediate `Participation` object. If the server responds to the pattern `http://example/degradation ? ?` by retrieving the entire interaction object, it cannot send information about the participants unless it later receives a request for the triple pattern `http://example/degradation/TetR ? ?`, even if that information is already available when responding to the first request. If the server does not maintain any state between requests, in the worst case this could mean retrieving the interaction object twice — once to match the triple pattern for the interaction, and then again for the participation.

The second JBEI-ICE example (Fig. 4.13) suffers from this problem. The query has to follow two layers of indirection to reach the sequence elements: first by evaluating `<part> sbol:sequence ?sequence`, and then `<sequence> sbol:elements ?na`. Both of these queries result in the part being retrieved from the ICE API.

One possible solution would be to implement a layer of caching for upstream results, meaning even if the same upstream resource is requested multiple times, it only has to be retrieved once. As the same resource is likely to be referenced multiple times in the same SPARQL query, such caching could be very short-lived and still likely result in fewer upstream requests.

Client-side optimisation

While the LDF client was not developed as part of this work, it is the central component for enabling SPARQL queries over LDF servers. The client functions by reordering SPARQL queries into an order that results in the most efficient set of pattern matches. For example, the following DistroWatch query:

```
SELECT ?s WHERE {
    ?s distrowatch:ostype "BSD" ;
        ?s distrowatch:status "Active" .
} LIMIT 5
```

is evaluated by the LDF client by evaluating `?  ostype "BSD"` and then evaluating, for each subject in turn, `<s> status "Active"` , or:

- Evaluate `?s distrowatch:ostype "BSD"` and `?s distrowatch:status "Active"` *(2 queries)*
- Select the smallest result set (set of possible subjects) from the two triple patterns — in this case, `?s distrowatch:ostype "BSD"`
- *For each possible subject*, evaluate the other triple pattern by binding s: `<s> distrowatch:status "Active"` *(20 queries)*

It would be far more efficient (in terms of number of pattern evaluations) to evaluate both `?  ostype "BSD"` and `?  status "Active"` and then take the intersection of the two result sets, i.e.:

- Evaluate `?s distrowatch:ostype "BSD"` and `?s distrowatch:status "Active"` *(2 queries)*
- Select `?s` where `?s` is common to both result sets *(0 queries)*

Conclusion

This chapter described how the recent prior work of Linked Data Fragments can be leveraged to develop novel data integration strategies for RDF. A method was defined for formalising an existing, non-RDF Web service in terms of RDF triple patterns, and for deriving missing patterns from those that are available. This method was implemented in the ldf-facade server software, which allows LDF servers to have intelligent logic for specific triple patterns rather than relaying RDF data from an existing datasource.

While this chapter only demonstrated a proof-of-concept, ldf-facade could potentially enable many legacy datasources to be exposed for use as part of federated graph queries with dynamically harmonized semantics. This notion is particularly relevant to synthetic biology, where there are many datasets with varying, incompatible representations which could be relevant to the design process. It may also have wider implications for Linked Data and the Semantic Web in general; if the method defined here can be proven to work at scale, it may contribute to a solution for the lack of availability of existing data as RDF.

5. Conversion and Enrichment of the iGEM Registry

5.1 Introduction

The iGEM Registry of Standard Biological Parts currently provides access to knowledge regarding over 20,000 BioBricks [78]. However, as described in section 2.4.1, while BioBricks have a well-defined *physical* standard, their *data* standard is still not formalised. The representation of parts used by the Registry does not yet have a standardized data model or computational API.

The need for a standardized representation of biological parts is exactly what SBOL is designed to address. Furthermore, as previously discussed in section 3.2, one of the challenges of encouraging the adoption of SBOL is "bootstrapping": there is no advantage to standardization unless there are data with which to integrate. This presents an opportunity to solve two problems at once. The development of an SBOL representation could bring standardization and machine-tractability to the iGEM Registry, and also help to seed the SBOL standard with a large dataset of knowledge about engineered parts.

Additionally, an SBOL representation of the iGEM Registry would potentially enable parts to be described with a broader scope of computationally tractable information. The data representation of the iGEM Registry is DNA-centric in the same manner as FASTA, GenBank, and SBOL version 1. Since the release of version 2.0 in 2015, SBOL has had the ability to represent not just DNA sequences and their features, but also more complex engineered biological systems comprising proteins, RNAs, and biochemical interactions. Expanding the SBOL version of the iGEM Registry to include such information could make it possible to perform queries such as searching for BioBricks which code for a specific protein, or producing visualizations which contain depictions of interactions.

Manually re-annotating the entire iGEM Registry would be extremely laborious. However, given a DNA sequence, it should be possible to retroactively add annotation of predicted transcription, translation, and regulation using existing established bioinformatics tools. For example, promoters and CDSs can be identified using gene finding tools; gene product proteins can be inferred from the template sequence; and proteins can be automatically annotated using Hidden Markov Model (HMM) techniques and datasets such as Pfam [111]. The iGEM Registry presents a perfect test case to explore the viability of such an "enrichment" process.

The research goals of this chapter are to investigate (a) how the iGEM Registry can be mapped to the SBOL data model, and (b) how SBOL data, such as these converted parts, can be enriched using automated sequence annotation.

Attribution: The initial conversion of iGEM to SBOL was developed as part of this work, and was later further developed with help from Chris Myers. Enrichment was developed as part of this work.

5.2 Modelling the iGEM database as SBOL

The iGEM Registry has a basic keyword search and an "API" with which specific parts can be retrieved as XML, but it lacks querying capabilities which could be modelled as RDF triples to produce an RDF view using ldf-facade. Converting the registry to SBOL therefore requires the more traditional approach of performing the conversion ahead of time and warehousing the results.

The iGEM Registry provides the entire dataset for download as a snapshot of a MySQL database. Setting up a local instance of the database from the snapshot is complicated by the snapshot being serialized using the MySQL "XML dump" format, which is not intended to be used to restore a live database. Fortunately, a modified version of the XMLDumpRestore script[1] can be used when combined with pre-processing to remove invalid bytes from the input file. A Docker image which automates this approach is available on GitHub[2].

In order to convert the Registry to SBOL, an *igem2sbol*[3] script was written. This script connects to the iGEM MySQL database, traverses the tables, and uses the sboljs library described in section 3.3 to generate SBOL.

There are two relevant tables in the iGEM Registry SQL database: `parts`, where each row corresponds to a BioBrick; and `parts_seq_features`, where each row corresponds to the annotation of a sequence feature. Composition between of parts is accomplished using a special type of sequence feature labelled "BioBrick", which contains the identifier of the nested part.

The columns of the `parts` table are mapped to the properties of SBOL `ComponentDefinition` (Table 5.1), and the columns of the `parts_seq_features` table is mapped to the properties of SBOL `SequenceAnnotation` (Table 5.2).

MySQL Column	**Description**	**SBOL Mapping**
`part_id`	Numeric database identifier for the part	
`ok`	Unknown	
`part_name`	Part name (i.e. `BBa_*`)	`dcterms:title`
`short_desc`	Short description	`dcterms:description`
`description`	Long description	`synbiohub:mutableDescription`

[1]`http://github.com/n8maninger/XMLDumpRestore`
[2]`https://github.com/udp/igemparts-mysql-docker`
[3]`http://github.com/udp/igem2sbol`

part_type	Role, e.g. Terminator	sbol:role
author	Free-text authorship	dc:creator
owning_group_id	N/A	igem:owning_group_id
status	Unknown	igem:status
dominant	1 if this is the preferred part in relation to its duplicates, if any.	igem:dominant
informational	Unknown	igem:informational
discontinued	1 if the part is discontinued	igem:discontinued
part_status	Unknown	igem:part_status
sample_status	“In stock” if iGEM has a physical sample of the BioBrick	igem:sample_status
p_status_cache	Unknown	Not mapped
s_status_cache	Unknown	Not mapped
creation_date	Created timestamp	dcterms:created
m_datetime	Modified timestamp	dcterms:modified
m_user_id	N/A	igem:m_user_id
uses	Number of times the part is used as a sub-part	igem:uses
doc_size	Unknown	igem:doc_size
works	Unknown	igem:works
favorite	Unknown	igem:favorite
specified_u_list	Unknown	igem:specified_u_list
deep_u_list	Unknown	igem:deep_u_list
deep_count	Unknown	igem:deep_count
ps_string	Unknown	igem:ps_string
scars	Unknown	igem:scars
default_scars	Unknown	igem:default_scars
owner_id	Unknown	igem:owner_id
group_u_list	Unknown	igem:group_u_list
has_barcode	Unknown	igem:has_barcode
notes	Unknown	igem:notes

source	Unknown	synbiohub:mutableProvenance
nickname	Unknown	igem:nickname
categories	Unknown	See section 5.2.1
sequence	DNA sequence	sbol:sequence
sequence_update	Unknown	igem:sequence_update
review_result	Unknown	igem:review_result
review_count	Unknown	igem:review_count
review_total	Unknown	igem:review_total
flag	Unknown	igem:flag
sequence_length	Length of the sequence — redundant?	Not mapped
temp_1	Unknown	Not mapped
temp_2	Unknown	Not mapped
temp_3	Unknown	Not mapped
temp4	Unknown	Not mapped
rating	Unknown	igem:rating

Table 5.1: *Mapping of the iGEM database* parts *schema to the SBOL data model. Columns with no SBOL counterpart are represented as custom terms in an* igem *namespace. Columns listed as "Unknown" had no obvious mapping and therefore cannot be mapped without documentation.*

MySQL Column	**Description**	**SBOL Mapping**
feature_id	Numeric database identifier for the feature	Not mapped
feature_type	Type, e.g. promoter	See table 5.4
start_pos	Start sequence offset (1-based)	sbol:start
end_pos	End sequence offset (1-based)	sbol:end
label	Feature title	dcterms:title
part_id	Identifier of the part that has the feature	N/A
type	Another type specifier, usually (but not always) the same as feature_type	See table 5.4

`label2`	The same as `label` if present, empty otherwise	Not mapped
`mark`	Unknown	Not mapped
`old`	Unknown	Not mapped
`reverse`	1 if the annotation is reverse complement	`sbol:orientation`

Table 5.2: *Mapping of the iGEM database* `parts_seq_features` *schema to the SBOL data model. This mapping was jointly developed with Chris Myers (University of Utah).*

iGEM Type	**SO Mapping**	**SO Description**
`Basic`	Not mapped	
`Cell`	Not mapped	
`Coding`	`SO:0000316`	CDS
`Composite`	Not mapped	
`Conjugation`	`SO:0000724`	`oriT`
`Device`	Not mapped	
DNA	`SO:0000110`	DNA
`Generator`	Not mapped	
`Intermediate`	Not mapped	
`Inverter`	Not mapped	
`Measurement`	Not mapped	
`Other`	`SO:0000110`	DNA
`Plasmid`	`SO:0000155`	`plasmid`
`Plasmid_Backbone`	`SO:0000755`	`plasmid_vector`
`Primer`	`SO:0000112`	`primer`
`Project`	Not mapped	
`Protein_Domain`	`SO:0000417`	`polypeptide_domain`
RBS	`SO:0000139`	`ribosome_entry_site`
`Regulatory`	`SO:0000167`	`promoter`
`Reporter`	Not mapped	
RNA	`SO:0000834`	`mature_transcript_region`
`Scar`	`SO:0001953`	`restriction_enzyme_assembly_scar`

Signalling	Not mapped	
T7	SO:0001207	T7_RNA_Polymerase_Promoter
Tag	SO:0000324	tag
Temporary	Not mapped	
Terminator	SO:0000141	terminator
Translational_Unit	Not mapped	

Table 5.3: *Mapping of iGEM part types to Sequence Ontology (SO) terms to populate the* `sbol:role` *property. Part types without a corresponding SO term are mapped to URIs in a custom iGEM namespace. This mapping was jointly developed with Chris Myers (University of Utah).*

iGEM Type	SO Mapping	SO Description
barcode	SO:0000807	engineered_tag
binding	SO:0001091	non_covalent_binding_site
BioBrick	SO:0000804	engineered_region
dna	SO:0000110	sequence_feature
misc	SO:0000110	sequence_feature
mutation	SO:0001059	sequence_alteration
polya	SO:0000553	polyA_site
primer_binding	SO:0005850	primer_binding_site
protein	SO:0000316	CDS
s_mutation	SO:1000008	point_mutation
start	SO:0000318	start_codon
stop	SO:0000319	stop_codon
tag	SO:0000324	tag
promoter	SO:0000167	promoter
cds	SO:0000316	CDS
operator	SO:0000057	operator
terminator	SO:0000141	terminator
conserved	SO:0000330	conserved_region
rbs	SO:0000139	ribosome_entry_site
stem_loop	SO:0000313	stem_loop

Table 5.4: *Mapping of iGEM feature types to Sequence Ontology (SO) terms to populate the* `sbol:role` *property. Feature types without a corresponding SO term are mapped to URIs in a custom iGEM namespace. This mapping was jointly developed with Chris Myers (University of Utah).*

5.2.1 Categories

When parts are submitted to the iGEM Registry, they can be placed into categories. For example, the part BBa_B0015 (double terminator) has the categories `//direction/forward` and `//terminator/double`. The categories can have multiple levels of nesting, e.g. `//function/metalsensing/iron` and `//rnap/prokaryote/ecoli/sigma70`.
In the SBOL conversion, the category specifier for each part is split into individual categories, and a nested hierarchy of collections is created. For example, a part with the category `//function/metalsensing/iron` would result in the creation of:

- An SBOL `Collection` for "function"
- An SBOL `Collection` for "metalsensing", contained as an `sbol:member` of the "function" collection
- An SBOL `Collection` for "iron", contained as an `sbol:member` of the "metalsensing" collection
- An SBOL `ComponentDefinition` for the part, created as an `sbol:member` of the "iron" collection

Breaking down iGEM categories into SBOL collections creates a tree which can be browsed at any node (e.g. retrieve all of the parts in any sub-category of `//function/metalsensing`).

5.2.2 De-flattening

The `parts_seq_features` table maintains a large amount of redundant data because of how composition is represented. The "transitive" model of annotation, as used by SBOL, is that if part A incorporates part B via composition and part B has an annotation, the annotation implicitly also applies to part A and does not need to be specified twice. iGEM does not follow this approach, instead duplicating annotations from composed parts to each part composing them so that each part has a "flattened" set of annotations. For example:

- BBa_B0010 (part ID 603) has a stem loop feature from 12..55 (feature ID 4184)
- BBa_B0015 (part ID 202) incorporates BBa_B0010 as a sub-part (feature ID 1916610)
- BBa_B0010 also has a stem loop feature (feature ID 1916611), which is actually a copy of feature ID 4184 from BBa_B0010

While this is a small-scale example, there are parts with hundreds of feature annotations in the Registry. Each time they — or a part incorporating them — is used in composition, all of these feature annotations are duplicated. It is possible that the maintainers of the iGEM Registry have a method to update all occurrences of a sequence annotation if it is found to be mis-annotated, and to propagate the addition or deletion of annotations from composed parts to the parts into which they are incorporated. However, in order to avoid this entire class of problem and create an intuitive SBOL representation, these duplicate annotations were flattened so that each sequence annotation would only be specified once.

Additionally, there are many cases of composition in the iGEM dataset where the composed part is represented in the `parts_seq_features` table as a feature annotation rather than explicitly as the composition of another BioBrick. In order to preserve a faithful representation and avoid changing the data by creating new relations, these were not altered in the current SBOL conversion. The prevelance of such mislabeled composition is explored in section 5.3.2.

5.3 iGEM-SBOL graph queries

The iGEM Registry provides basic search functionality based on the name or description of parts, but not the ability to search using criteria based on other information contained in the database, such as the type and location of features within parts. Converting the iGEM Registry to SBOL means that it can now be used in conjunction with the SBOL Stack graph database described in chapter 3, enabling the execution of powerful graph queries. This capability was tested with a number of example queries.

5.3.1 Use of transcription factor binding sites in the iGEM Registry

One of the central components of genetic circuit design are DNA-binding proteins which function as transcriptional activators or repressors [21]. Many of the parts in the iGEM Registry depend on such regulatory interactions, with the regulated promoters pTet and pLac both in the top 10 most used parts [112].

In the iGEM Registry, features in the `parts_seq_features` table can be labelled with a `feature_type` of `binding`, which is typically used to indicate that the sequence is a transcription factor binding site. Such features are mapped in the iGEM-SBOL conversion to the standardised Sequence Ontology term of `SO:0001091 non_covalent_binding_site`. Extracting such annotations is trivial with a SPARQL query over SBOL-RDF (Fig. 5.1).

There are 2282 annotations with the role `SO:0001091 non_covalent_binding_site` in the iGEM-SBOL dataset derived from the 6 October 2017 iGEM Registry dump (Fig. 5.1). Of these annotations, 495 are under 10 nucleotides in length making them unlikely to be annotations of actual TFBSs [113]. 75 are over 100 nucleotides in length, including instances of the entire part being annotated as binding. Excluding these leaves 1712 annotated potential binding sites. Sub-graphs from a network generated by global sequence alignment of these sequences

```
SELECT (COUNT(?sequenceAnnotation) as ?count) WHERE {
   ?sequenceAnnotation a sbol:SequenceAnnotation .
   ?sequenceAnnotation sbol:role <http://identifiers.org/so/SO:0001091> .
   ?sequenceAnnotation sbh:topLevel ?topLevel .
   <https://synbiohub.org/public/igem/igem_collection/1> sbol:member ?topLevel .
}
```

Figure 5.1: *SPARQL query to count all of the sequence annotations with the role SO:0001091 (binding) that are part of the iGEM parts collection, made possible by the conversion of the Registry to SBOL-RDF.*

using EMBOSS [114] `needleall` (Fig. 5.2) can be used to identify the most commonly used sites:

- LacI $\geq$ 76 occurrences
- TetR $\geq$ 61 occurrences
- lambda cI 46 occurrences (OR1 + OR2)
- LuxR/HSL $\geq$ 41 occurrences
- EsaR $\geq$ 24 occurrences
- AraC $\geq$ 19 occurrences
- NorR $\geq$ 12 occurrences

5.3.2 Re-use of parts in the iGEM Registry

During the conversion, it was observed that there are many instances of composition, where a part includes another part, that are not marked as such in the database. For example, BBa_I1041 incorporates pTet (BBa_R0040), but because the annotation of pTet is marked as "operator" in the `parts_seq_features` table rather than as "BioBrick" with a link to BBa_R0040, there is no link between BBa_I1041 and BBa_R0040.

The scale of re-use of existing parts within iGEM has recently been a contentious issue, following an observation in a 2014 letter to Nature Biotechnology that the level of part re-use was low among finalist iGEM teams [115]. Unfortunately, iGEM-SBOL does not contain information about teams and whether they were finalists and therefore cannot be used to reproduce these results. However, the lack of explicit indication of composition in the database may be relevant to future analyses.

With the SBOL representation of iGEM, the prevelance of this mislabeled composition can be quantified using a SPARQL query (Fig. 5.3). Application of this query to the 6 October 2017 iGEM Registry dump reveals 8463 instances of composition where the dataset does not contain a link from the parent part to the composed part, filtered for parts greater than 10 bases in length to exclude the identification of "parts" such BBa_J58006 (a stop codon). Note that this query does not account for entirely undocumented reuse, where a part contains another part without any sequence annotation at all.

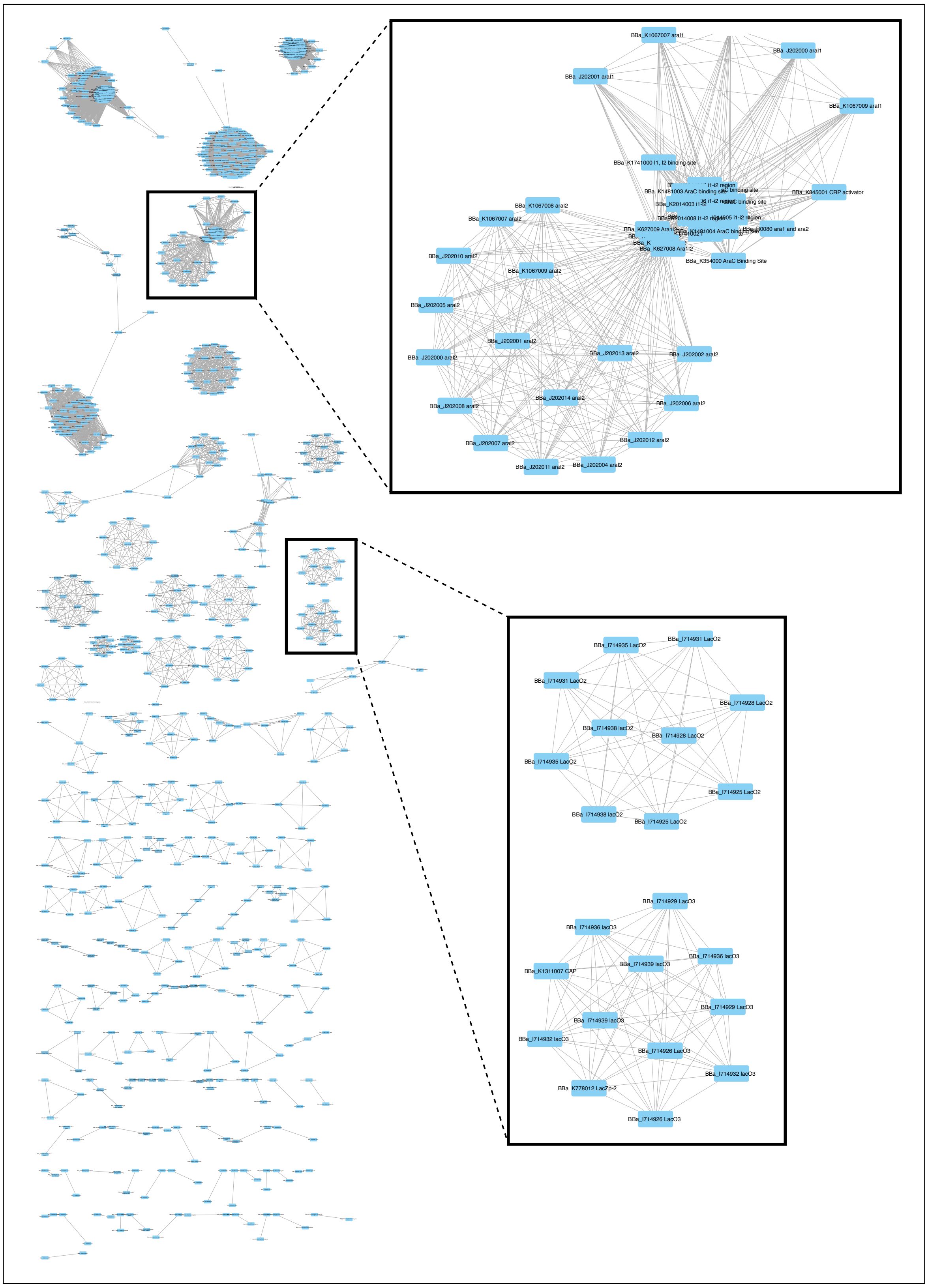


Figure 5.2: *A global sequence alignment of 1712 sub-sequences annotated as "binding" in the iGEM Registry using EMBOSS `needleall`, then used to generate a network of sequence similarity. Subgraphs for regions with distinct sequences are clearly visible. Extraction of the sub-sequences was made possible by the conversion of the Registry to SBOL-RDF.*

```
SELECT count(DISTINCT ?sa) WHERE {

    ?cd1 a sbol:ComponentDefinition .
    <https://synbiohub.org/public/igem/igem_collection/1> sbol:member ?cd1 .

    ?cd1 sbol:sequenceAnnotation ?sa .
    FILTER NOT EXISTS { ?sa sbol:component ?c . }

    ?sa sbol:location ?loc .
    ?loc sbol:start ?start .
    ?loc sbol:end ?end .

    ?cd1 sbol:sequence ?seq .
    ?seq sbol:elements ?na .

    bind (strlen(?na) as ?seqlen)
    filter(xsd:integer(?end) <= ?seqlen)
    filter(xsd:integer(?end) > xsd:integer(?start))

    bind (xsd:integer(?end) - xsd:integer(?start) + 1 as ?len)
    filter(?len > 10)
    bind (substr(?na, xsd:integer(?start), ?len) as ?nasub)

    ?seq2 a sbol:Sequence .
    ?seq2 sbol:elements ?na2 .
    <https://synbiohub.org/public/igem/igem_collection/1> sbol:member ?seq2 .

    FILTER(?na2 = ?nasub)

    ?cd2 sbol:sequence ?seq2 .

    FILTER(?cd2 != ?cd1)
}
```

Figure 5.3: *SPARQL query to count the number of sequence annotations which annotate a subsequence which is identical to another part, but where the composition is not made explicit using an* `sbol:component` *relation. The application of this SPARQL query was made possible by the conversion of the Registry to SBOL-RDF.*

5.4 Enrichment

SBOL2 provides a much richer data model than the original iGEM Registry representation. However, the process of converting the Registry data directly to their corresponding SBOL objects does not immediately add any additional information. The SBOL2 data model is used by the converted dataset, but only to represent the subset of knowledge that was originally represented by the Registry.

There are various approaches which could be used to add more information about parts to populate the richer SBOL2 data model. One time-consuming approach would be to simply read the part description and to manually create SBOL objects to document the function. A machine-learning approach may also be possible; in the Registry, gene products and interactions are often described in free-text using the wiki, so in theory it would be possible to mine the text and use it to populate the SBOL2 data model. An alternative approach which was explored as part of this work is to treat the annotation of iGEM parts as a bioinformatics problem, akin to the annotation of features in a natural genome.

The majority of the parts in the iGEM Registry are designed to be used as parts of transcriptional genetic circuits of the kind described in section 2.1. Annotating missing functional layers of a transcriptional genetic circuit design in the context of SBOL2 would require predicting these processes to fill in missing parts of the data model, such as creating components for gene products, transcription interactions, and binding interactions. Given just a DNA sequence for an iGEM part, an initial attempt to annotate its function would be to:

1. Locate and annotate possible CDSs in the DNA sequence
2. Translate these CDSs to protein sequences
3. Identify TF domains in the protein sequences
4. Locate any TFBSs in the DNA sequence
5. Create interactions to link the TF domains to TFBSs

While there are bioinformatics tools which can accomplish many of these tasks, no single tool is generally applicable to everything. For example, in prokaryotic DNA, CDSs are typically composed of a single open reading frame (ORF) beginning at a start codon such as AUG and ending at a stop codon such as UAG, UAA, or UGA [116, p. 334]. Finding a CDS in prokaryotic DNA can be as simple as searching for substrings in the nucleotide sequence. For eukaryotic DNA, the situation is often much more complicated: each ORF is composed of introns and exons, and the introns are spliced to give rise to the CDS [116, p. 317]. ORF finders for eukaryotes are highly complex, often using machine-learning algorithms trained for specific organisms [117]. Even just the initial step of "locate any CDSs in the DNA sequence", therefore, may require the results of a variety of software to be integrated.

This problem was addressed in this work with the development of *Enrichment*, a tool for automatic SBOL2 sequence annotation with the ability to "mix and match" the output of various existing bioinformatics tools. Enrichment is provided with an initial set of integrations chosen specifically for the annotation of iGEM parts, but is designed to provide an extensible foundation for automatic annotation of any SBOL data.

5.4.1 Building a knowledge base

The simplest way to accomplish a data workflow for automatic annotation is to develop a script to “glue” multiple tools together, where the output of each tool becomes the input of the next (Fig. 5.4). While this approach solves the immediate problem, it is not modular: removing or adding tools from the workflow would require changing the code. It is also highly dependent on the ordering of the tools; e.g., a tool that annotates protein domains would have to run after the tool that determined the protein sequences.

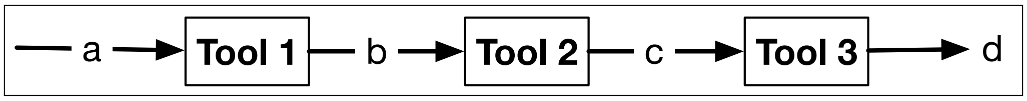


Figure 5.4: *The structure of a linear data workflow where the output of each tool is used as the input of the next. Tool 1 receives input a and produces output b; tool 2 receives input b and produces output c; and tool 3 receives input c and produces output d.*

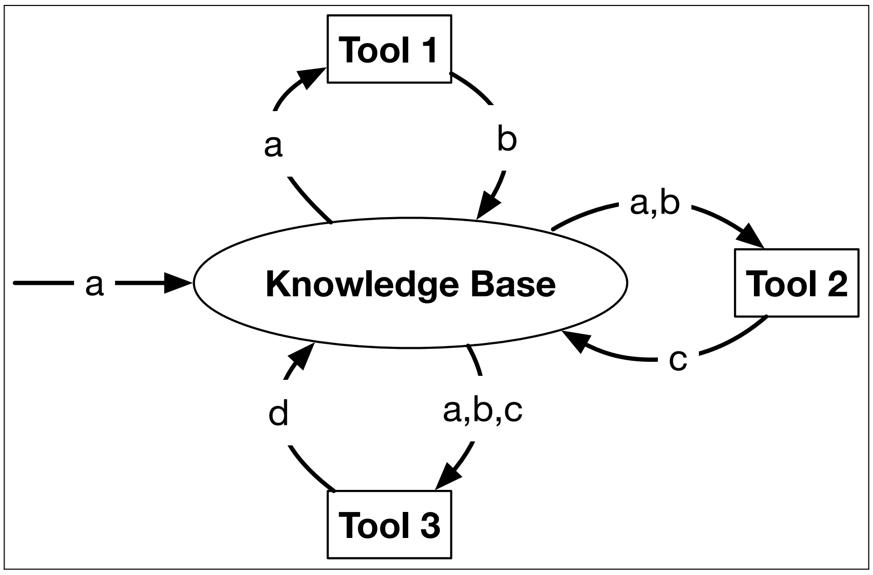


Figure 5.5: *The structure of the data workflow used by Enrichment, where each tool has access to the data generated by all other tools. Tool 1 receives input a and produces output b. Unlike the linear workflow described in Fig. 5.4, tool 2 now receives both a and b. As the knowledge base grows, each tool has the oppotunity to build upon any previous output.*

An alternative approach is to build a *knowledge base*, where each tool has access to the entire knowledge base, and therefore can both use the output of and become the input of any other tool (Fig. 5.5). Instead of writing a script with a specific order, each tool is implemented as an *integration*, which receives the knowledge base in its current state and can extend it with additional knowledge. For example, a CDS finder integration might look for any nodes in the knowledge base of type “Sequence”, and use them to emit nodes of type “CDS”. Subsequently, these nodes might be used by another integration to create "Protein" nodes for nodes of type "CDS". The relationship between the two specific integrations is never defined: they simply work over the same knowledge base, and it happens that the knowledge generated by one integration is useful for another.

The Enrichment knowledge base

The Enrichment knowledge base loosely follows the layout of a graph, where nodes, such as "a CDS" or "a protein", have relations such as "this CDS encodes this protein". The relations in Enrichment are not strictly graph edges, but fragments of knowledge which Enrichment internally refers to as "nuggets". The initial set of nodes defined for Enrichment to automatically annotate iGEM parts are:

- **DNASequence** — A nucleic acid sequence, equivalent to an SBOL2 `Sequence` object with the `encoding` property set to `http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html`. Described by the `na` string property which contains the sequence string.
- **SequenceFeature** — a feature of a DNA`Sequence`. Described by a set of sequence ranges.
- **ORF** (derived from `SequenceFeature`) — A potential open reading frame. Described by the `startCodon` and `stopCodon` sequence range properties and the `introns`, `exons`, and `CDSes` properties, each an array of sequence ranges.
- **Promoter** (derived from `SequenceFeature`) — A potential transcriptional promoter. Described by the `minus10`, `minus35`, and `tss` sequence range properties.
- **TFBS** (derived from `SequenceFeature`) — a potential binding site for a transcription factor, described by the `tf` property.
- **Terminator** (derived from `SequenceFeature`) — a potential transcriptional terminator.
- **Protein** — a conceptual protein. Described by the `aa` string property which contains the amino acid sequence.

These nodes are related using the following nuggets:

- **NORFMakesProtein** — indicates that an ORF node possibly codes for a `Protein` node
- **NProteinHasDegradationTag** — indicates that a `Protein` node has an SSRA degradation tag [118]
- **NProteinHasPfamHits** — indicates that a `Protein` node has hits in the Pfam [111] database
- **NProteinHasUniprotMatch** — indicates that a `Protein` node matches a protein found in UniProt

- **NProteinIsTF** — indicates that a protein is likely to be a specific transcription factor
- **NSequenceHasORF** — links `Sequence` nodes to ORF nodes
- **NSequenceHasPromoter** — links `Sequence` nodes to `Promoter` nodes
- **NSequenceHasTFBS** — links `Sequence` nodes to TFBS nodes
- **NTFBindsToTFBS** — indicates that a `Protein` node may bind to a TFBS node

The knowledge base is populated using a series of integrations, where each integration receives the entire knowledge base in its current state and has the opportunity to add additional information. For example, a gene finder uses DNA`Sequence` nodes to create ORF nodes. Another integration that performs translation can then extend the knowledge base containing ORF nodes by creating their corresponding `Protein` nodes, regardless of where the ORF nodes came from (Fig. 5.6). As an initial set of integrations designed for use with the regulatory circuit elements contained in the iGEM Registry, Enrichment provides:

- **IGFindORFs** — Search for open reading frames in DNA`Sequence` nodes, using either ve-sequence-utils [119] for prokaryota or Augustus [117] for eukaryota.
- **IGCreateProteinsForORFs** — Translate ORF nodes to create `Protein` nodes
- **IGSearchPfam** — Search Pfam [111] HMMs using `Protein` nodes to create `NProteinHasPfamHits` nuggets
- **IGCreateTFsFromPfamHits** — Create TF nodes for any `Protein` nodes with an associated `NProteinHasPfamHits` where the family refers to a TF family
- **IGFindDegradationTags** — Create `NProteinHasDegradationTag` nodes for `Protein` nodes with a sequence that have an ssRA degradation tag [118]
- **IGTransTerm** — Identify possible terminator regions in DNA`Sequence` nodes using TransTerm [120] and create `Terminator` nodes
- **IGPromoterHunter** — Identify possible promoter regions in DNA`Sequence` nodes using Promoter Hunter [121] and create `Promoter` nodes
- **IGFindTFBSsBlast** — Identify possible TFBS regions in DNA`Sequence` nodes using BLAST and create TFBS nodes
- **IGFindTFBSPWM** — Identify possible TFBS regions in DNA`Sequence` nodes using Position Weight Matrices (PWMs) and creates TFBS nodes

In addition to integrations, Enrichment also provides refinements. While integrations add to the knowledge base, refinements remove from it. For example, the `IGFindORFs` integration may add many potential shorter ORFs which are subsequences of a longer ORF. The `RLongestORF` refinement can be used to refine its contribution to the knowledge base by pruning any ORFs which overlap other ORFs. The initial set of refinements provided by Enrichment are:

- **RLongestORF** — Where there are overlapping ORF nodes, prunes all but the longest
- **RReverseOnly** — Prunes all sequence-related nodes that are not on the reverse strand
- **RCompleteTUsOnly** — Prunes any ORF nodes that are not surrounded by an identified `Promoter` and `Terminator` node
- **RForwardOnly** — Prunes all sequence-related nodes that are not on the forward strand
- **RMergeAdjacentTerminators** — Merges any overlapping `Terminator` nodes into one node

5.4.2 RDF representation

Once populated, the knowledge base can be used to generate a new SBOL2 document (Fig. 5.7). In order to allow the specification of interactions, the results are encapsulated by an SBOL `ModuleDefinition`. `FunctionalComponent` objects are created for nodes such as TFBS and Protein, and `Interaction` objects are created for nuggets such as `NTFBindsToTFBS`.

An important consideration for the RDF representation of Enrichment output was how to capture metadata concerning the provenance of added information.

As described in section 2.2.1, RDF is a simple data model consisting of subject-predicate-object triples. Unlike in graph databases such as Neo4j, it is not possible to annotate the edge between the subject and object with any additional information other than the predicate URI.

This problem is often addressed in RDF using *reification*, whereby the triple is "reified" as four triples using the `rdf:Statement` class and its `rdf:subject`, `rdf:predicate`, and `rdf:object` properties. The `rdf:Statement` can then be annotated with futher triples to provide metadata about the triple. However, this approach is highly verbose and cumbersome to query due to the additional layer of indirection.

Enrichment sidesteps this issue by using a simple, lightweight model for metadata based on PROV-O. In this model, resources are treated as immutable objects, where any addition of new properties to a resource A results in the generation of a copy of the resource A' with a new URI. The set of newly added triples can be inferred as the difference $B \quad A$, and the two resources A and B are connected using a `prov:Activity` which can be annotated to capture metadata about the enrichment step (Fig. 5.8).

5.4.3 Enrichment tool

The front-end of the Enrichment tool is a command-line utility. As input, it can take an SBOL1, SBOL2, FASTA, or GenBank file, which is used to create an initial knowledge base. An integration pipeline can then be constructed by specifying a list of steps to enable, which can be either integrations or refinements. As output, Enrichment produces SBOL files which can be used with any SBOL2 compliant software, including visualization tools — effectively providing a “FASTA to SVG” conversion.

Enrichment was tested with a number of parts from the SBOL-RDF conversion of the iGEM Registry. It was possible to produce and visualize SBOL2 from parts such as the Elowitz repressilator BBa_I5610 (Fig. 5.9), the araC-pBad system BBa_K808000 (Fig. 5.10), and the RFP transcriptional unit BBa_J04450 (Fig. 5.11b).

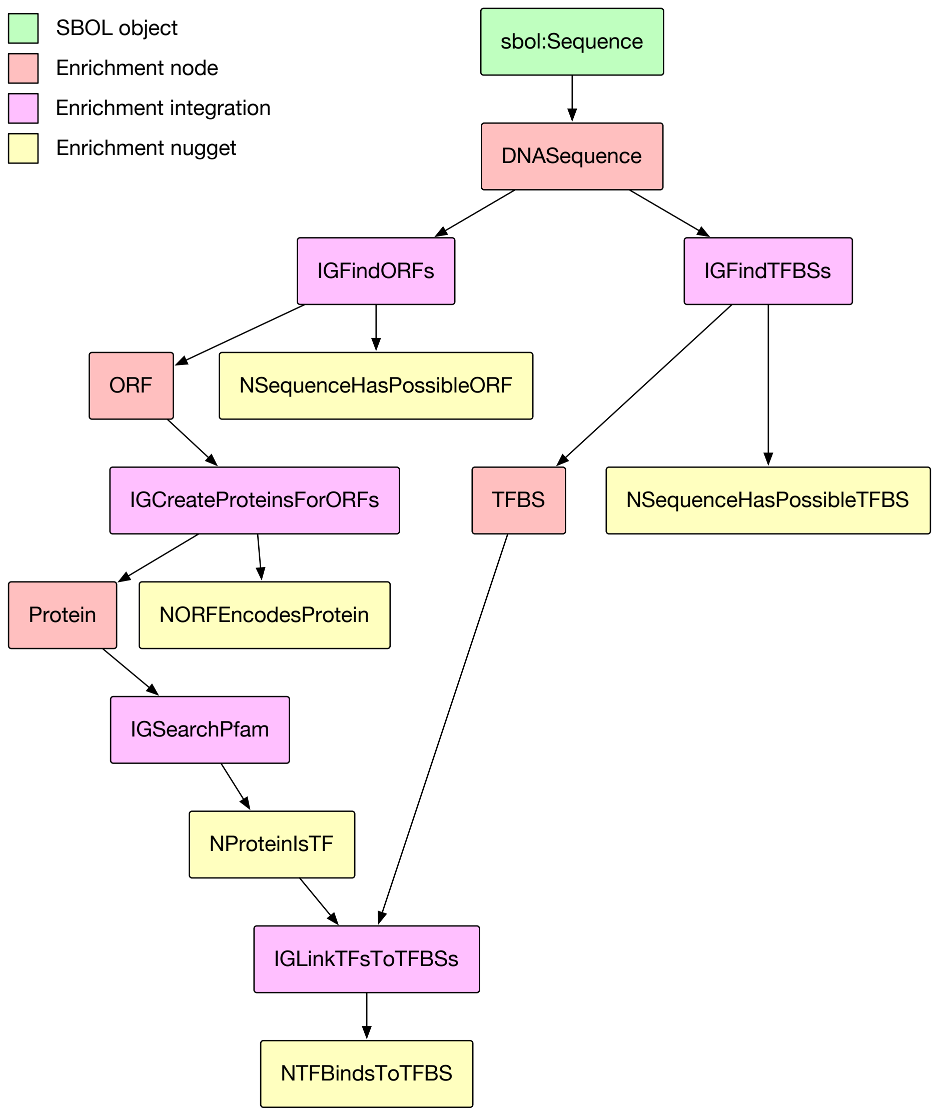


Figure 5.6: *An example of how a sequence leads to the creation of a knowledge base. From the sequence represented in SBOL, a DNASequence node is created. The DNASequence is then available to be used by two integrations: IGFindORFs, and IGFindTFBSs. IGFindORFs creates ORF nodes, along with NSequenceHasPossibleORF nuggets to link them to their respective DNASequence. Similarly, IGFindTFBSs creates TFBS nodes and NSequenceHasPossibleTFBS nuggets. The ORFs created by IGFindORFs are used by the IGCreateProteinsForORFs integration which performs translation, resulting in a Protein node and an NORFEncodesProtein node. The Protein node is then used by the IGSearchPfam node which identifies that it is part of a transcription factor family, resulting in an NProteinIsTF node. Finally, the IGLinkTFsToTFBSs integration maps the transcription factor to its corresponding binding site, resulting in the NTFBindsToTFBS nugget, which can later be used to generate an SBOL interaction.*

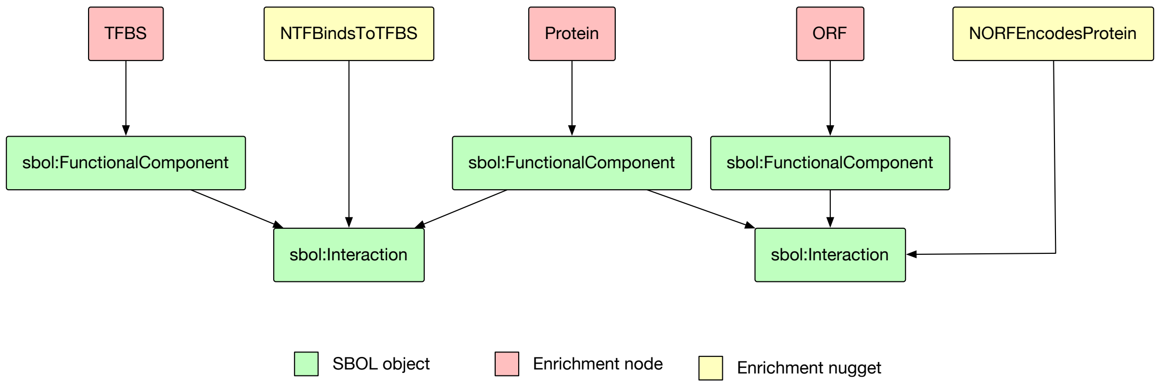
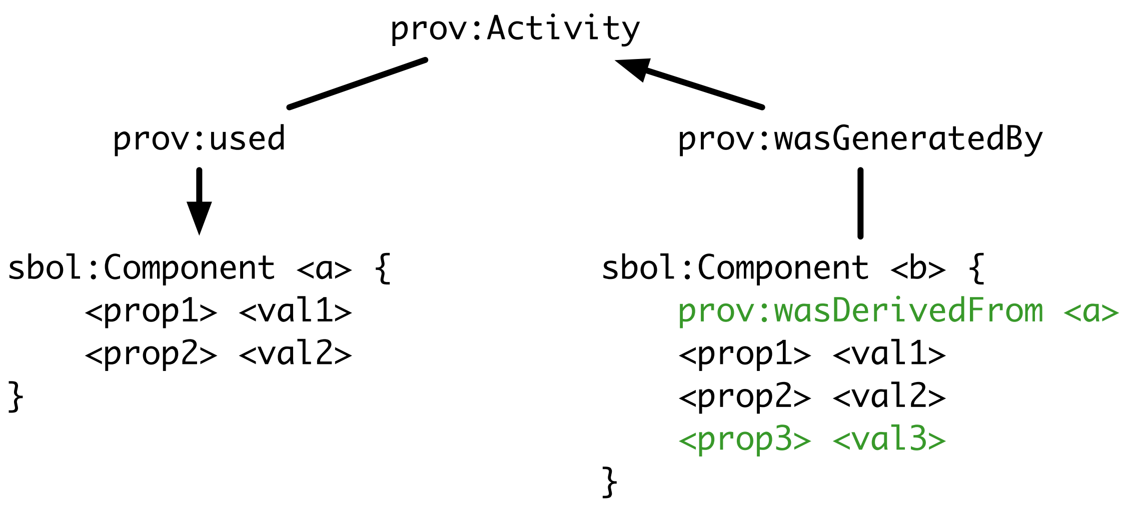


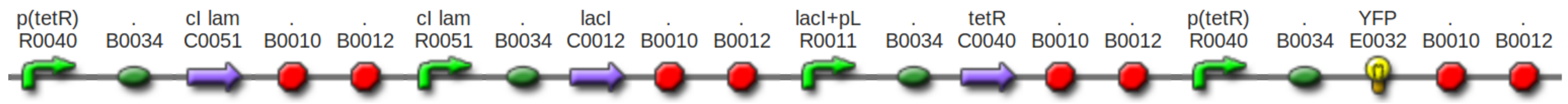
Figure 5.7: *Parts of the resulting knowledge base are mapped back to SBOL using the FunctionalComponent and Interaction classes. Nodes such as TFBS, Protein, and ORF map to FunctionalComponents, and nuggets such as NTFBindsToTFBS and NORFEncodesProtein map to Interactions.*



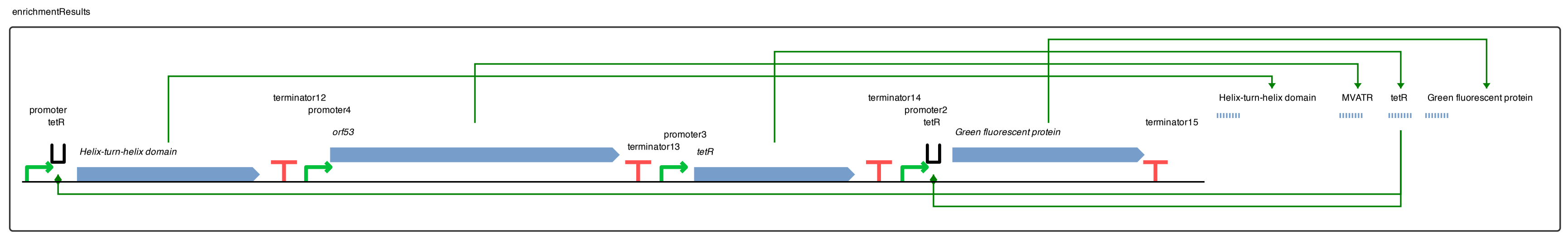
```
sbol:Component <a> {
    <prop1> <val1>
    <prop2> <val2>
}
```

```
sbol:Component <b> {
    prov:wasDerivedFrom <a>
    <prop1> <val1>
    <prop2> <val2>
    <prop3> <val3>
}
```

Figure 5.8: *Instead of using reification to provide metadata about added triples, Enrichment treats objects as immutable, creating a new copy when the object changes and connecting it to the original with a* `prov:Activity`*. The set of newly added triples can be inferred as the difference between the two objects.*

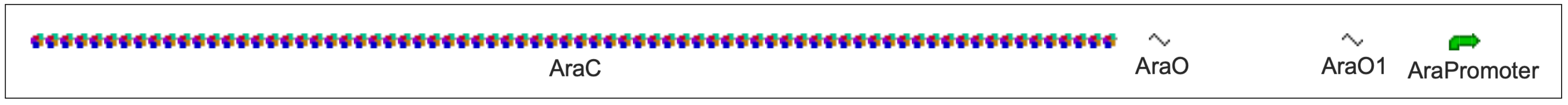


(a)

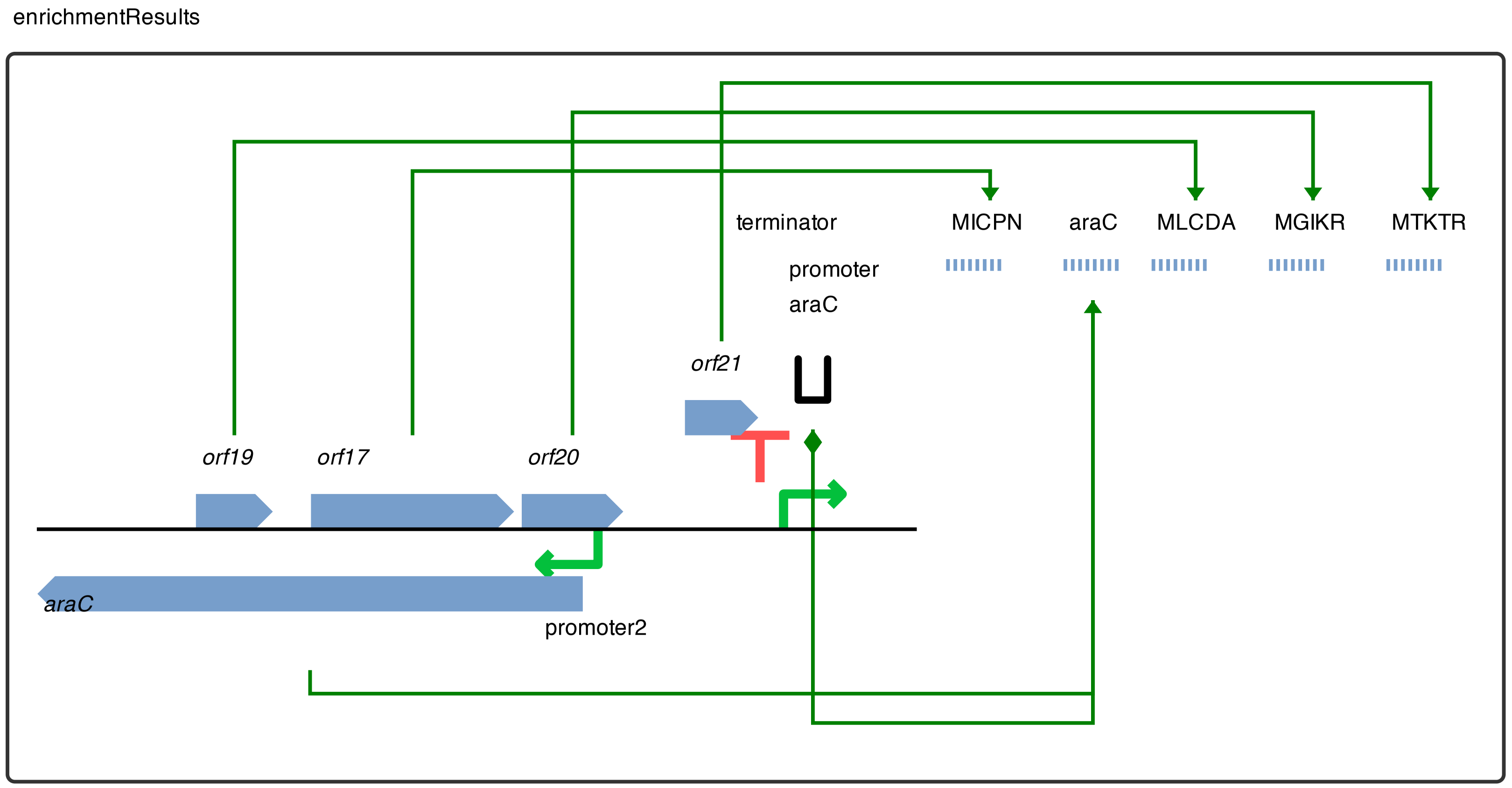


(b)

Figure 5.9: *The BioBrick BBa_I5610 as visualized by the iGEM registry (a), then stripped of its annotations and re-annotated as SBOL2 using Enrichment, and visualized using SynBioCAD (b). The integrations used were `IGFindORFs`, `IGPromoterHunter`, `IGTransTerm`, `RForwardOnly`, `RMergeAdjacentTerminators`, `RLongestORF`, `IGCreateProteinsForORFs`, `IGSearchPfam`, `IGCreateTFsFromPfamHits`, `IGFindTFBSsBlast`, and `IGLinkTFsToTFBSs`. The four transcriptional units have been identified, along with the tetR binding interactions.*

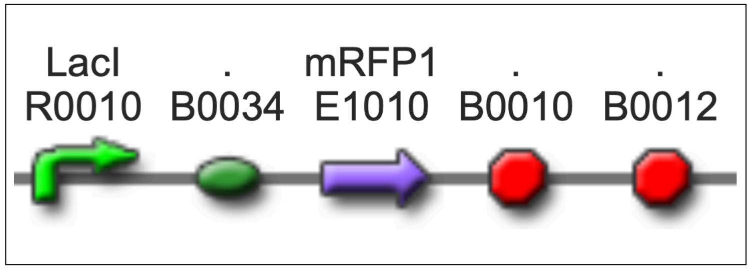


(a)

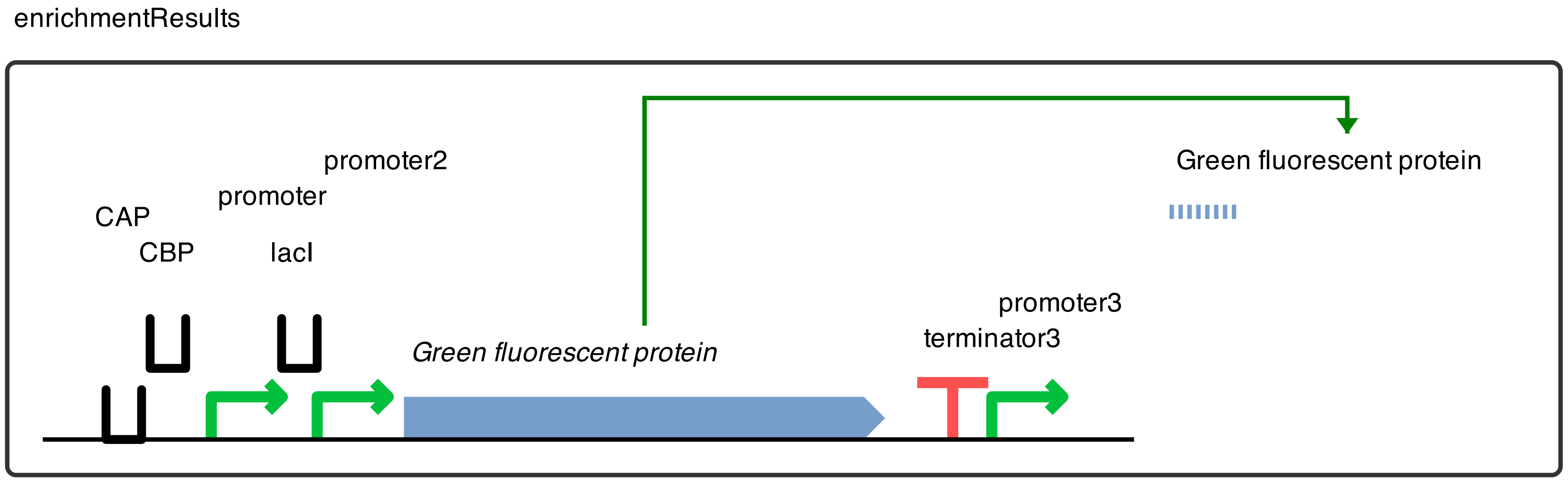


(b)

Figure 5.10: *The BioBrick BBa_K808000 as visualized by the iGEM registry (a), then stripped of its annotations and re-annotated as SBOL2 using aracpbadigem, and visualized using SynBioCAD (b). The integrations used were* `IGFindORFs`*,* `IGPromoterHunter`*,* `IGTransTerm`*,* `RMergeAdjacentTerminators`*,* `RLongestORF`*,* `IGCreateProteinsForORFs`*,* `IGSearchPfam`*,* `IGCreateTFsFromPfamHits`*,* `IGFindTFBSsBlast`*, and* `IGLinkTFsToTFBSs`*. The araC promoter, its binding site, and the araC promoter and CDS in the reverse direction have been identified. A spurious set of coding sites have been labelled on the forward strand, which resulted in the creation of unidentified "proteins" by* `IGCeateProteinsForORFs`*.*



(a)



(b)

Figure 5.11: *The BioBrick BBa_J04450 as visualized by the iGEM registry (a), then stripped of its annotations and re-annotated as SBOL2 using Enrichment, and visualized using SynBioCAD (b). The integrations used were `IGFindORFs`, `IGPromoterHunter`, `IGTransTerm`, `RForwardOnly`, `RMergeAdjacentTerminators`, `RLongestORF`, `IGCreateProteinsForORFs`, `IGSearchPfam`, `IGCreateTFsFromPfamHits`, `IGFindTFBSsBlast`, and `IGLinkTFsToTFBSs`. RFP has been mis-identified as GFP due to homology, and a spurious promoter has been labelled at the end of the part.*

5.5 Discussion & Conclusion

The research goals of this chapter were to investigate (a) how the iGEM Registry can be mapped to the SBOL data model, and (b) how SBOL data, such as these converted parts, can be enriched using automated sequence annotation. Goal (a) was realised by developing a conversion of the iGEM Registry to SBOL, by mapping its database representation to the Registry data model and its semantics to standardized ontology terms. Goal (b) was realised through the development of Enrichment[4], a tool for the automatic annotation of sequences using SBOL2; and its application to the SBOL version of the iGEM dataset.

There are several immediate advantages of converting a dataset to SBOL. SBOL can serve as a mediator to make data from different sources work together. For example, an iGEM part converted to SBOL could be used in a design alongside a part downloaded from JBEI-ICE, and benefit from compatibility with any SBOL-compliant tooling. Furthermore, SBOL is highly machine-tractable even where the source may not have been. For example, while the iGEM Registry has a basic search facility allowing questions such as "which parts contain a specific keyword" to be answered, the rich power of graph queries demonstrated earlier in this chapter make it possible to answer more complex questions about the dataset once in an SBOL representation.

Using the increased scope of the SBOL2 data model to enrich designs with additional knowledge is a logical next step following a direct conversion. The automated approach used by Enrichment has been shown to produce some convincing results, though it is dependent on the accuracy of tools to correctly predict sequence features and protein domains. There is much room for improvement in this area. However, the existence of a data standard capable of the representation of whole designs is a recent development, and there are large datasets of existing parts stored with reductive data representations. Finding ways to "fill in the blanks" will likely become an active area of research for the SBOL community, and Enrichment is an initial effort to explore what can be done by integrating tooling from the bioinformatics domain into an SBOL data workflow.

The SBOL version of the iGEM Registry is now available as a link from the page for each part in the official Registry (Fig. 5.12).

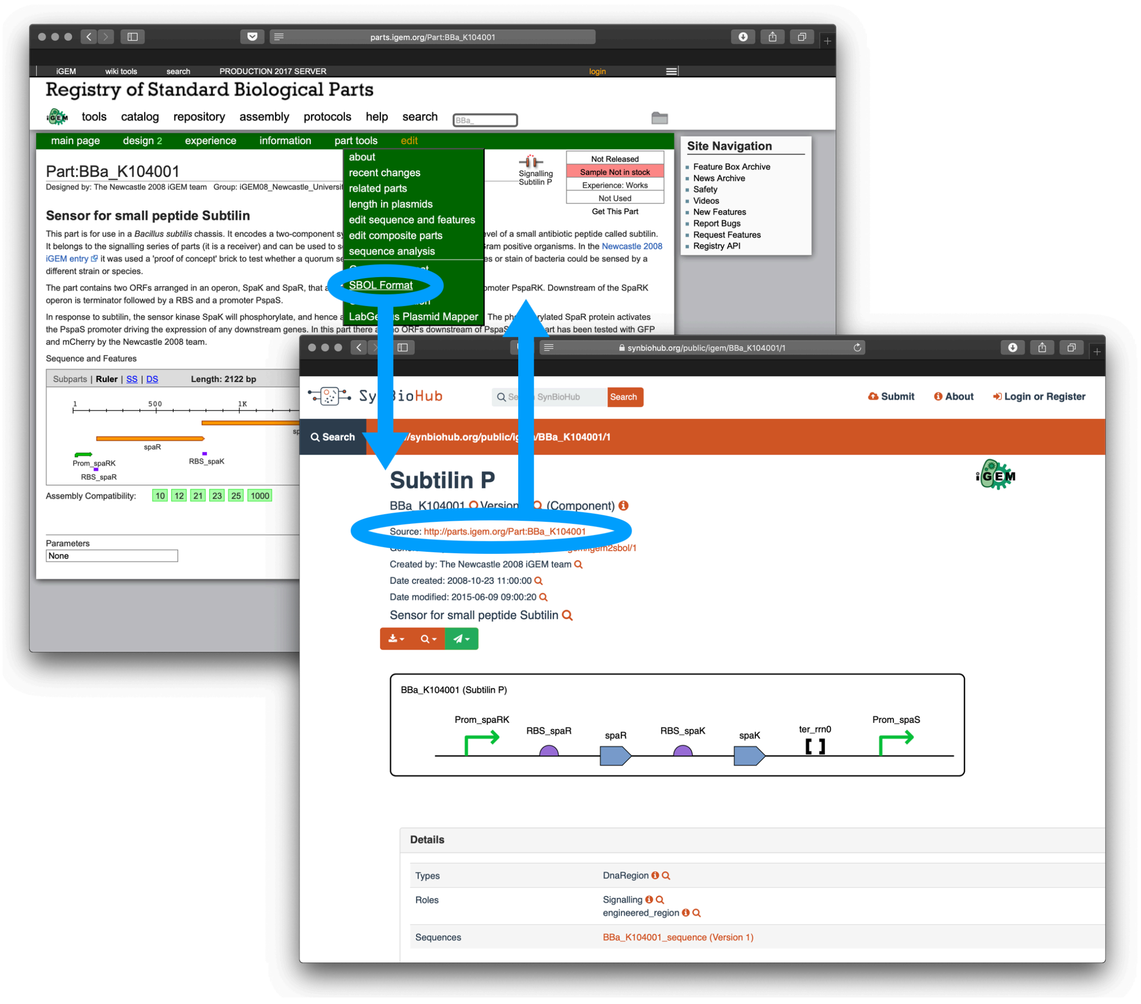
5.5.1 Future work

Upstream adoption of SBOL

The Registry database schema is undocumented, making some of the work in this chapter essentially a reverse engineering effort. The iGEM competition has contributed significantly to the *in vitro* standardization of BioBricks, but does not yet have a corresponding *in silico* standard. The Registry is in need of a formalised data model, whether a formalisation of the current database representation or an adoption of an existing standard. CORRECTION: what do you mean by in vitro

Ideally, the Registry would use SBOL as its data model and store it in a machine-tractable database such as an RDF triplestore, instead of using its own bespoke data representation and database schema. This would be beneficial to both the

[4]`https://github.com/udp/dnarichment`



Registry of Standard Biological Parts

Part:BBa_K104001

Designed by: The Newcastle 2008 iGEM team Group: iGEM08_Newcastle_Universit

Sensor for small peptide Subtilin

This part is for use in a *Bacillus subtilis* chassis. It encodes a two-component sy vel of a small antibiotic peptide called subtilin. It belongs to the signalling series of parts (it is a receiver) and can be used to s ram positive organisms. In the Newcastle 2008 iGEM entry it was used a 'proof of concept' brick to test whether a quorum se es or stain of bacteria could be sensed by a different strain or species.

The part contains two ORFs arranged in an operon, SpaK and SpaR, that a moter PspaRK. Downstream of the SpaRK operon is terminator followed by a RBS and a promoter PspaS.

In response to subtilin, the sensor kinase SpaK will phosphorylate, and hence a The ph rylated SpaR protein activates the PspaS promoter driving the expression of any downstream genes. In this part there a o ORFs downstream of PspaS art has been tested with GFP and mCherry by the Newcastle 2008 team.

Sequence and Features

Parameters

None

Subtilin P

BBa_K104001 Version (Component)

Source: http://parts.igem.org/Part:BBa_K104001

igem/igem2sbol/1

Created by: The Newcastle 2008 iGEM team

Date created: 2008-10-23 11:00:00

Date modified: 2015-06-09 09:00:20

Sensor for small peptide Subtilin

Details

Types	DnaRegion
Roles	Signalling engineered_region
Sequences	BBa_K104001_sequence (Version 1)

Figure 5.12: *The "SBOL Format" link in the official iGEM Registry now navigates to the iGEM-SBOL version of the part created as part of this work, hosted in the SynBioHub repository, also developed as part of this work and described in chapter 6.*

SBOL community and to iGEM: the Registry would benefit from more powerful querying capabilities, automatic integration with any tooling with SBOL support, and the machine-tractable representation of all aspects of a design; and the SBOL community would benefit from both having a vast repository of original SBOL data and the endorsement of iGEM.

Integration of further datasets

The iGEM Registry is only one of many potential datasets of parts that would be useful to have in an SBOL representation. Although as demonstrated in chapter 4 it is now possible to perform federated queries across SBOL and different data standards such as UniProt RDF, the integration could be even simpler if every dataset was already harmonized to use SBOL.

The script used for the conversion of the Registry to SBOL used was highly specific to the iGEM dataset. However, the work described in this chapter could potentially be generalised to become a well-defined pipeline for the standardization and enrichment of design knowledge. Rather than being justified only as "standardization", the conversion of a dataset to SBOL would have a clear purpose: to learn more about parts and designs by canonicalizing their representation using a data model with the ability to capture a rich set of design information, and then populating that data model using automated knowledge gathering.

Extending Enrichment

Boinformatics tools are traditionally used for the predicting features in naturally occurring sequences. The idea of applying them to *engineered* parts at first seems illogical: why would it be necessary to automatically annotate something that has been designed? The fact that this can be justified by the existence of large repositories of engineered sequences which can benefit from further analysis suggests that we are at a turning point in the history of synthetic biology. We are now engineering biology to an extent that natural sequences are no longer the only kind of sequences which can benefit from automatic annotation.

The ideas behind Enrichment are a proposed solution to a practical problem: SBOL2 has a wider scope than SBOL1, but the data does not. Filling in the remaining parts of the data model using bioinformatics tooling can mitigate this. The predictions made by bioinformatics tools such as promoter finders and HMM scanners are not completely accurate, but for applications such as visualisation they may be accurate enough.

While the first version of Enrichment provides a core set of integrations and refinements, the specific choice of tools and data sources used are for the purpose of proof-of-concept. There are countless relevant sources of data which could be used to build relevant integrations, with Enrichment serving as a modular open-source platform to connect them together.

SBOL integration

While Enrichment both reads and writes SBOL, it could have much more comprehensive support. Currently, existing sequence annotations in the input data are ignored, and the sequence is re-annotated from the ground up. Combining the existing annotations with predicted annotations would potentially result in the creation of a more accurate knowledge base in the event that annotations made by the user are present and accurate.

Enrichment also does not yet merge its output SBOL back into the source file; the output of Enrichment is essentially a separate collection of SBOL data from the original sequence. The process of merging two sets of SBOL data that describe the same resource in different levels of detail with possibly conflicting annotations has not yet been defined.

The ability to enrich existing design information in place, if developed, would also create the possibility of analysing designs *during* the design process. For example, a CAD tool such as SynBioCAD described in chapter 7 would have the ability to suggest design elements such as interactions dynamically, in a manner similar to code completion provided by programming environments.

Conclusion

This chapter described the conversion of an existing repository of design information, the iGEM Registry, to SBOL format; and explored how once in SBOL format, it is possible to automatically enrich design information by populating the SBOL data model using an integrated pipeline of bioinformatics tooling.

The SBOL conversion of the iGEM Registry is now publicly available on the SynBioHub repository [5], enabling iGEM Registry parts to be used with any SBOL-compliant tool such as SBOLDesigner [89] and the SynBioCAD tool described in chapter 7.

[5]`https://synbiohub.org/public/igem/igem_collection/1`

Part III

Sharing and Dissemination

6. SynBioHub: a standards-enabled design repository for synthetic biology

6.1 Introduction

Ever since the early days of computer science, one of the most pervasive problems has been the lack of reuse of existing software. In 1969, Richard Hamming famously said the following during his Turing Award lecture:

> *Indeed, one of my major complaints about the computer field is that whereas Newton could say, "If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.* [122]

While this problem has not been entirely solved in the nearly 50 years since, the situation has improved significantly. There are now vast repositories of open-source code such as GitHub [123], GitLab [124], and Bitbucket [125]. Most programming languages have package managers such as pip [126] and npm [127], which allow useful code libraries to be downloaded and made available to the developer in seconds.

It is easy to draw parallels between what Hamming complained of in 1969 and the situation more recently in synthetic biology. Publications about engineered biological systems have often omitted crucial information necessary for reproducibility — sometimes failing to include even sequences [8], which is equivalent to releasing a computer program without the source code. As highlighted in chapter 3, designs also often include a range of non-sequence information, such as compositional hierarchy, the relation between distinct parts, and computational models. Such information is important both to understand the motivation behind the design, and how to reproduce, modify, and — crucially — *reuse* it as part of larger systems.

One of the challenges to addressing this problem is that it has been technically impossible to comprehensively capture a design in a data file that can be distributed alongside a publication. Chapter 3 detailed how SBOL can help in this regard. But capturing the design data is only one part of the puzzle. Revisiting the software engineering analogy, SBOL provides the *source code language* for designs. It does not provide the means for the source code to be published and shared with others. Synthetic biology is essentially lacking an equivalent to GitHub.

While it is certainly possible to use existing software to share some information about designs, all of the existing solutions define their own data model, supporting SBOL only as an export or import option. Ideally, a community repository should be built directly upon a community standard, enabling the information captured by the repository to grow along with the information captured by the standard.

The research goal of this chapter is to investigate how SBOL can be used as a standard to create a user-facing platform for sharing designs. Specifically, this chapter describes the design and implementation of *SynBioHub*, a design repository for synthetic biology.

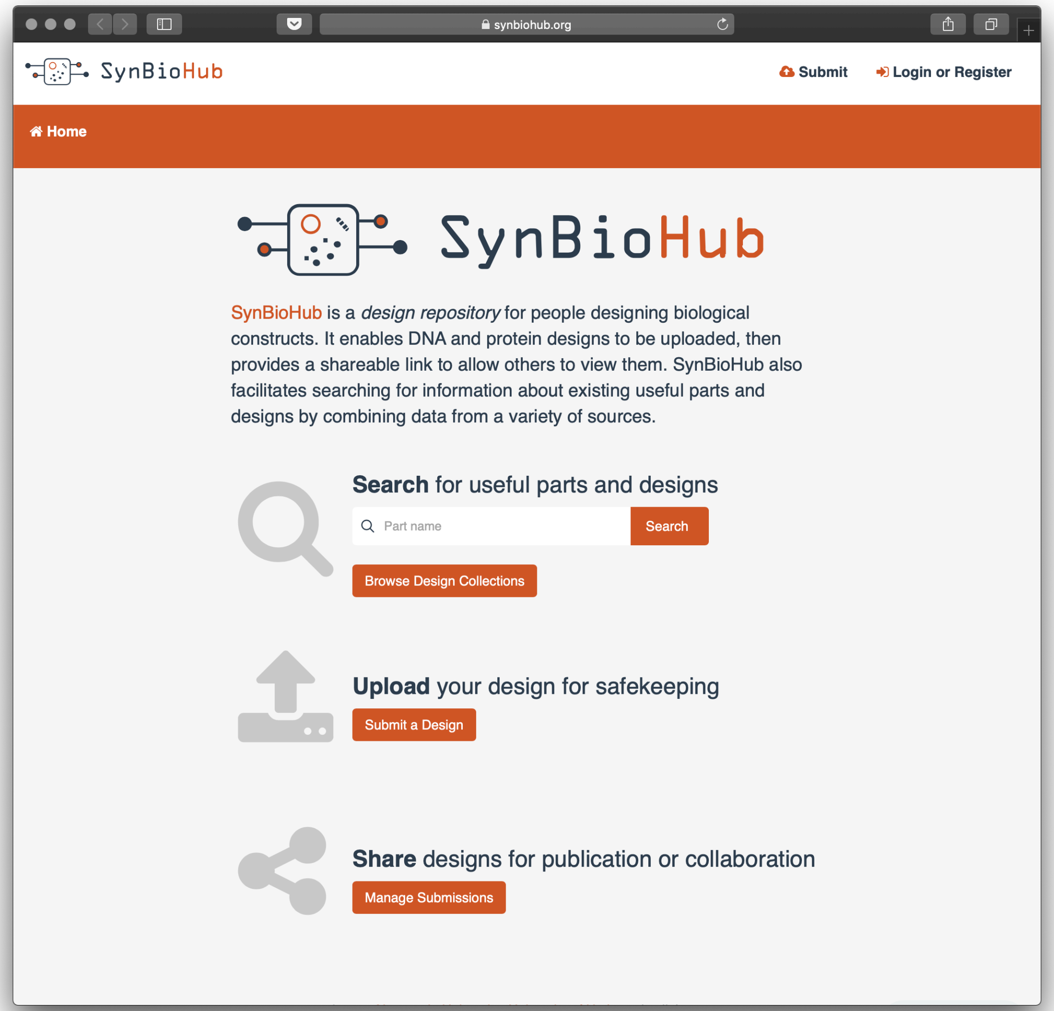
Attribution: While SynBioHub is now an active open-source project with many contributors worldwide, the initial version of SynBioHub was developed entirely as part of this work, based on a user interface design by Antarctic Design. The SynBioHub Lab project is being developed in collaboration with Christian Atallah.

6.2 Architecture of SynBioHub

In chapter 3, the application of RDF triplestores to SBOL was explored, resulting in the development of the SBOL Stack: a database for SBOL using an RDF triplestore as the backend. While a database alone does not satisfy the need for a user-facing repository, the SBOL Stack solves the first problem of "where to put the data". SynBioHub builds upon the SBOL Stack by providing a frontend user interface built using Web technology, much in the same way that a traditional Web application can be built atop a database such as MySQL. SynBioHub is implemented as a JavaScript application using node.js, the sboljs library described in chapter 3, and VisBOL [94] for design visualization. SynBioHub has three methods of communicating with the underlying SBOL Stack triplestore:

- Using SPARQL queries over SBOL/RDF to retrieve specific properties
- Fetching SBOL non-recursively using sboljs
- Fetching SBOL recursively using sboljs

The first approach, using SPARQL queries, is the most efficient. It takes advantage of the fact that SBOL is stored in an RDF triplestore, which enables SPARQL queries to be used to retrieve a specific subset of SBOL data. For example, the SynBioHub page which shows the members of a collection uses a SPARQL query is similar to:


SynBioHub is a *design repository* for people designing biological constructs. It enables DNA and protein designs to be uploaded, then provides a shareable link to allow others to view them. SynBioHub also facilitates searching for information about existing useful parts and designs by combining data from a variety of sources.

Search for useful parts and designs

Upload your design for safekeeping

Share designs for publication or collaboration

Figure 6.1: *The front page of SynBioHub, a design repository for synthetic biology. Users can choose to search for, upload, or share designs for publication or collaboration.*

```
SELECT ?member ?name ?description WHERE {

    <https://synbiohub.org/public/igem/igem_collection/1>
        sbol:member ?member .

    OPTIONAL {
        ?member dcterms:title ?name .
    }

    OPTIONAL {
        ?member dcterms:description ?description .
    }
}
```

Using SPARQL to retrieve data from the triplestore in this manner means that only the specific properties required to render the page are retrieved, and it is not necessary to load the SBOL using an SBOL library. The limitation of this approach is that every SBOL property that is required for the page has to be included in the SPARQL query and manually interpreted by SynBioHub, which repeats much of the implementation work already completed by SBOL software libraries.

The second approach is to non-specifically retrieve all of the triples associated with an SBOL object, and use sboljs to load them. This can be accomplished by filtering for triples with a specific subject, and then using SPARQL CONSTRUCT to create an RDF graph:

```
CONSTRUCT {
    ?s ?p ?o
} WHERE {
    ?s ?p ?o .
    FILTER(?s = <https://synbiohub.org/public/igem/BBa_K1367005/1>)
}
```

The resulting graph is valid RDF/XML, which can be loaded directly by sboljs and navigated as with any SBOL file. This approach works well for rendering pages with basic information about an SBOL object, but it is limited by child objects not being included. For example, it would not have worked for the aforementioned collection example, as the immediate properties of an SBOL2 `Collection` are only the URI of each member, not its name and description.

The third approach is to recursively retrieve an entire SBOL top-level, including all of its child objects. While potentially much slower than either of the other methods, comprehensively retrieving a top-level is the only way to provide an SBOL "document" which is ready to be loaded by tooling outside of SynBioHub. Originally, retrieving SBOL was implemented using the recursive strategy defined by the SBOL Stack (section 3.6.2). However, for large SBOL documents, repeating SPARQL queries for every unresolved URI can quickly become expensive. Instead, SynBioHub now adds a `synbiohub:topLevel` property to all SBOL objects upon submission. With this predicate, retrieving all of the triples associated with an SBOL top-level and its children can be accomplished in a single SPARQL query:

```
CONSTRUCT {
    ?s ?p ?o
} WHERE {
    ?s ?p ?o .

    FILTER(
        ?s
        synbiohub:topLevel
        <https://synbiohub.org/public/igem/BBa_K1367005/1>
    )
}
```

6.2.1 SynBioHub vocabulary

As the SBOL data is stored directly as RDF in the triplestore, there is no SynBioHub-specific data model. Where SynBioHub needs to add additional information which is not covered by the SBOL data model it can simply be added as additional triples. For example, SynBioHub has the concept of an attachment, where additional files (e.g. characterisation data) can be uploaded alongside SBOL. These are added to the triplestore using a custom vocabulary: the attachment is assigned the object type `synbiohub:Attachment`, and assigned to an SBOL object using the predicate `synbiohub:attachment`. The namespace used for SynBioHub terms is `http://wiki.synbiohub.org/wiki/Terms/synbiohub#`, which resolves to a definition of each term on the SynBioHub wiki.

Classes

`synbiohub:Attachment` – A resource (e.g. file, or SBML [27] model) associated with an `sbol:TopLevel`. The metadata for the attachment is described in RDF, but only the SHA-1 sum of the file contents is stored, using the `synbiohub:attachmentHash` predicate. This hash can then be mapped to the file contents in a separate, compressed file store.

Predicates

`synbiohub:attachment` – Associates a `synbiohub:Attachment` with an `sbol:TopLevel`. `synbiohub:attachment` has the domain `sbol:TopLevel` and the range `synbiohub:Attachment`.

`synbiohub:attachmentHash` – The SHA-1 sum of a file associated with a `synbiohub:Attachment`, represented as a string. This allows the content of the file to be located in the SynBioHub file store, which is separate from the RDF triplestore. `synbiohub:attachmentHash` has the domain `synbiohub:Attachment` and the range `xsd:string`.

`synbiohub:attachmentSize` – The size of the file associated with a `synbiohub:Attachment`, in bytes. `synbiohub:attachmentSize` has the domain `synbiohub:Attachment` and the range `xsd:integer`.

`synbiohub:mutableDescription` – A description for a `sbol:TopLevel` that, unlike `dcterms:description`, is mutable. That is, modifying the `mutableDescription` in SynBioHub does not require creating a new version of the `sbol:TopLevel`. `synbiohub:mutableDescription` has the domain `sbol:TopLevel` and the range `xsd:string`.

`synbiohub:snapshotOf` – SynBioHub has the functionality to create snapshots of the triplestore which used to roll back the state of the repository. `synbiohub:snapshotOf` predicate indicates that an `sbol:TopLevel` is a point-in-time snapshot of another `sbol:TopLevel`. `synbiohub:snapshotOf` has the domain `sbol:TopLevel` and the range `sbol:TopLevel`.

`synbiohub:topLevel` — Used to indicate which is the nearest top-level to an SBOL object. `synbiohub:topLevel` has the domain `sbol:Identified` and the range `sbol:TopLevel`.

The only data that are not stored in the triplestore are sensitive information such as usernames and password hashes. These are stored in an entirely separate SQLite database to ensure no possibility of inadvertent access through SPARQL queries. The only connection between the SQLite database and the RDF graph is the URI of each user assigned by SynBioHub, which is used in both the RDF predicate `synbiohub:ownedBy` and the `uri` column in the SQLite user table.

6.2.2 Users and graphs

One of the defining features of code repositories such as GitHub is the ability to have both public and private repositories. Projects often begin as private repositories, and then are made public once they reach maturity. SynBioHub implements this functionality with the use of multiple RDF graphs: one large graph for everything that is public, and a separate graph for each user.

Upon uploading a file (FASTA, GenBank, or SBOL1/2) to SynBioHub, it is first converted to SBOL2. The URI prefix of the URIs in the resulting SBOL2 data is then changed to the URL of the user, and if a part is later made public the prefix is changed to the URL of the public store. For example:

- A user uploads a file containing an SBOL `ComponentDefinition` with the URI `http://testdesign/promoter/1`. The URI prefix `http://testdesign/` is discarded and replaced with the SynBioHub URL of the user, e.g. `https://synbiohub.org/user/james/promoter/1`, and then the RDF is stored in the graph assigned to the user.
- The user decides to make the design public. The URI prefix is again changed from `https://synbiohub.org/user/james/` to the SynBioHub public URL, e.g. `https://synbiohub.org/public/promoter/1`.

Changing the URI prefix to match SynBioHub URLs means that each SBOL URI is now also a resolveable URL, as opposed to fabricated URIs with prefixes such as `http://testdesign/`. All SynBioHub URLs follow the “compliant URIs” scheme defined by the SBOL specification, where the URI is constructed from the prefix combined with the SBOL `displayId` and `version` properties.

6.3 Interfaces

While the goal of SynBioHub is to provide a user-facing design repository akin to JBEI-ICE, the purpose of its development is to improve the computational tractability of design information. As such, SynBioHub provides two interfaces: a Web interface which functions as a visual abstraction layer, and a programmatic API which provides access to the underlying SBOL data model.

6.3.1 Web interface

The SynBioHub Web interface allows users to:

- Browse through collections of parts (Fig. 6.2). Collections are defined by the presence of SBOL2 `Collection` objects in the triplestore.
- Search for parts, either using the free text or advanced search pages, which generate SPARQL queries behind the scenes; or manually using SPARQL queries (Fig. 6.3).
- View information about parts, including their visualisations rendered using VisBOL (Fig. 6.5).
- Submit new parts from local SBOL2, GenBank, or FASTA files (Fig. 6.4). The parts are converted to SBOL2, and then stored as RDF in the triplestore.

6.3.2 Programmatic API

As SynBioHub is built atop an RDF triplestore, it provides a SPARQL interface which can be used with RDF tooling such as Linked Data Fragments [72].

SPARQL queries are very low-level, making operations which should be simple — such as retrieving and inserting parts — require an understanding of SPARQL, which is an unacceptable requirement for a software developer to interface with SynBioHub. To address this issue, SynBioHub also provides a programmatic HTTP API which abstracts over the underlying triplestore.

The API endpoint for retrieving SBOL for a part is simply the URL of the part. The page decides whether to display the SBOL representation or the rendered Web page depending on the `Accept` header of the HTTP request, where `Accept: text/html` as sent by browsers results in the rendered page and `Accept: application/rdf+xml` results in the SBOL RDF+XML. This approach is also used for endpoints such as `/search`, where `Accept: application/json` returns results as a JSON result set rather than displaying the search form. The complete set of API endpoints provided by SynBioHub is documented in the SynBioHub wiki [1].

In order to make the API easier to use from software, client libraries are also provided for Java and JavaScript (Fig. 6.6). Additionally, the Java library has now been incorporated into libSBOLj, meaning that users of libSBOLj can use SynBioHub immediately without the requirement of any additional library support — the first time libSBOLj has had support for access to remote resources.

6.4 Web of Registries

SynBioHub supports the Web of Registries (WoR) [128] concept. Both other SynBioHub instances and ICE instances can be added as remote repositories to which queries can be federated, enabling each SynBioHub instance to act as a portal to multiple distinct upstream sources.

[1]`http://wiki.synbiohub.org/wiki/HTTP_API`

One of the interesting results of SynBioHub using an RDF triplestore is that data integration techniques, such as the previously discussed innovation of Linked Data Fragments [71], can be applied. Using LDF, it is possible to view the entire Web of Registries as a single, unified database over which federated SPARQL queries can be executed. This enables the design process to be distributed. Instead of collecting all of the design information into a single place, knowledge about the different components that make up a design can be split across multiple SynBioHub instances and aggregated using federated LDF SPARQL queries (Fig. 6.8).

6.5 SynBioHub Lab

When SynBioHub was originally developed, the scope of SBOL was limited to describing designs. Partly as a result of this work, SBOL is now also able to document knowledge from other stages of the synthetic biology lifecycle, such as information about constructs and experimental data. However, the SynBioHub user interface is heavily design-focused. While it can display information about the SBOL `Implementation`, `Experiment`, and `ExperimentalData` classes, it does not provide the means to populate them.

SynBioHub Lab (Fig. 6.10) is a re-design of SynBioHub with an emphasis on capturing whole-lifecycle knowledge for synthetic biology. In addition to an updated user interface, SynBioHub Lab also takes advantage of recent software innovations to greatly reduce the size and complexity of its codebase, with a complete re-implementation of the code in TypeScript and a port of SBOL-related functionality to the new sbolgraph library.

6.5.1 Whole-lifecycle knowledge management

The motivation for the development of SynBioHub Lab was to extend the scope of SynBioHub from a design repository to a “one-stop shop” for recording information about the entire synthetic biology lifecycle.

In the original version of SynBioHub, design information is submitted by uploading an SBOL, GenBank, or FASTA file. As part of the submission process, the user specifies a name and description for the design, which become the name and description of an SBOL `Collection` object in the triplestore. While further files can later be uploaded to the same collection, the process of creating a collection is strongly linked to uploading design information.

In SynBioHub Lab, the process of creating a collection (a “project” in SynBioHub Lab) is entirely separate from uploading design information. The form to create a project only requests a name and description. The subsequent project view (formerly the view for collection members in SynBioHub) is split into three sections: designs, constructs, and experiments (Fig. 6.12). The button to add a design leads to an upload form which accepts the same file formats as the original SynBioHub. The button to add a construct leads to a separate form to create and populate an SBOL `Implementation`, and the button to add an experiment leads to a form to create and populate an SBOL `Experiment`.

6.5.2 Nomenclature

The purpose of the SynBioHub user interface is to provide an abstraction layer that hides the complexity of the underlying SBOL data model from the user. In the original version of SynBioHub, this is not a perfect abstraction — something highlighted in a recent review paper of SynBioHub and JBEI-ICE, which said of SynBioHub that "the semantic annotation process should be biologist-friendly and hide the underlying RDF predicates" [129]. There are still many occurrences of SBOL nomenclature which could be replaced with more intuitive terminology. For example, what SynBioHub refers to as a "collection" could be renamed to a "project", which would bring it in line with other software such as Benchling [55]. SynBioHub Lab removes all SBOL nomenclature from the user interface. For example:

- Collections are now referred to as "projects"
- Components are now referred to depending on their type. For example, a component with a protein type is referred to as "Protein". A DNA component with a role of promoter is referred to as "Promoter".
- Implementations are now referred to as "constructs"

6.5.3 Architectural changes

As described in section 6.2, the original version of SynBioHub has three methods of communicating with the triplestore: directly via SPARQL queries, by non-recursively retrieving SBOL, and by recursively retrieving SBOL. An example of where the first approach is necessary is for retrieving the name and description of all members of a collection. Non-recursively retrieving the collection would not include the details of its members, and recursively retrieving the collection would result in at least one separate SPARQL query for each member. Consequently, SynBioHub issues a SPARQL query which specifically retrieves the name and description of all members, and iterates the result set manually to render the collection page. The result set must be iterated manually as it cannot be loaded by sboljs; it is only a partial view of the SBOL data model, and would not contain enough information to populate the class hierarchy.

In SynBioHub Lab, the code is greatly simplified by using the sbolgraph library described in section 3.3. Unlike previous SBOL libraries, sbolgraph is implemented using a set of classes which provide views over an underlying RDF graph, rather than as a class hierarchy which is populated by loading a document. The process of rendering a page is simply to issue a SPARQL CONSTRUCT query that retrieves the relevant data (e.g., the members of a collection and their metadata); load the result set into the RDF graph; and use the sbolgraph facade classes to access the partially populated data model. There is no requirement to manually iterate SPARQL result sets.

SynBioHub Lab also changes how design visualizations are produced. Instead of generating a VisBOL display list and using VisBOL client-side in the browser to render SVG, designs are rendered server-side using the SynBioCAD visualizer described in chapter 7, and the resulting SVG is included as part of the design page. Unlike VisBOL, SynBioCAD has comprehensive support for compositional hierarchy, resulting in significantly more detailed design visualizations (Fig. 6.11).

6.6 Discussion & Conclusion

The ability to effectively discover and share information is the foundation of science. Any improvement in this area has the potential to accelerate research. In synthetic biology, the need is particularly great: there is no platform that can be used to share a comprehensive description of a design using a machine-tractable data representation.

The richness of the SBOL standard addresses the issue of the data representation, but it does not provide the platform for sharing. While there are existing repositories that are user-accessible and support SBOL — such as JBEI-ICE — they stop short of exposing SBOL as a queryable RDF knowledge graph, undermining its machine-tractability. The SBOL Stack solves this problem by facilitating the storage of SBOL in an RDF triplestore, but is only a database for programmatic access and does not provide a user interface.

SynBioHub is an attempt to provide the best of both words: a graphical user interface constructed atop an underlying RDF triplestore. While users of SynBioHub do not need to understand SBOL or RDF, the knowledge that they submit is automatically compatible with other SBOL-compliant tooling, and tractable as a graph using SPARQL queries. It can be argued that this is exactly how SBOL should be used: to standardize knowledge representation behind the scenes, while remaining invisible to users that are only concerned with using the software.

The research goal of this chapter was to investigate how SBOL can be used as a standard to create a user-facing platform for sharing designs. SynBioHub, the realisation of this goal, is now a significant part of synthetic biology data infrastructure, with multiple successful deployments worldwide:

- The reference instance at synbiohub.org[2] hosts the complete iGEM Registry in SBOL format, and is linked to from the official Registry page for each part
- SynBioHub is one of the tools incorporated by the Synergistic Discovery and Design Environment (SD2E) project [10], serving the DARPA Synergistic Discovery and Design (SD2) program [130]
- The Living Computing Project, a synthetic biology project recently awarded \$10M by the National Science Foundation [131], uses SynBioHub [9]
- The SevaHub repository [132] uses SynBioHub to provide access to the Standard European Vector Architecture (SEVA) [133] plasmids in SBOL format
- The Java client for the SynBioHub programmatic API, developed as part of this work, has now been integrated into libSBOLj, allowing any user of libSBOLj to use SynBioHub without any additional dependencies
- The SBOLDesigner [89] tool for genetic circuit CAD integrates directly with the SynBioHub programmatic API to allow users to retrieve parts to incorporate into designs.

SynBioHub was the subject of a recent review by Urquiza-García et al., which compared SynBioHub and JBEI-ICE as management platforms for synthetic biology

[2]`https://synbiohub.org/`

designs [129]. The review concluded that SynBioHub and JBEI-ICE are both complementary solutions with "synergistic potential". Advantages cited of SynBioHub included its knowledge graph based representation of metadata, the ability to represent abstract designs, and versioning of parts. The main disadvantage identified was that SBOL is difficult to author, asserting that "SynBioHub must allow full online editing and not depend on external SBOL editors". It is possible that the Web-based editor for SBOL described in chapter 7 could be integrated into SynBioHub to satisfy this need.

6.6.1 The FAIRdom principles

The issue of unpublished data is widespread in the life sciences. A 2011 Peccoud et al. letter to Nature biotechnology [8] complained that missing sequence information in papers was problematic for reproducibility and reuse in synthetic biology, and served as the initial motivation for the development of SBOL. More recently, the systems biology community has pioneered the concept of "FAIRdom" as an approach to enable semantic interoperability between data, operating procedures, and models [134]. FAIRdom has since become a worldwide initiative across many different scientific domains, and is endorsed by the intergovernmental data integration effort ELIXIR [135]. FAIRdom consists of four principles: that data should be **F**indable, **A**ccessible, **I**nteroperable, and **R**eusable. These principles are clearly aligned with the complaints of Peccoud et al., and serve as a useful set of guidelines for what a data repository should strive to enable.

Findability and Accessibility

SynBioHub directly addresses the issues of findability and accessibility by storing data about biological parts in a highly machine-tractable RDF triplestore with powerful graph querying capabilities.

The Web of Registries support also enables SynBioHub to participate in a wider ecosystem of part repositories. SynBioHub instances can act as a gateway to other repositories of knowledge, therefore increasing findability and accessibility not only of parts contained in their own instance, but also parts stored elsewhere.

Interoperability

Unlike other repositories which use their own bespoke data representations, all of the data stored in a SynBioHub instance is represented using SBOL — an open community standard. Users of SynBioHub can have confidence that their parts will be interoperable with any other parts represented in SBOL, regardless of where or how they are stored. There is no "lock-in" to a SynBioHub data model; parts can easily be migrated to any other repository with the ability to store SBOL.

Reusability

Parts stored in SynBioHub are inherently reusable, as the SBOL standard both supports and encourages — through composition and provenance – the reuse of existing parts.

Furthermore, the repository itself is also re-usable: as an open source project, it can be customised to meet the needs of individual projects.

6.6.2 Dereferenceable URIs

As discussed in section 4.7, there are two approaches to capturing a design in SBOL: the monolithic approach of copying information about constituent parts into the same place, or the modular approach of only including immediately relevant design information, providing references to any composed parts rather than including their details. Integrating SBOL with other RDF resources using data integration technology as demonstrated in chapter 4 solves one of the practical problems with the modular approach: that SBOL designs often need to link out to non-SBOL resources such as UniProt proteins, which cannot be easily de-referenced by SBOL tooling.

The other logistical challenge for modular design using SBOL is that the SBOL objects themselves often cannot be de-referenced. The examples in the SBOL specification almost exclusively use URIs which look like URLs but are actually invented and unresolveable, such as `https://sbolstandard.org/example/toggleswitch`. Even with the querying power of RDF and Linked Data, it is difficult to find out more information about a resource without an indication of where the information can be found.

SynBioHub provides a solution to this problem by re-mapping the URIs of uploaded SBOL to resolveable URLs. If an SBOL design references a part hosted on SynBioHub by URI, it is guaranteed that that URI can also be used as an HTTP URL to retrieve the SBOL data for the part. As a direct result of SynBioHub providing this facility, libSBOLj has recently added support to use SBOL URIs as URLs in order to download missing parts of an "incomplete" SBOL document, which interestingly mirrors dependency resolution in software: an SBOL design can now reference an existing part using its SynBioHub URL instead of respecifying it.

6.6.3 Future work

Distributed design knowledge

The application of LDF to the Web of Registries enables SBOL design knowledge spread across multiple instances of SynBioHub to be queried as a single knowledge graph of knowledge. Figure 6.8 showed an example of how this federated approach could be used to specify a design in one repository, while the definitions of its constituent components remain in another.

There are many possibilities for distributing design knowledge in this manner. For example, one repository could define a part, and another repository could contain characterisation data for the same part URI. Any part of the SBOL data model can potentially be split across multiple repositories. For example, there is no reason that a separate repository would not be able to provide additional `SequenceAnnotation` objects that refer to a `ComponentDefinition` located elsewhere. Exploring the implications of SBOL moving away from a document-based approach and toward a federated knowledge graph, and what potential effect such a development would have on the design process, would be an interesting area for future research.

Sharing options

In SynBioHub, parts can currently be either private to a user (i.e. contained in the user graph), or public (i.e. contained in the public graph). This sharing model is simple, but also not necessarily well-suited to how the design process works in reality. A design often belongs not to a single user, but to a *group* of users; e.g., a group representing a lab.

While it is possible to work around this issue in SynBioHub by sharing a single user account among a group of users, it is not ideal; it would be necessary to share the login credentials, and there would be no way to attribute changes to specific users. The Urquiza-García et al. review of SynBioHub and JBEI-ICE cites as an advantage of JBEI-ICE that "multiple lab members can update the records for individual parts" [129].

A future version of SynBioHub could implement the concept of groups, where a group has its own graph and multiple users can have membership. Permission for each user in the group (e.g. read access, write access, and the ability to create new parts) would be configurable.

Profile pages

SynBioHub assigns URIs to users, which are used e.g. to populate the `synbiohub:ownedBy` property, and to provide a URI prefix for parts located in the private graph. For example, the SynBioHub user `james` would be assigned the URI `https://synbiohub.org/user/james`, and parts in the private graph of that user would be assigned URIs such as `https://synbiohub.org/user/james/mydesign`.

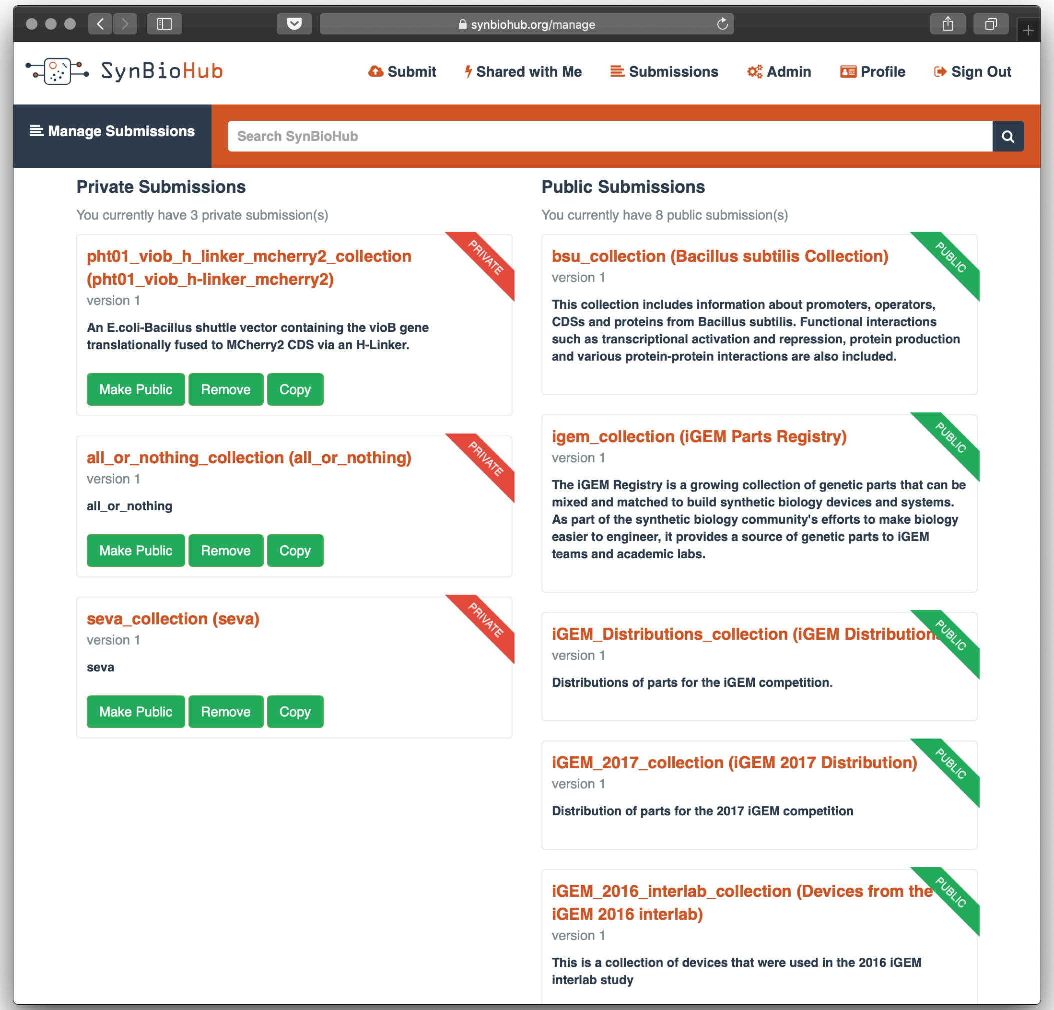
In code repositories such as GitHub, there is typically a user profile page which shows information about a user and a list of their contributions. SynBioHub could implement this functionality by making user URIs dereferenceable URLs in the same manner as part URIs. The same could be applied to the aforementioned potential feature of groups, which would allow a lab to set up a SynBioHub group to provide a list of their members and published designs — mirroring the GitHub concept of organizations.

While not directly related to improving the machine-tractability of design knowledge, features such as profile pages could help to make SynBioHub even more of a "hub" for the synthetic biology community, thereby encouraging its use and consequently the availability of design information in SBOL.

Conclusion

This chapter concerned the development of SynBioHub, an open-source Web-based repository for the sharing and dissemination of synthetic biology designs. Unlike existing repositories, SynBioHub is built atop an RDF triplestore using the SBOL data model, meaning that data uploaded to SynBioHub is highly machine-tractable, represented using an open standard with a detailed specification. The SynBioHub user interface functions as an abstraction layer over this underlying RDF knowledge representation, enabling users to publish design knowledge in a format amenable to data integration without additional effort.

SynBioHub contributes toward both of the overarching aims of this work. By providing a database and user interface for SBOL, it enables any existing parts converted into SBOL (e.g. the iGEM registry) to be accessed in a standardised and computationally tractable manner — thus contributing to the short-term aim of improving access to existing knowledge. Going forward, it also provides a platform for future parts to be published using SBOL, which means that future designers and design automation software can take advantage of a standardised repository of parts.



Private Submissions

You currently have 3 private submission(s)

pht01_viob_h_linker_mcherry2_collection (pht01_viob_h-linker_mcherry2)
version 1
An E.coli-Bacillus shuttle vector containing the vioB gene translationally fused to MCherry2 CDS via an H-Linker.

all_or_nothing_collection (all_or_nothing)
version 1
all_or_nothing

seva_collection (seva)
version 1
seva

Public Submissions

You currently have 8 public submission(s)

bsu_collection (Bacillus subtilis Collection)
version 1
This collection includes information about promoters, operators, CDSs and proteins from Bacillus subtilis. Functional interactions such as transcriptional activation and repression, protein production and various protein-protein interactions are also included.

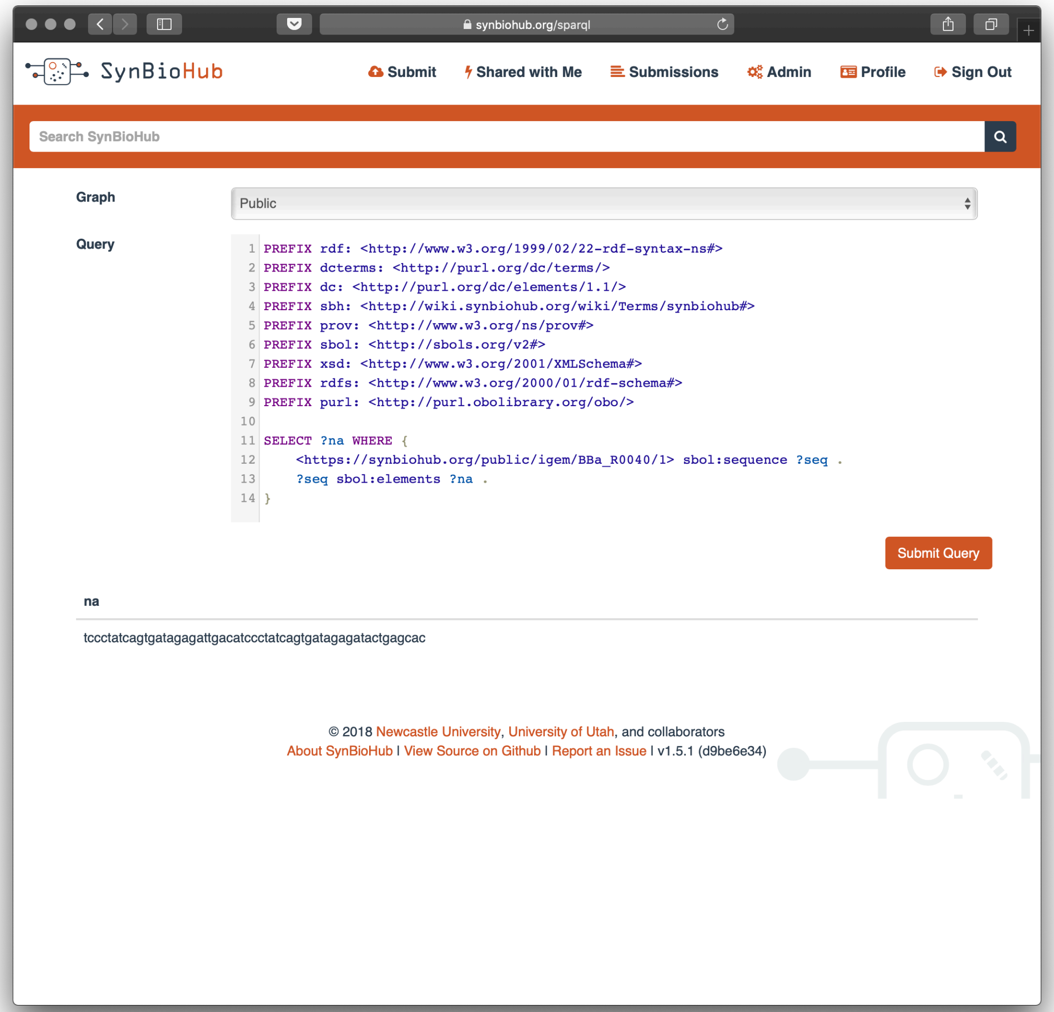
igem_collection (iGEM Parts Registry)
version 1
The iGEM Registry is a growing collection of genetic parts that can be mixed and matched to build synthetic biology devices and systems. As part of the synthetic biology community's efforts to make biology easier to engineer, it provides a source of genetic parts to iGEM teams and academic labs.

iGEM_Distributions_collection (iGEM Distribution
version 1
Distributions of parts for the iGEM competition.

iGEM_2017_collection (iGEM 2017 Distribution)
version 1
Distribution of parts for the 2017 iGEM competition

iGEM_2016_interlab_collection (Devices from the iGEM 2016 interlab)
version 1
This is a collection of devices that were used in the 2016 iGEM interlab study

Figure 6.2: *The Submissions view in SynBioHub provides an overview of uploaded collections from both the private graph, and from the public graph where the public collection is owned by the current user.*



Graph: Public

Query

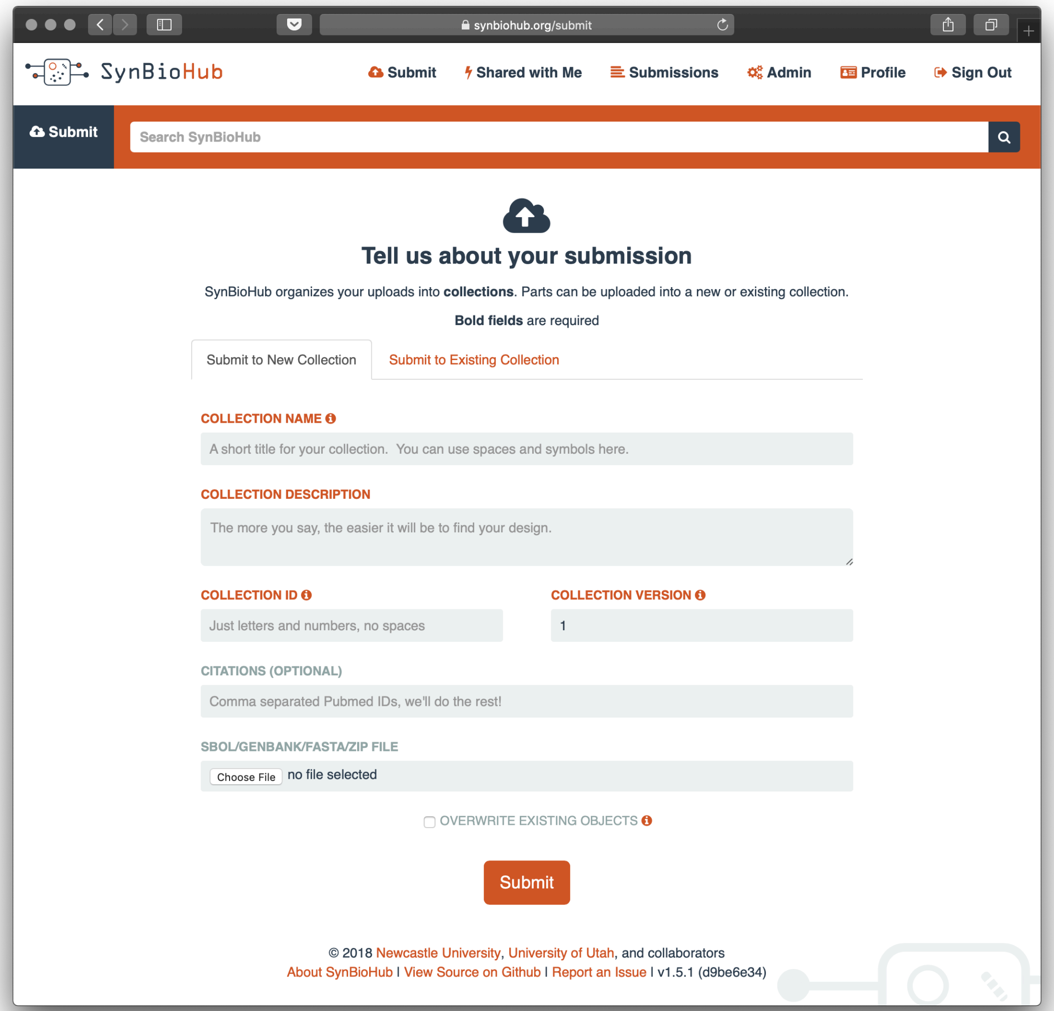
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX sbh: <http://wiki.synbiohub.org/wiki/Terms/synbiohub#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX sbol: <http://sbols.org/v2#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX purl: <http://purl.obolibrary.org/obo/>

SELECT ?na WHERE {
    <https://synbiohub.org/public/igem/BBa_R0040/1> sbol:sequence ?seq .
    ?seq sbol:elements ?na .
}
```

na
tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcac

© 2018 Newcastle University, University of Utah, and collaborators
About SynBioHub | View Source on Github | Report an Issue | v1.5.1 (d9be6e34)

Figure 6.3: *In addition to free-text search, SynBioHub also supports SPARQL queries. Queries are pre-processed by SynBioHub before being sent to the triplestore, in order to verify that they do not contain any potentially malicious constructs such as the GRAPH keyword (which would allow access to private user graphs) or INSERT and DELETE.*



Tell us about your submission

SynBioHub organizes your uploads into **collections**. Parts can be uploaded into a new or existing collection.

Bold fields are required

Submit to New Collection | Submit to Existing Collection

COLLECTION NAME
A short title for your collection. You can use spaces and symbols here.

COLLECTION DESCRIPTION
The more you say, the easier it will be to find your design.

COLLECTION ID
Just letters and numbers, no spaces

COLLECTION VERSION
1

CITATIONS (OPTIONAL)
Comma separated Pubmed IDs, we'll do the rest!

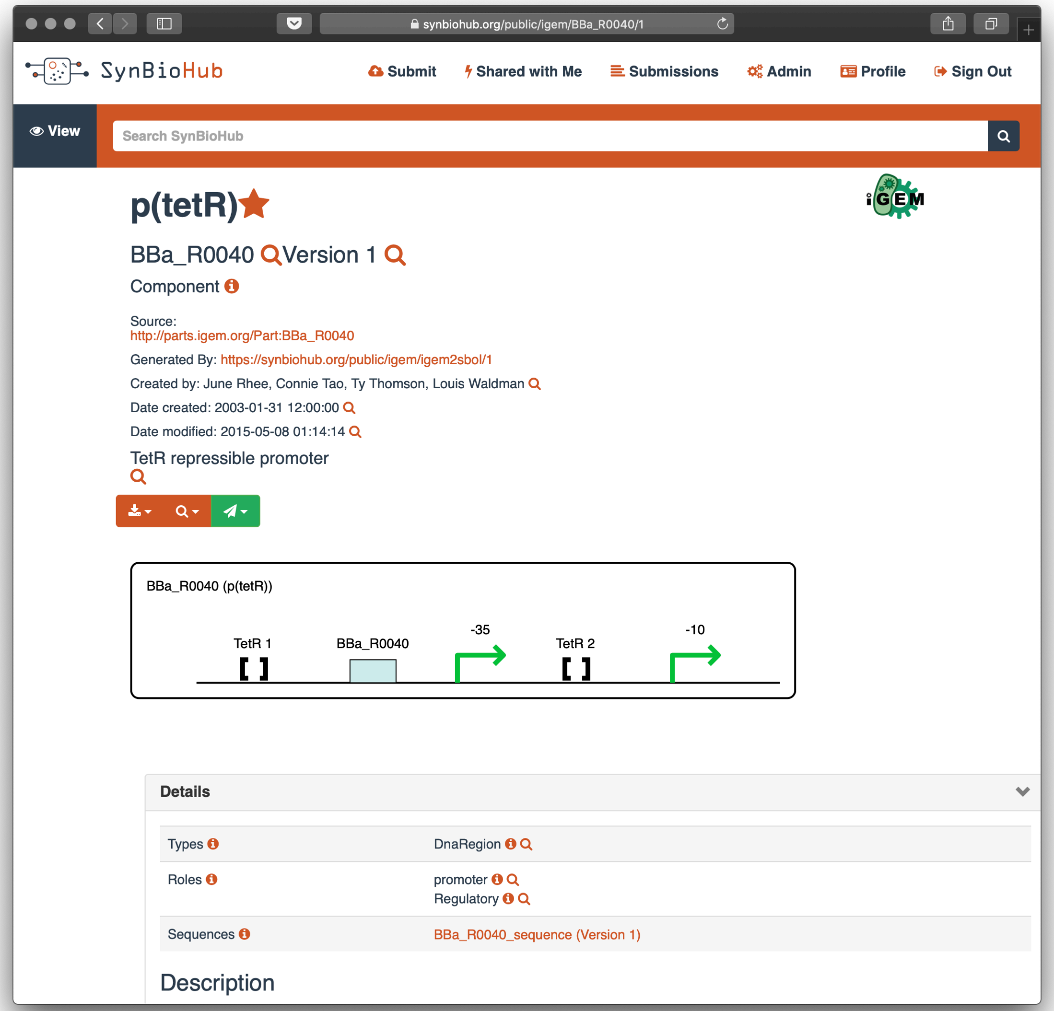
SBOL/GENBANK/FASTA/ZIP FILE
Choose File no file selected

OVERWRITE EXISTING OBJECTS

Submit

© 2018 Newcastle University, University of Utah, and collaborators
About SynBioHub | View Source on Github | Report an Issue | v1.5.1 (d9be6e34)

Figure 6.4: *The Submit page of SynBioHub. Submission in SynBioHub both creates an SBOL Collection object, and optionally adds data to the collection from a file in SBOL, GenBank, or FASTA format.*


synbiohub.org/public/igem/BBa_R0040/1

SynBioHub Submit Shared with Me Submissions Admin Profile Sign Out

View Search SynBioHub

p(tetR)

BBa_R0040 Version 1

Component

Source:
http://parts.igem.org/Part:BBa_R0040

Generated By: https://synbiohub.org/public/igem/igem2sbol/1

Created by: June Rhee, Connie Tao, Ty Thomson, Louis Waldman

Date created: 2003-01-31 12:00:00

Date modified: 2015-05-08 01:14:14

TetR repressible promoter

Details

Types	DnaRegion
Roles	promoter Regulatory
Sequences	BBa_R0040_sequence (Version 1)

Description

Figure 6.5: *A part from the iGEM-SBOL conversion described in chapter 5 displayed in SynBioHub, with its VisBOL visualisation.*

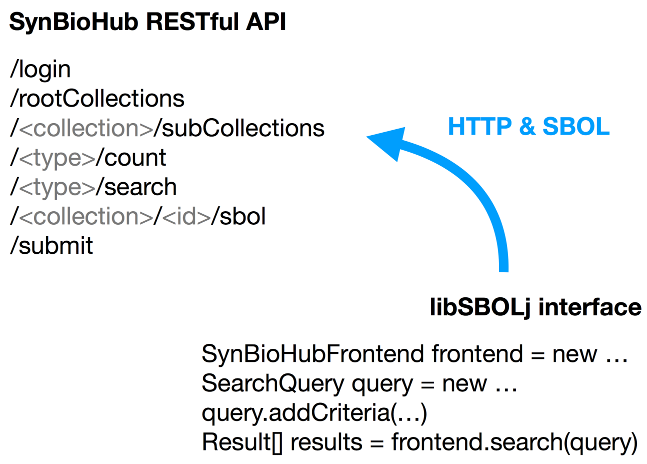


Figure 6.6: *An example of some of the functionality that SynBioHub provides through its Application Programming Interface (API) and the relevant Java pseudo-code. libSBOLj communicates with SynBioHub using the RESTful API over HTTP, and SBOL as the data exchange format.*

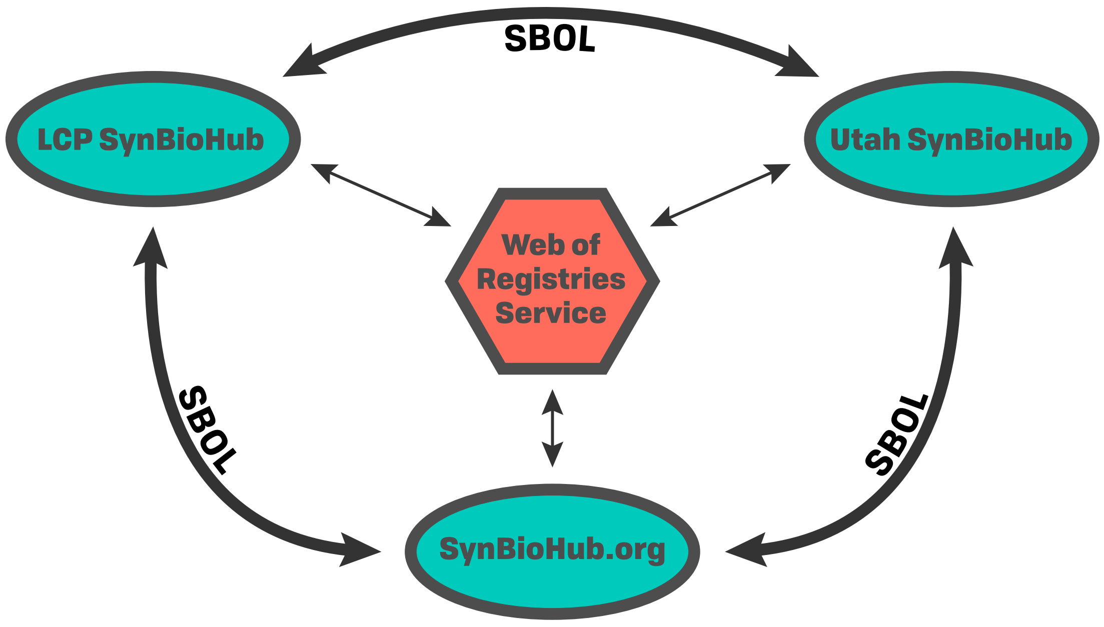


Figure 6.7: *The Web of Registries enables communication between SynBioHub instances. Any SynBioHub can access the Web of Registries to determine information about all registered SynBioHub instances. If a design references a part within another SynBioHub instance, the information about this part can be fetched in order to render this design information locally and provide links to the corresponding design information page for this part. Figure credit: Zach Zundel*

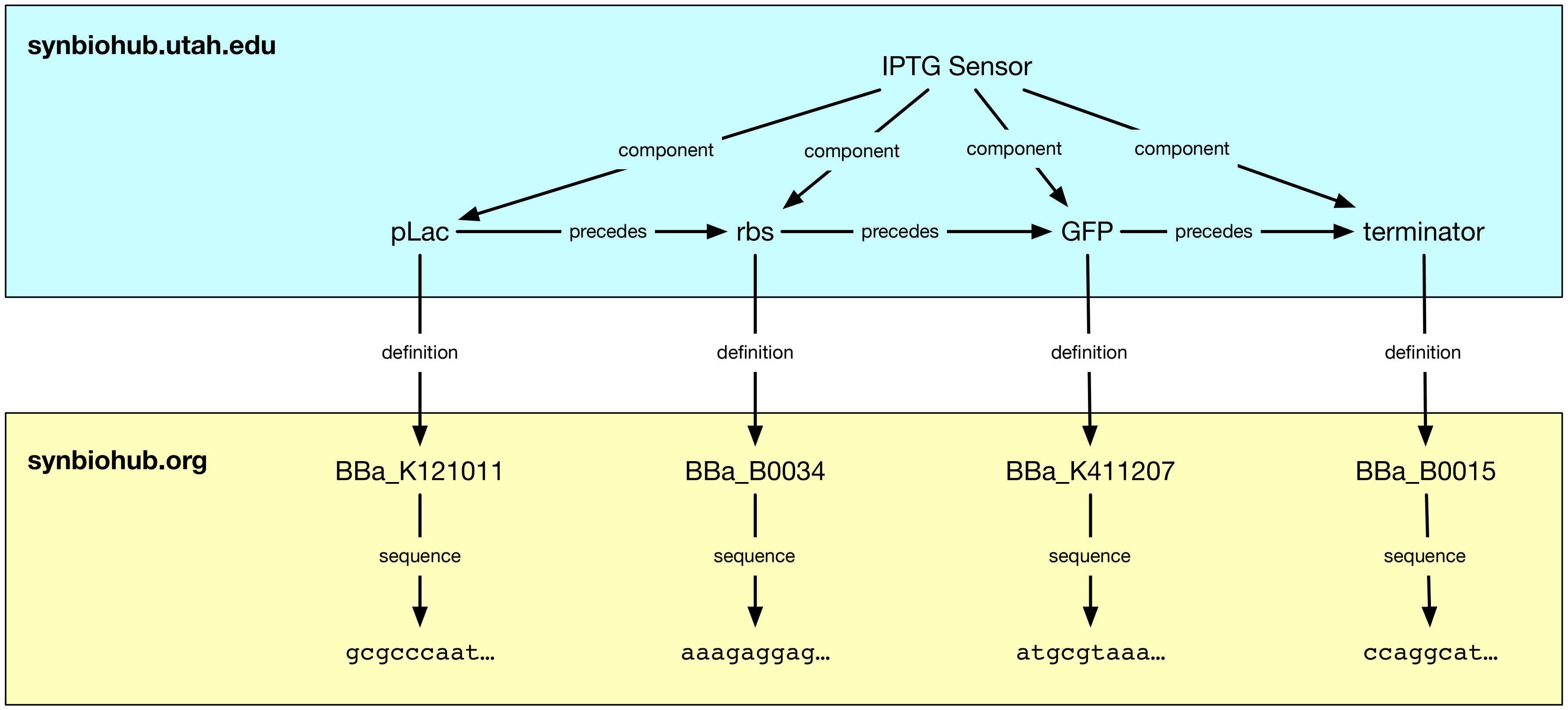


Figure 6.8: *An example SBOL design of an IPTG sensor, split across two different SynBioHub instances. The sensor component itself is submitted to SynBioHub Utah, but the definition of its constituent parts are located in the synbiohub.org repository. Despite the parts existing in separate locations, the Web of Registries combined with federated LDF SPARQL queries allows the design to be queried as a unified RDF graph.*

```
PREFIX sbol2: <http://sbols.org/v2#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX ncbi: <http://www.ncbi.nlm.nih.gov#>

SELECT ?Collection ?name ?description ?displayId ?version WHERE {
    ?Collection a sbol2:Collection .
    FILTER NOT EXISTS { ?otherCollection sbol2:member ?Collection }
    OPTIONAL { ?Collection dcterms:title ?name . }
    OPTIONAL { ?Collection sbol2:displayId ?displayId . }
    OPTIONAL { ?Collection dcterms:description ?description . }
    OPTIONAL { ?Collection sbol2:version ?version }
}
```

Figure 6.9: *SPARQL query to select all of the "root" collections; that is, all of the collections that are not contained by another collection.*

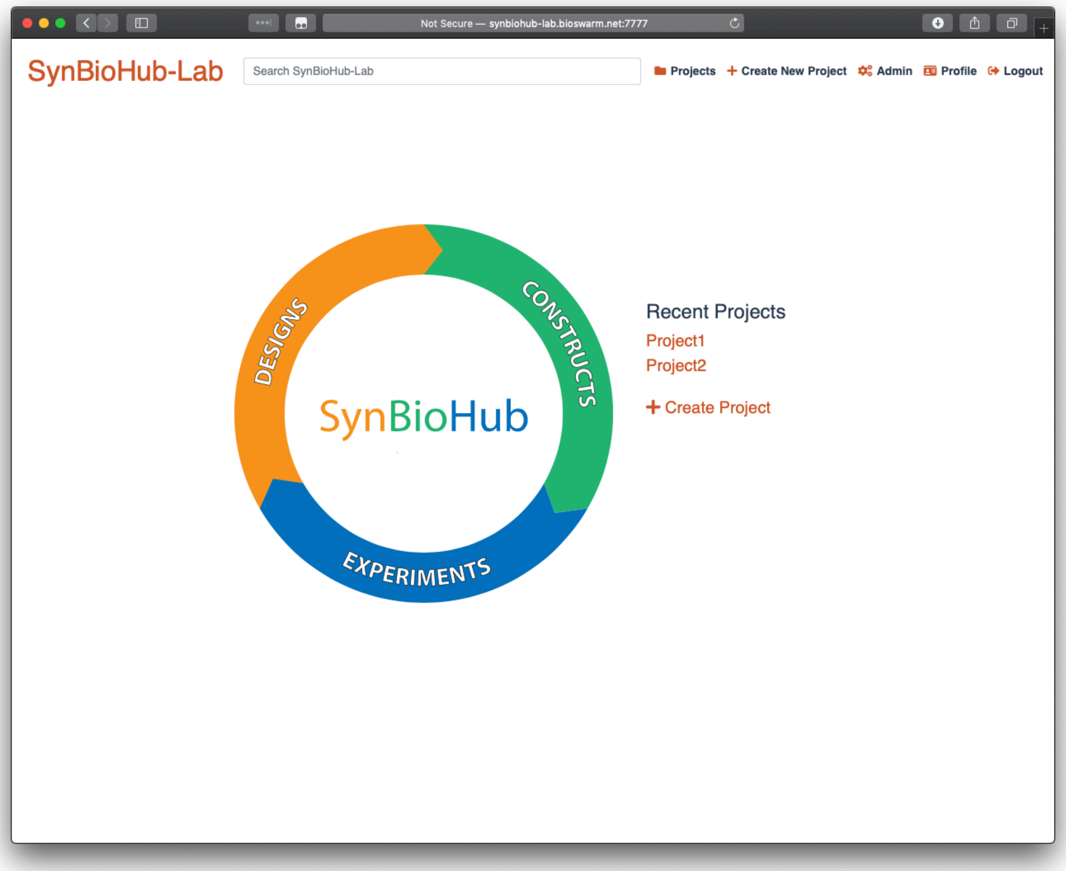
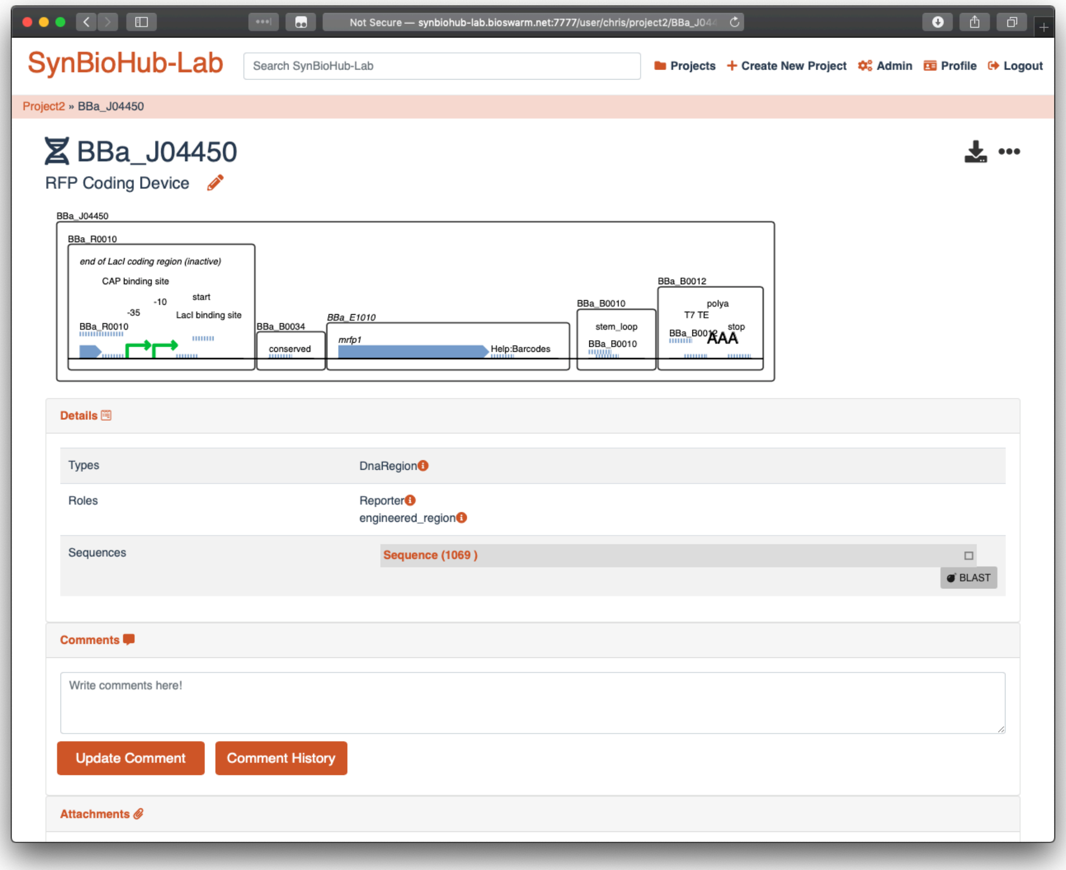


Figure 6.10: *The homepage of SynBioHub Lab, a comprehensive re-design of the SynBioHub repository with an emphasis on capturing whole-lifecycle knowledge for synthetic biology.*


Not Secure — synbiohub-lab.bioswarm.net:7777/user/chris/project2/BBa_J04

SynBioHub-Lab

Search SynBioHub-Lab

Projects + Create New Project Admin Profile Logout

Project2 » BBa_J04450

BBa_J04450

RFP Coding Device

Details

Types	DnaRegion
Roles	Reporter engineered_region
Sequences	Sequence (1069)

BLAST

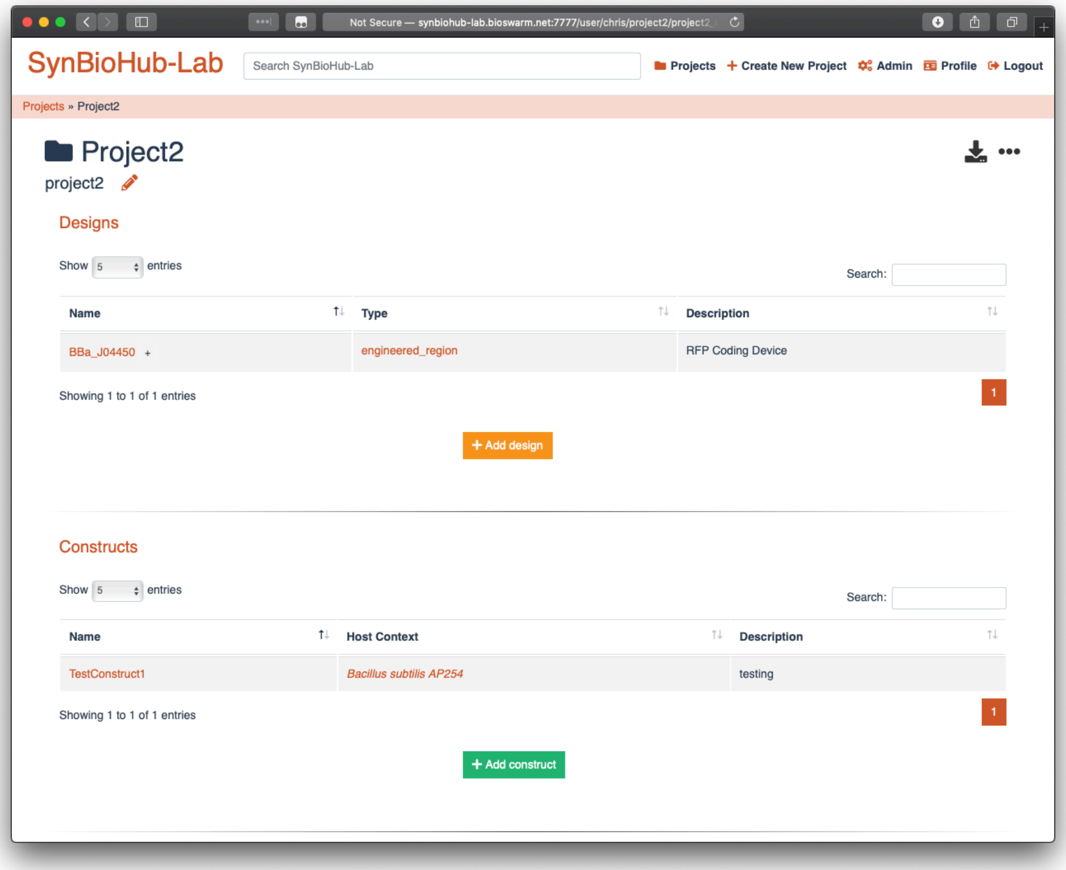
Comments

Write comments here!

Update Comment Comment History

Attachments

Figure 6.11: The part view in SynBioHub Lab, showing a design visualization using SynBioCAD in place of VisBOL. Unlike VisBOL, SynBioCAD renders the compositional hierarchy of the design.



Projects » Project2

Project2

project2

Designs

Show 5 entries

Search:

Name	Type	Description
BBa_J04450 +	engineered_region	RFP Coding Device

Showing 1 to 1 of 1 entries

1

Constructs

Show 5 entries

Search:

Name	Host Context	Description
TestConstruct1	*Bacillus subtilis AP254*	testing

Showing 1 to 1 of 1 entries

1

Figure 6.12: *The project view in SynBioHub Lab (formerly the view for collection members in SynBioHub) is split into three sections: designs, constructs, and experiments.*

7. SynBioCAD: a standards-enabled design tool for synthetic biology

7.1 Introduction

One of the two aims of this work is to improve access to knowledge about biological parts *in the long-term*. Unless synthetic biologists can easily document their parts and devices in a manner which is standardised and computationally tractable, the issue of knowledge fragmentation will continue in perpetuity.

Recent versions of SBOL provide a comprehensive data standard including concepts such as composition, modularity, and functional relationships. However, these concepts are largely ignored by tooling with SBOL support. For example, while interactions were added to SBOL in 2015, to date there exists no user-facing tool that can be used to *create* SBOL data incorporating interactions. The examples of interactions in the SBOL2 specification were created by writing code to manually interact with an SBOL library, a workflow which is prohibitively inaccessible to non-programmers.

This situation is representative of a wider problem in the synthetic biology community. Synthetic biology is an interdisciplinary pursuit, and necessarily includes researchers from many different domains. While it is not typically expected that computer scientists perform experiments in the wet lab, it is often necessary for experimentalists to navigate data models and terminology developed by computer scientists. This is counter-productive: if the ultimate goal is to get as much information as possible from the designers into a computational standard such as SBOL, it is essential that process is as accessible as possible and does not require computer science domain knowledge.

Therefore, the research goal of the final chapter of this thesis is to investigate how SBOL can be abstracted as a visual, user-facing tool. This chapter culminates in the development of *SynBioCAD*, an open-source CAD tool for synthetic biology built on the SBOL standard[1,2]. SynBioCAD provides an intuitive graphical interface that can be used to design both from a "top-down" (parts first) and a 'bottom-up" (sequence first) perspective. SynBioCAD acts as both the entry point to the tooling provided in this thesis — in that user-created designs can be augmented with additional information using Enrichment and published using SynBioHub — and also as the final destination, as it can be used to visualise and explore SBOL obtained from any source, including the

[1]`https://github.com/SynBioCAD`
[2]`https://biocad.io`

iGEM conversion and Enrichment framework described in chapter 5 and the SynBioHub repository described in chapter 6.

A note on SBOL versions

As discussed in section 2.5, there is a significant disparity between what the SBOL2 data model can represent and what SBOL Visual tooling supports. It is possible that one of the reasons for this disparity that SBOL2 essentially consists of two parallel data models: one consisting of components, which is almost entirely supported by SBOL Visual tooling; and one consisting of modules, which is almost entirely unsupported. While one of the reasons for this lack of support may simply be that the module functionality was added more recently, there is a more fundamental problem with SBOL2 visualization:

1. As described in section 3.5, with the introduction of modules, the SBOL data model added additional "functional" hierarchy which is separate from the hierarchy of the design.
2. In a visualisation, the "functional" hierarchy is often irrelevant as it does not help to communicate the design. For example, if in order to add interactions to a component it was first necessary to wrap the component in a module and create `MapsTo` relations, the module-component hierarchy and `MapsTo` relation are uninteresting to users and should not be displayed.
3. Visualisation software is therefore forced to provide a visual *abstraction* over the SBOL data rather than simply rendering one visual entity for each SBOL entity, as is possible with the component data model.

Visualisation tools can tackle this problem by first reducing SBOL to a simpler data model where components and modules share the same hierarchy and then producing the visualisation (Fig. 7.1). However, this means that each visualisation tool is forced to define an ad-hoc data model, which undermines the motivation of SBOL to create a standardised representation. The problem can be framed as "If this is how we draw it, this is how we think about it, and if this is how we think about it, why is it not reflected in the data model?"; or, in other words, should the disparity between the structure of the data and the desired structure of its visual depiction be interpreted as an indication that the data model should be changed?

Consequently, the work described in this chapter is built on a hypothetical "SBOL3" in which the the following SEPs defined in section 3.5 have been accepted:

- SEP 010 – simplify description of sequence features and sub-parts
- SEP 025 – Merge ComponentDefinition and ModuleDefinition
- SEP 015 – Simplification of SBOL class names

Conversion to and from the hypothetical data model is implemented in sbolgraph, and therefore SBOL2 is fully supported by SynBioCAD despite its internal use of "SBOL3". Assuming the SEPs are accepted, SynBioCAD will already have complete support for SBOL3 upon release.

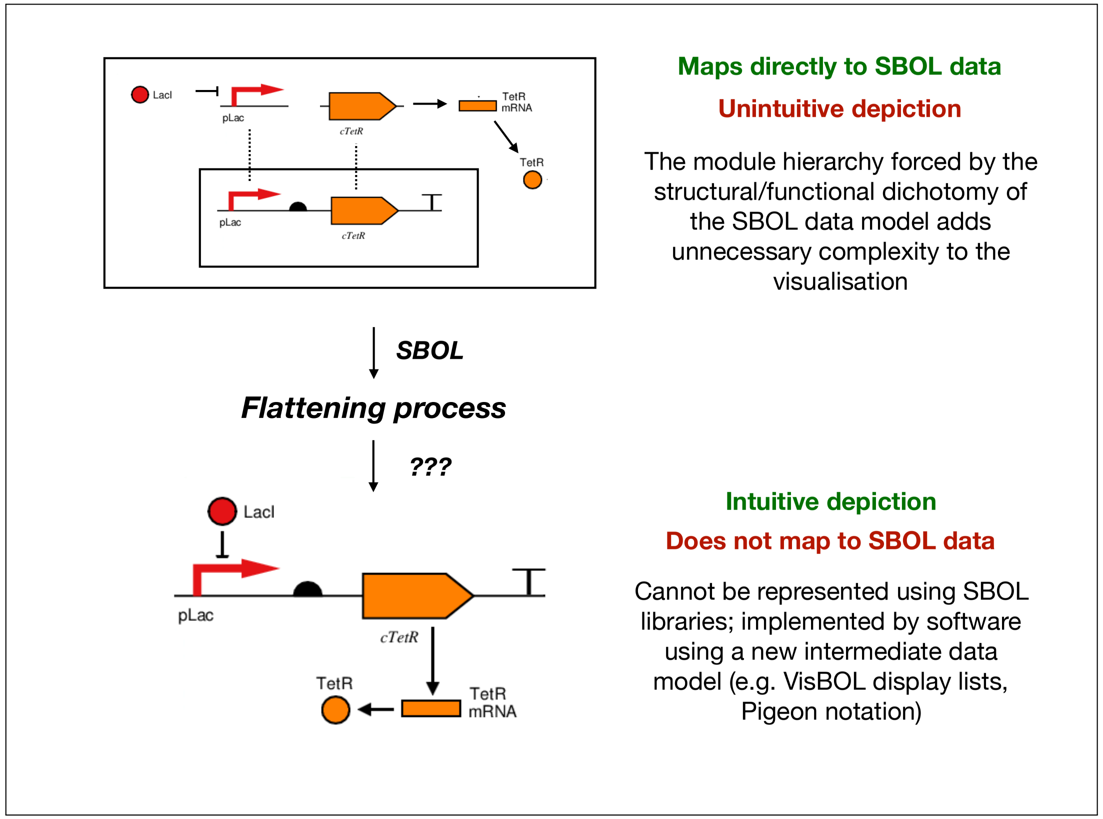

LacI
pLac
cTetR
TetR mRNA
TetR
pLac
cTetR
Maps directly to SBOL data
Unintuitive depiction
The module hierarchy forced by the structural/functional dichotomy of the SBOL data model adds unnecessary complexity to the visualisation
SBOL
Flattening process
???
LacI
pLac
cTetR
TetR
TetR mRNA
Intuitive depiction
Does not map to SBOL data
Cannot be represented using SBOL libraries; implemented by software using a new intermediate data model (e.g. VisBOL display lists, Pigeon notation)

Figure 7.1: *Implementing a visualization tool for SBOL is difficult because it is not always possible to map SBOL data directly to visual concepts. SBOL forces hierarchy on designs due to the separation of its structural aspects (e.g.* `ComponentDefinition`*) and its functional aspects (e.g.* `Interaction`*, which can only exist in the context of* `ModuleDefinition`*), and rendering this imposed hierarchy would result in an unintuitive visualization. Visualisation software is consequently forced to first map SBOL to a flattened representation, which is illegal in the current SBOL2 data model and therefore cannot be represented using SBOL libraries, resulting in the specification of ad-hoc "flattened" data models such as VisBOL display lists and Pigeon notation.*

7.2 Layouts for genetic circuits

The first challenge in developing a visual tool for authoring SBOL is to work out how to bridge the gap between SBOL and SBOL Visual. Specifying a *layout* in order to render a design has a different set of requirements to specifying the design itself. While SBOL provides the facilities necessary to capture essential design information such as sequences, function, and composition, a layout requires visual attributes such as the location, size, and even color of design components.

As described in section 2.5, there have been at least two attempts to describe such information in the context of SBOL. Pigeoncad [136] defined "Pigeon notation", a simple text-based format in which each line either lists a glyph type, name, and color; or the name of two glyphs and an arc type (either positive or negative regulation). VisBOL [94] defined a JSON display list format, which has a similar scope to Pigeon notation but also supports the specification of multiple distinct backbones.

Both of these attempts are very high-level, in that they specify the list of glyphs to display but without specific coordinates, and then rely on the rendering software to determine size and positioning. Essentially, they both function as an abstraction over the SBOL data model rather than a specification of layout.

With the proposed simplified data model and improved library support developed in part 3, such an abstraction over SBOL is no longer necessary. Instead, it is possible to use SBOL as the list of parts to display, and then a layout with a specific visual configuration can be derived directly from the SBOL. This approach was used in this work in the SynBioCAD Layout data model. SynBioCAD Layouts are not derivative of the SBOL like the aforementioned display list formats, but instead complementary; both the SBOL and the layout would be required to produce a visualization, with a clear separation of concerns between the part knowledge captured by the SBOL data model and the visual configuration described by the layout.

7.2.1 Specifying layouts

It was initially tempting to add layout information directly to the SBOL data model; i.e., by annotating objects such as `ComponentDefinition` with additional RDF triples to describe their position and size. However, this approach was not viable because the SBOL data model is fundamentally built on the principle of composition, whereby each component can reference other components as sub-components. If the same component is used multiple times in a design and that component also has sub-components, the sub-components will appear multiple times in the visual rendering but will each have only a single corresponding SBOL entity.

It is therefore necessary to define a new distinct class to represent the *depiction* of an SBOL object. Unlike SBOL entities such as `ComponentDefinition` which capture information such as the sequence of a part, depictions concern the position and size of a rendering within a layout. The base SynBioCAD Layout `Depiction` class has the following properties:

- `offset` (x, y): The offset in layout units from the parent `Depiction`, or the absolute offset if no parent.

- `size` (x, y): The size of the depiction in layout units
- `opacity`: whether to show the details inside the depiction ("whitebox") or not ("blackbox")

Several different of sub-classes `Depiction` are also defined:

- `LocationableDepiction`
- `BackboneDepiction`
- `LabelDepiction`

Two sub-classes are defined for `LocationableDepiction`:

- `ComponentDepiction`
- `FeatureLocationDepiction`

The majority of the existing tools for SBOL visualization have the visual concept of a *backbone*. Though not explicitly described by the SBOL Visual specification, a backbone is typically a horizontal line upon which glyphs can be placed representing one or two strands of DNA. The undocumented convention is that glyphs placed atop the backbone represent features on the forward strand, and glyphs placed below the backbone represent features on the reverse complementary strand. The SBOL data model, however, does not have such a concept. To address this, the `BackboneDepiction` class was added. A `BackboneDepiction` can optionally be used as a child of a `ComponentDepiction` to visually group the "sequence-bound" children (either sub-components or sequence features) of the component. A child is considered sequence-bound if it has one or more locations that specify a sequence location (e.g. using a `Range`), or if it is the subject or object of a `SequenceConstraint`.

7.2.2 Layout to SBOL references

Even with the simplified data model, it is not possible to accomplish a true 1:1 mapping where each SBOL URI maps to exactly one visual depiction. The reason for this is the same reason that layout information cannot simply be added directly to the SBOL data: if the same component is used multiple times in a design and that component also has sub-components, the sub-components will appear multiple times in the visual rendering but will each have only a single corresponding URI (Fig. 7.3).

A workaround for this issue used in the SynBioCAD Layout data model is to map depictions back to SBOL not by URI, but instead by URI "chains". Each depiction holds a URI chain containing its own URI, then the URI of the object that instantiated it, recursively (Fig. 7.4).

An important distinction to make is that the chain of instantiation is not the same as the chain of object ownership. For example, the chain of *instantiation* `C;C.1;B.1` in Fig. 7.4 corresponds to the component C using a sub-component `C.1` to instantiate the component B, which has a sub-component `B.1` to instantiate A, providing a unique

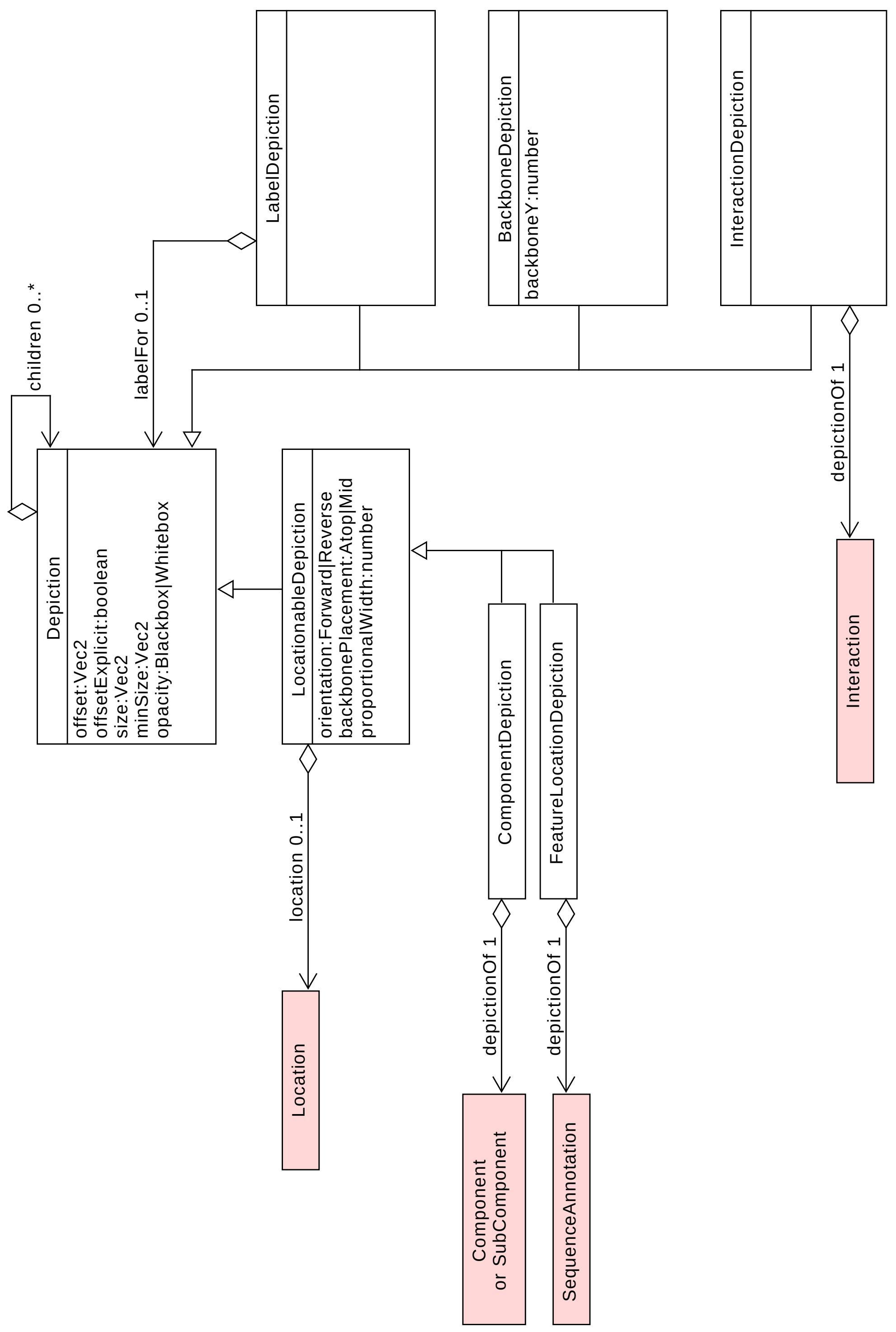


Figure 7.2: *A data model to specify layouts for the Synthetic Biology Open Language (SBOL). Visual information about SBOL entities is described using subclasses of the* `Depiction` *class.*

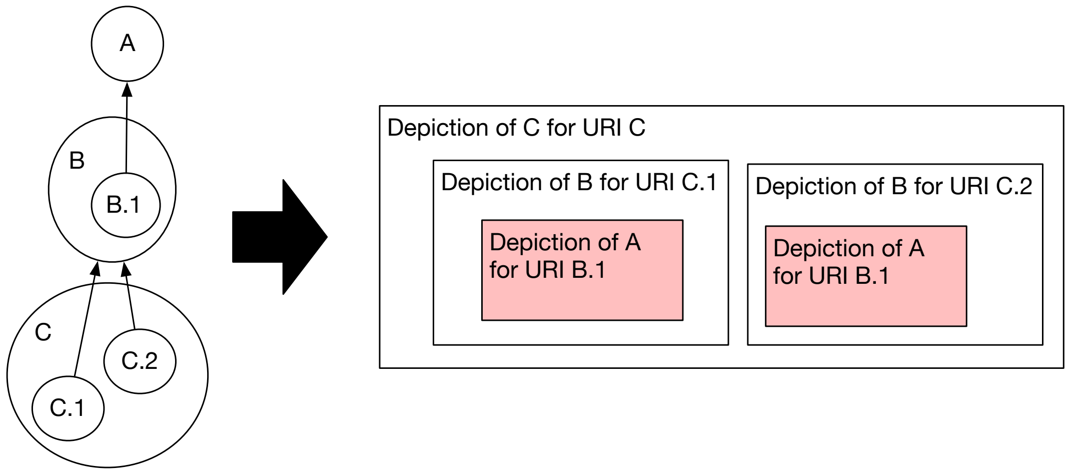

A
B
B.1
C
C.2
C.1
Depiction of C for URI C
Depiction of B for URI C.1
Depiction of A for URI B.1
Depiction of B for URI C.2
Depiction of A for URI B.1

Figure 7.3: *If the same component is used multiple times in a design and that component also has sub-components, the sub-components will appear multiple times in the visual rendering but will each have only a single corresponding URI, meaning that there is no longer a 1:1 mapping where one SBOL object maps to one depiction.*

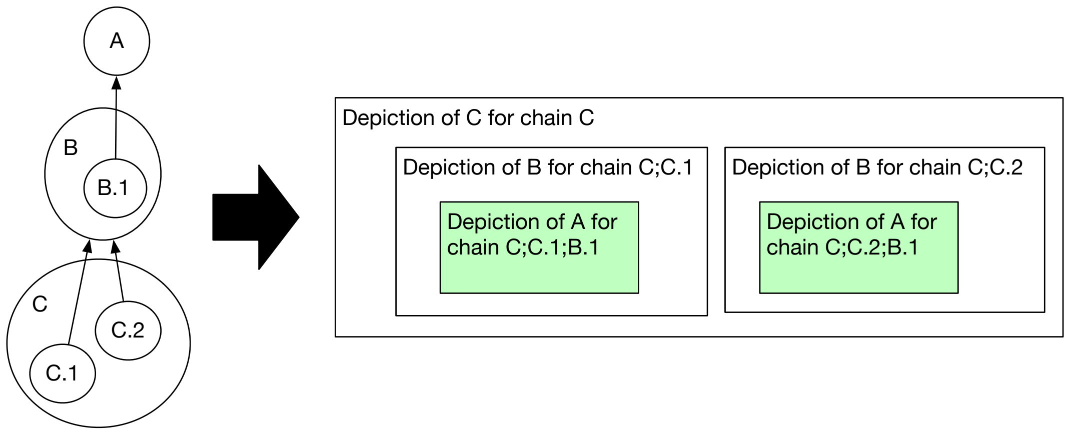

A
B
B.1
C
C.2
C.1
Depiction of C for chain C
Depiction of B for chain C;C.1
Depiction of A for chain C;C.1;B.1
Depiction of B for chain C;C.2
Depiction of A for chain C;C.2;B.1

Figure 7.4: *A workaround for the issue shown in Fig. 7.3 is to point back to SBOL from depictions using chains of URIs instead of a single URI. The chain of composition allows each depiction to umambiguously point to a particular instance of an SBOL object.*

identifier for one specific depiction of A. The chain of *object ownership* for the same object, however, would invariably be `B;B.1`, as `B.1` is a sub-component of B but the context of B being instantiated as a sub-component is not respected.

7.2.3 Creating layouts for SBOL data

The process of populating the SynBioCAD Layout data model using data from SBOL is mostly a straightforward case of creating the correct corresponding `Depiction` subclass for each SBOL object. One complication is the issue of avoiding duplicate depictions of the same object. Consider a design represented by a component with two sub-components. Intuitively, one attempting to visualise SBOL would simply draw each component along with its sub-components. However, if a component has already been drawn *as part of another component*, creating another depiction for it as a top-level would result in duplication (Fig. 7.5). This can be avoided by identifying the root components and working inwards:

1. Identify the set of "root" components R. These are `Component` objects are never instantiated as part of another `Component` using a `SubComponent`; i.e., components where the pattern `?subComponent sbol:definition ?component` has no matches.
2. For each of these root components $c \in R$, create a `ComponentDepiction` d where $depicts(d, c)$
3. For each sub-component c' of c, create a `ComponentDepiction` d' where $depicts(d', c')$ and $child(d', d)$
4. For each sequence annotation c' of c, create a `FeatureLocationDepiction` d' where $depicts(d', c')$ and $child(d', d)$
5. Repeat steps 3, 4, and 5 for each newly created `ComponentDepiction` and `FeatureLocationDepiction`

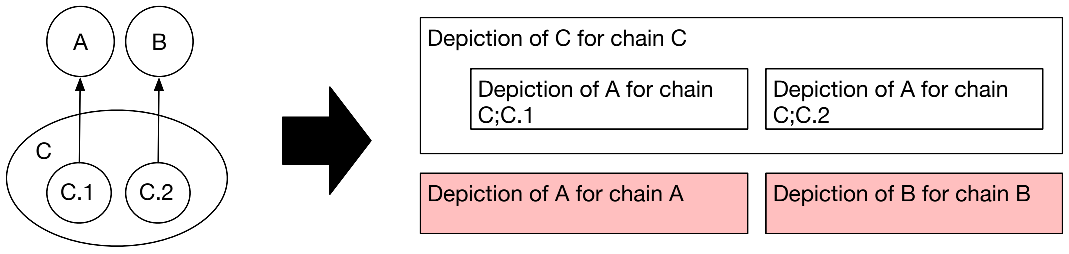

A
B
C
C.1
C.2
Depiction of C for chain C
Depiction of A for chain C;C.1
Depiction of A for chain C;C.2
Depiction of A for chain A
Depiction of B for chain B

Figure 7.5: *Creating a single Depiction for each SBOL object results in additional depictions being created at the root level for objects which were already included in compositional hierarchy.*

Once the layout has been generated, it is desirable to be able to update it if the SBOL it was generated from changes, rather than generating an entire new layout. This is particularly important if the layout contains specific positioning or styling information which would be lost if the layout was replaced. The synchronisation process can be performed as follows:

1. Assign a unique version number to represent this iteration of the synchronisation
2. For each component, start with a URI chain containing only the URI of the component. If the layout has a depiction corresponding to the chain, synchronise its attributes with the SBOL (e.g. orientation) and set its version number to the current version. Otherwise, create a new depiction.
3. For each sub-component of the component, extend the URI chain with the URI of the sub-component. Synchronise or create a depiction as above, then repeat this step for each sub-component of the definition of the sub-component.
4. Iterate through all of the depictions in the layout, removing any that do not match the current version number; as they were not visited, they are no longer present in the SBOL.

7.2.4 Configurating layouts

Once the layout has been generated, it is not yet ready to render. First, an offset and size must be assigned to all depictions. This process, termed herein as "configurating" the layout, requires the combination of a number of different strategies to accommodate requirements such as hierarchical composition and the layout of parts on backbones.

Hierarchical composition

The composition of hybrid parts from multiple smaller components is a key requirement for the top-down design of genetic circuits. This presents a challenge in the implementation of a visual rendering, in that the dimensions of each rendered part must accommodate the dimensions of its contained parts. Therefore, the dimensions of any part can not be established until the dimensions of all contained parts have been established.

In order to implement these rules programatically, the parts can be represented as a tree, where each node represents a part in the circuit, and edges represent composition. The tree repesentation enables depth to be used to layout in an order where the dimensions of leaves are known first before the more composed parts.

1. Perform a depth sort on the list of depictions. For sorting purposes, the depth of any given depiction d can be calculated as 1 greater than the maximum of the depth of its children $max(depth(children(d))+1$, or just 1 if there are no children.

2. Iterate through the list of depictions, which now begins with the lowest depth depictions (i.e. the leaves of the tree). Each depiction can calculate and store its size taking into account the size of its children, which will already have been calculated.

Backbone strategy

The purpose of backbone depictions is to convey locations within a sequence. SBOL has two mechanisms to express location information:

1. *Locations* allow the location of a region to be specified using numeric sequence offsets.

2. *Sequence constraints* allow the location of a region to be expressed as a function of the location of another region.

Locations are fairly straightforward to implement. The most common type of location in SBOL is `Range`, which specifies a start and an end offset. Constraints specify a subject, an object, and a restriction (e.g., `precedes`). The backbone layout strategy described here makes a "best effort" of supporting both, and does not any attempt to identify cases where the locations contradict the constraints or vice-versa. The strategy is as follows:

1. **Define a set of layers**. Initially, this set will be empty. Each set has an index, where negative indexes represent layers above the backbone and positive below; a height; and a set of occupied ranges, also initially empty.

2. **Set the initial length of the backbone.** The initial length is the length of the sequence multiplied by the scale factor.

3. **Place all explicitly positioned depictions.** These are depictions that have a specific offset assigned (i.e., `offsetExplicit` is true), and should not be subject to automatic layout.

4. **Place all depictions with fixed locations.** Any depictions with a `Range` are designated as fixed. Depictions already positioned due to explicit positioning are excluded.

5. **Place all depictions with sequence constraints that reference already placed depictions.** Perform this step recursively until all depictions are placed.

If there are any depictions left over, they must have no explicit position, fixed location, or constraints that refer to already placed depictions. The only other possibility is that they have constraints that refer only to each other (as is the case for much of the SBOL exported from SBOLDesigner). These remaining depictions can simply be sorted as a list with respect ot their constraints, and placed in order on the backbone.

Bin-packing strategy

The layout strategy used for a component with sub-components is based on a bin-packing algorithm, in an attempt to display all sub-components in the smallest possible space. As any sequence-bound sub-components are children of a `BackboneDepiction`, they are already configurated by the backbone strategy, and so the entire `BackboneDepiction` parent can be bin-packed as with any other child depiction. The strategy is as follows:

1. **Place the depictions into groups.** First, place depictions that are participants of the same interaction into the same group. Then place any remaining depictions each into their own group.

2. **Horizontally tile the children of each group.** A simple horizontal tiling strategy is used, i.e. for each child c_n, let $offset(c_n)$ equal $offset(c_{n-1}) + padding$

3. **Create layers above and below the group to allow for interaction arcs.** The height of the above layer is determined by the number of overlapping interactions to display above the components, and the height of the below layer by the number of overlapping interactions to display below.

4. **Use a bin packing algorithm to position the groups.** Jake Gordon's binary tree bin-packing implementation [137] is used, which allows packing into a growing space rather than a fixed space

5. **Route interactions within their layers.** The interactions populate the previously allocated layer space.

7.2.5 JSON representation

While SBOL data is represented as RDF because of its connection with ontologies and ease of querying, there would be little use for such functionality in the representation of layouts. Instead, SynBioCAD Layouts are represented using a lightweight JSON serialization (Fig. 7.6). The serialization has two top level properties: `size`, which contains the dimensions of the layout; and `depictions`, which contains a list of the depictions.

Each depiction has a `class` property to indicate which specific `Depiction` subclass it is instantiating, and the properties of the depiction as described in section 7.2.

7.3 Interactive genetic circuit visualizations for the Web

Rendering a layout is the process of converting it into an actual image for display. Previous SBOL Visual tooling has used a variety of visualization methods: Pigeoncad uses the TikZ LaTeX library; dnaplotlib uses matplotlib [138]; and VisBOL uses Scalable Vector Graphics (SVG) [139]. Of these approaches, SVG has a number of advantages. First, SVG is resolution-independent: instead of being constructed of pixels, an SVG image comprises a set of instructions to draw an image, and thus can be rendered at any size without loss of quality. Secondly, SVG is natively supported by all modern Web browsers. Rendering designs to SVG elements that can be incorporated into a Web page enables the development of interactive, Web-based applications that incorporate SBOL Visual.

7.3.1 Rendering layouts with Scalable Vector Graphics (SVG)

As the layout already contains offset and size information from the configuration process, producing an SVG visualization is fairly straightforward. The depictions are depth sorted as in 7.2.4, and then drawn starting with the highest depth so that the nested-most depictions are drawn last. SVG allows objects to be manipulated using translation matrices, which can be used to apply positioning, sizing, and rotation (in the case of reverse complement orientation). For a depiction d, the absolute offset $absOffset(d)$ can be calculated as $offset(parent(d)) + offset(d)$ in the case that the depiction has a parent, or simply $offset(d)$ otherwise. The translation matrix to render a depiction is then:

$$\begin{bmatrix} size(d)_x \cdot gx & 0 & \textit{absOffset}(d)_x \cdot gx \\ 0 & size(d)_y \cdot gy & \textit{absOffset}(d)_y \cdot gy \\ 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

In the case of rendering in reverse complement orientation, the depiction is rotated

180° about the center using a standard rotation matrix:

$$\begin{bmatrix} cos(180) & sin(180) & size(d)_x \cdot 2 \\ -sin(180) & cos(180) & size(d)_y \cdot 2 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

The specific glyphs, such as the glyph for a promoter, are rendered using functions derived from VisBOL [94]. These functions essentially work like a font does for text: they are supplied a bounding box and draw the glyph, so SynBioCAD does not require any knowledge of glyph geometry.

7.3.2 Interactivity

The most basic user interactivity — the ability to move and resize glyphs — can be implemented with the addition of two new properties to the layout data model:

- In order to enable resizing glyphs, a `minSize` property is added to `Depiction`, which is set to the new size when the glyph is resized. The configurate stage will not allow a glyph to become smaller than the `minSize`, which defaults to $(0, 0)$. When higher levels configurate, they are then forced to accommodate the new size.

- In order to enable dragging glyphs, an `offsetExplicit` property is added to `Depiction`. If the glyph is moved, the new offset is assigned to the `offset` property and `offsetExplicit` is set to true. The configurate stage will not automatically assign an offset to a glyph with `offsetExplicit` set to true.

The addition of these properties means the layout can still be automatically configured, but with additional user constraints. For example, if a glyph is moved within its parent, its siblings will be forced to rearrange to accommodate the new position of the glyph during automatic configuration.

SynBioCAD also supports more sophisticated interactivity through “drag operations” (Table 7.1). Each drag operation is a function that receives a source layout and accompanying SBOL graph; a destination layout and accompanying SBOL graph; and the target bounding box (i.e., where the depiction has been dragged to). The source and destination are usually the same layout and graph, but may be different e.g. if a part is being placed after being imported from SynBioHub. The drag operation can return a new layout (if the interaction caused a change in the visualization, e.g. resizing a glyph), a new SBOL graph (if the interaction changed the underlying data, e.g. altering hierarchical composition), or none of the above if the operation was not applicable.

The depth-sorted list of depictions used to calculate sizes is also useful for user interaction. In the event of interaction (e.g. a mouse click), the top depiction can be retrieved by iterating the depictions starting with the lowest depth (i.e. the most nested) and checking the bounding box of each against the coordinate of the user interaction. It is guaranteed that if a depiction is obscured by another nested depiction, the obscured depiction will not be the target of the user interaction.

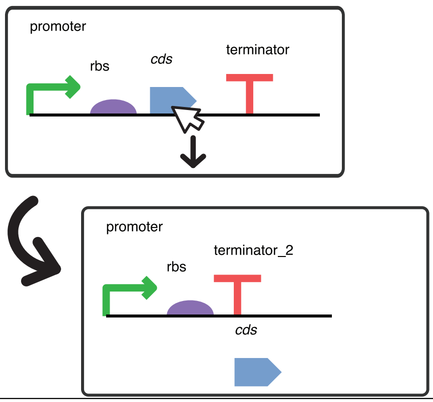
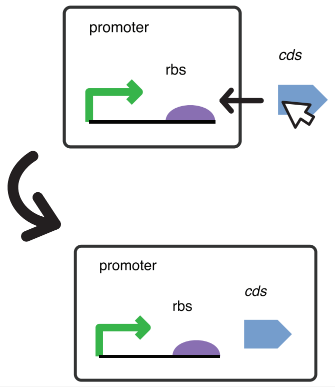
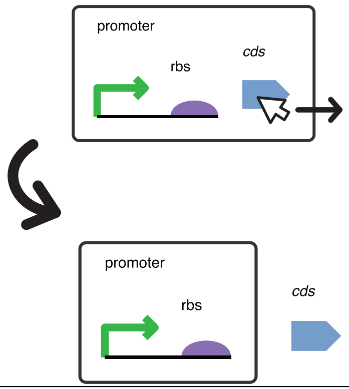
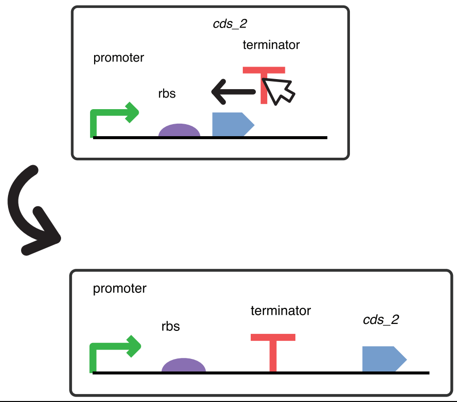
```
<?xml version='1.0' encoding='utf-8'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:sbol="http://sbols.org/v2#">
  <sbol:ComponentDefinition rdf:about="http://876d767d-9799-40fe-a5f4-fd0f522ca624/promoter/1">
    <sbol:persistentIdentity rdf:resource="http://876d767d-9799-40fe-a5f4-fd0f522ca624/promoter" />
    <sbol:version>1</sbol:version>
    <sbol:displayId>promoter</sbol:displayId>
    <sbol:role rdf:resource="http://identifiers.org/so/SO:0000167" />
    <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion" />
  </sbol:ComponentDefinition>
</rdf:RDF>
```

(a)

```
{
  "size": {
    "x": 85.375,
    "y": 54.25
  },
  "depictions": [
    {
      "class": "ComponentDepiction",
      "uid": 3,
      "offset": {
        "x": 10.5625,
        "y": 11.0125
      },
      "offsetExplicit": true,
      "size": {
        "x": 2,
        "y": 1.6
      },
      "minSize": {
        "x": 0,
        "y": 0
      },
      "opacity": 0,
      "isExpandable": false,
      "depictionOf": "http://876d767d-9799-40fe-a5f4-fd0f522ca624/promoter/1",
      "identifiedChain": "http://876d767d-9799-40fe-a5f4-fd0f522ca624/promoter/1",
      "children": [],
      "orientation": 0,
      "backbonePlacement": "top",
      "proportionalWidth": 2,
      "label": 4
    },
    {
      "class": "LabelDepiction",
      "uid": 4,
      "offset": {
        "x": 10.5625,
        "y": 10.0125
      },
      "offsetExplicit": false,
      "size": {
        "x": 4,
        "y": 1
      },
      "minSize": {
        "x": 0,
        "y": 0
      },
      "opacity": 1,
      "isExpandable": true,
      "depictionOf": null,
      "identifiedChain": null,
      "children": [],
      "labelFor": 3
    }
  ]
}
```

(b)

Figure 7.6: *An example of a SynBioCAD layout for a promoter (a) serialized in JSON (b). The URIs of parts are temporary URIs generated by SynBioCAD, which would be changed e.g. to SynBioHub URLs on publication. Information such as the names of parts are not included in the layout as they are already specified by the SBOL data to which this layout is complementary.*

DOpEnterParent When a `ComponentDepiction` is dragged into a parent `ComponentDepiction`, make the associated SBOL `SubComponent` a child of the parent `Component`.	 promoter rbs cds terminator promoter terminator_2 rbs cds
DOpEnterSibling When a `ComponentDepiction` is dragged into a whitebox sibling `ComponentDepiction`, create an SBOL `SubComponent` as a child of the sibling `Component`.	
DOpEnterWorkspace When a `ComponentDepiction` is dragged out of a containing `ComponentDepiction` into a space where it would have no parent, delete its SBOL `SubComponent` to allow it to be rendered as a root depiction.	
DOpMoveInBackbone When a `ComponentDepiction` is dragged horizontally within a `BackboneDepiction`, change its X offset only and allow the backbone layout to reposition sibling depictions.	 cds_2 terminator promoter rbs promoter terminator cds_2 rbs

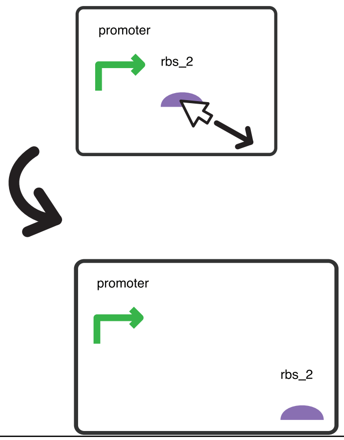
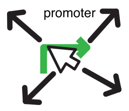
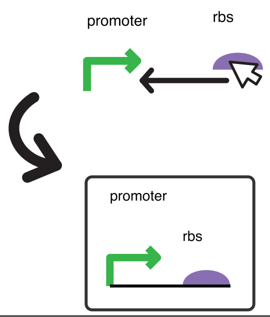
DOpMoveInParent When a `ComponentDepiction` is dragged within a parent, change its offset and allow the parent to resize to accommodate its new position.	
DOpMoveInWorkspace When a `ComponentDepiction` is dragged and has no parent, change its offset only.	
DOpTwoBlackboxesMakeConstraint When a blackbox `ComponentDepiction` is dragged over a sibling blackbox `ComponentDepiction`, wrap the two components in a parent component and create an SBOL `SequenceConstraint`.	

__Table 7.1:__ The current set of drag operations implemented by SynBioCAD.

Performance caveats

Ideally, the implementation of interactivity would be as straightforward as:

- If user interaction causes a change in the layout but not the underlying SBOL data, re-configurate the layout to accommodate for the change, then re-render the SVG
- If user interaction causes a change in the SBOL data (i.e., the graph), re-synchronise the layout using the algorithm described in section 7.2.3, re-configurate it, then re-render the SVG

Unfortunately, a practical implementation also has the consideration of performance. Both configuration of the layout and rendering the SVG are expensive operations — particularly in the context of a Web browser, where rendering alone can cause the deletion and creation of hundreds of elements in the page. This can be limited by restricting the configurate and render stages to only part of the layout. To implement this, each `Depiction` is also assigned a `version` property. If a depiction is modified, it increments its own version and that of all ancestors. The configurate process can then traverse the depiction hierarchy and only configurate depictions where the version has been modified. Subsequently, only depictions which have been reconfigurated need be re-rendered.

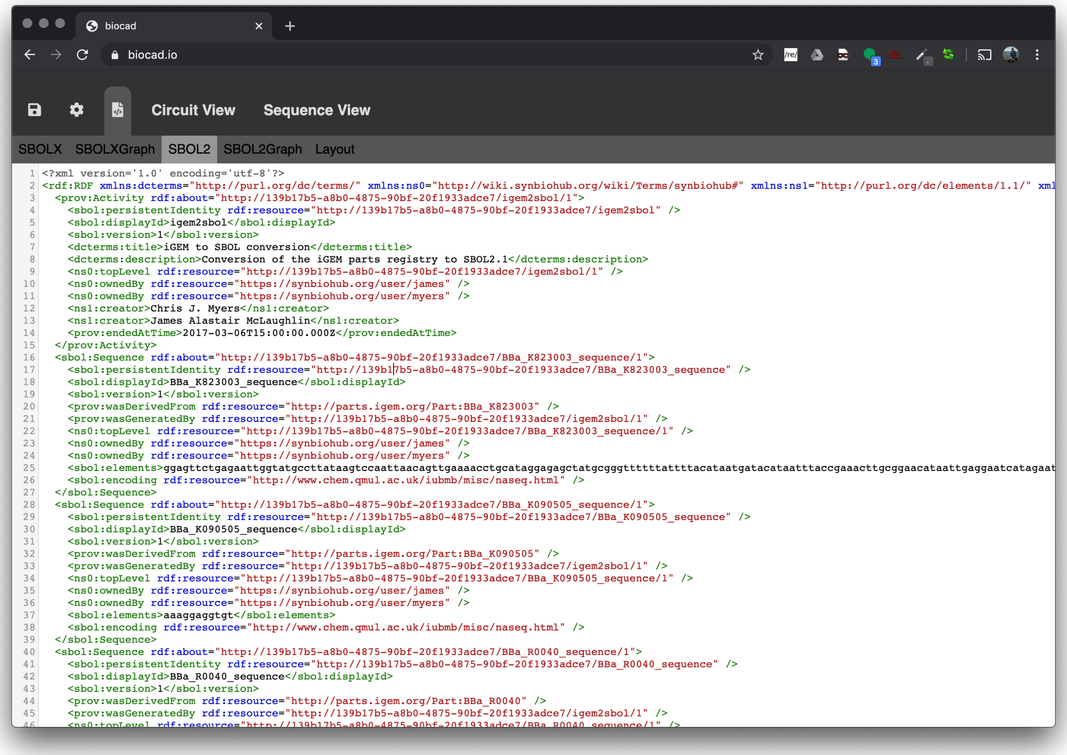
7.4 The SynBioCAD application

SynBioCAD[3,4] is a Web-based editor for SBOL developed as part of this work, building on the sbolgraph library and the new proposed version of the SBOL standard described in chapter 3, the SynBioHub repository described in chapter 6, and the visual layout data model defined in this chapter. The initial feature set of SynBioCAD comprises:

- Visualization of SBOL2 designs, including designs using the `ModuleDefinition` and `Interaction` classes
- Modifying designs by the drag and drop manipulation of interactive SBOL Visual glyphs
- Editing designs at a sequence level using an embedded sequence editor
- Integration with the SynBioHub design repository described in chapter 6, providing access to the conversion of the iGEM Registry described in chapter 5 (Fig. 7.10)

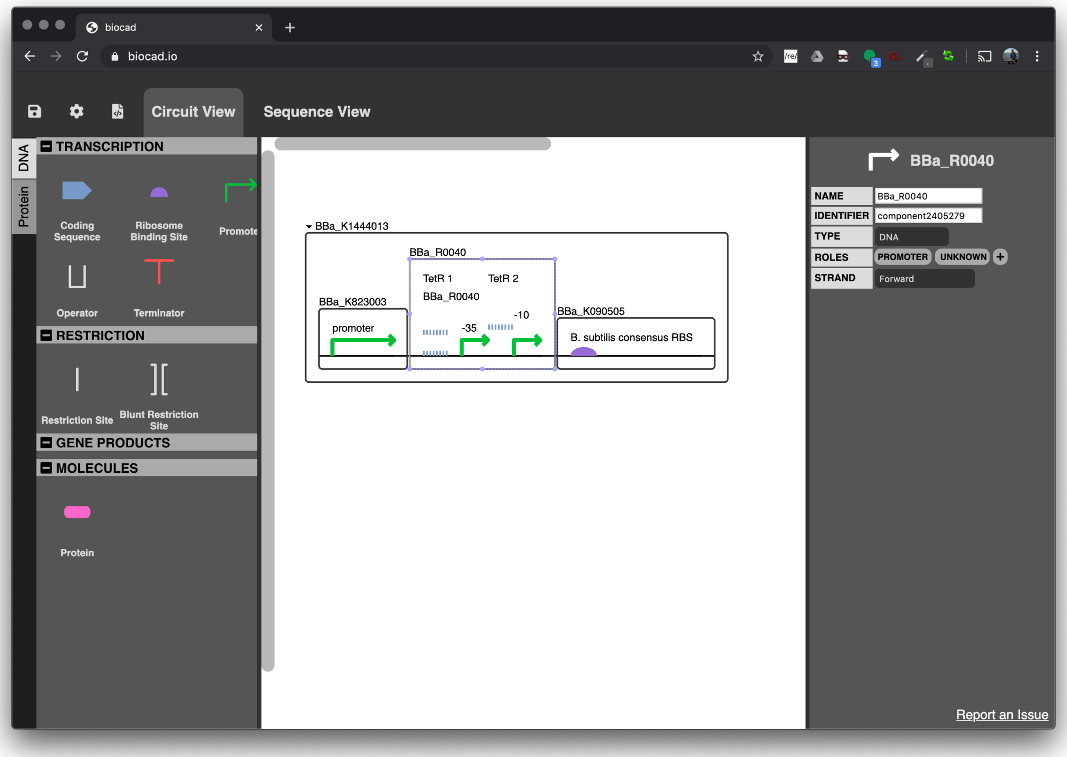
The SynBioCAD user interface has three design views: the circuit view (Fig. 7.8), the sequence view (Fig. 7.9), and the source view (Fig. 7.7). All of these views use the same underlying SBOL data, represented by the sbolgraph library. The synchronisation approach described in section 7.2.3 is used to propagate changes in the SBOL to changes in the visualization, which is rendered directly as SVG nodes in the Web browser.

[3]`https://biocad.io`
[4]`https://github.com/SynBioCAD/biocad`



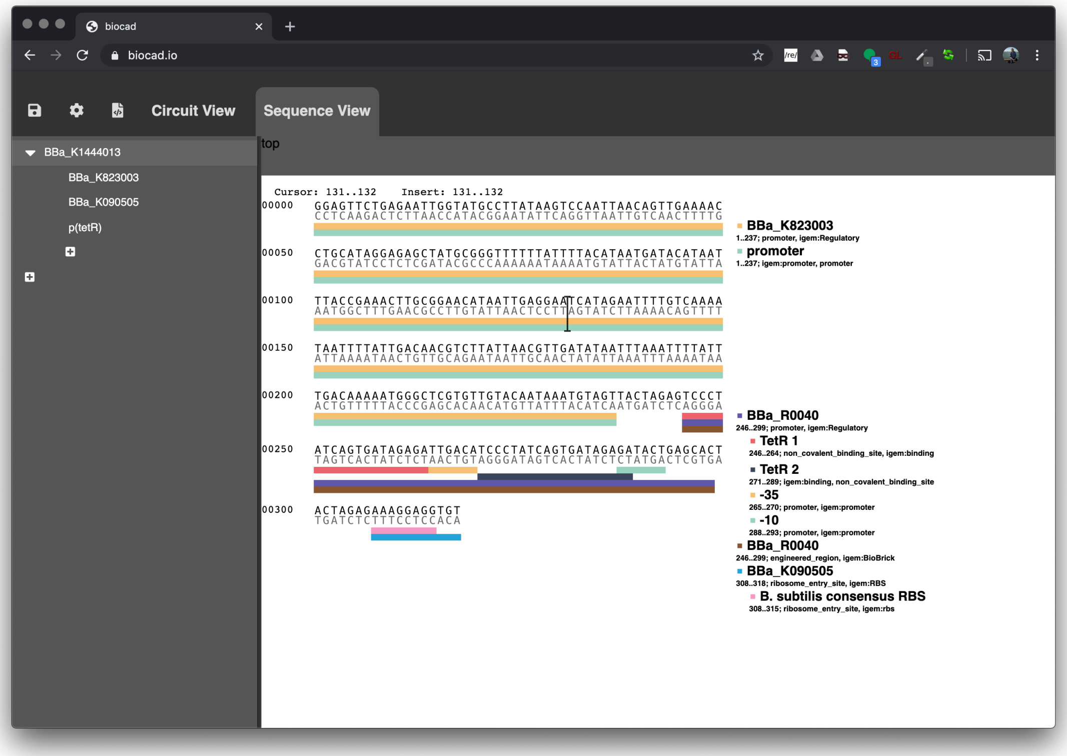
```xml
<?xml version='1.0' encoding='utf-8'?>
<rdf:RDF xmlns:dcterms="http://purl.org/dc/terms/" xmlns:ns0="http://wiki.synbiohub.org/wiki/Terms/synbiohub#" xmlns:ns1="http://purl.org/dc/elements/1.1/" xml
  <prov:Activity rdf:about="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol/1">
    <sbol:persistentIdentity rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol" />
    <sbol:displayId>igem2sbol</sbol:displayId>
    <sbol:version>1</sbol:version>
    <dcterms:title>iGEM to SBOL conversion</dcterms:title>
    <dcterms:description>Conversion of the iGEM parts registry to SBOL2.1</dcterms:description>
    <ns0:topLevel rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol/1" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/james" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/myers" />
    <ns1:creator>Chris J. Myers</ns1:creator>
    <ns1:creator>James Alastair McLaughlin</ns1:creator>
    <prov:endedAtTime>2017-03-06T15:00:00.000Z</prov:endedAtTime>
  </prov:Activity>
  <sbol:Sequence rdf:about="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K823003_sequence/1">
    <sbol:persistentIdentity rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K823003_sequence" />
    <sbol:displayId>BBa_K823003_sequence</sbol:displayId>
    <sbol:version>1</sbol:version>
    <prov:wasDerivedFrom rdf:resource="http://parts.igem.org/Part:BBa_K823003" />
    <prov:wasGeneratedBy rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol/1" />
    <ns0:topLevel rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K823003_sequence/1" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/james" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/myers" />
    <sbol:elements>ggagttctgagaattggtatgccttataagtccaattaacagttgaaaacctgcataggagagctatgcgggtttttttattttacataatgatacataatttaccgaaacttgcggaacataattgaggaatcatagaat
    <sbol:encoding rdf:resource="http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html" />
  </sbol:Sequence>
  <sbol:Sequence rdf:about="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K090505_sequence/1">
    <sbol:persistentIdentity rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K090505_sequence" />
    <sbol:displayId>BBa_K090505_sequence</sbol:displayId>
    <sbol:version>1</sbol:version>
    <prov:wasDerivedFrom rdf:resource="http://parts.igem.org/Part:BBa_K090505" />
    <prov:wasGeneratedBy rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol/1" />
    <ns0:topLevel rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_K090505_sequence/1" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/james" />
    <ns0:ownedBy rdf:resource="https://synbiohub.org/user/myers" />
    <sbol:elements>aaaggaggtgt</sbol:elements>
    <sbol:encoding rdf:resource="http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html" />
  </sbol:Sequence>
  <sbol:Sequence rdf:about="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_R0040_sequence/1">
    <sbol:persistentIdentity rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_R0040_sequence" />
    <sbol:displayId>BBa_R0040_sequence</sbol:displayId>
    <sbol:version>1</sbol:version>
    <prov:wasDerivedFrom rdf:resource="http://parts.igem.org/Part:BBa_R0040" />
    <prov:wasGeneratedBy rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/igem2sbol/1" />
    <ns0:topLevel rdf:resource="http://139b17b5-a8b0-4875-90bf-20f1933adce7/BBa_R0040_sequence/1" />
```

Figure 7.7: *The source view of SynBioCAD shows the underlying data in SBOLX (the proposed version of SBOL used internally by SynBioCAD) or SBOL2 format. It can also show the current layout in JSON format.*



NAME	BBa_R0040
IDENTIFIER	component2405279
TYPE	DNA
ROLES	PROMOTER UNKNOWN
STRAND	Forward

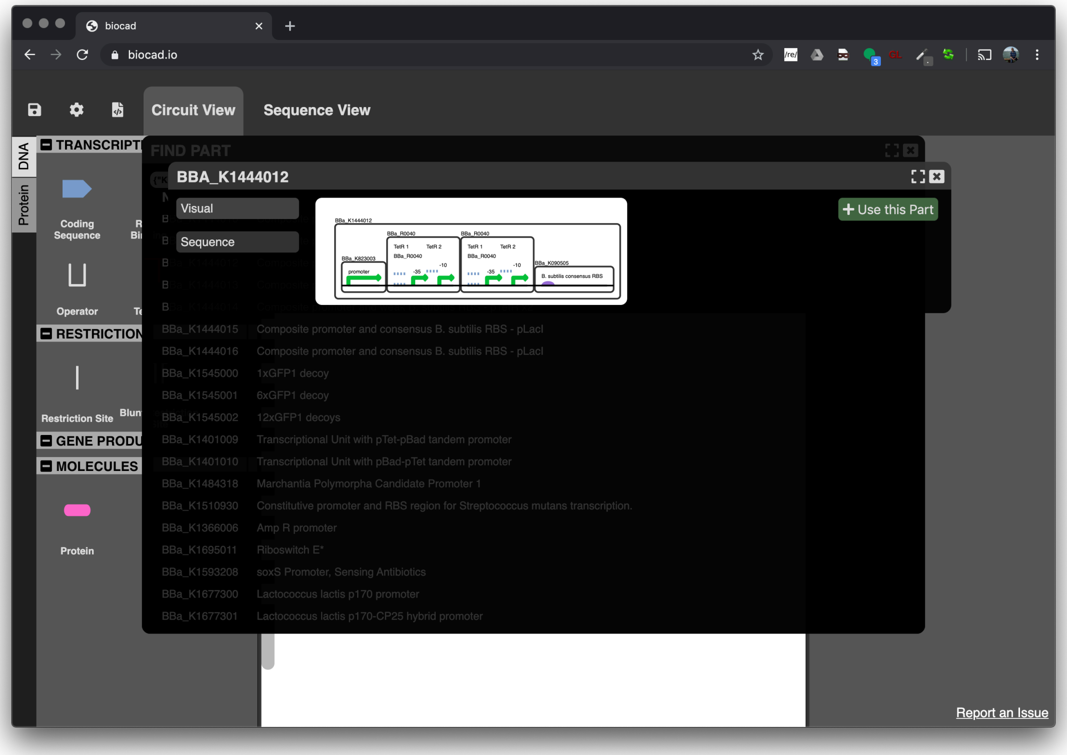
Figure 7.8: *The circuit view of SynBioCAD allows designs to be visualized and editedby "drag and drop" using SBOL Visual glyphs. It shares the same SBOL data as the sequence view.*



```
Cursor: 131..132    Insert: 131..132
00000   GGAGTTCTGAGAATTGGTATGCCTTATAAGTCCAATTAACAGTTGAAAAC
        CCTCAAGACTCTTAACCATACGGAATATTCAGGTTAATTGTCAACTTTTG
00050   CTGCATAGGAGAGCTATGCGGGTTTTTTATTTTACATAATGATACATAAT
        GACGTATCCTCTCGATACGCCCAAAAAATAAAATGTATTACTATGTATTA
00100   TTACCGAAACTTGCGGAACATAATTGAGGAATCATAGAATTTTGTCAAAA
        AATGGCTTTGAACGCCTTGTATTAACTCCTTAGTATCTTAAAACAGTTTT
00150   TAATTTTATTGACAACGTCTTATTAACGTTGATATAATTTAAATTTTATT
        ATTAAAATAACTGTTGCAGAATAATTGCAACTATATTAAATTTAAAATAA
00200   TGACAAAAATGGGCTCGTGTTGTACAATAAATGTAGTTACTAGAGTCCCT
        ACTGTTTTTACCCGAGCACAACATGTTATTTACATCAATGATCTCAGGGA
00250   ATCAGTGATAGAGATTGACATCCCTATCAGTGATAGAGATACTGAGCACT
        TAGTCACTATCTCTAACTGTAGGGATAGTCACTATCTCTATGACTCGTGA
00300   ACTAGAGAAAGGAGGTGT
        TGATCTCTTTCCTCCACA
```

- **BBa_K823003**
  1..237; promoter, igem:Regulatory
- **promoter**
  1..237; igem:promoter, promoter
- **BBa_R0040**
  246..299; promoter, igem:Regulatory
  - **TetR 1**
    246..264; non_covalent_binding_site, igem:binding
  - **TetR 2**
    271..289; igem:binding, non_covalent_binding_site
  - **-35**
    265..270; promoter, igem:promoter
  - **-10**
    288..293; promoter, igem:promoter
- **BBa_R0040**
  246..299; engineered_region, igem:BioBrick
- **BBa_K090505**
  308..318; ribosome_entry_site, igem:RBS
  - **B. subtilis consensus RBS**
    308..315; ribosome_entry_site, igem:rbs

Figure 7.9: *The sequence view of SynBioCAD allows sequence data to be modified directly. It shares the same SBOL data as the circuit view.*



BBa_K1444015	Composite promoter and consensus B. subtilis RBS - pLacI
BBa_K1444016	Composite promoter and consensus B. subtilis RBS - pLacI
BBa_K1545000	1xGFP1 decoy
BBa_K1545001	6xGFP1 decoy
BBa_K1545002	12xGFP1 decoys
BBa_K1401009	Transcriptional Unit with pTet-pBad tandem promoter
BBa_K1401010	Transcriptional Unit with pBad-pTet tandem promoter
BBa_K1484318	Marchantia Polymorpha Candidate Promoter 1
BBa_K1510930	Constitutive promoter and RBS region for Streptococcus mutans transcription.
BBa_K1366006	Amp R promoter
BBa_K1695011	Riboswitch E*
BBa_K1593208	soxS Promoter, Sensing Antibiotics
BBa_K1677300	Lactococcus lactis p170 promoter
BBa_K1677301	Lactococcus lactis p170-CP25 hybrid promoter

Figure 7.10: *In the SynBioCAD circuit view, parts can be imported from the SynBioHub repository described in chapter 6. This screenshot shows a part imported from the iGEM-SBOL conversion described in chapter 5.*

7.5 Discussion & Conclusion

The adoption of SBOL could make the representation of designs more standardized and machine-tractable. However, most designs are not yet represented in SBOL. Chapter 5 addressed this problem in the short-term by exploring how an existing dataset can be converted to SBOL. But the foundation of synthetic biology is the creation of *new* designs. Unless these new designs are documented using standards from the outset, it will be perpetually necessary to find ways to integrate design knowledge.

While there are some existing visual tools for the creation of SBOL data such as SBOLDesigner [89], SynBioCAD brings new functionality such as rendering of hierarchical composition; support for SBOL2 `ModuleDefinition` and `Interaction`; and an entirely Web-based editor which can be used without the need for any software.

SynBioCAD also attempts to accommodate for the fact that “Lego” potential of SynBio — where parts can be taken and plugged together to create new devices — is exciting, but also reductive. Much of the design process still takes place at the sequence level, as evidenced by the widespread success of Benchling compared to the relative obscurity of “top-down” tools such as SBOLDesigner.

For point-and-click design with interactive visual glyphs to be taken seriously, it must co-exist with the ability to work directly with the underlying sequence. A computing parallel is the prevelance of UNIX-like systems among software developers. While macOS and modern Linux distributions provide friendly graphical user interfaces, they still have a text-based Terminal emulator providing access to low-level command line tools. While the initial version of the SynBioCAD sequence editor is not comparable to a fully fledged editor such as Benchling [55], it demonstrates the idea of having different levels of abstraction for editing the same design.

7.5.1 Future work

Standardization of layouts

The difficulty of the development of SBOL Visual tooling should be a clear indicator that something needs to be done by the SBOL community to make their two standards — SBOL and SBOL Visual — work together more effectively.

Standardization of a data representation for layouts to complement the SBOL data model would be a logical step in this direction. Whether this representation would be derived from the layout data model proposed in this chapter or a proposal from elsewhere in the community, a formal specification for layouts could help to prevent repeated efforts and to accelerate the development of tooling that integrates both SBOL and SBOL Visual.

Continued development of SynBioCAD

SynBioCAD is a research project, and will require significant further development to turn it into a fully fledged design tool for synthetic biology. While the engineering groundwork — such as the sbolgraph library and specification of layouts — is in place, there is still much to be done. Ideas for future versions of SynBioCAD include:

- As the editor is Web-based, it can be embedded into other Web-based software. The recent review of SynBioHub [129] highlighted the need for the integrated ability to edit SBOL. SynBioCAD could potentially be implemented as the SynBioHub "submit" page to allow authors to document designs on the fly.
- Chapter 5 introduced Enrichment, a tool to enrich SBOL data using bioinformatics analysis. It may be possible to integrate Enrichment into SynBioCAD to suggest design elements such as interactions as the design is being constructed, in a manner similar to code completion provided by programming environments.
- The Virtual Parts Repository [80] provides a repository of SBOL components, each of which has an SBML [27] model attached. There exist various simulators for SBML such as COPASI [140] and iBioSim [141]. If SynBioCAD composed models at the same time as composing SBOL, it would potentially be possible to embed a simulator to provide a "Play" button for genetic circuit designs.

7.5.2 Conclusion

This chapter described a data model to bridge the SBOL data model and SBOL Visual by specifying layouts to link visual offsets and sizes to the descriptions of biological parts; and SynBioCAD, a Web-based tool for visualizing and editing SBOL built on much of the technology developed as part of this work.

SynBioCAD is the first visual authoring tool for SBOL with support for the SBOL2 "functional" concepts of modules and interactions. By abstracting away the SBOL data model, SynBioCAD is designed to serve not just the SBOL community, but synthetic biology in general — enabling users to capture designs both as a visualization, and using the machine-tractable data representation provided by SBOL.

Conclusions

8. Discussion & Conclusion

The two main research aims of this work were:

1. To explore how access to **existing knowledge** can be improved for the synthetic biology design process
2. To propose data standards and software infrastructure to make **future knowledge** about designs more accessible

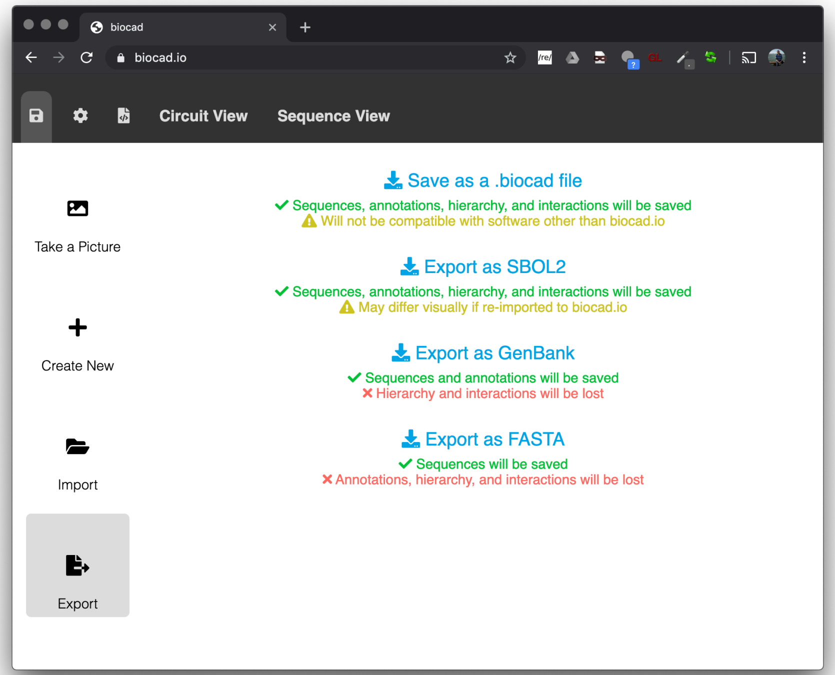
8.1 The groundwork: Machine-tractable design

The prerequisite for either of these aims was to determine what knowledge about designs *actually is*, and how it can be represented computationally. An ad-hoc description of a design in a Word processor is not something a computer can easily interpret. The representation of knowledge in a format which is amenable to comprehension by software — or, *machine-tractability* — is absolutely essential if we expect software to be able to do anything useful with that knowledge. The future of biodesign is likely to be knowledge-augmented, with computer software able to inspect a design and make suggestions about how to improve it or suggest parts that might be helpful. We might also expect that one day, the design process will be *entirely* automated: a user writes a specification for an engineered organism, and hits the "print" button. However, before we can expect any of this to be possible, we must make the design process transparent to machines.

One of the reasons that it is likely that this problem has not yet been adequately addressed is that the discipline of bioinformatics is well-established, and consequently has significant associated infrastructure, both in software and in data. It is very easy to look at the file formats typically used by bioinformatics tooling, such as the venerable FASTA and GenBank, and assume that they are adequate for synthetic biology. A recurring theme in this work was the need to challenge this assumption. Engineered systems are not the same as natural sequences. While they do share some things in common, such as potentially a DNA sequence, engineered systems have an entire set of additional properties which do not exist in natural sequences such as design intent, provenance, and composition.

It is possible to make-do with reductive data representations at the expense of machine-tractability. The iGEM Registry is a clear example of this: the only machine-tractable knowledge in the database is the DNA sequence of each part with annotated regions. All of the rich information that teams collect about parts is stored in free-text. There is little incentive *not* to take this approach: free-text is easier to write than learning to use a data standard, and what advantage does using a data standard actually bring to the designer of the part?

This considered, it is unsurprising that efforts such as SBOL struggle to gain traction. Expecting users to adopt a data standard out of altruism is unrealistic. For a standard to be adopted, it has to be *easier to use the standard than not to use it.* An example of where this might be the case with SBOL is the export option of SynBioCAD (Fig. 8.1). The user is presented with the option of exporting as SBOL, in which case they will keep all of the information about the design if it is re-loaded; as GenBank, where hierarchy and interactions will be lost; or as FASTA, where everything but the sequence will be lost. If other tools also supported SBOL, it would simply become the most frictionless option, much in the way that an Excel file is preferable to CSV for spreadsheets because formatting and formulas are retained.



✓ Sequences, annotations, hierarchy, and interactions will be saved
⚠ Will not be compatible with software other than biocad.io

✓ Sequences, annotations, hierarchy, and interactions will be saved
⚠ May differ visually if re-imported to biocad.io

✓ Sequences and annotations will be saved
✗ Hierarchy and interactions will be lost

✓ Sequences will be saved
✗ Annotations, hierarchy, and interactions will be lost

Figure 8.1: *The export options in SynBioCAD, demonstrating the value added by the SBOL standard. The user is presented with the option of exporting as SBOL, in which case they will keep all of the information about the design if it is re-loaded; as GenBank, where hierarchy and interactions will be lost; or as FASTA, where everything but the sequence will be lost.*

The problem is that many other tools *do not* support SBOL. As described in section 2.2.2, SBOL suffers from several interconnected issues which prevent it from gaining momentum. Implementing SBOL support in tools has been prohibitively difficult due to the only complete implementation being a Java library, libSBOLj — excluding software written in more common languages such as Python and JavaScript. Implementing new libraries for SBOL to address this problem has also proven very difficult due to the complexity of the data model. Even with a software library, understanding how to use SBOL is not an easy task.

The situation today is significantly better. The software libraries described in chapter 3 not only bring SBOL support to more programming languages, but also demonstrate that by leveraging the RDF foundation of SBOL, it can be much easier to implement the standard than the specification might suggest. Meanwhile, other developers in the SBOL community have also implement new libraries and abstractions, such as libSBOL [97].

The implementation of software libraries for a data standard may seem far removed from the general problem of machine-tractable design. But the work presented here has the potential to accelerate the adoption of SBOL, encouraging support to be added to more tools. Eventually, perhaps the majority of synthetic biology designs will be represented in SBOL, with vast repositories of SBOL data and a wealth of tooling with SBOL support.

Standards should be seen not as an exercise in imposing regulation, but as a tool to enable data and tooling to interoperate more effectively. Without co-operating on the development of standards, the synthetic biology community — like many other communities before it — will be forced to solve the same problems with standardization in perpetuity. The chapters of this work concerning data harmonization, while interesting from a research perspective, should not have been necessary. Biology is already extremely complicated, and adding layers of complexity by disagreeing on data representation creates more friction rather than less.

8.2 Aim 1: Access to existing knowledge

Previous attempts for data integration in the context of SynBio have mostly been concerned with improving our understanding of natural organisms. SynBioMine [58], for example, is almost entirely concerned with integrating data about genome features, collecting information about engineered parts from only one source.

It is likely that knowledge about engineered parts has simply not been considered worth integrating. Synthetic biology is still a fairly new field, and so engineered parts that are documented were documented fairly recently; surely the dataset is too small and too recent to require the application of data integration? However, this is simply not the case: as discussed in section 2.3, there exist large repositories of engineered parts which are disparate, heterogeneous, and computationally intractable.

SBOL can help in this regard by more than just providing yet another standard to capture designs. The SBOL data model is backed by RDF, and with RDF comes decades of research into knowledge representation and data integration. If designs are represented in SBOL and stored in RDF triplestores such as the SBOL Stack, it is not necessary to develop new SynBio-specific tooling to make design information tractable, because the infrastructure is already there and applicable to *any* RDF dataset.

Consequently, chapter 4 of this thesis briefly departed from the synthetic biology domain, focusing on the fundamentals of data integration using RDF. In this respect, the recent innovation of Linked Data Fragments (LDF) is a game-changer. Prior to LDF, federated querying was extremely verbose: a federated query had to manually specify which remote datasources to query for specific triple patterns, rather than being able to query multiple datasources simultaneously. The burden on servers of providing a full SPARQL endpoint is also tremendous, which has resulted in RDF resources having a reputation for being unreliable and having low availability. LDF solves both issues at once by moving the complexity of evaluating queries from the server to the client.

LDF is a very recent development, and not all of the possibilities of moving query evaluation to the client have yet been explored. Chapter 4 demonstrated one such interesting possibility. LDF servers are not encumbered with the processing of complex SPARQL queries and only have to respond to triple patterns. So, do those triple patterns really need to come from a real RDF datasource, or can triples be generated dynamically from something else? It is shown that this can indeed be accomplished by formalising a service such as a Web form or API into potential triple patterns and building an intelligent server that translates to and from RDF. The ldf-facade framework for developing such servers can be used to provide RDF views over non-RDF datasets, which is applicable both to the SynBio domain and any other knowledge on the Web.

Returning to more domain-specific applications of data integration, chapter 5 explored how the largest repository of engineered biological parts — the iGEM Registry — can be converted to SBOL format. This process involved mapping its database fields to fields in the SBOL data model, aligning its ad-hoc terminology to well-defined ontological terms, and de-flattening the hierarchy of parts to make composition more explicit. The resulting dataset is a significant resource for the SBOL community, growing the number of parts available in SBOL2 from a handful of examples in the specification to over 20,000 from the Registry. The advantages of an SBOL representation in making the Registry data more machine-tractable are also significant. While the Registry provides only a very basic keyword-driven search facility, in the SBOL representation the parts are now available in RDF, with all of the powerful querying capability enabled by SPARQL.

Converting knowledge from one representation to another can provide value when the new representation is easier to work with, as in the case of iGEM-SBOL. However, even if the new representation can capture a broader scope of knowledge than the original, the scope of the knowledge remains the same if it converted from a reductive form. The iGEM to SBOL conversion is an example of this: even though SBOL2 supports the description of additional layers of biology such as gene products and interactions, the iGEM Registry is restricted only to DNA sequences and therefore so is its resulting SBOL2 representation.

One method of enriching designs by "filling in" the SBOL2 data model is to predict the function of the part. Predicting function given a DNA sequence might be is the pursuit of the long-established field of bioinformatics. The technology developed by bioinformaticians, such as models trained to predict the locations of sequence features and protein domains, is traditionally used for the annotation of natural genomes. Engineered parts have been designed for a specific purpose, so their re-annotation using bioinformatics should be unnecessary. Nevertheless, we find ourselves with a vast repository of parts represented only by their sequence, and a data standard with the ability to capture much more.

The Enrichment tool described in chapter 5 allows additional information to be added to SBOL designs by constructing and executing a pipeline of knowledge integrations. As a proof-of-concept, an initial set of integrations were constructed from common bioinformatics tools and datasources such as PromoterHunter [121], TransTerm [120], and Pfam [111]. These integrations are used to build a knowledge base for the design, where each integration can build on information generated by prior integrations. Enrichment was shown to be able to re-annotate SBOL2 designs from a sequence level with some accuracy, and combined with SynBioCAD can even be used to produce a diagram from nothing but a DNA sequence.

The accuracy of Enrichment was not explored in detail in this thesis because the tools and datasets Enrichment integrates are not part of this work. They are intended to be interchangeable. It does not matter which tool, for example, finds ORFs in a sequence, because downstream integrations to translate the ORFs will still work with the same concepts in the knowledge base. What the development of Enrichment demonstrates is that the concept of using bioinformatics to fill in the gaps in knowledge about engineered designs is a viable approach.

8.3 Aim 2: Accessibility of future knowledge

Improving access to knowledge in the short-term through data integration may help to make the design process easier today by making it easier to collect knowledge about existing parts. But considering the purpose of synthetic biology is to engineer new designs, it is important to consider how *those* designs will be published for future designers. How does someone creating a design today publish it in a computationally tractable form?

The issues raised in the Peccoud et al. letter [8] — that publications about designs often do not include sequences — are finally being taken seriously by journals, and SBOL is central to the response. ACS Synthetic Biology now recommends that all authors use SBOL to represent their designs [54]. While this development is well-intentioned and definitely a step in the right direction, it cannot be an easy task for authors today. First, an author would have to learn enough about the highly complex SBOL data model to work out exactly how their design can be represented. They would then have to find some way to actually generate the SBOL, which could require learning to write Java code. The simplified data model and new tooling developed as part of this work may help to lower the burden on authors by making the process of building SBOL representations of designs easier.

The ACS Synthetic Biology recommendations fall short of recommending *where to put* the SBOL. The obvious place would be the supplementary material, but supplementary material is a set of files that can be downloaded alongside a paper — far removed from triplestores with rich querying capabilities as described throughout this work. JBEI host an instance of JBEI-ICE dedicated to ACS Synthetic Biology[1], but as discussed in section 2.4 it suffers from a similar problem, treating SBOL as a file format rather than as an RDF data model.

Using SBOL in this manner as essentially an improved format for sequences does not take full advantage of what an RDF-based representation of design knowledge could enable. The Semantic Web ambition of all of the data on the Internet being represented in RDF may or may not be realistic, but the application of Semantic Web technology to the specific domain of SynBio design is a powerful solution to an immediate problem,

[1]`https://acs-registry.jbei.org`

particularly with recent data integration technology such as Linked Data Fragments.

SynBioHub is an attempt to address this problem by combining the rich data representation of a triplestore with a Web-based user interface. Since its publication in 2018, SynBioHub has already found success in the SynBio community, having been adopted by projects such as the NSF Living Computing Project (LCP) [9] and the DARPA Synergistic Discovery & Design Project (SD2) [10]. Every new user of SynBioHub expands the ecosystem of highly queryable semantic design knowledge built around SBOL.

The final problem is how to create SBOL data in the first place. The recent review of SynBioHub and JBEI-ICE highlighted this issue, asserting that "SynBioHub must allow full online editing" [129]. Existing tools, such as SBOLDesigner [89], provide are restricted mainly to the features of SBOL1. SynBioCAD provides a Web-based, open-source editor for SBOL2, serving as an illustration of much of the work described herein: it is built upon the sbolgraph library described in chapter 3, and provides access to the parts converted from the iGEM Registry in chapter chapter 5 using the SynBioHub repository described in chapter 6.

8.3.1 Conclusion

The research question motivating this work was how the machine-tractability of knowledge can be improved in order to make the synthetic biology design process more efficient. Both short-term and long-term approaches were explored: short-term solutions to improve the tractability of knowledge that already exists, and long-term solutions to ensure that future knowledge can be documented using a machine-tractable data standard.

The Synthetic Biology Open Language (SBOL) served as the foundation for both of these approaches. Its data model is a formalisation of what constitutes a synthetic biology design, which is a prerequisite of any attempt to work with design knowledge. The contributions made to SBOL as part of this work are intended to facilitate the development of a smaller and easier to implement standard for the next iteration of SBOL, SBOL 3.0 — thereby encouraging adoption of SBOL by software, and in turn the long-term availability of design information in a standardized, machine-tractable format.

SBOL is an example of a Resource Description Framework (RDF) standard, meaning all SBOL data is also a knowledge graph which can be navigated using RDF tooling. The existing software infrastructure for SBOL rarely took advantage of this capability. This work has demonstrated that SBOL can be used directly as an RDF data model, using RDF graph libraries and RDF triplestores. Additionally, a novel method for making existing non-RDF knowledge tractable through dynamic harmonization was proposed. While the motivation was to use this method with the SBOL RDF standard, it could in theory be used to turn nearly any non-RDF datasource into a SPARQL endpoint.

The remainder of this work made use of the fundamental data representation provided by SBOL to realise a suite of tools which can be used as part of a standards-enabled data workflow for synthetic biology design. SynBioHub provides a platform for any user to share design knowledge, and when combined with the conversion of the iGEM Registry makes the largest SBOL dataset of parts to-date readily available online. Enrichment can automatically add additional information to designs by integrating the output of bioinformatics analyses. SynBioCAD provides Web-based visualization and editing capabilities using the SBOL and SBOL Visual standards, enabling any user to create SBOL data regardless of familiarity with the standard.

Together, the contributions of this work help to further standardization of knowledge representation in the synthetic biology design process, potentially improving its efficiency and ultimately the efficiency of the synthetic biology lifecycle as a whole.

Bibliography

[1] Endy, D. “Foundations for engineering biology”. In: *Nature* 438.7067 (2005), p. 449.

[2] Medema, M. H., Breitling, R., Bovenberg, R., and Takano, E. “Exploiting plug-and-play synthetic biology for drug discovery and production in microorganisms”. In: *Nature Reviews Microbiology* 9.2 (2011), p. 131.

[3] Verseux, C. N., Paulino-Lima, I. G., Baqué, M., Billi, D., and Rothschild, L. J. “Synthetic biology for space exploration: promises and societal implications”. In: *Ambivalences of Creating Life*. Springer, 2016, pp. 73–100.

[4] SynbiCITE. *Synthetic Biology Investors*. URL: `http://www.synbicite.com/collaboration/investors/` (visited on 07/10/2019).

[5] URL: `https://bbsrc.ukri.org/funding/grants/priorities/synthetic-bio/` (visited on 07/17/2019).

[6] Carlson, R. H. *Biology Is Technology: The Promise, Peril, and New Business of Engineering Life*. Harvard University Press, 2010. ISBN: 978-0-674-05362-5.

[7] Doan, A., Halevy, A., and Ives, Z. *Principles of Data Integration*. Morgan Kaufmann, 2012. ISBN: 9780124160446.

[8] Peccoud, J., Anderson, J. C., Chandran, D., Densmore, D., Galdzicki, M., Lux, M. W., Rodriguez, C. A., Stan, G.-B., and Sauro, H. M. “Essential information for synthetic DNA sequences”. In: *Nature biotechnology* 29.1 (2011), pp. 22–22.

[9] Project, L. C. *Living Computing Project Resources*. URL: `https://www.programmingbiology.org/resources` (visited on 09/14/2019).

[10] (TACC), T. A. C. C. *Synergistic Discovery and Design Environment*. URL: `https://sd2e.org/about/` (visited on 09/14/2019).

[11] Prakash, C. S. “The genetically modified crop debate in the context of agricultural evolution”. In: *Plant physiology* 126.1 (2001), pp. 8–15.

[12] Wright, D. “The Genetic Architecture of Domestication in Animals”. In: *Bioinform Biol Insights* 9.Suppl 4 (2015), pp. 11–20.

[13] Dijkstra, E. W. “The humble programmer”. In: *Commun. ACM* 15.10 (1972), pp. 859–866.

[14] Andrianantoandro, E., Basu, S., Karig, D. K., and Weiss, R. “Synthetic biology: new engineering rules for an emerging discipline”. In: *Molecular systems biology* 2.1 (2006).

[15] Kelwick, R., MacDonald, J. T., Webb, A. J., and Freemont, P. "Developments in the tools and methodologies of synthetic biology". In: *Frontiers in bioengineering and biotechnology* 2 (2014), p. 60.

[16] Suryanarayana, G., Samarthyam, G., and Sharma, T. "Chapter 5 - Modularization Smells". In: *Refactoring for Software Design Smells*. Boston: Morgan Kaufmann, 2015, pp. 93–122. ISBN: 978-0-12-801397-7.

[17] Chen, D. and Vernadat, F. B. "Enterprise interoperability: A standardisation view". In: *International Conference on Enterprise Integration and Modeling Technology*. Springer. 2002, pp. 273–282.

[18] Wheelwright, S. C. and Clark, K. B. "Accelerating the design-build-test cycle for effective product development". In: *International Marketing Review* 11.1 (1994), pp. 32–46.

[19] Wang, J., Xiong, Z., Meng, H., Wang, Y., and Wang, Y. "Synthetic biology triggers new era of antibiotics development". In: *Reprogramming Microbial Metabolic Pathways*. Springer, 2012, pp. 95–114.

[20] Chen, Y. Y. and Smolke, C. D. "From DNA to targeted therapeutics: bringing synthetic biology to the clinic". In: *Science translational medicine* 3.106 (2011), 106ps42–106ps42.

[21] Brophy, J. A. and Voigt, C. A. "Principles of genetic circuit design". In: *Nat. Methods* 11.5 (2014), pp. 508–520.

[22] Elowitz, M. B. and Leibler, S. "A synthetic oscillatory network of transcriptional regulators". In: *Nature* 403.6767 (2000), pp. 335–338.

[23] Gardner, T. S., Cantor, C. R., and Collins, J. J. "Construction of a genetic toggle switch in *Escherichia coli*". In: *Nature* 403 (2000), pp. 339–342.

[24] Jacob, F. and Monod, J. "Genetic regulatory mechanisms in the synthesis of proteins". In: *Journal of molecular biology* 3.3 (1961), pp. 318–356.

[25] Jusiak, B., Cleto, S., Perez-Pinera, P., and Lu, T. K. "Engineering synthetic gene circuits in living cells with CRISPR technology". In: *Trends in biotechnology* 34.7 (2016), pp. 535–547.

[26] Beal, J. "Bridging the gap: a roadmap to breaking the biological design barrier". In: *Frontiers in bioengineering and biotechnology* 2 (2015), p. 87.

[27] Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., et al. "The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models". In: *Bioinformatics* 19.4 (2003), pp. 524–531.

[28] Le Novere, N., Bornstein, B., Broicher, A., Courtot, M., Donizelli, M., Dharuri, H., Li, L., Sauro, H., Schilstra, M., Shapiro, B., et al. "BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems". In: *Nucleic acids research* 34.suppl_1 (2006), pp. D689–D691.

[29] *International Workshop on Bio-Design Automation*. URL: `https://www.iwbdaconf.org/` (visited on 09/27/2019).

[30] Nielsen, A. A., Der, B. S., Shin, J., Vaidyanathan, P., Paralanov, V., Strychalski, E. A., Ross, D., Densmore, D., and Voigt, C. A. “Genetic circuit design automation”. In: *Science* 352.6281 (2016), aac7341.

[31] Pearson, W. R. “Rapid and sensitive sequence comparison with FASTP and FASTA”. In: (1990).

[32] Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., and Sayers, E. W. “GenBank”. In: *Nucleic acids research* 41.D1 (2013), pp. D36–D42.

[33] Burks, C., Fickett, J. W., Goad, W. B., Kanehisa, M., Lewitter, F. I., Rindone, W. P., Swindell, C. D., Tung, C.-S., and Bilofsky, H. S. “CABIOS REVIEW: The GenBank nucleic acid sequence database”. In: *Bioinformatics* 1.4 (1985), pp. 225–233.

[34] “ECMA-404: The JSON Data Interchange Format.” In: (2013).

[35] *Extensible Markup Language (XML).* URL: `https://www.w3.org/XML/` (visited on 09/19/2019).

[36] *JSON Schema.* URL: `https://json-schema.org` (visited on 09/19/2019).

[37] *W3C XML Schema Definition Language (XSD).* URL: `https://www.w3.org/TR/xmlschema11-1/` (visited on 09/19/2019).

[38] *How can I access resources on this website programmatically?* URL: `https://www.uniprot.org/help/api` (visited on 09/19/2019).

[39] W3C. *RDF 1.1 Concepts and Abstract Syntax.* Feb. 25, 2014. URL: `https://www.w3.org/TR/rdf11-concepts` (visited on 08/21/2019).

[40] Weibel, S., Kunze, J., Lagoze, C., and Wolf, M. “Dublin core metadata for resource discovery”. In: (1998).

[41] *Ontologies.* URL: `https://www.w3.org/standards/semanticweb/ontology` (visited on 09/19/2019).

[42] Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., et al. “Gene ontology: tool for the unification of biology”. In: *Nature genetics* 25.1 (2000), p. 25.

[43] Eilbeck, K., Lewis, S. E., Mungall, C. J., Yandell, M., Stein, L., Durbin, R., and Ashburner, M. “The Sequence Ontology: a tool for the unification of genome annotations”. In: *Genome biology* 6.5 (2005), R44.

[44] Redaschi, N., Consortium, U., et al. “Uniprot in RDF: Tackling data integration and distributed annotation with the semantic web”. In: (2009).

[45] Fu, G., Batchelor, C., Dumontier, M., Hastings, J., Willighagen, E., and Bolton, E. “PubChemRDF: towards the semantic annotation of PubChem compound and substance databases”. In: *Journal of cheminformatics* 7.1 (2015), p. 34.

[46] Jupp, S., Malone, J., Bolleman, J., Brandizi, M., Davies, M., Garcia, L., Gaulton, A., Gehant, S., Laibe, C., Redaschi, N., et al. “The EBI RDF platform: linked open data for the life sciences”. In: *Bioinformatics* 30.9 (2014), pp. 1338–1339.

[47] Galdzicki, M., Clancy, K. P., Oberortner, E., Pocock, M., Quinn, J. Y., Rodriguez, C. A., Roehner, N., Wilson, M. L., Adam, L., Anderson, J. C., et al. “The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology”. In: *Nature biotechnology* 32.6 (2014), pp. 545–550.

[48] Courtot, M., Juty, N., Knüpfer, C., Waltemath, D., Zhukova, A., Dräger, A., Dumontier, M., Finney, A., Golebiewski, M., Hastings, J., et al. “Controlled vocabularies and semantics in systems biology”. In: *Molecular systems biology* 7.1 (2011).

[49] Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., and Zhao, J. “Prov-o: The prov ontology”. In: *W3C recommendation* 30 (2013).

[50] Weibel, S. “The Dublin Core: a simple content description model for electronic resources”. In: *Bulletin of the American Society for Information Science and Technology* 24.1 (1997), pp. 9–11.

[51] Mısırlı, G., Hallinan, J., Pocock, M., Lord, P., McLaughlin, J. A., Sauro, H., and Wipat, A. “Data integration and mining for synthetic biology design”. In: *ACS synthetic biology* 5.10 (2016), pp. 1086–1097.

[52] Galdzicki, M., Wilson, M. L., Rodriguez, C. A., Adam, L., Adler, A., Anderson, J. C., Beal, J., Chandran, D., Densmore, D., Drory, O. A., Endy, D., Gennari, J. H., Grünberg, R., Ham, T. S., Kuchinsky, A., Lux, M. W., Madsen, C., Misirli, G., Myers, C. J., Peccoud, J., Plahar, H., Pocock, M. R., Roehner, N., Smith, T. F., Stan, G.-B., Villalobos, A., Wipat, A., and Sauro, H. M. *BBF RFC 84: Synthetic Biology Open Language (SBOL) Version 1.0.0.* Tech. rep. Sept. 2011. DOI: `1721.1/60088`.

[53] W3C. *RDF 1.1 XML Syntax.* Feb. 25, 2014. URL: `https://www.w3.org/TR/rdf-syntax-grammar/` (visited on 09/10/2019).

[54] Hillson, N. J., Plahar, H. A., Beal, J., and Prithviraj, R. *Improving Synthetic Biology Communication: Recommended Practices for Visual Depiction and Digital Submission of Genetic Designs.* 2016.

[55] *Benchling.* URL: `https://www.benchling.com` (visited on 09/24/2019).

[56] Kwok, R. “Five hard truths for synthetic biology”. In: *Nature News* 463.7279 (2010), pp. 288–290.

[57] Lapatas, V., Stefanidakis, M., Jimenez, R. C., Via, A., and Schneider, M. V. “Data integration in biological research: an overview”. In: *Journal of Biological Research-Thessaloniki* 22.1 (2015), p. 9.

[58] Department of Genetics, U. o. C. *SynBioMine.* URL: `http://www.synbiomine.org/` (visited on 08/21/2019).

[59] Smith, R. N., Aleksic, J., Butano, D., Carr, A., Contrino, S., Hu, F., Lyne, M., Lyne, R., Kalderimis, A., Rutherford, K., et al. “InterMine: a flexible data warehouse system for the integration and analysis of heterogeneous biological data”. In: *Bioinformatics* 28.23 (2012), pp. 3163–3165.

[60] Department of Genetics, U. o. C. *SynBioMine: Data Categories.* URL: `http://www.synbiomine.org/synbiomine/dataCategories.do` (visited on 08/21/2019).

[61] Bultelle, M., Murieta, I. S. de, and Kitney, R. "Introducing SynBIS-The synthetic biology information system". In: *IET/SynbiCITE Engineering Biology Conference.* IET. 2016, pp. 1–2.

[62] Codd, E. F. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[63] W3C. *SPARQL Query Language for RDF.* Jan. 15, 2008. URL: `https://www.w3.org/TR/rdf-sparql-query` (visited on 08/21/2019).

[64] Broekstra, J., Kampman, A., and Van Harmelen, F. "Sesame: A generic architecture for storing and querying rdf and rdf schema". In: *International semantic web conference.* Springer. 2002, pp. 54–68.

[65] Erling, O. and Mikhailov, I. "RDF Support in the Virtuoso DBMS". In: *Networked Knowledge-Networked Media.* Springer, 2009, pp. 7–24.

[66] McBride, B. "Jena: Implementing the rdf model and syntax specification". In: *Proceedings of the Second International Conference on Semantic Web-Volume 40.* CEUR-WS. org. 2001, pp. 23–28.

[67] LLC, S. *BlazeGraph.* URL: `https://www.blazegraph.com` (visited on 09/27/2019).

[68] Berners-Lee, T., Hendler, J., and Lassila, O. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 28–37.

[69] Berners-Lee, T. *Linked Data — Design issues.* July 27, 2006. URL: `https://www.w3.org/DesignIssues/LinkedData.html` (visited on 08/21/2019).

[70] W3C. *SPARQL 1.1 Protocol.* Mar. 21, 2013. URL: `https://www.w3.org/TR/sparql11-protocol` (visited on 08/21/2019).

[71] Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., and Van de Walle, R. "Web-Scale Querying through Linked Data Fragments." In: *LDOW.* Citeseer. 2014.

[72] Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., and Van de Walle, R. "Querying datasets on the web with high availability". In: *International Semantic Web Conference.* Springer. 2014, pp. 180–196.

[73] *GenBank and WGS Statistics.* URL: `https://www.ncbi.nlm.nih.gov/genbank/statistics/` (visited on 05/28/2019).

[74] *Current Release Statistics.* URL: `https://www.ebi.ac.uk/uniprot/TrEMBLstats` (visited on 05/28/2019).

[75] Zhu, B. and Stülke, J. "SubtiWiki in 2018: from genes and proteins to functional network annotation of the model organism Bacillus subtilis". In: *Nucleic Acids Research* 46.D1 (Oct. 2017), pp. D743–D748. ISSN: 0305-1048. DOI: `10.1093/nar/gkx908`.

[76] Guo, A. C., Jewison, T., Wilson, M., Liu, Y., Knox, C., Djoumbou, Y., Lo, P., Mandal, R., Krishnamurthy, R., and Wishart, D. S. "ECMDB: the E. coli Metabolome Database". In: *Nucleic acids research* 41.D1 (2012), pp. D625–D630.

[77] Consortium, U. "UniProt: a hub for protein information". In: *Nucleic acids research* 43.D1 (2014), pp. D204–D212.

[78] *Registry of Standard Biological Parts.* URL: `http://parts.igem.org/Main_Page` (visited on 09/19/2019).

[79] Ham, T. S., Dmytriv, Z., Plahar, H., Chen, J., Hillson, N. J., and Keasling, J. D. "Design, implementation and practice of JBEI-ICE: an open source biological part registry platform and tools". In: *Nucleic acids research* 40.18 (2012), e141–e141.

[80] University, N. *Virtual Parts.* URL: `http://virtualparts.org/parts` (visited on 08/21/2019).

[81] Herscovitch, M., Perkins, E., Baltus, A., and Fan, M. "Addgene provides an open forum for plasmid sharing". In: *Nature biotechnology* 30.4 (2012), pp. 316–317.

[82] Brown, J. "The iGEM competition: building with biology". In: *IET Synthetic Biology* 1.1 (2007), pp. 3–6.

[83] Galdzicki, M., Rodriguez, C., Chandran, D., Sauro, H. M., and Gennari, J. H. "Standard biological parts knowledgebase". In: *PloS one* 6.2 (2011), e17005.

[84] Kamens, J. "The Addgene repository: an international nonprofit plasmid and data resource". In: *Nucleic Acids Research* 43.D1 (Nov. 2014), pp. D1152–D1157. ISSN: 0305-1048.

[85] Misirli, G., Hallinan, J., and Wipat, A. "Composable modular models for synthetic biology". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 11.3 (2014), p. 22.

[86] Boutillier, P., Maasha, M., Li, X., Medina-Abarca, H. F., Krivine, J., Feret, J., Cristescu, I., Forbes, A. G., and Fontana, W. "The Kappa platform for rule-based modeling". In: *Bioinformatics* 34.13 (June 2018), pp. i583–i592. ISSN: 1367-4803.

[87] Quinn, J., Beal, J., Bhatia, S., Cai, P., Chen, J., Clancy, K., Hillson, N., Galdzicki, M., Maheshwari, A., Pocock, M., et al. "Synthetic biology open language visual (SBOL Visual), Version 1.0.0". In: (2013).

[88] Der, B. S., Glassey, E., Bartley, B. A., Enghuus, C., Goodman, D. B., Gordon, D. B., Voigt, C. A., and Gorochowski, T. E. "DNAplotlib: programmable visualization of genetic designs and associated data". In: *ACS synthetic biology* 6.7 (2016), pp. 1115–1119.

[89] Zhang, M., McLaughlin, J. A., Wipat, A., and Myers, C. J. "SBOLDesigner 2: an intuitive tool for structural genetic design". In: *ACS synthetic biology* 6.7 (2017), pp. 1150–1160.

[90] Van Assche, E., Van Puyvelde, S., Vanderleyden, J., and Steenackers, H. P. "RNA-binding proteins involved in post-transcriptional regulation in bacteria". In: *Front. Microbiol.* 6 (2015).

[91] Kushwaha, M. and Salis, H. M. "A portable expression resource for engineering cross-species genetic circuits and pathways". In: *Nat. Commun.* 6 (2015).

[92] Bhatia, S. and Densmore, D. "Pigeon: a design visualizer for synthetic biology". In: *ACS synthetic biology* 2.6 (2013), pp. 348–350.

[93] Chandran, D., Bergmann, F. T., Sauro, H. M., et al. "TinkerCell: modular CAD tool for synthetic biology". In: *J Biol Eng* 3.1 (2009), p. 19.

[94] McLaughlin, J. A., Pocock, M., Mısırlı, G., Madsen, C., and Wipat, A. "VisBOL: web-based tools for synthetic biology design visualization". In: *ACS synthetic biology* 5.8 (2016), pp. 874–876.

[95] Zhang, Z., Nguyen, T., Roehner, N., Misirli, G., Pocock, M., Oberortner, E., Samineni, M., Zundel, Z., Beal, J., Clancy, K., Wipat, A., and Myers, C. J. "libSBOLj 2.0: A Java Library to Support SBOL 2.0". In: *IEEE Life Sci Lett* 1.4 (Dec. 2016), pp. 34–37. DOI: 10.1109/LLS.2016.2546546.

[96] *Developer Survey Results 2019.* URL: https://insights.stackoverflow.com/survey/2019 (visited on 06/27/2019).

[97] Bartley, B. A., Choi, K., Samineni, M., Zundel, Z., Nguyen, T., Myers, C. J., and Sauro, H. M. "pySBOL: A Python Package for Genetic Design Automation and Standardization". In: *ACS synthetic biology* (2018).

[98] Gómez, J., García, L. J., Salazar, G. A., Villaveces, J., Gore, S., García, A., Martín, M. J., Launay, G., Alcántara, R., del-Toro, N., Dumousseau, M., Orchard, S., Velankar, S., Hermjakob, H., Zong, C., Ping, P., Corpas, M., and Jiménez, R. C. "BioJS: an open source JavaScript framework for biological data visualization". In: *Bioinformatics* 29.8 (Feb. 2013), pp. 1103–1104.

[99] Zundel, Z., Samineni, M., Zhang, Z., and Myers, C. J. "A validator and converter for the synthetic biology open language". In: *ACS synthetic biology* 6.7 (2017), pp. 1161–1168.

[100] Roehner, N., Oberortner, E., Pocock, M., Beal, J., Clancy, K., Madsen, C., Mısırlı, G., Wipat, A., Sauro, H., and Myers, C. J. "Proposed data model for the next version of the Synthetic Biology Open Language". In: *ACS synthetic biology* ().

[101] *RDFLib.* URL: https://github.com/RDFLib/rdflib (visited on 09/27/2019).

[102] *Apache Commons RDF.* URL: https://commons.apache.org/proper/commons-rdf/ (visited on 09/27/2019).

[103] *rdf-ext.* URL: https://github.com/rdf-ext/ (visited on 09/27/2019).

[104] Microsoft. *TypeScript.* URL: https://www.typescriptlang.org (visited on 09/26/2019).

[105] Madsen, C., Moreno, A. G., P, U., Palchick, Z., Roehner, N., Atallah, C., Bartley, B., Choi, K., Cox, R. S., Gorochowski, T., Grünberg, R., Macklin, C., McLaughlin, J., Meng, X., Nguyen, T., Pocock, M., Samineni, M., Scott-Brown, J., Tarter, Y., Zhang, M., Zhang, Z., Zundel, Z., https:andandorcid.organd0000-0002-1663-5102, J. B. iD: Bissell, M., Clancy, K., Gennari, J. H., Misirli, G., Myers, C., Oberortner, E., Sauro, H., and Wipat, A. "Synthetic biology open language (SBOL) version 2.3. 0". In: *Journal of Integrative Bioinformatics* 16 (2019).

[106] Beazley, D. M. "Automated scientific software scripting with SWIG". In: *Future Generation Computer Systems* 19.5 (2003), pp. 599–609.

[107] Misirli, G., Taylor, R., Goñi-Moreno, A., Mclaughlin, J. A., Myers, C. J., Gennari, J., Lord, P., and Wipat, A. "SBOL-OWL: An ontological approach for formal and semantic representation of synthetic biology information". In: *ACS synthetic biology* (2019).

[108] Limited, A. A. *DistroWatch.com: Put the fun back into computing. Use Linux, BSD.* URL: `https://distrowatch.com` (visited on 09/08/2019).

[109] Bartley, B., Beal, J., Clancy, K., Misirli, G., Roehner, N., Oberortner, E., Pocock, M., Bissell, M., Madsen, C., Nguyen, T., et al. "Synthetic biology open language (SBOL) version 2.0. 0". In: *Journal of integrative bioinformatics* 12.2 (2015), pp. 902–991.

[110] Wimalaratne, S. M., Bolleman, J., Juty, N., Katayama, T., Dumontier, M., Redaschi, N., Le Novère, N., Hermjakob, H., and Laibe, C. "SPARQL-enabled identifier conversion with Identifiers.org". In: *Bioinformatics* 31.11 (Jan. 2015), pp. 1875–1877. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/btv064`.

[111] Bateman, A., Coin, L., Durbin, R., Finn, R. D., Hollich, V., Griffiths-Jones, S., Khanna, A., Marshall, M., Moxon, S., Sonnhammer, E. L., et al. "The Pfam protein families database". In: *Nucleic acids research* 32.suppl_1 (2004), pp. D138–D141.

[112] *Frequently Used Parts.* URL: `http://parts.igem.org/Frequently_Used_Parts` (visited on 09/22/2019).

[113] Stewart, A. J., Hannenhalli, S., and Plotkin, J. B. "Why transcription factor binding sites are ten nucleotides long". In: *Genetics* 192.3 (2012), pp. 973–985.

[114] Rice, P., Longden, I., and Bleasby, A. *EMBOSS: the European molecular biology open software suite.* 2000.

[115] Vilanova, C. and Porcar, M. "iGEM 2.0—refoundations for engineering biology". In: *Nature biotechnology* 32.5 (2014), p. 420.

[116] Bruce, A., Johnson, A., Lewis, J., Morgan, D., Raff, M., Roberts, K., and Walter, P. *Molecular Biology of the Cell Sixth Edition.* Garland Science, 2015. ISBN: 9780815344322.

[117] Stanke, M., Steinkamp, R., Waack, S., and Morgenstern, B. "AUGUSTUS: a web server for gene finding in eukaryotes". In: *Nucleic acids research* 32.suppl_2 (2004), W309–W312.

[118] Keiler, K. C., Waller, P. R., and Sauer, R. T. "Role of a peptide tagging system in degradation of proteins synthesized from damaged messenger RNA". In: *Science* 271.5251 (1996), pp. 990–993.

[119] *ve-sequence-utils. DNA/RNA/AA sequence manipulation utility functions.* URL: `https://github.com/TeselaGen/ve-sequence-utils` (visited on 09/19/2019).

[120] Jacobs, G. H., Chen, A., Stevens, S. G., Stockwell, P. A., Black, M. A., Tate, W. P., and Brown, C. M. "Transterm: a database to aid the analysis of regulatory sequences in mRNAs". In: *Nucleic acids research* 37.suppl_1 (2008), pp. D72–D76.

[121] Klucar, L., Stano, M., and Hajduk, M. "phiSITE: database of gene regulation in bacteriophages". In: *Nucleic acids research* 38.suppl_1 (2009), pp. D366–D370.

[122] Hamming, R. W. "One man's view of computer science". In: *Journal of the ACM (JACM)* 16.1 (1969), pp. 3–12.

[123] GitHub, I. *Build software better, together.* URL: https://github.com (visited on 09/12/2019).

[124] GitLab. *The first single application for the entire DevOps lifecycle.* URL: https://about.gitlab.com/ (visited on 09/12/2019).

[125] Atlassian. *The Git solution for professional teams.* URL: https://bitbucket.org (visited on 09/12/2019).

[126] Foundation, P. S. *The PyPA recommended tool for installing Python packages.* URL: https://pypi.org/project/pip/ (visited on 09/12/2019).

[127] npm, I. *npm is the package manager for javascript.* URL: https://www.npmjs.com (visited on 09/12/2019).

[128] *JBEI Registry: Towards a Distributed Web of Registries.* 2009. URL: https://www.iwbdaconf.org/2009/docs/iwbda2009proceedings.pdf.

[129] Urquiza-García, U., Zieliński, T., and Millar, A. J. "Better research by efficient sharing: evaluation of free management platforms for synthetic biology designs". In: *Synthetic Biology* (2019).

[130] DARPA. *Synergistic Discovery and Design Environment.* URL: https://www.darpa.mil/program/synergistic-discovery-and-design (visited on 09/14/2019).

[131] University, B. *Project to Engineer Cells That Compute Awarded $10M NSF Grant.* Jan. 8, 2016. URL: http://www.bu.edu/eng/2016/01/08/project-to-engineer-cells-that-compute-awarded-10m-nsf-grant-2/ (visited on 09/14/2019).

[132] University, N. *SEVAhub is an instance of the SynBioHub design repository providing access to SEVA-DB converted to the Synthetic Biology Open Language (SBOL).* URL: http://sevahub.es/ (visited on 09/14/2019).

[133] Durante-Rodríguez, G., Lorenzo, V. de, and Martínez-García, E. "The standard European vector architecture (SEVA) plasmid toolkit". In: *Pseudomonas Methods and Protocols.* Springer, 2014, pp. 469–478.

[134] Krebs, O., Wolstencroft, K., Stanford, N. J., Morrison, N., Golebiewski, M., Owen, S., Nguyen, Q., Snoep, J. L., Mueller, W., and Goble, C. A. "FAIRDOM approach for semantic interoperability of systems biology data and models." In: *ICBO.* 2015.

[135] ELIXIR. *FAIR Service Architecture.* URL: `https://elixir-europe.org/about-us/implementation-studies/fair-service-architecture` (visited on 09/14/2019).

[136] Bhatia, S. and Densmore, D. "Pigeon: a design visualizer for synthetic biology". In: *ACS synthetic biology* 2.6 (2013), pp. 348–350.

[137] Gordon, J. *Binary Tree Bin Packing Algorithm.* URL: `https://codeincomplete.com/posts/bin-packing/` (visited on 09/23/2019).

[138] Hunter, J. D. "Matplotlib: A 2D graphics environment". In: *Computing in science & engineering* 9.3 (2007), p. 90.

[139] Ferraiolo, J., Jun, F., and Jackson, D. *Scalable vector graphics (SVG) 1.0 specification.* iuniverse, 2000.

[140] Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., and Kummer, U. "COPASI—a complex pathway simulator". In: *Bioinformatics* 22.24 (2006), pp. 3067–3074.

[141] Myers, C. J., Barker, N., Jones, K., Kuwahara, H., Madsen, C., and Nguyen, N.-P. D. "iBioSim: a tool for the analysis and design of genetic circuits". In: *Bioinformatics* 25.21 (2009), pp. 2848–2849.

A. SEPs

SEP 009 -- SBOL Provenance

SEP	009
Title	SBOL Provenance
Authors	Matthew Pocock (turingatemyhamster@gmail.com), James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk), Goksel Misirli (goksel.misirli@ncl.ac.uk), Owen Gilfellon (o.gilfellon@ncl.ac.uk), Anil Wipat (anil.wipat@ncl.ac.uk)
Editor	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk)
Type	Data Model
SBOL Version	2
Status	Accepted
Created	20-Sep-2016
Last modified	
Issue	#24

Abstract

It is essential to track the provenance of designs, to track attribution and modification. The PROV-O ontology is a mature data model for tracking provenance of digital artifacts. SBOL 2 already uses the `prov:wasDerivedFrom` predicate to allow an `Identifiable` instance to point back to a resource that it was logically derived from. This SEP describes an extended sub-set of PROV-O suitable for capturing a more detailed provenance trail for SBOL top-level entities.

Table of Contents

- 1. Rationale <or, if you prefer, 'Motivation'>
- 2. PROV-O
- 3. Specification
  - [3.1 light-weight provenance] (#lightweight)
  - 3.2 full provenance
- 4. Example or Use Case
- 5. Backwards Compatibility
- 6. Discussion
  - 5.1 discussion point
  - 5.2 discussion point
- 7. Competing SEPs
- References
- Copyright

1. Rationale

Provenance is central to a range of quality control and attribution tasks within the Synthetic Biology design process. Tracking attribution and derivation of one resource from another is paramount for managing intellectual property purposes. Source designs are often modified in systematic ways to generate derived designs, for example, by applying codon optimization or systematically removing all of a class of restriction enzyme sites, and documenting the transformation used, and any associated parameters, makes this explicit and potentially allows the process to be reproduced systematically. If a design has been used within other designs, and is later found to be defective, it is paramount that all uses of it, including uses of edited versions of the design, can be identified, and ideally replaced with a non-defective alternative. When importing data from external sources, it is important not only to attribute the original source (for example, GENBANK), but also the tool used to perform the import, as this may have made arbitrary choices as to how to represent the source knowledge as SBOL. All these activities have in common that it is necessary to track what resource, and what transformation process was applied by whom to derive an SBOL design.

The PROV-O ontology (https://www.w3.org/TR/prov-o/) already defines a data model for provenance and can be adopted to describe these activities.

2. PROV-O

PROV-O defines three core classes: `Entity` , `Activity` and `Agent` . An agent runs an activity to generate one entity from another. Provenance can be captured at two levels of detail using these classes.

![Alt PROV-O data model] (images/sep_009_prov-o.png)

The more abstract level links the generated `Entity` directly to the `Agent` responsible for producing it with the `wasAttributedTo` relation, and to the `Entity` that it was generated from with the `wasDerivedFrom` relation. In the more detailed representation, the generated `Entity` is linked through a `qualifiedDerivation` relation to a `Derivation` . This `Derivation` instance is then linked to an `Entity` that acts as the source in this process. The `Derivation` is also linked to `Activity` instances, which are used to provide more detailed information about activities, such as which `Agent` is used.

Although, PROV-O is an ontology and provides machine access to provenance information, in order to simplify the relationshis between different PROV-O classes, a human redable notation, PROV-N, has also been developed. For simplicity, we will use PROV-N and diagrams to show different example.

PROV-N example below shows how entity2 is derived from entity1 using activity1.

```
entity (entity1)
entity (entity2)
wasDerivedFrom (derivationId; entity2, entity1, activity1)
```

This derivation example can be represented in the RDF/Turtle format as below:

```
:entity1 a prov:Entity.
:entity2 a prov:Entity;
   prov:qualifiedDerivation :derivationId.

 :derivationId a prov:Derivation ;
    prov:entity :entity1;
    prov:hadActivity :activity1 .
```

An `Activity` is used to qualify how an `Agent` was used. An `Activity` is linked through a `qualifiedAssociation` to an an `Association` . An `Association` is linked to an `Agent` through the `agent` link and is linked to a `Role` through the `hadRole` link. The `Activity` can then be annotated with various other predicates to describe when and how the used entity was transformed into the generated one, together with other entities that are used. How different entities are used in an `Activity` is specified using the `Usage` class, which is linked from an `Activity` through the `qualifiedUsage` . A `Usage` is then linked to an `Entity` through the `entity` link and is linked to a `Role` through the `hadRole` link.

The PROV-N example shows an example activity, which was run between 10am and 11am on 9th September 2016. Agent1 with the role of 'modifier' was involved in the activity. The entity1 was used with the'source' role in this activity.

```
activity(activity1,2016-09-14T10:00:00, 2016-09-14T11:00:00)
wasAssociatedWith (associationId; activity1, agent1, - ,
[prov:role=”modifier”])
used (usage1; activity1, entity1, [prov:role=”source”])
```

This activity example can be represented in the RDF/Turtle format as below:

```
:activity1 a prov:Activity ;
   prov:startedAtTime  "2016-09-14T10:00:00"^^xsd:dateTime ;
   prov:endedAtTime  "2016-09-14T11:00:00"^^xsd:dateTime ;
   prov:qualifiedAssociation :associationId ;
   prov:qualifiedUsage :usage1 .

:associationId a prov:Association ;
   prov:agent :agent1 ;
   prov:hadRole :modifier.

:usage1 a prov:Usage ;
   prov:entity :entity1 ;
   prov:hadRole :source.

:modifier a prov:Role.

:source a prov:Role .

:agent1 a prov:Agent.
```

The `wasDerivedFrom` relation can be inferred by following `qualifiedDerivation` then `entity` . The `wasAttributedTo` relation can be inferred by following the relationships between the derived `Entity` and the `Agent` used in the `Activity` .

So in the case that a `Derivation` and a `Acitivity` are explicitly provided, these direct relations are redundant.

Below is a more concrete example about preparing a cheese and tomatoe sandwich. `Bob` , the agent was involved in the `fillingthebread` activity with the `maker` role. `Cheese` and `tomatoe` were `used` in this activity as `fillings` . The `bread` entity was `used` as a `container` .

```
entity (sandwich)
entity (tomatoe)
entity (cheese)
entity (bread)
agent(Bob)

wasDerivedFrom(sandwich,-,fillingthebread)
activity(fillinthebread, 2016-09-14T10:00:00, 2016-09-14T11:00:00)
wasAssociatedWith (fillinthebread, Bob, - , [prov:role=”maker”])
used (fillinthebread, cheese, [prov:role=”filling”])
used (fillinthebread, tomatoe, [prov:role="filling"])
used (fillinthebread, tomatoe, [prov:role="container"])
```

3. SBOL Provenance Proposal

This proposal suggests that all `sbol2:Identified` types to be potentially annotated with provenance information. As such, the rdfs for `sbol2:Identified` would be extended with:

```
sbol2:Identified a prov:Entity
```

Instances of the other following PROV-O classes are also imported into SBOL.

Class
prov:Derivation
prov:Activity
prov:Agent
prov:Association
prov:Usage

3.1 light-weight provenance

The `sbol2:Identified` class already has the `prov:wasDerivedFrom` property in SBOL 2.0. The `prov:wasAttributedTo` is added additionally to support light-weight provenance capture:

Property	cardinality	type	description
prov:wasDerivedFrom	0..1	URI	the `Entity` was derived from this resource
prov:wasAttributedTo	0..1	URI	the `Entity` was derived by this agent

3.2 full provenance

The following data model property is added to `Identified` to support rich provenance:

Property	cardinality	type	description
prov:qualifiedDerivation	0..1	Derivation	the `Entity` was derived by an activity

The `qualifiedDerivation` property contains a `Derivation` instance.

3.2.1 Derivation class

Property	cardinality	type	description
prov:entity	0..1	URI	the `Entity` was used to derive the new entity
prov:hadActivity	1..*	URI	Each hadActivity property points to an `Activity` instance that is used in the derivation of the new entity

3.2.2 Activity class

Property	cardinality	type	description
prov:qualifiedAssociation	1	URI	Points to an `Association` providing information about how an agent was used in the activity
prov:qualifiedUsage	1	URI	Points to a `Usage` providing information about how an `Entity` was used in the activity when deriving the new `Entity`
startedAtTime	0..1	DateTime	the `Activity` started at this time
endedAtTime	0..1	DateTime	the `Activity` ended at this time

When using `Activity` to capture how an entity was derived, it is expected that any additional information needed will be attached to the `Activity` as annotations. This may include software settings or textual notes.

3.2.3 Association class

Property	cardinality	type	description
prov:agent	1	URI	Points to an `Agent`
prov:hadRole	1	URI	Points to a role specifying how the agent was used in the activity

3.2.4 Usage class

Property	cardinality	type	description
prov:entity	1	URI	Points to an `Entity` that is used when deriving the new entity
prov:hadRole	1	URI	Points to a role specifying how the entity was used

3.3 Best practice

This specification does not state what the newly added properties must point to. As long as they are resources that are consistent with the prov property domains, they are legal.

In the special case where the `wasDerivedFrom` or `entity` URI is itself an SBOL `Identified` instance, the types should make sense. For example, a `ComponentDefinition` may be derived from another `ComponentDefinition` , but it would probably not make sense for it to be derived from a `Collection` .

When the `Activity` is the result of running some software, the `Agent` pointed to by `agent` properties should refer to a resource representing the software. This should in turn be annotated with any additional information, such as software version, needed to be able to run the same software again.

Providers of provenance information are free to make use of more of PROV-O than is described here. This specification describes the minimal subset of PROV-O that a provenance-aware SBOL tool must be able to handle. However, it is acceptable for tools that understand more than this subset and use as much as they are able. Tools that only understand this sub-set must treat any additional data as annotations. Tools that are not aware of SBOL provenance at all must maintain and provide access to this information as annotations.

When extending SBOL entities with provenance, instances of `Derivation` , `Activity` , `Usage` and `Association` classes that are imported from PROV-O should be embedded within those SBOL entities. However, agents should be serialized as `sbol2:GenericTopLevel` entities in order to allow their reuse in different `Activity` instances.

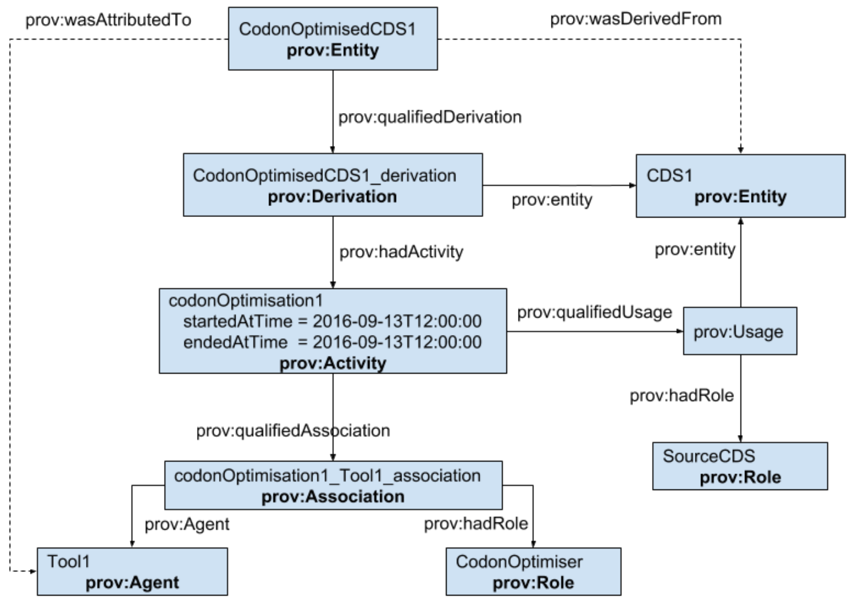
Where possible, the light-weight provenance scheme should be avoided and the proposed full provenance scheme should be used. This would ensure capturing detailed provanence information as much as possible. However, SBOL libraries can generate the light-weight provenance as a read-only attributes in order to simplify the tracking of provenance, and querying of related data.

4. Examples

4.1 Codon optimisation example

As a practical real-world example where provenance predicates can be applied, consider codon optimization of a coding sequence. In the current specification, the relationship between the original CDS and the codon-optimized version could simply be represented using a `prov:wasDerivedFrom` predicate. However, this does not allow for additional information to be attached to the relation: for example, what was the software used to codon-optimize the reading frame, and what parameters were used?

With more comprehensive use of the PROV ontology, the codon optimization can be represented as a `prov:Activity` . A codon-optimized CDS entity and the original CDS entity through a `Derivation` . An `Activity` is linked to this `Derivation` and specifies how the original CDS was used as a source. The `Activity` can then add additional information, such as the `prov:Agent` responsible (in this case, codon-optimizing software). The `wasDerivedFrom` and `wasAttributedTo` predicates can be inferred from the relationships between `Entities` , the `Agent` , through the `Derivation` and the `Activity` .



As an RDF/XML serialization, this example can be represented as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:dcterms
    xmlns:prov="http://www.w3.org/ns/prov#" xmlns:sbol="http://sbols.org/v2#">
    <sbol:ComponentDefinition rdf:about="http://cds/codon-optimized">
        <dcterms:title>Codon Optimized CDS</dcterms:title>
        <prov:qualifiedDerivation>
            <prov:Derivation rdf:about="http://cds/codon-optimized-derivation"
                    <prov:entity rdf:resource="http://cds/non-codon-optimized"
                    <prov:hadActivity>
                            <prov:Activity rdf:about="http://codon-optimizatio
                                <dcterms:title>Codon Optimization Activity</dc
                                <prov:qualifiedUsage>
                                    <prov:Usage rdf:about="http://codon-optimi
                                        <prov:entity rdf:resource="http://cds/
                                        <prov:hadRole rdf:resource="http://sbo
                                    </prov:Usage>
                                </prov:qualifiedUsage>
                                <prov:qualifiedAssociation>
                                    <prov:Association rdf:about="http://codon-
                                        <prov:agent rdf:resource="http://codon
                                        <prov:hadRole rdf:resource="http://sbo
                                    </prov:Association>
                                </prov:qualifiedAssociation>
                            </prov:Activity>
                        </prov:hadActivity>
                </prov:Derivation>
            </prov:qualifiedDerivation>
        </sbol:ComponentDefinition>
        <sbol:ComponentDefinition rdf:about="http://cds/non-codon-optimized">
            <dcterms:title>Non Codon Optimized CDS</dcterms:title>
        </sbol:ComponentDefinition>
        <prov:Agent rdf:about="http://codon-optimization-software">
            <dcterms:title>Codon Optimization Software</dcterms:title>
        </prov:Agent>
    </rdf:RDF>
```

The same information is shown using PROV-N below:

```
wasDerivedFrom (codon-optimized,non-codon-optimized, codon-optimisation-
activity)

activity (codon-optimisation-activity, 2016-9-13T10:00:00, 2016-9-
13T11:00:00)

used (codon-optimisation-activity, non-codon-optimized,
[prov:role="source"])

wasAssociatedWith (codon-optimisation-activity, codon-optimization-software,
- , [prov:role="codonoptimiser"])

agent (codon-optimization-software, [prov:type="prov:SoftwareAgent"])
```

4.2 Deriving strains

Bacterial strains are often derived from parent strains through modifications such as gene knockouts or mutations. *Bacillus subtilis* 168 is a laboratory strain and has several advantages as a model organism in synthetic biology. *B. subtilis* 168 strains are easy to transform and are not motile, facilitating the analysis of engineered cells easily. This strain was derived from the Marburg strain in the 1940s through x-radiation. The Marburg strain, on the other hand, is motile and also more difficult to transform compared to the 168 strain. In this example, how the 168 was derived from the Marburg strain is shown.

```
entity (BacillusSubtilis168, [rdf:type="sbol:ComponentDefinition"])
entity (BacillusSubtilisNcib3610, [rdf:type="sbol:ComponentDefinition"])
wasDerivedFrom(BacillusSubtilis168,BacillusSubtilisNcib3610,
xraymutagenesis)
activity(xraymutagenesis, 1947, -)
wasAssociatedWith (xraymutagenesis, x-ray, - , [prov:role="mutagen"])
```

4.3 Provenance of annotating SBOL entities

It is important to capture how annotations are produced and the track these annotations between different software tools. In this example, a protein component definition is annotated with additional properties. An `Activity` is used to document the software generating the annotations and how this process was carried out. For simplicity, the relationships are only shown in PROV-N:

```
entity(protein_v1, [rdf:type="sbol:ComponentDefinition",
sbol:type="biopax:Protein"])
entity(protein_v2, [rdf:type="sbol:ComponentDefinition",
sbol:type="biopax:Protein"])
wasDerivedFrom(protein_v2,protein_v1, annotationActivity)
activity(ac1, 2016, 2016)
wasAssociatedWith (annotationActivity, pepstat, - ,
[prov:role="calculatedpI", prov:role="addedAnnotations"])
agent(pepstat, [prov:type=prov:SoftwareAgent])
```

4.4 The Provenance of new SBOL entities

It is also important to track how SBOL entities are generated. In this example, a promoter component is searched for restriction sites for which SBOL's 'SequenceAnnotation' entities are created.

```
entity(promoterv1, [rdf:type="sbol:ComponentDefiniton"])
entity(promoterv2, [rdf:type="sbol:ComponentDefiniton",
sbol:sequenceAnnotation=seqAnnotation1])
entity(seqAnnotation1,
[rdf:type="sbol:SequenceAnnotation",sbol:role="so:restrictionSite"])
wasDerivedFrom(promoterv2,promoterv1, activity)
wasAssociatedWith (activity, rtsFinderSoftware, - , [prov:role="predicted
rts", prov:role="annotatedSequence"])
used (activity, seqAnnotation1, [prov:role="generated"])
agent(rtsFinderSoftware, [prov:type=prov:SoftwareAgent])
```

4.5 Creating models for biological parts

In this example, a promoter model is created using two different parameter estimation activities. Each activity uses two different experimental data and parameters are fitted using the COPASI modelling tool. Parameter estimation was initially carried out using the first activity. The second activity was used to refine the results.

```
entity(promoterA, [rdf:type="sbol:ComponentDefiniton"])
entity(experiment1, [rdf:type="sybiont:Experiment"]
entity(experiment2, [rdf:type="sybiont:Experiment"]
entity(experiment3, [rdf:type="sybiont:Experiment"]
entity(experiment4, [rdf:type="sybiont:Experiment"]
wasDerivedFrom(promoterA, - , activity1)
wasDerivedFrom(promoterA, - , activity2)

wasInformedBy(activity2, activity1)

wasAssociatedWith (activity1, COPASI, - , [prov:role="parameterEstimation"])
used (activity1, experiment1, [prov:role="experiment"])
used (activity1, experiment2, [prov:role="experiment"])

wasAssociatedWith (activity2, COPASI, - , [prov:role="parameterEstimation"])
used (activity2, experiment3, [prov:role="experiment"])
used (activity2, experiment4, [prov:role="experiment"])

agent(COPASI, [prov:type=prov:SoftwareAgent])
```

5. Backwards Compatibility

6. Discussion

5.1 discussion point

Summarize discussion, also represent dissenting opinions

5.2 discussion point

7. Competing SEPs

At the time of writing, there are no competing or conflicting SEPs.

References

Copyright

SEP 014 -- Using SBOL to Model the Design-Build-Test Cycle

SEP	
Title	Using SBOL to Model the Design-Build-Test Cycle
Authors	Bryan Bartley (bartleyba@sbolstandard.org), Ali Humphries, James McLaughlin, Goksel Misirli, Matthew Pocock, James Skelton, Angel Goni-Moreno, Chris Myers, Jake Beal, and Anil Wipat
Editor	
Type	Data Model
SBOL Version	2.3
Status	Withdrawn, Replaced by SEP 019
Created	20-Jun-2017
Last modified	19-Jul-2017
Issue	#31

Abstract

Linking experimental data with SBOL designs is becoming critical to a number of important synthetic biology projects. Therefore this SEP introduces a data model for SBOL that supports Design-Build-Test-Learn reasoning. As the synthetic biologist proceeds around the Design-Build-Test-Learn cycle, each iteration generates new understanding that must be integrated at each stage. This SEP will help synthetic biologists close the loop in the Design-Build-Test-Learn cycle.

Table of Contents

- Motivation
- Specification
- Example
- Backwards Compatibility
- Discussion
- Relation to Other Proposals
- References
- Copyright

Motivation

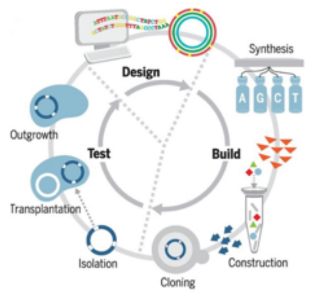
A critical bottleneck for many large-scale projects in synthetic biology is the inability to integrate data through successive stages of the design-build-test lifecycle. Up to now, SBOL has been used exclusively as a language for design, but in order to serve the advancement of synthetic biology, we must now extend SBOL to describe what happens when a design moves out of the designer's imagination and takes form in the lab as a real biological sample. Indeed a design may be realized in the lab as many distinct unique samples with distinct histories and biological variations. After building a design, experimentalists may run quality control and performance tests on their biological constructs. These procedures generate large volumes of data that can be difficult for a synthetic biologist to organize, track, and manage without automated tools. The SBOL standard could make a big impact in synthetic biology by implementing support for design-build-test automation.

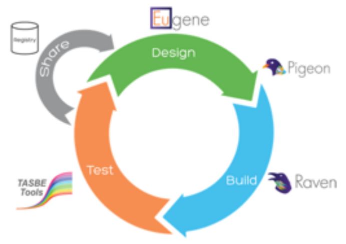
Many lines of evidence suggest that it is time for the SBOL community to act on this if it wants to stay relevant. Recently, one SBOL developer brought attention to this issue, because linking experimental data with designs is becoming "critical" to his projects, such as inter-lab measurement standardization. In our own lab, linking designs with data about actual instances (or "clones") of plasmid constructs has also been an issue. From outside the SBOL community, our Chair has also been hearing from audiences about this problem at many international synthetic biology conferences.

The field of synthetic biology borrows from more conventional engineering fields a framework for problem-solving known as the design-build-test cycle. The design-build-test cycle is essentially the scientific method applied in the context of engineering. Stages in the cycle include designing an inital prototype, testing that prototype, analyzing its performance against specific metrics, learning what worked and what did not work, designing a new prototype based on what was learned, and completing the cycle again. Ideally each cycle generates new understanding that feeds back into new cycles as alternative approaches or reformulated problems. Thus, this problem-solving strategy is also sometimes called the design-build-test-learn cycle.

The design-build-test cycle is especially important in the context of synthetic biology because trial and failure is an essential part of the process. Due to the unpredictability of biological systems, many prototypes (biological variants) of a genetic construct are often tested before arriving at the desired system behavior. Thus, synthetic biologists must routinely manage many cycles of designing, building, and testing genetic systems. The biodesign automation community was thus born out of a need to connect collaborators at different stages of synthetic biology with automated workflows. The synthetic biology cycle includes stages like mining DNA parts from online databases, assembling new genetic programs from DNA sequences, synthesizing and assembling DNA, quality control, quantitative characterization of a DNA part's encoded behavior, and submitting characterized parts to inventories so that other engineers may reusethem.

The origins of the design-build-test framework for engineering can be traced back to a business practice called product life-cycle management. In the 1980s the American Motor Corporation began developing a business practice called product life-cycle management (PLM) to help teams of engineers collaborate. Their automated PLM systems supported design-build-test workflows for engineers using off-the-shelf components, computer-aided design tools, and networked databases containing design specifications and documentation [1, 2]. The design-build-test cycle appears frequently in literature about PLM from the 1990s[1, 3]. Later, in the early 2000s, the design-build-test problem-solving framework became incorporated into educational pedagogy, especially in the context of engineering education[4, 5]. Nowadays, the design-build-test problem-solving framework has become a fundamental principle for synthetic biology [6, 7]. Two conceptualizations of this are shown below.

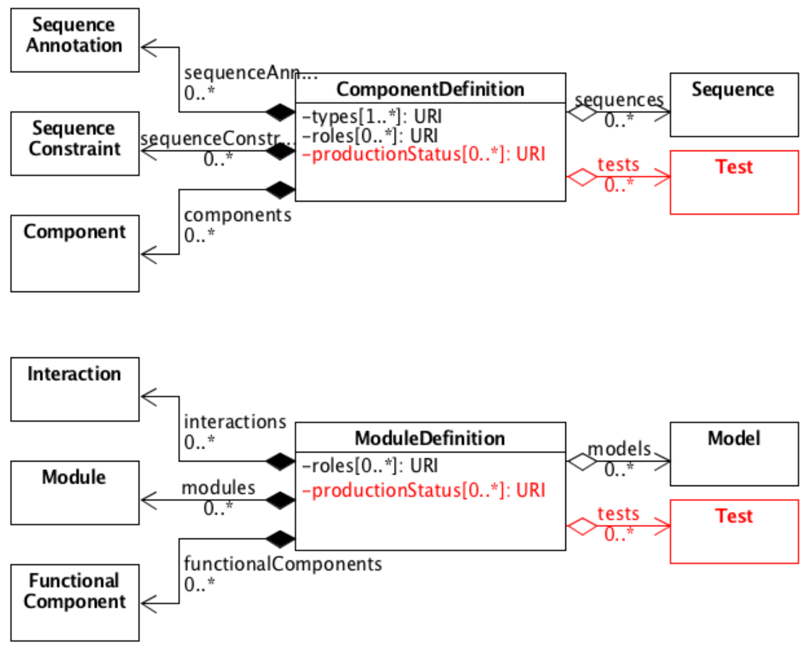




As the SBOL Chair recently observed (on 6/17/17), SBOL does not distinguish between designs that have been fabricated as DNA or subsequently tested. The design-build-test data model introduced in this SEP makes these stages explicit, and can be broadly understood inside and outside the SBOL community. As we expand the SBOL data model, it is important that we use simple and easy-to-understand concepts in order to promote broad interest and adoption.

Specification

This specification describes how the design-build-test cycle for synthetic biology may be represented using SBOL. Proposed changes to the data model include the introduction of two new properties, `productionStatus` and `tests`, which will be added to both `ComponentDefinition` and `ModuleDefinition`. The same properties must be duplicated in both classes in order to maintain symmetry between structural and functional layers of representation. In addition, a new class called `Test` is introduced. The `Test` class contains an external link to data files and some meta-data about those files. This specification clarifies where quality control and functional characterization data should be placed. Test data for quality control (eg, sequencing data) should be linked from a `ComponentDefinition` while functional data, (eg, flow cytometry) should be linked from a `ModuleDefinition`. Thus, this SEP establishes links between SBOL designs and experimental data.



Indicating the Production Status

The `productionStatus` property of a `ModuleDefinition` or `ComponentDefinition` MAY optionally contain one of the following ontology terms, as defined in the table below. In the rest of this specification, we use `design` and `build` as shorthand to refer to any `ComponentDefinition` or `ModuleDefinition` that contains the corresponding URI in its `productionStatus` field.

productionStatus	Description
http://sbols.org/v2#design	Objects with this term represent the designer's *conceptual* rendering of a synthetic construct expressed in SBOL
http://sbols.org/v2#build	Objects with this term describe a *real* biological sample or physical artifact that currently exists or existed previously

If no term is given, the object is assumed by default to represent a `design`. Hence, SBOL objects from versions prior to 2.3 will be grandfathered in as `designs`. In the future, SBOL developers may wish to add additional terms, or adopt extant ontologies, to describe different stages of production.

For a given design many builds may be generated. In general, the design should serve as a reference to which builds are compared either for quality control (sequence verification) or comparison of observed versus expected output (experimental data vs. model predictions). Therefore, as a best practice, a user SHOULD NOT recursively copy all the Components and Modules which describe the compositional hierarchy of a design over to each new build generated, as this would be inefficient and redundant. A build object SHOULD be a simple `ComponentDefinition` or `ModuleDefinition` containing no subparts. If a `build` is augmented with any annotations or compositional elements, these should be interpreted as an empirical description of reality that describes the corresponding physical symbol. For example, one might add `Component` substructure to a `build` to indicate structural reorganization of a plasmid due to an unexpected recombination error.

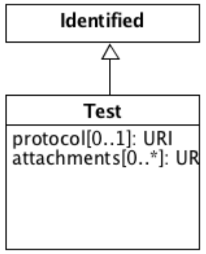
A `design` and `build` may be linked by `wasDerivedFrom` relationship. A `design` and `build` may also be linked by PROV-O classes. This link however requires several degrees of indirection through `Activity` and `Usage` classes and may not be very intuitive for some users. An experimental protocol can be captured by a Plan object and the person who performed the protocol may be represented by an Agent. However, the preferred ontology terms and other fine details about how to represent experiments with PROV-O is still open for discussion and outside the scope of this SEP.

Linking Experimental Data with `Test` Objects

The `Test` class is simply a wrapper object that provides a link to an external data file, with analogy to the `Model` class. A `Test` class represents empirical data, while a `Model` represents a simulation. For purposes of knowledge representation, the `Test` and `Model` concepts are considered distinct stages of production (see Example 3) in synthetic biology workflows. However, the specification of `Test` and `Model` classes have some similarity because they both represent links to external files with meta-data about those files. Thus, the specification of these classes are similar, leading some in the SBOL development community to suggest abstracting out a general `Attachment` class to represent external data files. Specification of an `Attachment` class is beyond the scope of this SEP, but this proposal is compatible with that vision, as discussed in Relation to Other Proposals.

`Test` is subclassed from `Identified`. A `Test` object contains two properties:

- `protocol` : The protocol contains a single URI which, in the simplest case, may refer to a file containing a written laboratory protocol. Alternatively, the URI may point to an instance of a provenance class, such as Plan or Activity. This specification is intentionally vague to allow users to experiment.
- `attachments` : A list of one or more data files that may be associated with this test. Data files might include .csv, .xlsx, .fcs, .abi, .png, etc. At this stage, tooling will have to infer the file type from a file extension. In the future, the URIs contained in this property may refer to `Attachment` objects.



Currently this specification does not require any further meta-data about the attachments, but this is discussed in Relation to Other Proposals.

Validation Rules

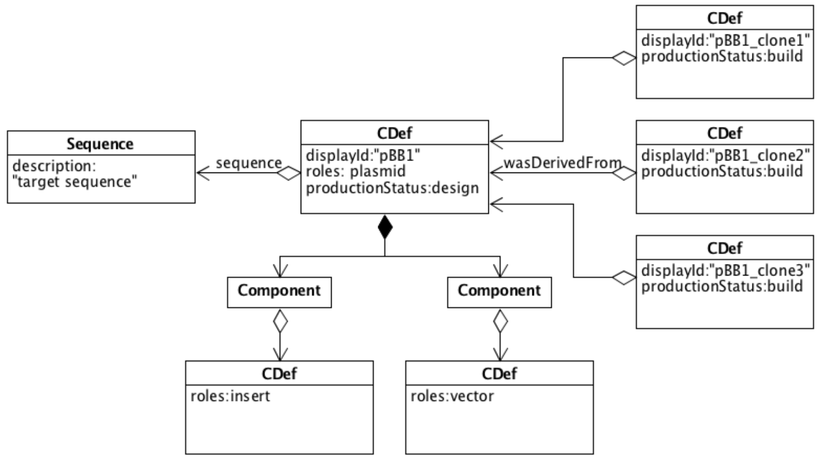
- Objects in different stages of production (`design` vs `build`) MUST be related through PROV-O annotations. In other words, a `build` object MUST link back to a single `design` object, either through a simple `wasDerivedFrom` relationship, or through a more detailed provenance using Activity, Agent, and other classes to describe the assembly process.
- The flag `design` or `build` MUST be assigned to the highest level object in an abstraction hierarchy (defined through `component` or `module` subpart relationships). This is where client tools should look for the term. It is OPTIONAL if other subparts in an abstraction hierarchy have a `productionStatus` specified.
- Objects in different stages of production (`design` vs `build`) MUST NOT be mixed with each other in hierarchical compositions defined through `modules` and `components` subpart relationships.
- If the SBOL object at the highest level of a compositional hierarchy does not have a `productionStatus` , it is assumed to be `productionStatus: design` .
- A `ModuleDefinition` with `productionStatus:design` MUST NOT link to a `Test` object. In other words, the `tests` property MUST BE empty. It is appropriate for a `design` to link a `Model` , however.
- A `ModuleDefinition` with `productionStatus:build` MUST NOT link to a `Model` object. In other word, the `models` property MUST BE empty. It is appropriate, however, for a `build` to link to a `Test` .

Best Practices

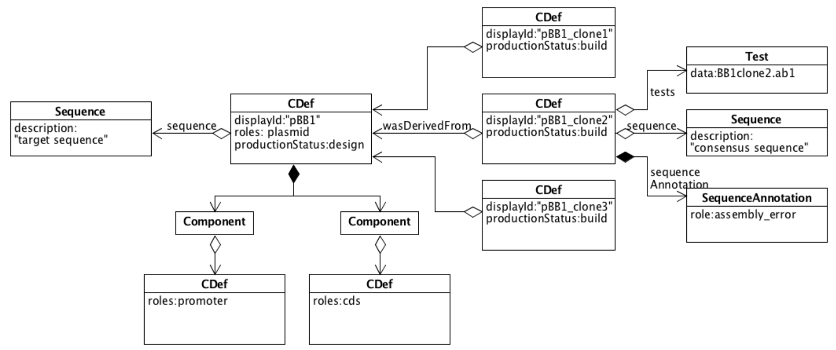
- A `design` should be expect to describe an abstraction hierarchy of `Modules` or `Components` related through `modules` and `components` subpart relationships. In contrast, a `build` MAY be a flat data structure. This is because it is redundant and inefficient to copy the full abstraction hierarchy for each `build` generated. A `design` may generate a large number of builds.

Example

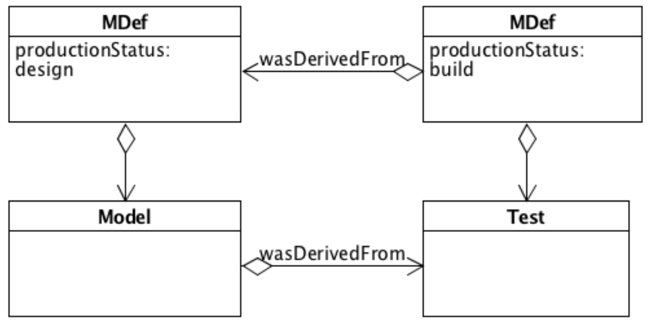
For a given `design` many `builds` may be generated. In this example an abstraction hierarchy of `Components` is associated with the `design` . The following shows an SBOL data structure representing a plasmid construct consisting of an insert and vector. Three distinct clones of this plasmid are represented.



Quality control data such as a sequencing trace file can be linked via a `Test` object. Sequence verification, no matter what the sequencing platform, generates a consensus `Sequence` for the `build` . The `build` may then be augmented with quality control `SequenceAnnotations` .



Some conceptualization of the Design-Build-Test cycle include a fourth step, Learn. Thus information gained through the process feeds back and informs new designs. To close the loop in the Design-Build-Test-Learn cycle, `Test` and `Model` must be connected. A `Test` may be linked to a `Model` by Provenance classes describing the data reduction and fitting procedure used to generate a Model.



Backwards Compatibility

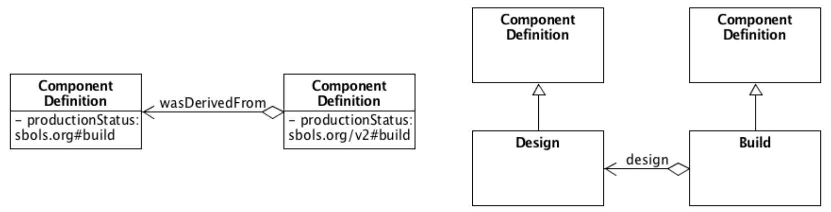
This SEP does not affect backwards compatibility.

To ensure reverse compatibility, all `ComponentDefinitions` are assumed to be a `design` unless explicitly indicated that is a `design` . Thus any `ComponentDefinitions` current populating SBOL repositories such as JBEIR-ICE and SynBioHub should not be affected. Any `ComponentDefinitions` lacking a `productionStatus` property are grandfathered in as `design` objects.

Discussion

In the motivation section, I have included links to the original thread which motivated this SEP.

This proposal was robustly debated and discussed at the HARMONY workshop at University of Washington, Seattle, June 2017. Discussion was moderated according to formal guidelines which seemed to be successful in encouraging expression of different philosophical views and constructive problem-solving. In principle, all present consented to the shared view that the SBOL standard should support automation of design-build-test workflows.



Based on group feedback, the Specification proposed here has been modified from the original SEP. In the original, `Design` and `Build` were explicitly represented by classes (subclassed from `ComponentDefinition`), while this modified proposal introduces internal SBOL ontology terms (`http://sbols.org/v2#design` and `http://sbols.org/v2#build`). The original proposal did not allow a `ModuleDefinition` to be distinguished as a design or build, while this proposal allows both `ComponentDefinitions` and `ModuleDefinitions` to be assigned a term indicating their production status. This was the most important deciding factor for the current approach, though other factors were considered.

Our discussion searched out a compromise balance between pragmatism and idealism. Pragmatists value easy tool migration and minimal modifications to the data model, while idealists favor more explicit knowledge-representation and formalized approaches to ontology development. Ultimately these philosophical differences were bridged by a mutual desire for compromise and progress.

The original proposal requires introducing new classes which presents some implementation challenges for library developers. In addition SynBioHub and ICE repositories already contain `ComponentDefinition` objects which would have to be redefined as `Design` or `Build` objects. Thus the introduction of two new terms `http://sbols.org/v2#design` and `http://sbols.org/v2#build` into the SBOL ontology provides a more expedient migration path that does not require implementing new classes.

Thus, this proposal introduces new ontology terms rather than new classes, which also presents challenges worth discussing. The minimalist approach, which requires no changes to the data model, would be to put these new terms in the `ComponentDefinition::type` property, an approach similar to that taken for SEP 011 -- Specify DNA / RNA topology with type fields.

- Overloading an existing field with too many different ontologies may be difficult for tooling to interpret and the semantics may become confusing for programmers
- If we choose to overload the `types` field, we would have to add a `types` field to `ModuleDefinition` (to maintain symmetry between structural and functional levels of representation)
- We shouldn't use the `roles` field either because that is used by client applications to determine the SBOL Visual symbol for display
- In other CAD standards, information about a product's development status is fundamental [8]
- In the field of ontology development, the difference between Concept vs Reality is fundamental
- Having a dedicated field for production status makes sense, as we may later want to introduce more granularity in describing stages of production, and there may be extant ontologies in the manufacturing domain that represent production stages

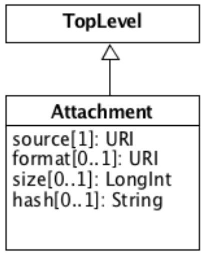
Based on our best synthesis of all the views and concerns express, we, the authors, conclude that a new field indicating `productionStatus` should be added to both `ComponentDefinition` and `ModuleDefinition`.

It is also necessary to link data. Since quality control data, ie, DNA sequencing data, describes the structural features and manufacturing outcomes, it is appropriate to attach these data to ComponentDefinitions. Other data containing characterization data of functional behavior are more appropriate to attach to ModuleDefinitions. Therefore, we conclude that a property called `tests` should be added to both `ComponentDefinition` and `ModuleDefinition` which contains URIs to corresponding `Test` objects. This ensures consistency and symmetry between the structural and functional layers of representation in the SBOL data model.

Discussion about where to put data describing `Context` was also important. Since `Context` is a broad concept, we discussed different kinds of `Context` such as `Host` and `Environment`. The reason that Context enters into this dicussion is because it is necessary to specify between an intended `Context` for a `design` versus the actual `Context` in which a synthetic biological construct is actually built and deployed. As a simple example, the repressilator was originally designed for the MG1655Z1 strain of *E. coli* while a researcher may choose to evaluate its behavior in another organism entirely, such as *Lactobacillus*.

Relation to Other Proposals

This SEP was devised to be consistent with future plans for developing a general `Attachment` class. `Attachment` objects would wrap external file links and provide meta-data that helps client tooling interface with the files. In anticipation of future plans, this SEP does not specify any metadata about files referenced by a `Test` class. That's because a constituency of SBOL developers now see justification for specifying a general `Attachment` class that includes meta-data about external data files. Some of the



proposed propoerties of this class are as follows:

- `source` : The attached file
- `format` : An optional URI indicating the data format. The EDAM ontology, which is currently used to indicate `Model.language` , extends beyond modeling formats and includes a variety of experimental data formats as well.
- `size` : The byte size of the linked file. It is especially helpful to know this when operating over networks with large files.
- `hash` : A file hash used by databases like SynBioHub for caching files

Currently, specification of `Attachment` is beyond the scope of this proposal. An issue which is still open with regard to `Attachments` is where they belong in the data model. For example, some have argued that it would be helpful to attach to `Collections` . In the SynBioHub repository, `Attachments` (a generic annotation class used internally) can be associated with any object. In a future proposal, we recommend that `Attachment` be added as a new `TopLevel` class. In the future a `Test` could refer to `Attachment` objects through its `attachments` property. Thus this SEP is expected to be consistent with future development of an `Attachment` class.

References

[1] Gerald I Susman. Integrating design and manufacturing for competitive advantage. Oxford University Press, 1992

[2] S Hill. How to be a trendsetter: Dassault and ibm plm customers swap tales from the plm front. COE Newsnet, 2003.

[3] Wheelwright SC, Clark KB. Accelerating the design-build-test cycle for effective product development. International Marketing Review. 1994 Feb 1;11(1):32-46.

[4] Elger DF, Beyerlein SW, Budwig RS. Using design, build, and test projects to teach engineering. InFrontiers in Education Conference, 2000. FIE 2000. 30th Annual 2000 (Vol. 2, pp. F3C-9). IEEE.

[5] Hermon P, McCartan C, Cunningham G. Group design-build-test projects as the core of an integrated curriculum in product design and development. engineering education. 2010 Dec 1;5(2):50-8.

[6] Hutchison CA et al. `Design` and synthesis of a minimal bacterial genome. Science. 2016 Mar 25;351(6280).

[7] http://2014.igem.org/Team:BostonU/Chimera

[8] Sudarsan Rachuri, Eswaran Subrahmanian, Abdelaziz Bouras, Steven J Fenves, Sebti Foufou, and Ram D Sriram. Information sharing and exchange in the context of product lifecycle management: Role of standards. Computer-Aided Design, 40(7):789–800, 2008.

Copyright

SEP 015 -- Simplification of SBOL class names

SEP	015
Title	Simplification of SBOL class names
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk); Raik Grünberg (raik.gruenberg@gmail.com)
Editor	
Type	Data Model
SBOL Version	3.0
Replaces	
Status	Draft
Created	28-Jun-2017
Last modified	28-Jul-2019
Issue	#32

Abstract

The number of classes in the SBOL data model grew sharply with the introduction of SBOL 2.0. Prior to generalizations such as turning DnaComponent into ComponentDefinition, many discussions took place about the nomenclature, and eventually a quorum was reached and the class names were accepted.

SBOL2 has been released for some time now, and the community is now familiar with using the data model in practice. This SEP suggests that, given our experience actually using SBOL2, we reconsider the nomenclature once more for SBOL3.0. The intention would be to simplify names and to convey meaning more effectively.

1. Motivation

SBOL2 can be very confusing to newcomers, and the verbosity of its class names doesn't help. The most common class in SBOL is likely `ComponentDefinition` , which is a verbose name usually abbreviated to "CD" or simply "component". Confusingly though, SBOL 2.x also defines a class `Component` which, however, actually describes a subcomponent. Similar issues exist with `ModuleDefinition` and `Module` .

The rationale for this SEP, using Component as an example, are that:

- In SBOL2, `Component` *must* be contained by a `ComponentDefinition` and is therefore *always* a subcomponent. Changing the name to `SubComponent` is **more expressive**.
- This change leaves the name `Component` free for `ComponentDefinition` to be renamed to. Removing "Definition" makes the name **less verbose** and **less confusing**. The simplified name `Component` matches what programmers currently use in colloqial conversations about SBOL.
- In programming languages, classes are definitions *implicitly* and do not need to be explicitly named as such. For example, the `File` class in Java is called `File` , not `FileDefinition` , even before an instance of it has been created. Bringing SBOL in line with this makes SBOL **more intuitive**.

The `FunctionalComponent` class, though also confusingly named, will remain as-is for the purposes of this SEP.

2. Specification

The following classes will be renamed:

- `ComponentDefinition` to `Component`
- `Component` to `SubComponent`
- `ModuleDefinition` to `Module`
- `Module` to `SubModule`
- `definition` to `instanceOf`

3. Backwards Compatibility

This is a major change, but only to the names of classes. (Note: a separate SEP will deal with removing FunctionalComponent). Consequently, updating the libraries should be straightforward. It is important to defer this change to SBOL 3.0 so that the namespace is different (otherwise, it would be ambiguous whether a Component was an SBOL 2.0 Component instance or a SBOL 3.0 Component).

More generally, tools solely programmed for SBOL 2.x will, in any case, run into trouble when opening an SBOL 3.0 document. SBOL2 documents use the URI prefix `http://sbols.org/v2#` for terms. Presumably, SBOL3 will use the `http://sbols.org/v3#` prefix, which will prevent SBOL2 and SBOL3 from clashing.

4. Discussion

5.1 Related SEPs

The following SEPs make complementary suggestions for further simplification of the SBOL data model:

- SEP 10 (Issue) -- a proposal to separate sequence feature annotations from sub-component (part) compositions
- SEP 25 (Issue) -- merge Module(Definition) with Component(Definition) and remove FunctionalComponent

5.2 Implementation cost

Library developers may be concerned at the time cost of implementing such a significant change. However, as the proposed change is only to the nomenclature and not the semantics, it should generally be no less complicated than a find/replace operation.

5.2 Distinction between SBOL 2.x and 3.x documents

SBOL2 documents use the URI prefix `http://sbols.org/v2#` for terms. Presumably, SBOL3 will use the `http://sbols.org/v3#` prefix, which will prevent SBOL2 and SBOL3 from clashing. This has yet to be discussed.

6. Competing SEPs

None.

References

Copyright

SEP 019 -- Using SBOL to model the Design-Build-Test-Learn Cycle

SEP	
Title	SEP 019 -- Using SBOL to model the Design-Build-Test-Learn Cycle
Authors	Bryan Bartley (bartleyba@sbolstandard.org), Jake Beal (jakebeal@gmail.com), Raik Gruenberg (raik.gruenberg@gmail.com), James McLaughlin (j.a.mclaughlin@newcastle.ac.uk), Chris Myers (myers@ece.utah.edu), Nicholas Roehner (nicholasroehner@gmail.com), and Anil Wipat (anil.wipat@newcastle.ac.uk)
Editor	Nicholas Roehner (nicholasroehner@gmail.com)
Type	Data Model
SBOL Version	2.3
Replaces	SEP 014, SEP 016, SEP 017, SEP 018
Status	Accepted
Created	10-Nov-2017
Last modified	11-Dec-2017
Issue	#43

Abstract

Linking experimental data with SBOL designs is becoming critical to a number of important synthetic biology projects. Therefore this SEP introduces a data model for SBOL that supports Design-Build-Test-Learn reasoning. As the synthetic biologist proceeds around the Design-Build-Test-Learn cycle, each iteration generates new understanding that must be integrated at each stage. This SEP will help synthetic biologists close the loop in the Design-Build-Test-Learn cycle.

Table of Contents

- 1. Rationale
- 2. Specification
  - 2.1 Implementation
    - 2.1.1 Implementation.built
  - 2.2 Design, Build, Test, and Learn
  - 2.3 Best Practices
    - 2.3.1 Usage Roles
    - 2.3.2 Versioning versus Provenance Semantics
  - 2.4 Validation Rules
- 3. Examples
  - 3.1 Use Cases
  - 3.2 Design-Build-Test-Learn Provenance
  - 3.3 Assembly Provenance
  - 3.4 Implementation
  - 3.5 Attachment
- 4. Backwards Compatibility
- 5. Discussion
- 6. Relation to Other SEPs
- References
- Copyright

Rationale

- Decouple the design-build-test-learn process, according to the foundational principles for engineering biology.
- Specify where to add experimental data
- Provide an SBOL representation of biological instances of a design that can link to LIMS systems
- Provide clear guidance for using PROV-O with illustrative examples.
- Capture workflow provenance, a description of the events of a workflow, which is crucial for scientific reproducibility
- Support model-based design.

This SEP was initiated in response to the ["Design-Build-Test" thread] on sbol-dev.

Specification

Here we propose to add a new class, `Implementation` , to allow users to represent physical realizations of biological designs. For example, an `Implementation` could be used to represent an aliquot of DNA, a cell clone, or a lysate. The `Implementation` class is meant to serve as a connection point between theoretical designs of biological systems and descriptions of their actual structure and/or function following their physical construction.

Here we also define four new SBOL ontology terms (`sbol:design` , `sbol:build` , `sbol:test` , and `sbol:learn`) to serve as values for the `hadRole` properties of the PROV-O classes `Usage` and `Association` . In addition, we present tentative best-practice validation rules for the use of these ontology terms and PROV-O to represent design-build-test-learn cycles. These best practices will undergo continuing development to support new use cases as they arise.

2.1 Implementation

An `Implementation` represents a real, physical instance of a synthetic biological construct which may be associated with a laboratory sample. An `Implementation` describes the thing which was built during the Build phase of a D-B-T-L workflow. An `Implementation` may be linked back to its original design (either a `ModuleDefinition` or `ComponentDefinition`) using the `wasDerivedFrom` and/or `wasGeneratedBy` properties inherited from the Identified superclass. An `Implementation` may also link to a `ModuleDefinition` or `ComponentDefinition` that specifies its actual realized structure and/or function as described in Section2.1.1.

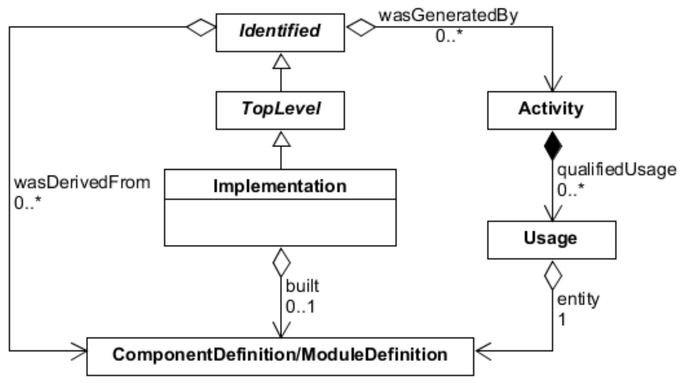


Figure 1: Diagram of the `Implementation` class and its associated properties

2.1.1 Implementation.built

The `built` property is OPTIONAL and MAY contain a URI that MUST refer to a `TopLevel` object that is either a `ComponentDefinition` or `ModuleDefinition` . This `ComponentDefinition` or `ModuleDefinition` is intended to describe the actual physical structure and/or functional behavior of the `Implementation` . When the `built` property refers to a `ComponentDefinition` that is also linked to the `Implementation` via PROV-O properties such as `wasDerivedFrom` , it can be inferred that the actual structure and/or function of the `Implementation` matches its original design. When the `built` property refers to a different `ComponentDefinition` , it can be inferred that the `Implementation` has deviated from the original design. For example, the latter could be used to document when the DNA sequencing results for an assembled construct do not match the original target sequence.

2.2 Design, Build, Test, and Learn

The ontology terms `design` , `build` , `test` , and `learn` are OPTIONAL values of the `hadRole` properties of the `Usage` and `Association` classes. These properties describe how entities (such as samples, data, or models) are used in an `Activity` and what role an `Agent` or `Plan` (such as a person, software tool, or protocol) plays in an `Activity` , respectively.

In natural language, these terms indicate the following:

- "design" describes a process by which a conceptual representation of an engineer's imagined and intended design is derived, possibly from a model or pre-existing design.
- "build" describes a process by which a sample is derived in accordance with a conceptual design, often from other samples.
- "test" describes a process by which raw data or observations are derived via experimental measurement of samples.
- "learn" describes a process by which a theoretical model, analysis, datasheet, etc. is derived, usually from experimental data or another model.

2.3 Best Practices

2.3.1 Usage Roles

To facilitate data exchange across different domains of synthetic biology, a user MAY use one of the terms "design", "build", "test", or "learn" to specify `Usage` roles.

A user MAY also specify additional Usage roles that correspond to their own home-made ontologies for specifying recipes and protocols.

2.3.2 Versioning versus Provenance Semantics

A new object which is the product of an `Activity` links back to the `Activity` which generated it via the `wasGeneratedBy` field. By the W3 PROV-O specification, generation is defined as:

> `...the completion of production of a new entity by an activity. This entity did not exist before generation and becomes available for usage after this generation.`

Provenance semantics are somewhat different from the versioning semantics defined in the SBOL specification. The SBOL specification defines a new version of an object as an update of a previously published object (and therefore a previously existing object). In contrast, an SBOL object which is "generated" from another SHOULD BE regarded as a new entity, not a new version by the PROV-O specification above. However, this distinction is somewhat subjective (see Theseus paradox). Therefore we RECOMMEND as a best practice that objects linked by Activities not be successive versions of each other, though we leave this at the discretion of users and library developers.

2.4 Tentative Validation Rules

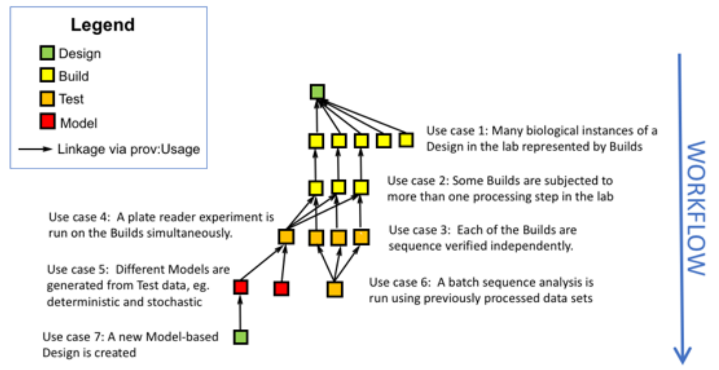
A design-build-test-learn process would typically generate new SBOL objects in the order of one or more `ModuleDefinitions` and/or `ComponentDefinitions` , followed by one or more `Implementations` , followed by one or more `Collections of Attachments` , followed by one or more `Models` . This order of operations (among others) is compatible with the following best-practice validation rules:

- An `Activity` that contains an `Association` of role "design" SHOULD be referred to by the `wasGeneratedBy` property of at least one `ModuleDefinition` , `ComponentDefinition` , or `Sequence` . If this `Activity` contains one or more `Usage` objects, then at least one of them SHOULD be of role "learn" or "design".
- An `Activity` that contains an `Association` of role "build" SHOULD be referred to by the `wasGeneratedBy` property of at least one `Implementation` . If this `Activity` contains one or more `Usage` objects, then at least one of them SHOULD be of role "design" or "build".
- An `Activity` that contains an `Association` of role "test" SHOULD be referred to by the `wasGeneratedBy` property of at least one `Collection of Attachment` objects. If this `Activity` contains one or more `Usage` objects, then at least one of them SHOULD be of role "build".
- An `Activity` that contains an `Association` of role "learn" SHOULD be referred to by the `wasGeneratedBy` property of at least one `Model` , `Collection of Attachment` objects, `ModuleDefinition` , `ComponentDefinition` , or `Sequence` . If this `Activity` contains one or more `Usage` objects, then at least one of them SHOULD be of role "test".
- A `Usage` of role "design" SHOULD refer to a `ModuleDefinition` , `ComponentDefinition` , or `Sequence` .
- A `Usage` of role "build" SHOULD refer to an `Implementation` .
- A `Usage` of role "test" SHOULD refer to a `Collection of Attachment` objects.
- A `Usage` of role "learn" SHOULD refer to a `Model` .

Examples

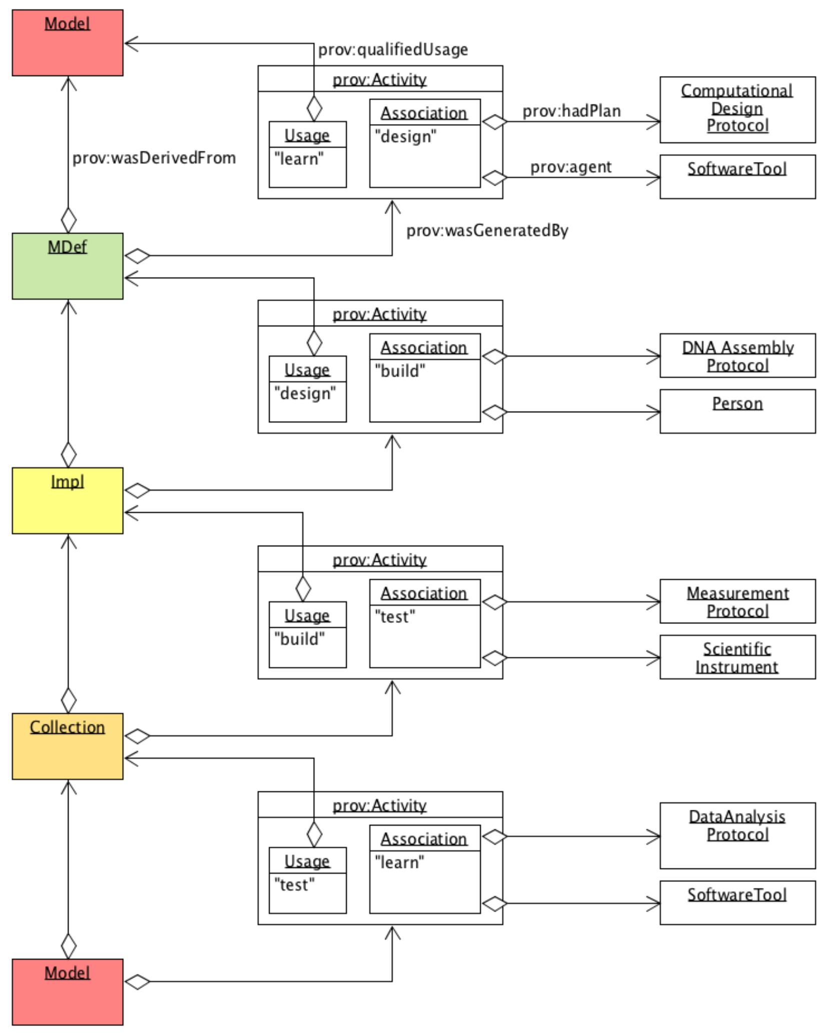
See Discussion for an in-depth explanation of these examples.

3.1 Use Cases


Legend
🟩 Design
🟨 Build
🟧 Test
🟥 Model
Linkage via prov:Usage
Use case 1: Many biological instances of a Design in the lab represented by Builds
Use case 2: Some Builds are subjected to more than one processing step in the lab
Use case 3: Each of the Builds are sequence verified independently.
Use case 4: A plate reader experiment is run on the Builds simultaneously.
Use case 5: Different Models are generated from Test data, eg. deterministic and stochastic
Use case 6: A batch sequence analysis is run using previously processed data sets
Use case 7: A new Model-based Design is created
WORKFLOW

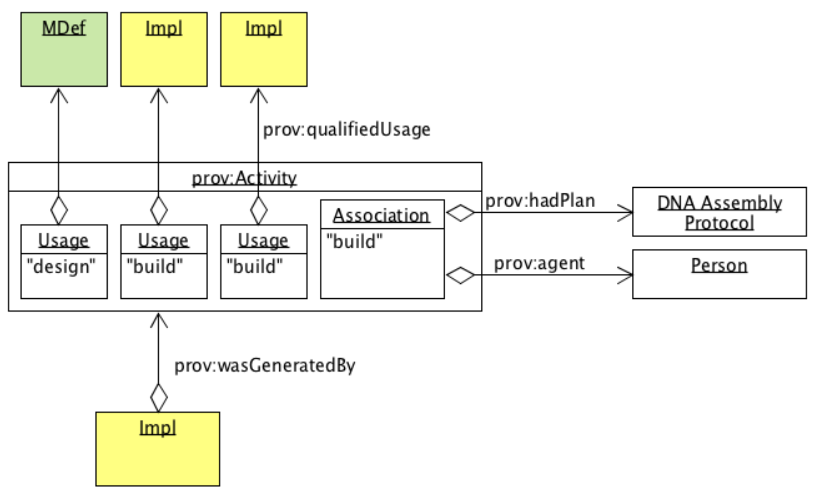
Example 1: A hypothetical workflow for model-based design that demonstrates a variety of use cases which are supported by this proposal

3.2 Design-Build-Test-Learn Provenance



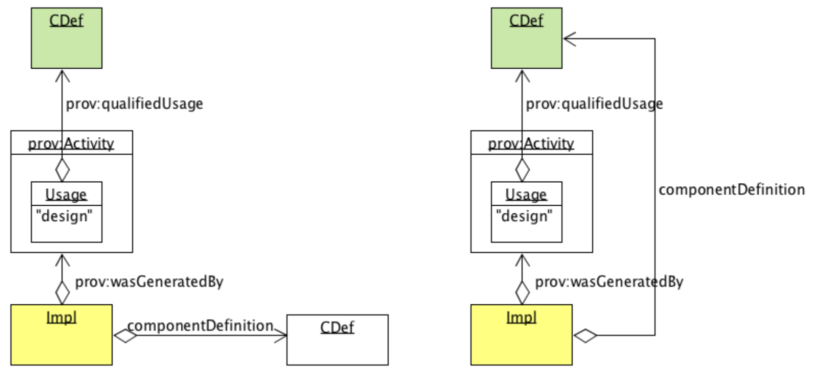
Example 2: A reproducible workflow for model-based design represented using PROV-O classes and annotations

3.3 Assembly Provenance



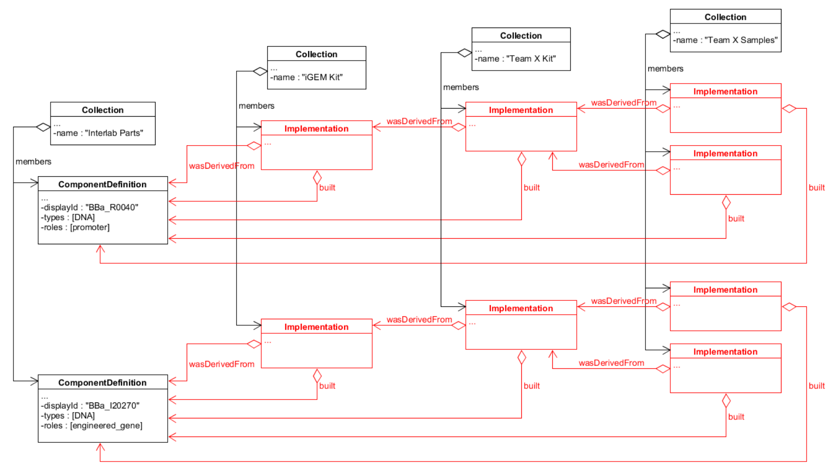
Example 3: A detailed representation of assembly provenance. Assembly provenance allows a synthetic biologist to track all the physical components or samples which go into a DNA assembly protocol.

3.4 Implementation



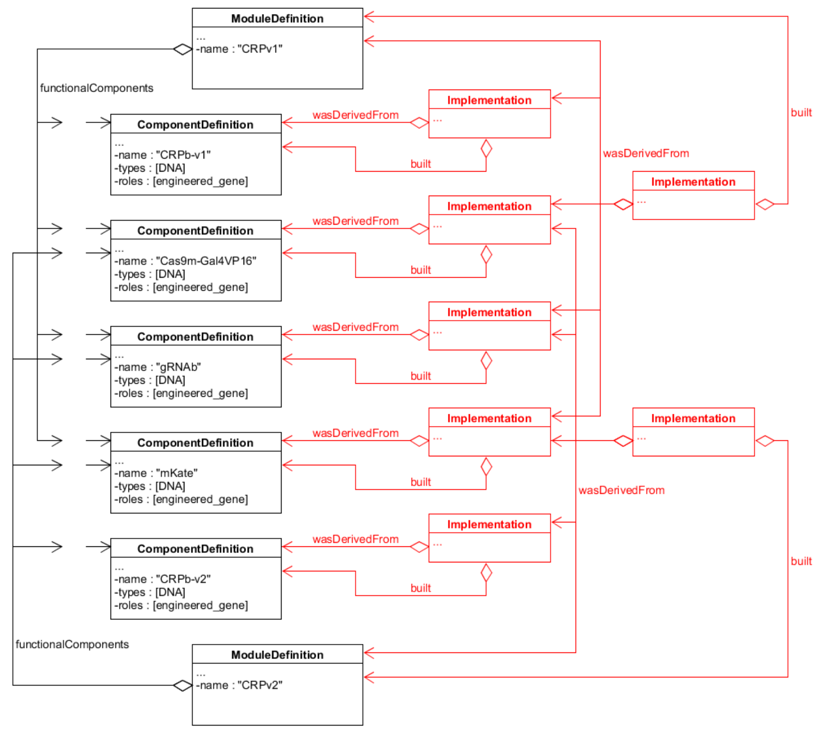
Example 4: An Implementation links a design ComponentDefinition to its build. In the left example, the build does not match its design specification. In the right hand example, the build does match its specification.

3.5 Derivation of Biological Replicates for iGEM Interlab Study



Example 5: An initial sample kit based on part designs from the iGEM Registry is replicated and split into different samples.

3.6 Co-Transfection of Constructs for CRISPR Repression Module



Example 6: Two sets of overlapping constructs that implement the components of a CRISPR-based circuit are transfected to implement two different versions of the circuit.

Backwards Compatibility

This proposal does not affect backwards compatibility.

Discussion

5.1 Design-build-test-learn

The design-build-test-learn cycle is a common theme in synthetic biology and engineering literature. The cycle is a generalized abstraction of an ideal engineering workflow and approach to problem-solving. Thefefore, the design-build-test-learn cycle is a de facto ontology upon which to base an SBOL data model for workflow abstraction. Other workflow activities in synthetic biology, such as analyzing, modeling, verifying, and evolving, by and large fit into the design-build-test-learn abstraction.

This specification enables synthetic biologists to capture workflow provenance with SBOL. The aim is to capture a complete description of evaluation and enactment of computational and laboratory protocols in a workflow. This is crucial to verification, reproducibility, and automation in synthetic biology. By specifying a coherent model for workflow provenance, we can also address other critical use cases such as specifying where to add experimental data and performing model-based design.

Provenance classes were previously adopted in SBOL 2.1, but a gap in the specification left some ambiguity in how these classes should be used. The 2.1 specification indicates that the `hadRole` field on `Usage` and `Activity` classes MUST be provided, but does not define any terms to use in these fields. Because such terms are not specified, it is currently not possible for tool-builders to interpret or exchange provenance histories. It is expected that users will use their own ontology terms to specify how objects were used in a recipe, protocol, or computational analysis. However, these home-made ontologies will be very domain specific and may not be intelligible to users working in another domain. For example a modeler should not be expected to understand an ontology of Usage roles for DNA assembly. The terms "design", "build", "test", and "learn" provide a high level workflow abstraction that allows tool-builders to track provenance within their domain as well as to track the flow of data between domains.

Example 1 illustrates the various use cases that are possible with this provenance representation. Use case 1 describes linking many biological instances (plasmid clones, cellular clones, etc.) to the design which generated them. Use case 2 describes linking sequential stages of a build process which requires more than one processing step in the laboratory. Use case 3 describes a simple case of linking experimental data, such as sequencing data, to a sample. Use case 4 describes performing an experimental test on multiple samples at once in batch, for example in a 96-well plate. Use case 5 describes a scenario in which different models (eg. deterministic vs stochastic) are derived from experimental data. Use case 6 describes a case in which data from multiple experiments are integrated into a single analysis. Use case 7 describes the creation of a model-based design.

The Design, Build, and Test objects illustrated in Example 1 are included for purposes of discussion, but this specification does not actually introduce new classes by these names. Rather, this specification introduces the `Implementation` class to represent "Builds". Also this specification introduces `Attachments` in order to wrap experimental data files generated by a "Test". A "Design" is represented by a `ModuleDefinition` .

Example 2 shows explicitly how SBOL objects can be linked via PROV-O in an idealized design-build-test-learn workflow. This particular example represents model-based design. The workflow begins with a Model which is used to generate a `ModuleDefinition` ("Design") using a computational tool such as iBioSim. This ModuleDefinition is then used to generate a new `Implementation` ("Build") via an AssemblyProtocol. Now that a "Build" has been constructed in the laboratory, a "Test" may be performed by running a MeasurementProtocol on laboratory equipment, thus generating a `Collection` of data files or `Attachments` . Finally, a new `Model` might be derived from these data. This `Model` may not match the beginning `Model` , as our observation may not match the prediction.

For illustrative purposes, Example 2A shows how `Agent` or `Plan` might be extended to represent the important players in a synthetic biology worflow. For example, a ComputationalDesignProtocol may document that a Python script was executed in order to optimize a DNA sequence for synthesis. An AssemblyProtocol may describe a laboratory protocol (for example, either manual or automated). A SoftwareTool or Person are `Agents` that assist in an `Activity` . These examples illustrate where a developer might want to hook in their own application-specific data model. The PySBOL and libSBOL libraries allow users to define their own extension classes to `Plan` and `Agent` through inheritance relationships.

In addition to specifying a procedural workflow, this specification also allows us to interpret other kinds of provenance. Example 3 illustrates an example of "assembly provenance". This allows the user to track all of the physical instances of components which may be used in a DNA assembly procedure as well as the original design used to generate the new build. Other kinds of provenance are also possible. For example, provenance might also be used to track all the `Models` used to compose a more complicated `Model` .

Example 4 illustrates how the `Implementation` class is used to differentiate a conceptual design from a realized design. In the left hand example, the `Implementation` links via PROV-O to the `ComponentDefinition` that represents the original design, while its `componentDefinition` property links to the `ComponentDefinition` which describes the structure of the actual physical implementation. The right hand example demonstrates a case where the `Implementation` matches its design specification.

Example 5 provides an example of representing a sample history in SBOL. An initial sample kit based on part designs from the iGEM Registry is replicated and split into separate kits for each iGEM team. Multiple samples are then extracted from each team's kit for further construction and/or testing. Software tooling can trace the provenance of each sample to determine whether its original design (the `ComponentDefinition` from which its parent sample was derived) matches the description of what was built (the `ComponentDefinition` linked via its `built` property).

Example 6 provides an example of representing physical composition with SBOL. Two sets of overlapping constructs that implement the components of a CRISPR-based circuit are transfected to implement two different versions of the circuit.Software tooling can confirm that a given circuit `Implementation` satisfies its original design in two ways. First, a tool can compare the `ModuleDefinition` objects referred to by the `wasDerivedFrom` and `built` properties of the circuit `Implementation` . Second, a tool can confirm that the `ComponentDefinition` objects composed by the `ModuleDefinition` are the same as those from which the circuit component `Implementation` objects are derived.

Relation to Other SEPs

This proposal evolved from SEP 14 after discussion at 2017 HARMONY and COMBINE workshops, on the Github issue tracker, and on the ["Design-Build-Test" thread]: https://groups.google.com/forum/#!topic/sbol-dev/AnpwJP2_f5A

This proposal is competing with SEP 20. The most significant difference between these SEPs is that SEP 20 includes a mandatory design field on its Implementation class. This field is meant to serve as the primary means for documenting the intended structure and/or function of an Implementation, rather than PROV-O classes.

On the one hand, the design field could serve as a simpler, more explicit means of expressing these semantics. On the other hand, the design field could be seen as redundant given SBOL's adoption of a subset of PROV-O. In addition, the design field is currently mandatory as specified by SEP 20, which fails to account for use cases where there is no intended design, such as a sample of natural/unknown origin or one produced via directed evolution.

If SEP 19 passes, it would not preclude the addition of an optional design field to the Implementation class in the future.

References

Copyright

SEP 022 -- Rename displayId to id

SEP	022
Title	Rename displayId to id
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk); Christian Atallah (c.atallah2@ncl.ac.uk); Bradley Brown (b.bradley2@ncl.ac.uk); Anil Wipat (anil.wipat@ncl.ac.uk)
Editor	
Type	Data Model
SBOL Version	3.0
Replaces	
Status	Draft
Created	19-Jun-2018
Last modified	19-Jun-2018

Abstract

This SEP proposes renaming the `displayId` property of `Identified` to simply `id` in SBOL 3.0.

1. Motivation

The `displayId` property is very confusingly named, as it is not intended to be displayed (unlike the `name` property). Newcomers to SBOL are likely to assume `displayId` should be visible to users, or worse be explictly displayed in place of the name.

2. Specification

The following property will be renamed:

- `displayId` to `id`

3. Backwards Compatibility

This is a simple property name change and will be performed with the other breaking changes in SBOL3.

4. Discussion

5.1 Related SEPs

6. Competing SEPs

None.

References

Copyright

SEP 025 -- Merge ComponentDefinition and ModuleDefinition

SEP	026
Title	Merge ComponentDefinition and ModuleDefinition
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk); Chris Myers (myers@ece.utah.edu)
Editor	
Type	Data Model
SBOL Version	3
Replaces	
Status	Draft
Created	21-Jun-2018
Last modified	21-Jun-2018

Abstract

This SEP proposes merging ComponentDefinition and ModuleDefinition into a single class. This new, unified class would be able to have both structure (e.g. a sequence), and function (e.g. interactions).

Motivation

The SBOL2 data model has a split betwen structure, which is represented using ComponentDefinitions, and function, which is represented using ModuleDefinitions. ComponentDefinitions are mostly structural, in that they are intended to represent the primary sequence of a biopolymer and its hierarchical composition*. ModuleDefinitions, constrastingly, are functional groupings and can represent interactions such as genetic production and promoter regulation.

This split causes significant complexity when constructing SBOL objects. For example, figure 1 shows a simple transcriptional unit with a genetic production interaction. This is illegal in SBOL2, and instead a wrapper ModuleDefinition must be created as shown in figure 2.

** This split is not entirely clean: ComponentDefinitions and SequenceAnnotations have sequence ontology roles which are used to express function (e.g. "promoter").*

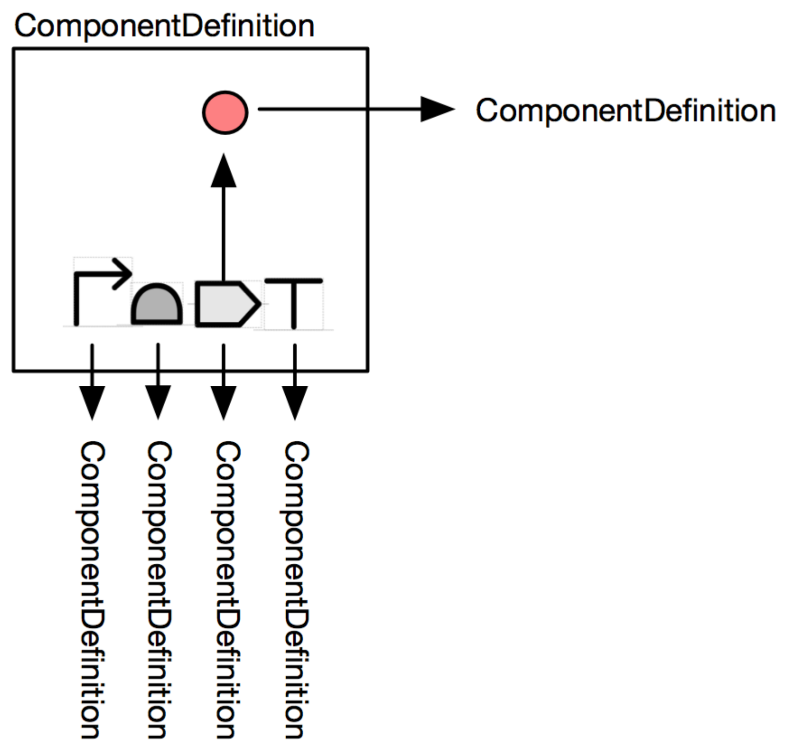


Figure 1: A transcriptional unit encodes a promoter. This is illegal in SBOL2 because the ComponentDefinition can not contain interactions.

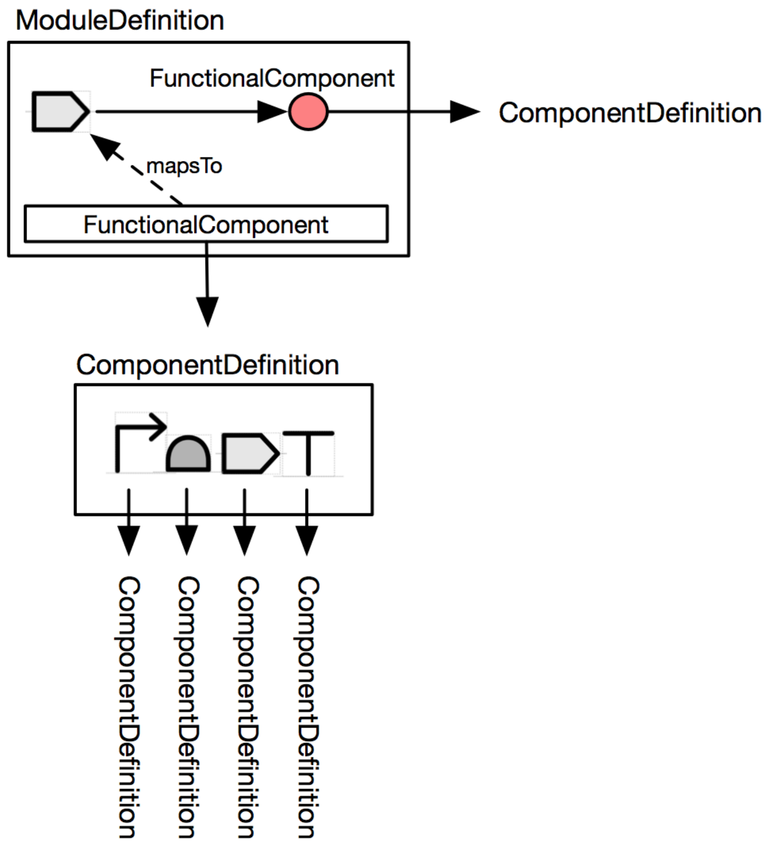


Figure 2: A transcriptional unit encodes a promoter in legal SBOL2. The transcriptional unit must be wrapped into a ComponentDefinition and then instantiated in a wrapping ModuleDefinition as a FunctionalComponent. The CDS can then be exposed to the wrapper ModuleDefinition using mapsTo. The protein is also instantiated as a FunctionalComponent, and finally the genetic production interaction can be created.
In this SEP, we are not suggesting that either of these approaches should be *disallowed*. The example in figure 2 remains perfectly valid in a hypothetical SBOL3 where ComponentDefinition and ModuleDefinition are merged; the only difference would be that the class name ModuleDefinition becomes ComponentDefinition. However, the example in figure 1 would now *also* be legal SBOL.

Advantage: MapsTo comes later

Constructing *any* design in SBOL2 other than composing simple DNA parts nearly always requires a MapsTo. MapsTo should only be necessary when *parts are being incoporated into larger designs*. This SEP would allow many simple parts to be constructed without any use of MapsTo.

Advantage: Flattening is now possible

"Flattening" operations have often been discussed by the SBOL community. Flattening involves taking a hierarchical design and "compiling" it to a design where everything is at the same depth. If this SEP is accepted, it will be possible to take a more modular design such as that described by figure 2, and flatten it to a simpler design as in figure 1. The flattened design provides an instant overview and can be more easily visualized.

Advantage: The spec gets shorter

Merging MD and CD will simplify the spec considerably. Classes such as FunctionalComponent can finally be removed.

Specification

The specification will be modified such that:

- All properties of ModuleDefinition are moved to ComponentDefinition
- FunctionalComponent is removed and any references to it are changed to references to Component (that is, SubComponent if SEP 15 is accepted)
- Module (that is, SubModule if SEP 15 is accepted) is removed and any references to it are changed to references to Component (that is, SubComponent if SEP 15 is accepted)
- ModuleDefinition is removed

Backwards Compatibility

Upconverting from SBOL2 can be implemented by treating both ComponentDefinitions and ModuleDefinitions as the same type when reading SBOL.

Downconverting to SBOL2 would require exporting any SBOL3 ComponentDefinitions with interactions as SBOL2 ModuleDefinitions, and any with a sequence or SequenceConstraints as SBOL2 ComponentDefinitions.

Discussion

5.1 Related SEPs

Competing SEPs

None.

References

Copyright

SEP 026 -- Add a link from Location to Sequence

SEP	026
Title	Add a link from Location to Sequence
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk); Chris Myers (myers@ece.utah.edu)
Editor	Zach Palchick
Type	Data Model
SBOL Version	2.3
Replaces	
Status	Complete
Created	21-Jun-2018
Last modified	03-Sep-2018

Abstract

In SBOL2, ComponentDefinitions can have multiple Sequences, and ComponentDefinitions can also have SequenceAnnotations. Currently, it is ambiguous which Sequence the SequenceAnnotation(s) refer to. Adding an optional link from Location to the Sequence it is annotating would solve this.

Motivation

As abstract

Specification

Add an optional `sequence` property to Location that links to a Sequence.

Example or Use Case

The below listed use cases are examples where there could be multiple Sequences associated with a ComponentDefinition, but it is not clear which Sequence a SequenceAnnotation would refer to.

- Different encodings of the same thing (e.g. smiles and inchi)
- DNA sequence + methylation patterns

Backwards Compatibility

When loading older SBOL:

- If there is exactly one sequence in a ComponentDefinition, make locations in the SequenceAnnotations of that ComponentDefinition point to that sequence
- If more than one sequence, warn that the annotations are ambiguous

Backporting to older SBOL:

- If exactly one sequence in a ComponentDefinition, remove the sequence property of locations in the SequenceAnnotations of that ComponentDefinition
- If more than one sequence, cannot be represented in older SBOL, throw an error.

Discussion

Competing SEPs

SEP 23 which advocates associating at most one Sequence with a ComponentDefinition. It has since been revoked.

References

Copyright

SEP 029 -- Update to issue procedure for SEPs

SEP	029
Title	Update to issue procedure for SEPs
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk)
Editor	Zach Palchick
Type	Procedure
Status	Final
Created	17-Oct-2018
Last modified	17-Oct-2018
Issue	#64

Abstract

SEP 001 defined the procedure for managing SEPs. This procedure currently mandates that all SEPs remain "open" on the issue tracker so that they are all visible by default. This SEP proposes that this requirement is changed so that SEPs issues can be closed when action is no longer required. This SEP also clarifies that the definitive text for an SEP is in the repository and not the issue tracker, solving the problem of unnecessary copy and pasting between the two. Finally, this SEP also recommends that SEPs concerning the specification are accompanied by a pull request to the SBOL specification repository.

Motivation

It is generally accepted (citation needed) when using GitHub for software development that an open issue is something that requires attention, and when attention is no longer required, the issue can be closed. As we are repurposing the GitHub issue tracker for keeping track of SEPs, it would therefore be logical that the issues SEPs that no longer require editorial attention (e.g. revoked, or merged into spec) can be closed.

Moreover, in the current system, the text of SEPs is duplicated; it is usually copied once for the markdown file contained in the repository itself, and once for the issue. In this SEP we recommend that the text of SEPs exist only in the repository. SEPs can then remain permanently in the repository for posterity, while their issues can eventually be closed by the editors. Using the repository as the definitive resource rather than the issue tracker is important for openness: issues are only accessible by the proprietary GitHub API, whereas the repository itself can be cloned or moved elsewhere. The current system also suggests that issue numbers correspond to SEP numbers, which has not been the case in practice.

Finally, there is a latency issue between the acceptance of SEPs and their merging into the SBOL specification. This SEP attempts to mitigate this by recommending that any SEP concerning the specification is accompanied by a pull request to the SBOL specification repository.

Specification

The following text of SEP 001 under "2.5 Decision":

> All SEPs will always remain "Open" on the issue tracker so that they are all visible by default. Editors attach and update issue labels to allow easy filtering for SEP Type and Status.

Is superceded by the following:

> SEPs remain "Open" on the issue tracker until they no longer require attention; i.e., have been either accepted and merged into the specification, rejected, or revoked.

The following text under "2.2 SEP Submission":

> If approved, an editor will submit the SEP to the dedicated github issue tracker, for which only SBOL editors have write-access. The github issue number then becomes the SEP number by which the proposal can be referenced.

Is superceded by the following:

> If approved, an editor will merge the pull request for the SEP into the dedicated SEPs repository on github, for which only SBOL editors have write-access.

After the steps for forking and opening a pull request to the SEPs repository in "2.2 SEP Submission", the following text is added:

> If the SEP concerns the SBOL specification, it is recommended that a corresponding pull request is also opened in the SBOL specification repository. This ensures that if the SEP is accepted, it can be swiftly merged into the specification.

Discussion

It should be noted that closed issues are not deleted and are always accessible and searchable through the same user interface as open issues.

Copyright

SEP 030 -- Best Practices for Multicellular System Designs

SEP	030
Title	Best Practices for Multicellular System Designs
Authors	Bradley Brown, Christian Atallah, James Alastair McLaughlin, Göksel Misirli, Ángel Goñi-Moreno, Nicholas Roehner, David James Skelton, Bryan Bartley, Jacob Beal, Chueh Loo Poh, Irina Dana Ofiteru, and Anil Wipat
Editor	
Type	Data Model
SBOL Version	2.4
Replaces	
Status	Accepted
Created	06-Nov-2018
Last modified	06-Nov-2018

Abstract

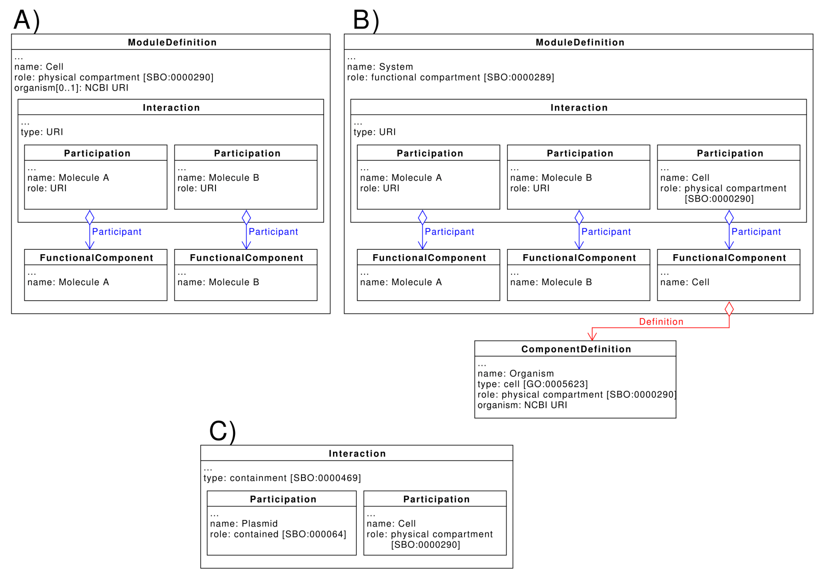
This SEP proposes some potential best practices for capturing information about multicellular designs.

1. Motivation

SBOL has been used extensively to represent designs in homogeneous systems, where the same design is implemented in every cell. However, in recent years there has been increasing interest in multicellular systems, where biological designs are split across multiple cells to optimise the system behaviour and function. Therefore, there is a need to define a set of best practices so that multicellular systems can be captured using SBOL in a standard way.

2. Specification

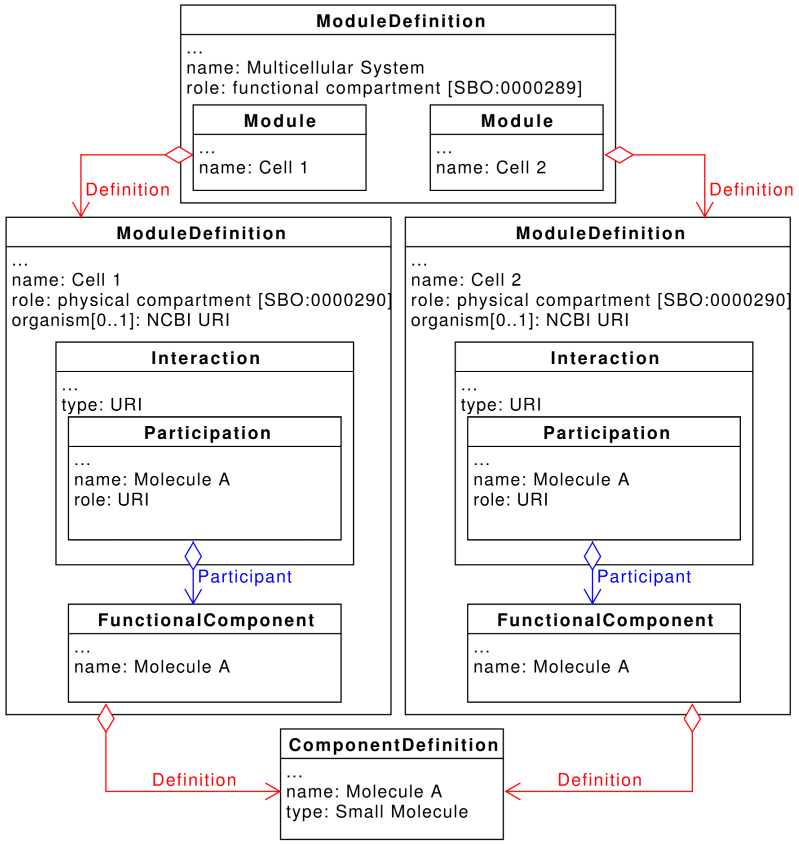
2.1 Representing Cells



The examples in A) and B) represent a cell which contains molecules 'A' and 'B', which interact in some way. **A)** First proposal for capturing cell designs in SBOL. A `ModuleDefinition` is used to represent the entire cell. This `ModuleDefinition` has a role of 'physical compartment' from the Systems Biology Ontology (SBO) and is annotated with a URI pointing to an entry in the NCBI Taxonomy Database. Interactions occurring within the cell are specified using `Interaction` classes. **B)** Second approach for capturing cell designs in SBOL. A `ComponentDefinition` annotated with a URI pointing to an entry in the NCBI Taxonomy Database is used to capture information about the cell's strain/species. The `ComponentDefinition` instance has a type of 'Cell' from the Gene Ontology (GO), and a role of 'physical compartment'. An instance of the `ModuleDefinition` class is used to represent a system in which the cell is implemented. Entities, including the cell, are instantiated as `FunctionalComponents`, and process are captured using the `Interaction` class. Process which are contained within the cell are represented by including the cell as a participant with a role of 'physical compartment'. **C)** Example of how the `Interaction` class can be used to explicitly confer that an entity is contained inside a cell. Here a plasmid is specified as being contained within a cell.

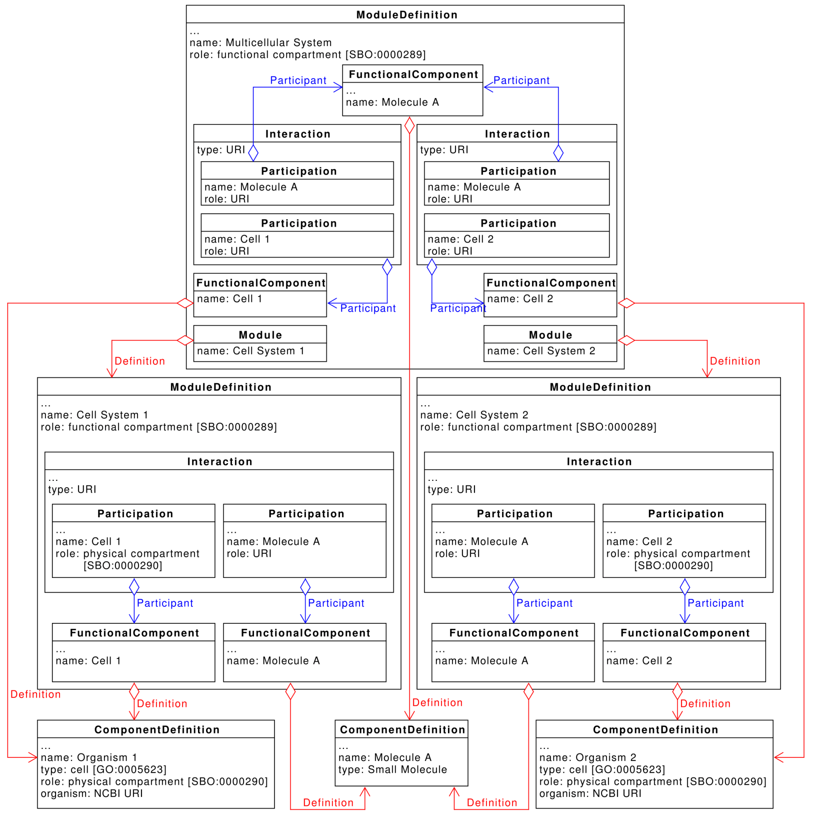
2.2 Multiple Cells in a Single Design

2.2.1 Approach 1



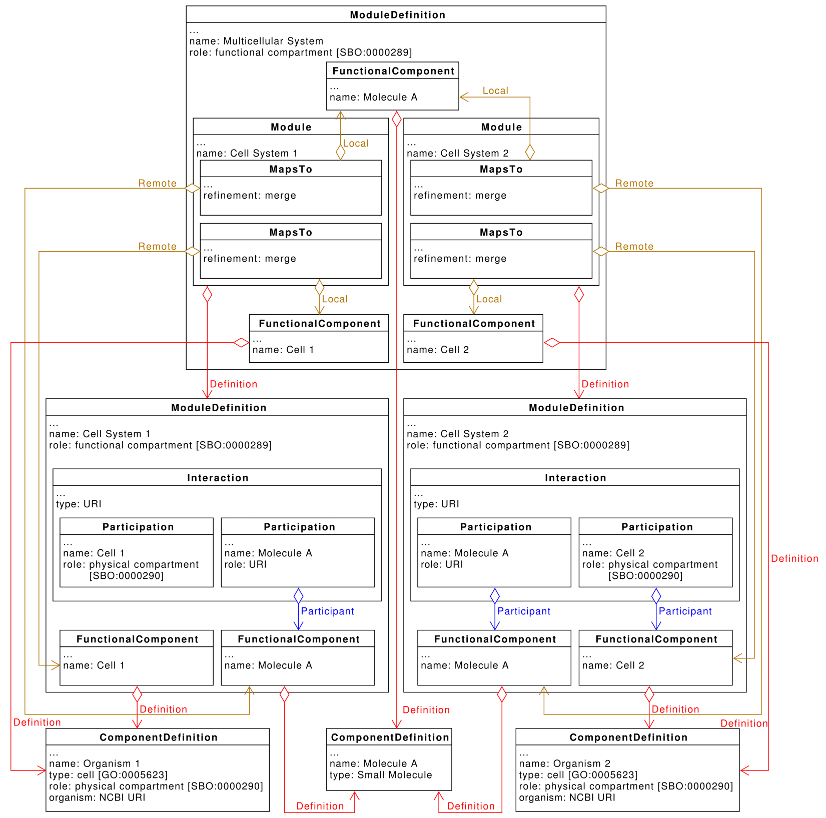
Captured here is a design involving two cells which both interact with the small molecule 'Molecule A'. Designs for Cell 1 and Cell 2 are captured using the approach depicted in Figure 1A, however this proposed approach is also compatible with the method shown in Figure 1B. The overall multicellular system is represented by a `ModuleDefinition` with a role of 'functional compartment', which is an SBO term. Both Cell 1 and Cell 2 are included in this design as an instance of the `Module` class. Interactions between the cells can be elucidated by comparing the entities which participate in processes defined within the cell designs. Cell 1 and Cell 2 in this design both have process which involve 'Molecule A', and hence can be deduced to have some form of intercellular interactions.

2.2.2 Approach 2



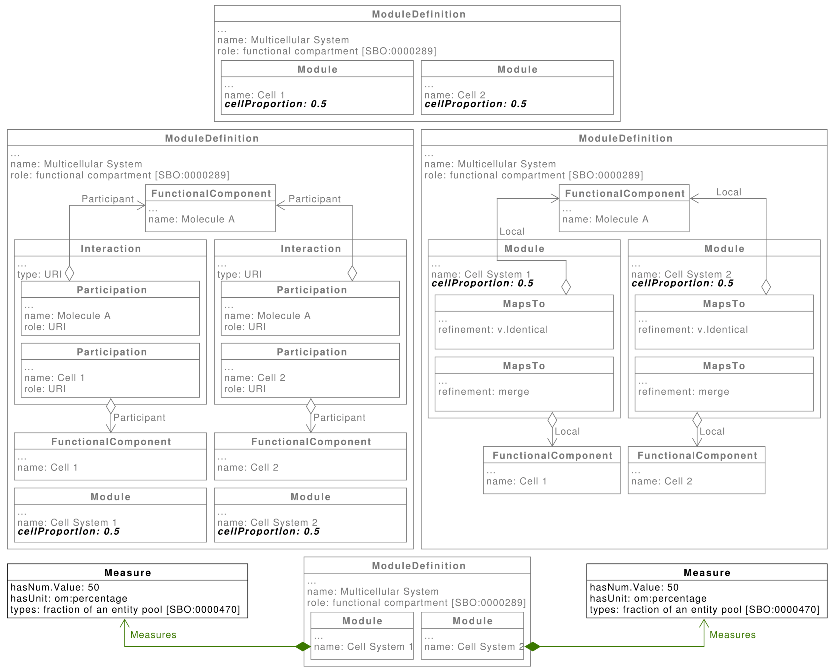
Captured here is a design involving two cells which both interact with the small molecule 'Molecule A'. Designs for Cell 1 and Cell 2 are captured using the approach depicted in Figure 1B, as this proposed approach is not compatible with the method shown in Figure 1A. The overall multicellular system is represented by a `ModuleDefinition` with a role of 'functional compartment'. Both Cell 1 and Cell 2 are included in this design as an instance of the `FunctionalComponent` class, which is defined by the `ComponentDefinition` instance which captures taxonomic information about the cell. Intercellular interactions are defined explicitly using the `Interaction` class. In this design, 'Molecule A' participates in processes within both Cell 1 and Cell 2.

2.2.3 Approach 3



Captured here is a design involving two cells which both interact with the small molecule 'Molecule A'. Designs for Cell 1 and Cell are captured using the approach depicted in Figure 1B, as this proposed approach is not compatible with the method shown in Figure 1A. The overall multicellular system is represented by a `ModuleDefinition` with a role of 'functional compartment', which is an SBO term. The two systems involving Cell 1 and Cell 2 are included in this multicellular design as instances of the `Module` class. Entities in the multicellular design, including those in the Cell 1 and Cell 2 system designs, are instantiated as a `FunctionalComponent`. Instances of the `MapsTo` class are used to map entities specified in both the individual cell systems, and the multicellular system.

2.3 Cell Ratios



A-C) demonstrate which class instances should be annotated with cell proportions for the approaches described in Figures 2-4 respectively. **D)** Alternative to annotating class instances with cellular proportions based on SEP 028. Instances of the `Measure` class are used to capture the percentage of each cell type present in the multicellular system design.

3. Backwards Compatibility

The best practices in this proposal do not affect backwards compatibility.

4. Discussion

5 Related SEPs

6. Competing SEPs

None.

References

Copyright

SEP 032 -- Do not rename ontologicially defined predicates

SEP	032
Title	Do not rename ontologically defined predicates
Authors	James Alastair McLaughlin (j.a.mclaughlin@ncl.ac.uk)
Editor	
Type	Data Model
SBOL Version	3
Replaces	
Status	Draft
Created	07-Dec-2019
Last modified	07-Dec-2019

Abstract

SBOL uses many terms from existing ontologies, such as Dublin Core and Prov-O. Currently, the specification is written in a manner such that those terms are given a new "SBOL alias" sometimes, but not always, distinct from the name assigned to them by the ontology. This SEP proposes that the name assigned by the ontology is always used.

Motivation

The SBOL specification intentionally uses terms from existing ontologies where appropriate. For example, instead of SBOL defining the concept of a "title" or "description", it instead uses the already accepted `dcterms:title` and `dcterms:description` properties from the Dublin Core ontology. This makes a lot of sense: rather than attempting to define everything from scratch, SBOL builds on decades of semantic web research in creating ontological definitions.

However, in the way that the SBOL specification is written, each ontology term first has to be given an "SBOL alias". For example, the `dcterms:title` property is first introduced as "name":

> The name property is OPTIONAL and has a data type of String.

And then later "mapped" to an ontology term in the "Serialization" section:

> name is serialized as dcterms:title

This approach has a number of issues:

- It assumes the SBOL is being serialized at all. In the days of SBOL1 this may have been the case, but with SBOL2 the SBOL is often contained in triplestores (e.g. synbiohub) and queried using SPARQL queries. In this case, there is no "name" alias but only "dcterms:title".
- It makes serialized SBOL extremely confusing to read, because the ontologically-defined names used in the serialization do not match the specification-defined names used by SBOL libraries
- As we have adopted new ontologies, it has resulted in clunky names for properties. For example, SBOL renames the `prov:wasDerivedFrom` property to "wasDerivedFroms" for consistency with other aliases used in the specification.
- It means that integrating terms from other ontologies requires a two-step process of writing their description as SBOL "aliases" and then writing their "serialization"

SBOL is likely to adopt new ontologies in the future, and this process should be as easy as possible. Aligning the SBOL specification as closely as possible with the ontologies it references, therefore, is desirable.

Specification

Properties renamed from their respective ontologies, e.g. "name", are changed back to their original names e.g. "title". Statements such as:

> The name property is OPTIONAL and has a data type of String

would be changed to

> The title property is OPTIONAL and has a data type of String. It is defined by the Dublin Core ontology and is located in the dcterms namespace.

with a link to the respective ontology. Statements describing "sets" such as:

> The wasDerivedFroms property is OPTIONAL and MAY specify a set of URIs.

would be changed to more abstract data model definitions such as:

> The wasDerivedFrom property is OPTIONAL and MAY be used to specify one or more URIs. It is defined by the Prov-O ontology and is located in the prov namespace.

Backwards Compatibility

This SEP would only change specification wording; serialization would remain identical, and is therefore fully forwards and backwards compatible. However, as it is a widespread change to the wording of the specification it has been provisionally grouped with the SBOL3 SEPs.

Discussion

5.1 Related SEPs

Competing SEPs

None.

References

Copyright

B. Examples of valid RDF/XML incompatible with libSBOLj

Input

```
<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:sbol="http://sbols.org/v2#"
         xmlns:dcterms="http://purl.org/dc/terms/" xmlns:prov="http://www.w3.org/ns/prov#">
    <sbol:ComponentDefinition rdf:about="http://example/promoter">
        <sbol:displayId>promoter</sbol:displayId>
        <sbol:persistentIdentity rdf:resource="http://example/promoter" />
        <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
        <dcterms:title>pLac</dcterms:title>
        <sbol:component rdf:resource="http://example/promoter/lacI_binding_site"/>
    </sbol:ComponentDefinition>
    <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
        <sbol:definition>
            <sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
                <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
                <sbol:displayId>lacI_binding_site</sbol:displayId>
                <sbol:persistentIdentity rdf:resource="http://example/lacI_binding_site" />
                <dcterms:title>lacI binding site</dcterms:title>
            </sbol:ComponentDefinition>
        </sbol:definition>
        <sbol:persistentIdentity rdf:resource="http://example/promoter/lacI_binding_site" />
        <sbol:displayId>lacI_binding_site</sbol:displayId>
        <dcterms:title>lacI binding site subcomponent</dcterms:title>
        <sbol:access rdf:resource="http://sbols.org/v2#public"/>
    </sbol:Component>
</rdf:RDF>
```

Output

```
Exception in thread "main" java.lang.ClassCastException: org.sbolstandard.core.datatree.NamedProperty$Impl
    cannot be cast to org.sbolstandard.core.datatree.IdentifiableDocument at
    org.sbolstandard.core2.SBOLReader.parseComponent(SBOLReader.java:3595) at
    org.sbolstandard.core2.SBOLReader.parseComponentDefinition(SBOLReader.java:1903) at
    org.sbolstandard.core2.SBOLReader.readTopLevelDocs(SBOLReader.java:1060) at
    org.sbolstandard.core2.SBOLReader.read(SBOLReader.java:727) at
    org.sbolstandard.core2.SBOLReader.read(SBOLReader.java:625) at
    org.sbolstandard.core2.SBOLReader.read(SBOLReader.java:512) at
    org.sbolstandard.core2.SBOLReader.read(SBOLReader.java:437) at
    org.sbolstandard.core2.SBOLReader.read(SBOLReader.java:422) at
    org.sbolstandard.core2.SBOLValidate.validate(SBOLValidate.java:2680) at
    org.sbolstandard.core2.SBOLValidate.main(SBOLValidate.java:2971)
```

In this example, a ComponentDefinition is instantiated inline as a child node of the sbol:definition predicate. While valid RDF/XML, it crashes libSBOLj with a ClassCastException.

Input

```
<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:sbol="http://sbols.org/v2#"
         xmlns:dcterms="http://purl.org/dc/terms/" xmlns:prov="http://www.w3.org/ns/prov#">
    <sbol:ComponentDefinition rdf:about="http://example/promoter">
        <sbol:displayId>promoter</sbol:displayId>
        <sbol:persistentIdentity rdf:resource="http://example/promoter" />
    </sbol:ComponentDefinition>
    <sbol:ComponentDefinition rdf:about="http://example/promoter">
        <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
        <dcterms:title>pLac</dcterms:title>
    </sbol:ComponentDefinition>
</rdf:RDF>
```

Output

```
sbol-10215: URI Compliance Warning:
The displayId property of a compliant Identified object is REQUIRED.
Reference: SBOL Version 2.3.0 Section 12.3 on page 61
: http://example/promoter

sbol-10502: Strong Validation Error:
The types property of a ComponentDefinition is REQUIRED and MUST contain a non-empty set of URIs.
Reference: SBOL Version 2.3.0 Section 7.7 on page 22
: http://example/promoter

Validation failed.
```

In this example, a ComponentDefinition is split across two description blocks. In RDF/XML, this is equivalent to having a single description block. However, it causes libSBOLj to fail with validation errors.

Input

```
<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:sbol="http://sbols.org/v2#"
         xmlns:dcterms="http://purl.org/dc/terms/" xmlns:prov="http://www.w3.org/ns/prov#">
    <sbol:ComponentDefinition rdf:about="http://example/promoter">
        <sbol:displayId>promoter</sbol:displayId>
        <sbol:persistentIdentity rdf:resource="http://example/promoter" />
        <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
        <dcterms:title>pLac</dcterms:title>
        <sbol:component>
            <rdf:Description rdf:about="http://example/promoter/lacI_binding_site">
                <rdf:type rdf:resource="http://sbols.org/v2#Component" />
                <sbol:definition>
                    <sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
                        <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
                        <sbol:displayId>lacI_binding_site</sbol:displayId>
                        <sbol:persistentIdentity rdf:resource="http://example/lacI_binding_site" />
                        <dcterms:title>lacI binding site</dcterms:title>
                    </sbol:ComponentDefinition>
                </sbol:definition>
                <sbol:persistentIdentity rdf:resource="http://example/promoter/lacI_binding_site" />
                <sbol:displayId>lacI_binding_site</sbol:displayId>
                <dcterms:title>lacI binding site subcomponent</dcterms:title>
                <sbol:access rdf:resource="http://sbols.org/v2#public"/>
            </rdf:Description>
        </sbol:component>
    </sbol:ComponentDefinition>
</rdf:RDF>
```

Output

```
sbol-10519: Strong Validation Error:
The components property of a ComponentDefinition is OPTIONAL and MAY contain a set of Component objects.
Reference: SBOL Version 2.3.0 Section 7.7 on page 22
: http://example/promoter

Validation failed.
```

In this example rdf:Description is used instead of sbol:Component. In RDF/XML, this use of rdf:Description would be equivalent to using sbol:Component. In libSBOLj, it causes a validation error.

Input

```
<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:sbol="http://not/sbol/" xmlns:ns="http://sbols.org/v2#"
    xmlns:dcterms="http://purl.org/dc/terms/" xmlns:prov="http://www.w3.org/ns/prov#">
    <ns:ComponentDefinition rdf:about="http://example/promoter">
        <ns:displayId>promoter</ns:displayId>
        <ns:persistentIdentity rdf:resource="http://example/promoter" />
        <ns:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
        <dcterms:title>pLac</dcterms:title>
        <ns:component rdf:resource="http://example/promoter/lacI_binding_site"/>
    </ns:ComponentDefinition>
    <ns:Component rdf:about="http://example/promoter/lacI_binding_site">
        <ns:definition rdf:resource="http://example/lacI_binding_site" />
        <ns:persistentIdentity rdf:resource="http://example/promoter/lacI_binding_site" />
        <ns:displayId>lacI_binding_site</ns:displayId>
        <dcterms:title>lacI binding site subcomponent</dcterms:title>
        <ns:access rdf:resource="http://sbols.org/v2#public"/>
    </ns:Component>
    <ns:ComponentDefinition rdf:about="http://example/lacI_binding_site">
        <ns:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
        <ns:displayId>lacI_binding_site</ns:displayId>
        <ns:persistentIdentity rdf:resource="http://example/lacI_binding_site" />
        <dcterms:title>lacI binding site</dcterms:title>
    </ns:ComponentDefinition>
</rdf:RDF>
```

Output

```
<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:ns="http://sbols.org/v2#"
    xmlns:sbol="http://not/sbol/" xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:prov="http://www.w3.org/ns/prov#" xmlns:om="http://www.ontology-of-units-of-measure.org/resource/om-2/">
  <sbol:ComponentDefinition rdf:about="http://example/promoter">
    <sbol:persistentIdentity rdf:resource="http://example/promoter"/>
    <sbol:displayId>promoter</sbol:displayId>
    <dcterms:title>pLac</dcterms:title>
    <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
    <sbol:component>
      <sbol:Component rdf:about="http://example/promoter/lacI_binding_site">
        <sbol:persistentIdentity rdf:resource="http://example/promoter/lacI_binding_site"/>
        <sbol:displayId>lacI_binding_site</sbol:displayId>
        <dcterms:title>lacI binding site subcomponent</dcterms:title>
        <sbol:access rdf:resource="http://sbols.org/v2#public"/>
        <sbol:definition rdf:resource="http://example/lacI_binding_site"/>
      </sbol:Component>
    </sbol:component>
  </sbol:ComponentDefinition>
  <sbol:ComponentDefinition rdf:about="http://example/lacI_binding_site">
    <sbol:persistentIdentity rdf:resource="http://example/lacI_binding_site"/>
    <sbol:displayId>lacI_binding_site</sbol:displayId>
    <dcterms:title>lacI binding site</dcterms:title>
    <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#DnaRegion"/>
  </sbol:ComponentDefinition>
</rdf:RDF>
```

In this example, the prefix “ns” is used for SBOL, and the prefix “sbol” is used for another namespace. In the libSBOLj output, tags correctly prefixed with the “ns” namespace pointing to SBOL are switched to the “sbol” namespace pointing to the other namespace.