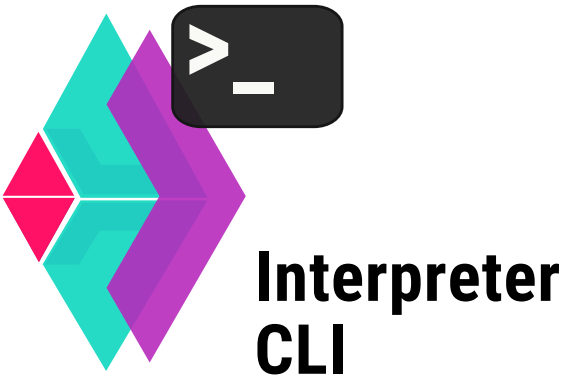

BDSL

BDSL User Manual

Bigraphical Domain-specific Language (BDSL): User Manual

BDSL Version: v1.0-SNAPSHOT

*Dominik Grzelak**
Software Technology Group
Technische Universität Dresden, Germany

Abstract

This report describes Bigraphical DSL (BDSL), a domain-specific language for reactive systems, rooted in the mathematical spirit of the bigraph theory devised by Robin Milner. BDSL is not only a platform-agnostic programming language but also a development framework for reactive applications, written in the Java programming language, with a focus on stability and interoperability. The report serves as a user manual mainly elaborating on how to write and execute BDSL programs, further covering several features such as how to incorporate program verification. Moreover, the manual procures some best practices on design patterns in form of code listings. The BDSL development framework comes with a ready-to-use interpreter and may be a helpful research tool to experiment with the underlying bigraph theory. The framework is further intended for building reactive applications and systems based on the theory of bigraphical reactive systems.

This report is ought to be a supplement to the explanation on the website `www.bigraphs.org`. The focus in this report lies in the basic usage of the command-line interpreter and mainly refers to the features available for the end-user, thus, providing a guidance for taking the first steps. It does not cover programmatic implementation details in great detail of the whole *BDSL Interpreter Framework* that is more suited to developers.

Acknowledgment This research project is funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

* Corresponding author; E-mail: `dominik.grzelak@tu-dresden.de`

Contents

1 Introduction

In this report, we describe a reactive, rule-based, and model-driven language termed Bigraphical Domain-specific Language (BDSL), which is both a platform-agnostic, application-independent programming language and framework, written in Java, with a focus on stability and interoperability. BDSL can not only be employed to experiment with bigraphs but can also be used to write and verify software programs based on the bigraph theory.

What are bigraphs?

Bigraphs [13], devised by Robin Milner, are not only a special class of graphs where two individual structures are combined (i.e., a forest and a hypergraph), allowing to express two semantic dimensions at the same time explicitly, but are also a model of computation that can resemble the two core elements of ubiquitous computing, namely, location and communication (see [3]). An example of an arbitrary bigraph is illustrated on the left-hand side in Figure 2.

A distinctive feature of this theory, and the BDSL language and framework as well, is the ability not only to model reactive systems but at the same time enabling mathematical verification of the program's behavior based on correctness properties. In other words, BDSL allows to check different concerns of a program, such as correctness, safety or security, by mathematical means.

Application Domains

The expressiveness of bigraphs makes it possible to use and exploit it in a variety of contexts. On the one hand, bigraphs can be used as any other graph structure to resemble data-structures, and on the other, found application in many fields such as for the modeling of context-aware systems [2], cloud systems [17, 14] wireless networks [4], sensor networks [18], indoor space models [22], spatial-temporal entities [20], or the formalization of the Web Services Business Process Execution Language (WS-BPEL) [9], to mention a few.

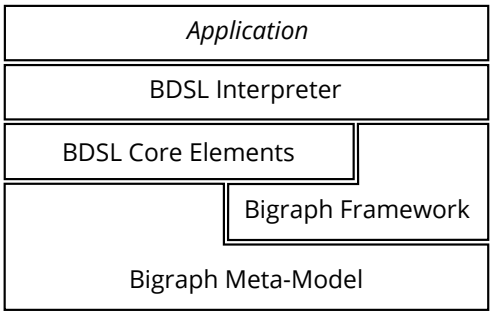

Figure 1: Diagram of the major building blocks of the BDSL environment.

Environment around BDSL

The major building blocks of the whole environment around BDSL is illustrated in Figure 1 which puts the BDSL-related frameworks, explained in a moment, into relation. More about bigraphs and the *Bigraph Framework* can be found on `www.bigraphs.org`.

BDSL Interpreter Framework is the name of the Java framework that contains, among other things, the underpinning functionality of the interpretation behavior and implements the command-line tool. The usage of both the language and the interpreter is the main focus of this document. The project is available from `https://git-st.inf.tu-dresden.de/bigraphs/bigraph-algebraic-interpreter`.

BDSL CE Framework is the name of the Java project that defines the grammar of BDSL and contains necessary language-specific implementations. The project is available from `https://git-st.inf.tu-dresden.de/bigraphs/bigraph-dsl-ce`.

1.1 Bigraphical Reactive Systems and Programming

Dynamic Behavior Generally, bigraphs can be reconfigured using so-called *reaction rules*. Thus, reaction rules enable the modeling of arbitrary behavior which resembles a basic form of programming. Having these rules at hand allows one to straightforwardly express dynamic behavior. Commonly, a rule has a left-hand side and a right-hand side called the *redex* and *reactum*, respectively. The redex is to be matched in a host graph where the match is afterwards replaced with the reactum. To control the rule application, they are applied based on a strategy, a fixed workflow, or before/after certain conditions, for example.

Reactive Systems *Reactive systems* perform their computation by reacting to stimuli from their environment. That means that to a great extend, the reactive system's behavior relies on external values and rules that are separate from the actual application source-code.

Bigraphical Reactive Systems A structure comprising a signature and a set of reaction rules is called a bigraphical reactive system (BRS, see [13]). Usually, it is augmented with a bigraph over the same signature, also termed *agent* here, that represents the initial state of the BRS to reconfigure with the given rules. Based on the facts given below, it will be apparent that BDSL provides a suitable foundation for programming reactive systems and BRSs in a canonical way.

1.1.1 Managing Complexity of Software

To make the software development of reactive sytems more productive, maintainable and less error-prone, suitable patterns and programming models shall be employed which allow semi- or full adaptivity by convenient change management of the program's code at design or runtime with minimal non-invasive actions. In line with this, rule-based programming and model-driven development are known to reduce the software's complexity (see Section 1.1.2). Making use of the notion and patterns of reactive systems is not only a suitable approach but also provides a more realistic view when dealing with distributed systems that behave non-deterministically in different contexts and strongly depend on external values from the environment (e.g., sensor values, user preferences, or location models).

→ Section 1.1.2

Modularity Bigraphs are naturally predestined for modular composition, which eases the creation of complex structures by simpler ones. This ability comes right from their mathematical underpinning by utilizing their categorical operators *composition* and *product* (see [13]).

Formal Verification BRSs (Section 1.1) provide the necessary means to model and verify the behavior of a reactive system. The building blocks for this mathematical reasoning are provided by the underlying operational semantics that bigraphs employ. BRSs are a model of computation and utilize so-called *Labeled Transition Systems* as their operational semantic model, which are often used for concurrent programs and reactive, distributed systems (cf. [6]). In particular, they "provide the basis for automatic system analysis and verification" [25]. We refer the reader to [16, 10, 1], because

a more detailed description of the subject called *formal methods* would go beyond the scope of this report.

1.1.2 Programming Models and Paradigms

Rule-based Programming

Commonly, expert systems or other AI-driven software apply rule engines in order to primarily model knowledge by means of rules. Rule-based Programming (RP) is regarded as programming paradigm or programming model in line with Object-oriented Programming (OOP) or Functional Programming (FP), for instance, where classes or functions, respectively, are the primary artifacts to program with. Thus, RP provides a different approach to dynamically implement, configure and change the behavior of a system, application or module in any architecture or software, primarily by means of rules.

RP is concerned with three main questions (see [24, p. 25]):

1. Where is the source of the rules?
2. When, how and where do rules change the behavior of the application?
3. Which effect has the changed behavior of the application before/whilst/after execution and appliance of the rules?

The reader may refer to [24], in order to get familiar with this kind of programming.

Model-driven Development

Compared with Model-driven Development (MDD), *models* are the primary artifacts and *transformations* are considered as the fundamental operations for and on these models. To obtain another view of the *bigraphical programming model* presented here, we can follow the general vision of MDD that in the broader sense *programming* is *model transformation*, and further, on a lower mathematical level that *graph rewriting*, which BRSs resemble, can be regarded as model transformation.

Thus, to close the loop with RP and MDD in mind, we may argue that *bigraphical reaction rules* can be regarded as a model transformation language for bigraphs; and that the bigraph theory represents a form of rule-based model-driven programming. The following analogy may help: Reaction rules are both rules in RP and model transformation specification in MDD.

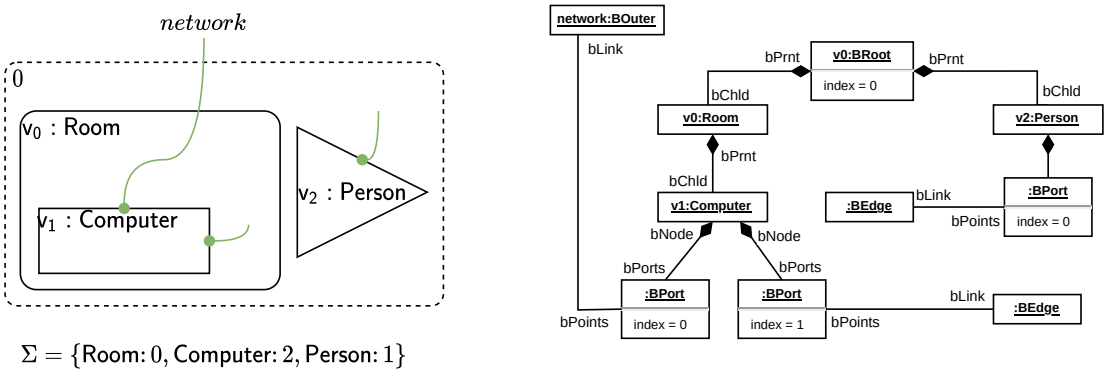

Figure 2: A bigraph B with signature Σ (left-hand side) and its corresponding type graph G over TG_Σ compatible to Σ (right-hand side). The metamodel of G is shown in Figure 3.

1.1.3 Yet Another Bigraph Tool?

BDSL was grown of the need to provide a more recent software exposition of a bigraphical tool that is developed with the two aspects *maintainability* and *extensibility* in mind for longevity. In fact, this is one of the major driver to contribute to the research of the bigraph theory and evaluation of real-world scenarios.

Alternative Tools Until now, the lack of sophisticated bigraphical tools impede building elaborated real-world applications based on the bigraph theory. Currently available bigraph tools can hardly be integrated in any software. The reason is that there are only a few working implementations of tools to compare. No systematic study has been reported in the scientific literature to evaluate all existing tools in detail. To fill this gap and understand the differences and commonalities of all available tools we are in the process of conducting a comparative study (not yet published) including a thorough analysis of these tools, namely, BigMC, BigraphER, BigRED, BigM, jLibBig, bigraphspace, DBtk and BPL Tool, to mention a few. In this study, we come to the preliminary conclusion that the majority of the examined tools do not possess the maturity to be used effortlessly and to be integrated in other software to build on the foundations of the, still young, bigraph theory. Most of them are outdated (4-10 years), not maintained anymore (e.g., BigMC, BigRED), or discontinued (e.g., DBtk). Moreover, the proprietary formats of these tools hamper the interoperability across these tools which make it difficult to build bigraphical tool chains (see [11]) or any other kind of software that wants to exploit the bigraph theory. Though, current bigraph tools such as BigraphER [19] are still suitable for experimentation.

Bigraph Metamodels To partly alleviate these shortcomings, a model-driven development approach was followed for the development of BDSL and also its underlying components such as *Bigraph Framework*[1]. Therefore, the generic metamodel described in [11] was employed, particularly to encode bigraphical models by the EMOF standard. To illustrate, Figure 3 depicts the bigraphical metamodel, also called type graph TG_Σ, of the typed graph G in Figure 2 (right-hand side), which is the representation of a concrete bigraph and compliant with the EMOF standard. This metamodel "serves

[1] Refer also to Figure 1, where its dependency relationships are depicted. More information on *Bigraph Framework* is available from `www.bigraphs.org`.

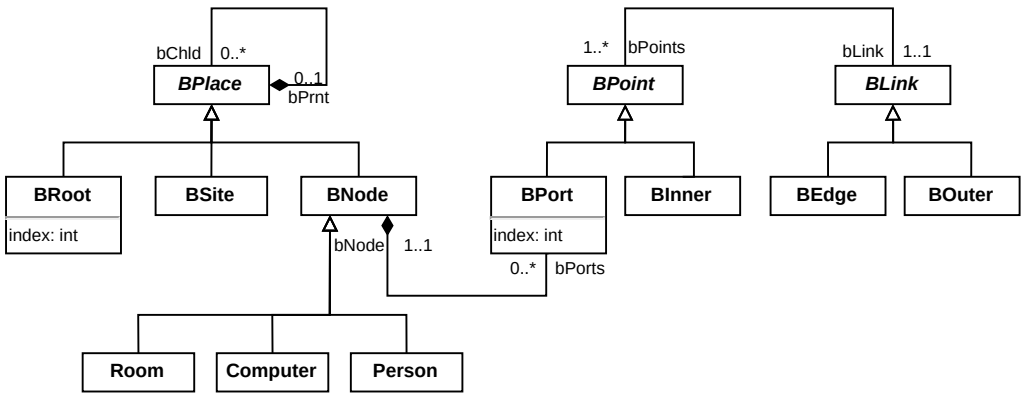

Figure 3: Type Graph TG_Σ compatible to Σ of the bigraph B and type graph G both depicted in Figure 2.

as an abstract syntax specification" [11, p. 19], which allows to derive instance models that syntactically conform to this metamodel.[2]

Furthermore, another metamodel was developed that describes the structure and semantic of BDSL programs and provides ample opportunities for extensions. It incorporates the generic bigraphical metamodel of Kehrer et. al. [11] for any bigraph expression: In what follows, any bigraph variable over a signature Σ (Section 3.4.2) is internally always an instance of a bigraphical metamodel, such as TG_Σ in Figure 3, for instance.

→ Section 3.4.2

1.2 Installation

First of all, BDSL is an open-source research project and thus primarily distributed as source code to be built manually. The source code can be obtained via this repository: `https://git-st.inf.tu-dresden.de/bigraphs/bigraph-algebraic-interpreter`.

Prerequisites

BDSL is written in the Java programming language. Therefore, the *JDK* or *JRE* (minimum version 11) must be installed depending on the intended use: Acquire the JDK 11 for active development or just the JRE if only running the tool is sufficient. Further *Git* is needed in order to clone the repository mentioned above. Also *Maven* (minimum version 3.6) is used for the build management.

1.2.1 Build Configuration

First, clone the Git repository by using the link provided above. After, change into the created directory. Then, execute the following commands inside this directory within a terminal to start the build process:

```
1 $ git clone -b 'v1.0-SNAPSHOT' --depth 1
    https://git-st.inf.tu-dresden.de/bigraphs/bigraph-
    algebraic-interpreter
2 $ cd ./bdsl-algebraic-bigraph-interpreter/
3 $ mvn clean install -DskipTests
```

The first line checks out the given tag `v1.0-SNAPSHOT` (i.e., the current version of BDSL) of the specified repository without cloning the complete commit history. In the last line, the build process of all interpreter components is started using Maven. After completion the tool can then be located inside the folder `./bigraph-algebraic-interpreter-cli/target/bdsl-1.0-SNAPSHOT.jar` from the root directory of the cloned repository.

1.2.2 Downloading the Binaries

Although, BDSL is primarily available as source-code, the binaries can also be obtained directly. The compiled tool is available from `https://git-st.inf.tu-dresden.de/bigraphs/bigraph-algebraic-interpreter/-/releases`. Nevertheless, to obtain the latest version the project can be build directly (see Section 1.2.1).

→ Section 1.2.1

This distribution includes a launcher script to run the tool effortlessly on most Unix-like platforms. Therefore, consult the `README.md` of the

[2] In turn, the metamodel of TG_Σ is a generic type graph called TG_{Base}, which contains no information about the bigraphical signature Σ (see [11]). Thus, TG_Σ is an extension of TG_{Base}.

BDSL Interpreter Framework repository (`https://git-st.inf.tu-dresden.de/bigraphs/bigraph-algebraic-interpreter`) for further instructions.

1.3 How to write and run BDSL programs?

Writing BDSL programs is possible with

- any text editor (e.g., Visual Code), or
- Theia BDSL (a browser application, see Section 6.3).

→ Section 6.3

To run BDSL programs the following options are available:

- Via the command-line tool, which is mainly used in this report (Section 1.2). It can also be accessed via the integrated terminal window in Theia BDSL.
- Via the *BDSL Interpreter Framework* to programmatically start BDSL programs in Java applications. This topic, however, is not elaborated here in great detail. Though, Section 6.2 provides a brief introduction.

→ Section 1.2

→ Section 6.2

1.4 Further Help

Learning by examples is a useful strategy to get acquainted very fast with the BDSL syntax and features:

- Section 5 illustrates by examples how to model typical problems in computer science the bigraphical way.
- Moreover, the website `www.bigraphs.org` provides further information on the underlying *Bigraph Framework* and other related components (refer also to Figure 1), which are particularly useful for developers.
- Robin Milner's book entitled "The Space and Motion of Communicating Agents" [13] is a self-contained resource about the bigraph theory and adequate to provide the necessary mathematical background.

→ Section 5

1.5 Remarks

A BDSL program, also simply referred to as program here, is also called a BDSL document.

LHS and RHS are short for left-hand side and right-hand side, respectively.

2 General Usage of the BDSL Interpreter Tool

2.1 The CLI Interpreter of BDSL

BDSL can be, as with most other DSLs, implemented by interpretation and code generation. As apparent from the name, the CLI interpreter implements the interpretation approach to execute BDSL programs.

Here, *interpretation* means that a BDSL program is treated like a script that is evaluated at run-time. That mans that no executable binary is created.

Generally, the BDSL interpreter can be used from the command-line by issuing the following command:

```
$ java -jar bdsl.jar <options...>
```

In order to make the execution more convenient under different operating systems, a start script for Unix-like operating systems is provided to ease the use of BDSL (see Section 1.2.2). For the rest of this manual, we will refer to the start scripts only. For example, instead of the call above, one can simply use the command "`bdsl <options...>`" in the terminal to achieve the same.

→ Section 1.2.2

When starting the CLI interpreter for the first time without arguments, it will first introduce itself with a banner and exit gracefully afterwards:

Listing 1: Starting the interpreter from the command-line.

```
$ bdsl
  ___    _                                          _          ___     ___    _
 | _ ) (_)  __ _    _ _    __ _    _ __   | |_      |    \   / __| | |
 | _ \ | | / _` |  | ' \  / _` |  | '_ \  | ' \     | |) | \__ \ | |__
 |___/ |_| \__, |  |_||_| \__,_|  | .__/  |_||_|    |___/  |___/ |____|
           |___/                  |_|
```

Disabling the Banner See Section 2.3 on how to disable the banner at startup.

2.1.1 Command-line Option Format

The BDSL interpreter tool accepts various command-line options. Options are passed to the interpreter by prefixing them with a dash `-` (*short* options) or a double dash `--` (*long* options), followed by the option's name.

Some options act as *switches* when passed without a value, whereas other options may be used to pass also values to a BDSL program. Therefore, the value for an option is specified using an equal sign `=` between the option's name and its value.

A list of all options can be printed when calling the interpreter with the `--help` switch. The output is shown below in Figure 4. More details on the meaning of some command-line options are given in Section 2.3.3.

→ Section 2.3.3

2.2 Supplying a BDSL Program to the Interpreter

In the current version, a BDSL program for the interpreter can be provided from the filesystem. Listing 2 shows how to supply a BDSL doc-

```
usage: BDSL interpreter tool
 -B <property=value>                                set the value for the given properties.
 -D <property=value>                                set the value for the given properties.
 -disableBanner,--disableBanner                     Disables the banner at startup
 -h,--help                                          print this message and exit
 -i,--include=<filename(s) of include file(s)>      Additional file(s) to include. The file order is also
                                                    the order in which the file(s) are loaded. Items are
                                                    separated by a comma.
 -iudf,--includeUdf=<filename(s) of udf archive(s)> Additional user-defined function archive(s) to include.
                                                    The file order is also the order in which the file(s)
                                                    are loaded. Items are separated by a comma.
 -m,--main=<main program file>                      the filename of the main BDSL file to interpret
 -outputDir,--outputDir=<name of output folder>     The base output folder for all generated files.
 -v,--verbose=<arg>                                 Verbosity of the output. Possible values are: TRACE,
                                                    DEBUG, INFO, WARN, ERROR, FATAL, OFF.
 -version,--version                                 print the version information and exit
```

Figure 4: The output of the interpreter tool is shown when called with the `-help` switch.

ument for the interpreter from the filesystem and also the exemplary output after its execution.[3]

Listing 2: Interpreting a BDSL program.

```
$ bdsl --main=test_bdsl_01.bdsl
      ___    _                                          _          ___      ___     _
   | _ ) (_)  __ _    _ _    __ _    _ __   | |_       |    \  / __| | |
   | _ \ | | / _` | | ' \  / _` | | ' \  | ' \      | |) | \__ \ | |__
   |___/ |_| \__, | |_|   \__,_| | .__/ |_||_|     |___/  |___/ |____|
                |___/             |_|
Printing bigraph variable as Ecore model ...
<?xml version="1.0" encoding="UTF-8"?>
<bigraphMetaModel:BBigraph xmi:version="2.0" xmlns:xmi="http://www.omg
    .org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bigraphMetaModel="de.tud.inf.st.bigraphs" xsi:schemaLocation
    ="de.tud.inf.st.bigraphs jar:file:/home/dominik/.m2/repository/de/
    tudresden/inf/st/bigraphs/model/bigraph-ecore-model/1.1.0/bigraph-
    ecore-model-1.1.0.jar!/model/bigraphBaseModel.ecore">
<bRoots>
<bChild xsi:type="bigraphMetaModel:a" name="v0"/>
<bChild xsi:type="bigraphMetaModel:b" name="v1"/>
<bChild xsi:type="bigraphMetaModel:b" name="v2"/>
</bRoots>
</bigraphMetaModel:BBigraph>
Translating BRS to BigMC ...
%passive a : 1;
%passive b : 1;


a | b | b -> b | a;

a | b | b ;

%check
```

The used BDSL example program `test_bdsl_01.bdsl` is printed in full length in the appendix in Listing 36. As apparent from the program's

[3] Future works considers loading and storing BDSL programs from and to a database (see Section 7.1).

call above, the argument `--main` points to the resource path of the BDSL document to be interpreted by the tool. Concerning the output, the tool first introduces itself with a banner, then prints the bigraph variable `$bigVar` as Ecore model to the console, before exporting the declared BRS variable `$example` to the console in the term language of BigMC [4].

Variables and methods are discussed later in the course of this user manual— before, the configuration concept of the BDSL interpreter shall be introduced.

2.3 Externalized Configuration

Different kinds of options, such as key-value pairs and flags, can be set to change various details of a BDSL program or its bigraph model declarations. To give an idea, it is possible to globally specify certain properties of the bigraph models (e.g., the namespace URI of the models, see Section 2.3.1) for the whole program, or the execution details of the interpreter, among other things.

→ Section 2.3.1

Variants and Precedence

All configurable properties can be adjusted by several means. For instance, configuring a BDSL program can be accomplished by setting various properties in an externalized configuration file. Moreover, the following mechanisms exist, which are in the order of precedence:

→ Section 2.3.3

1. **Command-line arguments** ,

→ Section 2.3.2

2. **External Configuration Files**, and

→ Section 2.3.1

3. **Magic Comments** .

! → Note that higher items in the list will override properties that are also specified by items lower in the list.

! → Note further that not all options can be specified by all three configuration mechanism above.

Overview of Configurable Properties

Therefore, Table 1 gives an overview of all available configurable properties, and under which configuration mechanism they can be accessed and modified. Their usage is explained in the following subsections.

Example: Disabling the Banner

As an example, we show how to disable the introductory banner of the CLI interpreter at startup (refer to Listing 2) via a command-line option.[5] To run the interpreter without showing the banner, add the following argument to the call:

```
1  bdsl --disableBanner ...
```

[4] `http://bigraph.org/bigmc/`, (last visited June 24, 2021).

[5] See also Table 1—disabling the banner can also be achieved by setting the right property in an external configuration file.

Table 1: An overview of configurable properties concerning the execution of a BDSL program.

	Property	Type (Description)	Configuration Variant		
			MC	CF	ARGS
General	Main file	`String` (filename)	□	■	■
	Include files	`String[]` (array of filenames)	□	■	■
	Include UDF archives	`String[]` (array of filenames)	□	■	■
	Verbosity	`Enum` (TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF)	□	■	■
	Banner visibility at startup	Flag/Switch	□	■	■
BRS Execution	Base export path	`String` (path)	□	■	■
	Transition system of a BRS	`String` (filename)	□	■	■
	States of the transition system	`String` (path)	□	■	■
	Measure execution time	`Boolean`	□	■	■
	Maximum transitions of a BRS	`Long`	□	■	■
	Labels of the transition system	`Boolean`	□	■	■
(Meta)Model	Encoding	see Section 2.3.1	■	■	■
	Namespace URI	see Section 2.3.1	■	■	■
	Namespace prefix	see Section 2.3.1	■	■	■
	Name of the instance model	see Section 2.3.1	■	■	■
	Location of metamodel file	see Section 2.3.1	■	■	■

MC means magic comments, CF means configuration file, and ARGS means command-line arguments.

2.3.1 Magic Comments

Magic comments provide a way to configure additional pre-defined parameters per BDSL document *within* a BDSL document's header. To be recognized, these comments have to be inserted in the first lines of a BDSL document. A typical magic comment section is shown in Listing 3.

Listing 3: All Available Magic Comment Directives

```
1 // encoding: UTF-8
2 // ns-uri: http://www.example.org
3 // ns-prefix: sample
4 // name: F
5 // schemaLocation: test-1.ecore
```

The meanings of each magic comment are as follows:

Magic Comment Directive	Description
`encoding`	The `encoding` directive specifies the encoding of all model resources that are loaded or serialized.
`ns-uri`	The namespace URI defines an identifier for the namespace to use for bigraph model elements.
`ns-prefix`	The namespace prefix to use for bigraph model elements (i.e., "tags") in the model file.
`name`	The name of the "bigraph" in the model file.
`schemaLocation`	The location of the metamodel. Useful for auto-validation of the Ecore instance models against the specified metamodel. It is only considered when a bigraph is exported.

2.3.2 Configuration file

A BDSL program can also be configured via an external configuration file with the default name `bdsl-execution.properties`. Two locations are supported by default (the list is ordered by precedence):

1. Within a `/config` sub-folder of the current directory.
2. The current directory

These locations are always scanned before, looking for `bdsl-execution.properties`. To clarify, configurable properties specified in the `config` sub-folder override properties specified in the current directory.

Available Properties Instead of describing each available configurable property that can be set in an external configuration file, a complete listing is given in Listing 35 in the appendix. The default values and additional comments on the usage are provided as well.

Default Location The default location for external configuration files can be configured. A different location can be specified by passing the command-line argument `-Dspring.config.location=PATH/TO/CONFIG/FILE`. As explained before in Section 2.3, configuration files specified in this way will have the

→ Section 2.3

highest precedence compared to properties specified in configuration files located in any of the listed locations above.

Example

Suppose that the default configuration called `bdsl-execution.properties` is placed next to the BDSL program to be executed. Further, an additional configuration file is supplied with the following argument: `-Dspring.config.location=new-config.properties`. Other than that, no other configurations are present. The locations are scanned in the following order:

- `bdsl-execution.properties` in the current folder of the BDSL program.
- `new-config.properties` in the current folder of the BDSL program.

Consequently, the respective properties contained in there are also evaluated in this order. This enables convenient overriding of previously defined default properties.

2.3.3 Command-line Options

Some properties can also be changed retrospectively, or in an ad-hoc fashion via program arguments. According to the configuration precedence described in Section 2.3, values supplied via arguments have the highest precedence and will overwrite all previous set parameters with the same name. This behavior allows to set default parameters via configuration files which can then be later conveniently modified via parameters over program arguments.

→ Section 2.3

General Options

The general configurable properties shown in Table 1 (top part) have to be passed according to this template: `--name=value`, with an exception of the `disableBanner` property. See Section 2.3.2 for a reference of all available property names that can be configured. For example, to pass the main BDSL program and additional include files, the call in the terminal is:

→ Section 2.3.2

```
$ bdsl --include=lib.bdsl --main=program.bdsl
```

The other properties are supplied in a slightly different format as described below.

Specifying BRS Execution Details

To supply BRS execution details presented in Table 1 (middle block) as arguments, these key-value properties have to be passed according to this template: `-Dname=value`. For example, to restrict the number of maximum transitions when executing BRSs (Sections 3.4.8 and 4.5), one has to pass:

→ Sections 3.4.8 and 4.5

```
$ bdsl ... -Dmodel-
    checking.transitionOptions.maximumTransitions
    =123 ...
```

The names after `-D` resemble *the same names* that are also used within an external configuration file (refer to Listing 35). The above call overrides the previously set value in a `bdsl-execution.properties` file under the same name—here it will be the option `model-checking.transitionOptions.maximumTransitions`. See Section 2.3.2 for a reference of all available property names that can be configured.

→ Section 2.3.2

Specifying Magic Comments

The prefix to supply magic comment directives is `-B` followed by the

→ Section 2.3.1

name of the respective directive according to key labels described in Section 2.3.1 and in Table 1 (bottom block). The value of the corresponding key must be set after an equal sign `=`. For example, to override the `ns-uri` and `encoding` property specified in the magic comment section of a BDSL program, the interpreter call is:

```
bdsl ... -Bns-uri=http//www.example.org -Bencoding=UTF-8
```

→ Section 2.3.2

Each key must be individually prefixed with `-B` when values of multiple keys shall be changed at the same time. See Section 2.3.2 for a reference of all available property names that can be configured.

3 BDSL Program Structure

3.1 Elements of a BDSL program

A BDSL program basically consists of four parts: **(i)** Namespaces and imports (Section 3.3); **(ii)** a `main` block (Section 3.2); **(iii)** several variable declarations of signatures (Section 3.4.1), bigraphs (Section 3.4.2), reaction rules (Section 3.4.6), predicates (Section 3.4.7) and BRSs (Section 3.4.8); and **(iv)** event listeners (Section 3.5).

Workspaces

BDSL possesses the notion of a *workspace*. The workspace is made up of a main program, i.e., a program containing a `main` block, and all additional included files, usually termed libraries here. These files can be supplied using the command-line interpreter (see properties "Main Files" and "Include Files" in Table 1). For example, workspaces enable to refer to elements in other files, see Section 3.3.1, and, as a result, facilitate the separation of different concerns as illustrated in Section 5.2 by an example.

→ Section 3.3.1
→ Section 5.2

Before each element of a BDSL program is explained in more detail, a quick syntax overview is given in Table 2.

Table 2: Short syntax overview for BDSL programs in `*.bdsl` files.

Syntax	Meaning
`signature Sig { }`	Signature declaration with name `Sig`.
`[active|passive|atomic] ctrl a arity 1`	Control definition inside a signature declaration. The status of a control can either be `active`, `passive` or `atomic`. The control label is of type `String`, the arity is of type `Integer`.
`val A(Sig) = {X}`	Bigraph variable declaration assigned the explicit signature `Sig`. `X` is a bigraph, where controls are drawn from `Sig`.
`react B(Sig) = {X}, {Y}`	Reaction rule declaration assigned the explicit signature `Sig`. `X` is the redex, `Y` the reactum. Both are bigraph expressions, or methods that return a bigraph.
`pred C(Sig):[iso|partial] = {X}`	Predicate declaration assigned the explicit signature `Sig`. `X` is the redex, `Y` the reactum. Both are bigraph expressions, or methods that return a bigraph. The type of a predicate can either be `iso` or `partial`.
`$R`	Reference to a bigraph, predicate, rule or BRS variable.
`brs D(Sig) = { agents=[$A], rules=[$B], preds=[$C] }`	BRS declaration, assigning a bigraph `A`, a rule `B` and a predicate `C`. The the name of the BRS is `D`. All entries accept an array of references.
`a * a`	Composition operator inside a bigraph expression.
`a - a`	Nesting operator inside a bigraph expression.
`a | a`	Merge operator inside a bigraph expression.
`a || a`	Parallel operator inside a bigraph expression.

3.2 Main Block

The `main` block provides the main entry point of any BDSL program. The statements inside the `main` block are executed from top to bottom in

the sequence they are given within the curly brackets: `main = { ... }`. However, "global" definitions outside the `main` block are possible and are always evaluated first.

3.3 Scoping, Namespaces and Imports

A scope can be defined as "the collection of valid targets for a reference" [21, p. 222]. The element's scope depends on its environment, for example, the namespace within which the element lives (see below), or possibly any other location, even non-structural in nature (see also [21]). Before discussing *scopes* in more detail, the basic concepts are introduced first. Therefore, we introduce the notion of *namespaces* in order to allow the referencing of elements in other BDSL programs.

Namespace

A namespace is similar to the notion of a package in the Java programming language. They are optional, and consequently, if omitted it is not possible to refer to elements in other files by using its *simple name*, also called *identifier* (ID).

Full Qualified Name

The *fully qualified name* is made up of the namespace—if specified—and the identifier of the element. The ID is usually a name specified by the user such that of a signature definition (Section 3.4.1) or a bigraph variable declaration (Section 3.4.2). If the element is further contained in another element then the ID of the container is appended before, which is done recursively for the whole containment hierarchy. In other words, the qualified name is a data structure which comprises the parent-child relationship of some element within a namespace.

For example, the mechanism for computing the qualified name for a bigraph variable named `bigraphVariableName` under a namespace declaration `my.namespace` specified in the same BDSL program is as follows. Its fully qualified will include the namespace and its ID, namely, `my.namespace.bigraphVariableName` in this example. This form holds for all elements in a BDSL file.

Referencing

As long as an element in BDSL has a name, it can technically be referred. Bigraphs (Section 3.4.2), reaction rules (Section 3.4.6), predicates (Section 3.4.7) and BRS declarations (Section 3.4.8) can be referenced from any place. For referencing, the qualified name is further prefixed by a dollar symbol `$` (see, for example, Line 6 in Listing 13), with an exception for signatures (see, for example, Line 1 in Listing 13).

In case the namespace is left out and the element is not contained anywhere, a valid qualified name may only include a single ID. Then the qualified name will be computed only using the ID of the element.

3.3.1 Scope

Global Scoping

The global scope is regarded as the outermost scope of a BDSL program. Global scoping automatically acquires elements in same workspace, without the need of an explicit import. As a result, we can refer to all public elements in a BDSL program by using the *qualified name*. The workspace concept is explained in Section 3.2.

→ Section 3.2

The scope is an essential data structure that comes into play when referring to elements. Given a certain location in a BDSL program, the scope describes all the reachable elements from their. Basically, it describes the visibility of elements depending on their context. Further, scopes are

Variable Shadowing

recursive and can be chained, leading to the notion of an outer and inner scope. This chain of responsibility allows *variable shadowing*.

Example

Given two bigraph variables with the same name declared in different locations (e.g., in the `main` block and outside the `main` block), they can still be uniquely referred to. This is shown in Listing 4.

Listing 4: Demonstration of the scoping mechanism.

```
1  signature Sig1 {
2    ctrl a arity 1
3    ctrl b arity 1
4  }
5
6  val a1(Sig1) = { a }
7
8  main = {
9    val a1(Sig1) = { b }
10   val bigVar = $a1
11 }
12
13 val bigVarOuter = $a1
```

The variable `a1` outside the `main` block has a node `a`, whereas the variable `a1` inside the `main` block contains the node `b`. When referring to `a1` inside the main-block, the declaration in Line 9 is used, because it lies in the inner scope. In contrast, the outer scope carries `a1` in Line 6. Bigraph variable declarations are described in more detail in Section 3.4.2.

→ Section 3.4.2

3.3.2 Namespace Mechanism

Sometimes it is useful to allow references to elements in a BDSL program using the *fully qualified name*. Especially, when external variables shall be referenced that have the same name and are specified in different scopes, or no import mechanism is employed (Section 3.3.3).

→ Section 3.3.3

The usage of the namespace mechanism is exemplified both in Listing 5 and Listing 6.

Listing 5: BDSL program 1.

```
1 namespace first.program
2
3 signature Sig1 {ctrl a arity 1}
4
5 val a1(Sig1) = { a }
```

Listing 6: BDSL program 2.

```
1 namespace second.program
2
3 signature Sig1 {ctrl b arity 1}
4 val a1(Sig1) = {b}
5 val b1(Sig1) = $first.program.a1
```

In Listing 5, the signature `Sig1` and bigraph variable `a1` are stored in the index (i.e., the global scope) with the qualified name `my.first.program.Sig1` and `my.first.program.a1`, respectively. This enables to refer to elements in a non-ambiguous way without using imports in Listing 6.

However, using the full qualified name is sometimes not always the shortest and most convenient way to reference elements. Below imports are introduced, to refrain from using the qualified name every time for referencing in other files.

3.3.3 Import Mechanism

BDSL provides an import mechanism, similar to the `import` statement in Java, that allows to import variables by their fully qualified names only once at the beginning of a program. Afterwards, they can be referred to by using only their simple name (i.e., their ID).

BDSL `import` statements also support wildcards `*` to import all variables of a specific namespace.

The following three listings illustrate the usage of BDSL imports.

Listing 7: BDSL program 1 to be imported

```
1 namespace my.first.bdsl.program
2
3 signature Sig1 {ctrl a arity 1}
4 val a1(Sig1) = { a }
5 val a2(Sig1) = { a }
```

Listing 8: BDSL program 2 to be imported

```
1 namespace my.second.bdsl.program
2
3 signature Sig1 {ctrl b arity 1}
4 val b1(Sig2) = { b }
5 val b2(Sig2) = { b }
```

Listing 9: BDSL program 3 with imports

```
1 namespace my.third.bdsl.program
2 import my.first.bdsl.program.a1
3 import my.second.bdsl.program.*
4
5 val c1 = $a1 // a1 is imported
6 val c2 = $my.first.bdsl.program.a2 // a2 is not imported above, but
    fully qualified name is used
7 val c3 = $b1 // b1 imported by wildcard
8 val c4 = $b2 // b2 imported by wildcard
```

! → As explained in Section 3.3.1, the variable resolution depends also on the scope. In Listing 9 only the global scope comes into play.

3.3.4 Elements of the same namespace

It is possible to reference elements of the same namespace in other files without the need of an `import` statement. This is illustrated in the following example below (see Listing 10 and Listing 11).

Listing 10: BDSL program 1

```
1 namespace my.bdsl
2 signature Sig1 {ctrl a arity 1}
3 val a1(Sig1) = { a }
```

Listing 11: BDSL program 2

```
1 namespace my.bdsl
2
3 val b1(Sig1) = $a1
```

Both programs are able to refer to each other without an explicit import because they are under the same namespace. In Listing 11, the bigraph variable `a` and signature `Sig1` from Listing 10 are used.

3.4 Classes and Variables

BDSL introduces many new features compared to currently available bigraph tools, such as the possibility to load bigraphical variables from

external resources (e.g., filesystem, or database[6]), the definition of *inner variables* or *listeners*.

3.4.1 Signature Declarations

The concept of a *class* as known from classical object-oriented programming is implemented currently only for signatures. More specifically, signature classes are *singletons*, which means that there can be only one instance of a signature class at a time. A signature is passed to bigraphs, rules and predicates, among others, in the course of a declaration, and, thus, always represents the same instance. All the modifications and updates that occur inside the same signature class will have the same effect on all instances that use this signature variable.

The basic definition of a bigraphical signature class is presented in Listing 12:

Listing 12: Example of a Bigraphical Signature Specification

```
1 signature Sig1 {
2   active ctrl a arity 1
3   passive ctrl b arity 1
4   atomic ctrl c arity 1
5 }
```

When defining a signature class, an instance of that singleton is automatically created with the identifier after the `signature` symbol and can be directly used afterwards. In the example of Listing 12 it is `Sig1`. It can also be considered as a constant.

The first letter of the signature's name should be capitalized. Otherwise, a warning is outputted, when the interpreter is executed.

Label and Arity Every entry inside the signature definition defines a control which is assigned a name and an arity (i.e., the number of ports of a control, see Figure 2).

Control Status Furthermore, a control can be `active`, `passive` or `atomic` which acts as some kind of constraint when building bigraphs and executing BRSs. It basically determines the *activity* or "reactiveness" of a control and its sub-structures later. This symbol can be omitted, with the result that the control's *status* is automatically set to `active`. For further details refer to [13, Def. 8.2].

Syntactic Sugar The specification of the status, label and arity of a control can be written also in a sugared form like this: `a: 1`. This expression exactly represents Line 2 in Listing 12. Here, `active` is always the default *status* for a control if not explicitly specified.

More sophisticated type systems are discussed in [5], which however, are not implemented yet in this tool.

3.4.2 Bigraph Declarations

→ Section 3.4.5

Bigraphs can be defined and constructed in many ways. For example, by using categorical operators (Section 3.4.5), by referencing other bigraph variables, or by assigning a function that returns a bigraphical structure

[6] Loading bigraphical resources from a database is an issue for future releases to explore.

→ Section 4
→ Section 3.4.3

(Section 4). Moreover, it is further possible to create local inner variables within a bigraph declaration (Section 3.4.3), which can be referenced too. Every bigraph variable has a name. The name of the bigraph can be used to refer to it later in order to conveniently build larger bigraphs or use them within another method.

For a brief overview, Listing 13 shows some viable declarations and assignments that are possible:

Listing 13: Some examples of bigraph variable declarations. The signature is ommitted from the code.

```
1 val bigVar2(Sig1):a = { a | b | b }
2
3 val bigVar1 = load(sig=Sig1, as=xmi, resourcePath="file:path/test.xmi")
4
5 main = {
6   val foo = $bigVar2 // signature will be Sig1
7   $foo = $bigVar1 // still Sig1
8   $foo = $bigSig2 // has now Sig2
9 }
```

→ Section 3.4.5

A bigraph contains *nodes*, possibly hierarchically structured using the available operators (Section 3.4.5). These nodes are created by directly using their *control* label which in turn are specified in the signature. Bigraphs must be assigned a signature before they can be created. The signature is passed to the bigraph declaration within the parenthesis `()` right after the variable name (see Line 1).

→ Section 3.3

The available controls for the bigraph expression definition are determined through the specified signature. That ensures to use only valid control labels inside the curly brackets of a declaration. Alternatively, the fully qualified name can also be used (see Section 3.3) to reference these controls. For example, Line 1 above can be rewritten to `val bigVar2:Sig1.a = { Sig1.a | Sig1.b | Sig1.b}`.

The control specification right after the colon `:` of a bigraph variable is a convenience feature that creates a node with the given control and automatically nests all further nodes under this one.

→ Section 1.1.3

Every bigraph variable implicitly instantiates a bigraph class (i.e., a metamodel) with the given signature and is a variable of that type. With regards to Listing 13 for example, the bigraph variable `bigVar2` in Line 1 is an instance of a type graph $TG_{\texttt{Sig1}}$ compatible to `Sig1`. Refer to the details given in Section 1.1.3, and Figure 3 which illustrates such a bigraph metamodel extending a bigraphical signature.

Attaching Links to Nodes

→ Section 3.4.4

In order to connect an outer name to a node (recall the bigraph in Figure 2), the name of the outer name has to be written inside square brackets `[]` right after the node's name. For instance, `a["network", "door"]`, creates a node named `a` that is connected to the two outer names `network` and `door`. The outer name is created automatically, when specified in this manner. See also Section 3.4.4, where special bigraphs are introduced that allow the creation of different kinds of linking structures easily. When attaching links to nodes, the arity of the control is respected (Section 3.4.1). If too many links are going to be connected, an error will be thrown.

! →
→ Section 3.4.1

Declaration Scope

Bigraphs can be defined in the *global scope* outside the `main` block and also within the `main` block.

Signature Overwriting

A bigraph is always defined over a signature. Therefore, the first argument is a signature `Sig1` which is automatically resolved from the BDSL document. Consider the declaration `val baz = {Sig1.a * Sig1.b}`. If no signature is explicitly defined, the interpreter tries to infer the signature according to this scheme:

1. Check if control type is explicitly defined with the qualified name after the colon `:`. Then its "container", i.e., its signature, will be used.
2. Otherwise, the bigraph expression definitions are inspected in order to find any control expression. Then, for the first one found, return the signature of this control.

In case the signature cannot be inferred, an error will be thrown. However, this also depends on the RHS of the bigraph variable declaration. If a bigraph variable (LHS) without an explicit signature definition is (re-)assigned a new bigraph (RHS, possibly from a `load()` method, see Section 4.2), the signature of the RHS will be used always. If for a bigraph reference (LHS) a reassignment is made and it has a different signature than the RHS, then the signature of the RHS will be used to overwrite the one on the LHS.

→ Section 4.2

If this behavior is not wanted, the signature of the bigraph declaration on the LHS should be explicitly specified. Then during the validation phase the signature will be checked against each other if they match, otherwise an error is thrown. This does not apply, however, for bigraph references on the LHS where the signature cannot be re-declared but only overwritten.

3.4.3 Inner Variables

A bigraph declaration supports also *inner variables*. They automatically infer the signature of their parent variable declaration in order to use the controls of the passed signature. Inner variables can be referenced in the same way as standard variables in BDSL. A brief demonstration on how to declare inner variables is given in Listing 14 (see also Listing 39 in the appendix for more examples).

Listing 14: Example of inner variable declarations. The signature is omitted.

```
1 val big1(Sig1) = {
2   val ex1:a = {
3     a - b
4     a - a
5   }
6   val ex2:b = { $ex1 } // creates a node with the control "b" and
        places "ex1" under it
7   val ex3:c = { } // creates a node with control "c"
8 }
```

Since the signature for inner variables is automatically inferred, the explicit specification is not necessary.

Moreover, they enable a convenient way of placing bigraphs side-by-side as the parallel product (Section 3.4.5) is implicitly used. This is shown in Line 2 to Line 7.

→ Section 3.4.5

The declaration of the inner variable `ex2` in Line 6 illustrates how to refer to the inner variable `ex1` that is contained in the same block under `big1`.

3.4.4 Elementary Bigraphs

Elementary bigraphs are node-free bigraphs and can be classified into *placings* and *linkings* (see [13]). They are a means for building complex bigraphical structures from elementary ones. Furthermore, they enable the expression of more sophisticated linking structures as explained in Section 3.4.2 in the paragraph "Attaching Links to Nodes".

BDSL provides special symbols in order to use them for a bigraph declaration:

- `barren()`: This function creates a placing comprising only one root node. It can also be written by using the short form `brn()`.
- `merge(i: int)`: This function returns a placing that has one root and `i` sites.
- `id(i: int)`: This identity function creates a place graph $\mathsf{id}_i : i \to i$ where the k-th site is connected to the k-th root ($k \in i$). The argument denotes how many "site-root-pairs" shall be created. Only positive integers are allowed. The index of both the site and root is determined automatically.
- `join()`: This function creates the elementary placing $join : 2 \to 1$. Two sites are nested under a root.
- `closure(s: string, ...)`: A function that creates a linking comprising only idle inner names. It accepts an arbitrary number of names. It can also be written by using the symbol `clsre(...)` or simply `/(...)`.
- `substitution(from: string, [to: string, ...])`: This function creates a substitution, where `from` is the label for the outer name to be created, and the second argument accepts a string array for inner names to be connected to `from`. It can also be written by using the symbol `subst(...)`.

The usage of these elementary bigraphs is illustrated in the appendix in Listing 37 and Listing 38.

3.4.5 Bigraphical Operators

So far, bigraph declarations were covered without actually constructing a complex hierarchical structure. However, in Listing 13, for example, the bigraph created in Line 1 contained the three nodes `a`, `b`, and `b` under a node `a`. This was accomplished by using special *operators*.

To effectively create complex bigraphs from simple ones, BDSL provides several operators, which resemble the categorical operators of bigraphs (see [13]):

Products — The operators `|` and `||` denote the merge and parallel product, respectively. They are used to place a bigraph side-by-side to another bigraph.

Composition — The operators `-` and `*` denote the nesting and composition operation, respectively. They allow to place a bigraph within another bigraph for expressing containment.

! → Note that composition is not valid if a node has an atomic control assigned

(see [13]). In this case, an error will be thrown by the interpreter.

3.4.6 Reaction Rule Declarations

Listing 15 shows some possible reaction rule declarations. A reaction rule $\mathsf{R} = (R, R')$ is a tuple comprising a redex and reactum (see [13]). In BDSL they are enclosed within curly brackets in the following format: `{REDEX},{REACTUM}`. It is possible to directly assign the redex and reactum either a bigraph variable reference, or a method which produces a bigraph (refer to Section 4).

→ Section 4

Listing 15: Reaction Rule Declarations

```
react ruleVar1(Sig1) = {a | b}, {a | b}
react ruleVar2(Sig1) = {
  // loading the redex from the filesystem, see  Section 4.2
  load(sig=Sig1, as=xmi, resourcePath="file:models/redex.xmi")
}, {
  a | b // specifying the reactum directly
}
react ruleVar3 = $ruleVar1 // referencing a rule variable
react ruleVar4(Sig1) = $ruleVar2
```

Signature Overwriting In the example above, `$ruleVar3` and `$ruleVar4` have no explicit signature specified and both will automatically get assigned the signature of the rule variable references, namely in this case, `Sig1`. If however the signature is explicitly given and does not match with the RHS, an error will be thrown.

3.4.7 Predicate Declarations

Predicates represent essential elements for the verification of BDSL programs and individual BRSs. They are mainly regarded as correctness properties that must hold for all or individual states during the program's execution.

Moreover, they can be employed as constrains to fire specific events when these hold `true` or `false` (refer to Section 6 for more details).

Listing 16: Predicate declaration examples.

```
pred pVar1(Sig1):partial = {a | b}
pred pVar2(Sig1) = load(sig=Sig1, as=xmi, resourcePath="file:test.xmi")
pred predVar3 = $pVar1
pred predVar4(Sig1):iso = {a | b | b}
```

Signature Overwriting In the example above, `$predVar3` has no explicit signature specified; thus, it will automatically get assigned the signature of the predicate variable references, namely in this case, `Sig1`. If however the signature is explicitly given and does not match with the RHS, an error will be thrown.

! →

Predicate Types Predicates serve as some kind of logical statements in a BRS that are evaluated at every state change. They come in two forms and are termed *partial* and *iso*. `Partial` means, that only the substructure of a bigraphical state of a BRS's transition system must match according to the predicate definition. Whereas, however, `iso` means that the whole bigraph definition is matched against the current bigraphical state of the transition system.

3.4.8 Bigraphical Reactive System Declarations

A BRS can be regarded as some container with a signature, and references to other variables (i.e., bigraphs, reaction rules and predicates).

Listing 17: BRS declaration example.

```
brs example(Sig1) = {
  agents = [$bigVar],
  rules = [$rule],
  preds = [$pred]
}
```

BRS variables are mainly declared by references to other variables. This allows to conveniently change the contents of a BRS by changing the referenced variables. That also means that explicit bigraph definitions are not supported.

The signature of a BRS must match the signature of the agents, rules and predicates. Otherwise, an error will be produced and the program is not executed.

Executing a BRS To *execute* a BRS, the `execute()` method (Section 4.5) is the right candidate.

3.5 Event Listeners/Callbacks

BDSL defines some special default *events*, where it is possible to react on these events by incorporating additional logic to be executed. These events are emitted in the course of the execution of a BRS (see Section 4.5),

→ Section 4.5

Event Types Currently available events are:

- Start and finish of a BRS evaluation,
- Rule matches (*single*, *many* or *all*), and
- Predicate matches (*single*, *many* or *all*).

These events are coupled with a BRS declaration. Listing 18 shows an excerpt of an usage example on how to integrate additional functionality after certain events occur.

Listing 18: BRS events.

```
    // signature and other variable declaration omitted
main = {
  brs example(Sig1) = {
    agents = [$agent],
    rules = [$rule1, $rule2],
    preds = [$pred1]
  }
  execute($example)
}

onReactiveSystemStarted($example) = {
  println("BRS execution started")
  j {
```

```
14      HelloUdfFunction()
15    }
16  }
17
18  onReactiveSystemFinished($example) = {
19    println("BRS execution finished")
20  }
21
22  listenForPredicateMatch($example, [$pred1]) = {
23    println("predicate matched!")
24  }
25
26  listenForRuleMatch($example, [$rule1, $rule2]) = {
27    println("Rule was applied")
28  }
```

→ Sections 4 and 6.1

The event listeners can be regarded as callback functions. Therein, all statements as in the `main` block are valid (see also Sections 4 and 6.1). For example, calling a *user-defined function* as demonstrated in Line 14 of Listing 18 is also possible. Refer also to Listing 33 for more usage examples on *user-defined functions*.

Aggregate Events

In order to aggregate multiple *events*, i.e. more than one predicate match (rule match), the respective variables can be supplied as an array between square brackets `[...]`. This is shown in Line 26 in Listing 18. Here, the same logic is executed when either `$rule1` or `$rule2` could be matched in the current "host" bigraph of the evolving transition system of the BRS variable `$example`.

4 Predefined Methods in BDSL

BDSL provides several standard methods:

→ Section 4.1
- `println(...)`

→ Section 4.2
- `load(...)`

→ Section 4.3
- `randomBigraph(...)`

→ Section 4.4
- `export(...)`

→ Section 4.5
- `execute(...)`

Where they can be used depends on their *function*. Some of them can only be called within the `main` block, within a callback block or in a completely different context such as in a bigraph declaration. Therefore, all methods are explained in the following subsections in more detail.

4.1 Printing to the Console

→ Sections 3.1 and 3.5

The `println()` method allows to print a string or a bigraph to the console. This method can only be called within the `main` block (Section 3.1) or in a event listener block (Section 3.5).

Listing 19: Println method examples.

```
println("Hello, BDSL!")
println($bigVar)
println($bigVar, xmi)
println($bigVar, ecore)
```

Parameters

The following parameters are available:

#	Argument	Description
1	`PrintableExpression`	A printable expression, either a char sequence or bigraph variable.
2	`Enum`	The model format to use when the 1st argument is a bigraph. Accepts either `xmi` or `ecore`. Default is `xmi`.

4.2 Loading Bigraphs

→ Sections 3.4.2, 3.4.6 and 3.4.7

The `load()` method is able to load arbitrary bigraph expressions from different resources. The result of a `load()` method can in some cases directly assigned to a bigraph, rule or predicate declaration (see Sections 3.4.2, 3.4.6 and 3.4.7).

Listing 20: Load method examples.

```
// loading two bigraph instances
val big1 = load(sig=Sig1, as=xmi, resourcePath="classpath:test1.xmi")
val big2(Sig1) = load(sig=Sig1, resourcePath="file:test2.xmi")

react(Sig1) = { // loading a reaction rule
  load(sig=Sig1, as=ecore, resourcePath="file:redex.xmi")
}, {
  load(sig=Sig1, as=ecore, resourcePath="file:reactum.xmi")
}
```

Parameters The following parameters must be supplied:

#	Argument	Description
1	`sig:Signature`	The signature of the bigraph to be loaded.
2	`as:Enum`	The Ecore format of the model to load. Either `xmi` or `ecore`.
3	`resourcePath:String`	A valid resource path with a specific prefix. See below.

It is necessary to specify the concrete signature because it is not stored in a model file. The argument `as` shall be supplied to indicate the model format, which can either be a metamodel (`ecore`) or instance model (`xmi`). It can be omitted when the file extension is provided. Then this argument is tried to be derived automatically.

Resource Locations Bigraph expressions can be loaded from several locations, where the resource location identifier must be accordingly specified:

Resource Type	Description
Filesystem	via `"file:"`; a model on the filesystem.
Classpath	via `"classpath:"`; a model in the classpath of the interpreter.

In every case the resource path follows the same schema, where slashes are used for navigation.

4.3 Synthesizing Random Bigraphs

The `randomBigraph()` method allows the creation of random bigraphs according to the algorithm proposed in [8]. An example is shown below on how to use this method.

Parameters The following parameters must be supplied:

#	Argument	Description
1	`sig:Signature`	The signature with the controls to use for the random bigraph
2	`t:int`	The number of roots the bigraph shall have (i.e., width of the forest).
3	`n:int`	The number of nodes in total
4	`p:float`	The fraction of nodes that should be linked randomly via an outer name or an edge (50 %-50 %). A value of 0.0 means that no nodes will be linked and 1.0 means that all nodes are tried to be linked.

Listing 21: Random bigraph generation example.

```
signature Sig1 {
  ctrl a arity 4
  ctrl b arity 4
  ctrl c arity 4
  ctrl d arity 4
}

signature Sig2 {
  ctrl a arity 4
```

```
10 }
11
12 val test3 = randomBigraph(sig=Sig2, n = 10, t = 1, p = 0.5)
13
14 main = {
15   val test2(Sig1) = randomBigraph(sig=Sig1, n = 20, t = 2, p = 1.0)
16   println($test2)
17   println($test3)
18 }
```

If the signature is specified on the left-hand side of the assignment, then it must match with the one specified in the `randomBigraph()` method. Otherwise, an error is thrown. You may also discard the explicit signature assignment as for example in Line 12. After evaluation, `$test3` will have the signature `Sig2`.

Some Notes on the Output

The outcome of the generated bigraphs strongly depend on the passed parameters. Note that the linking behavior is determined by the given signature and the assigned arity of each control. See also [8] for further details on how the random bigraph generation algorithm works.

4.4 Exporting Bigraph Variables

The `export()` method enables to export BDSL variables into several formats such as to BigMC [15], BigraphER [19], or even as a graphics file (`*.png`), and to different locations (whereby the console and filesystem is the only available option in the current version of BDSL). Listing 22 shows several examples on how to call this method.

Listing 22: Exporting a BRS into several formats.

```
1  brs example(Sig1) = {
2    agents = [$agent1],
3    rules = [$rule1],
4    preds = [$pred1]
5  }
6
7  export($example) // default is Ecore's *.xmi format
8  export($example, as=bigrapher) // output is printed to the console
9  export($example, as=bigmc, resourcePath="console:") // output is
       printed to the console
10 export($example, as=bigmc, resourcePath="file:./dump/test.bigmc")
11 export($example, as=ecore)
12 export($example, as=ecore, resourcePath="file:./dump/test.ecore")
13
14 export($agent1, as=png, resourcePath="file:./dump/bigraphVar.png")
15 export($pred1, as=png, resourcePath="file:./dump/bigraphVar.png")
```

If no resource path is specified, the output will be printed to the console. For the export format `png`, a resource path must be explicitly defined. Otherwise, an validation error will be thrown.

Parameters

The following parameters are available:

#	Argument	Description
1	`bigraph variable`	The bigraph, predicate or BRS to export.
2	`as:Enum`	The export format for the `bigraph variable`. Either `xmi`, `ecore`, `bigrapher`, or `bigmc`. To export the variable into a graphics file, the option `png` can be used.
3	`resourcePath:String`	The resource path to which the `bigraph variable` should be exported to. Refer to Section 4.2, whereby only "`file:` and `console:`" is available here.

! → → Sections 3.4.2 and 3.4.7

Exporting a variable to a graphic file (`*.png`) is only supported for bigraphs (Section 3.4.2) and predicates (Section 3.4.7) in the current version of BDSL.

4.5 Executing BRSs

→ Section 3.4.8

To explicitly execute a BRS declaration (Section 3.4.8), BDSL provides the `execute()` method, which accepts a BRS variable reference. Listing 23 shows how to call this method.

Listing 23: Executing a BRS.

```
1 brs example(Sig1) = {
2   agents = [$agent1],
3   rules = [$rule1],
4   preds = [$pred1]
5 }
6 execute($example)
```

→ Section 3.4.7

When a BRS is executed, a transition system is generated in the "background" and the given predicates (Section 3.4.7) are evaluated. A user might want to react on these events by actively listening to these, and for attaching additional logic. Refer to Section 3.5 to get more details on that topic.

→ Section 3.5

Transition System

→ Section 2.3

The generated transition system can be automatically exported to a file either by using an external configuration file, or by passing a specific command-line argument (see Section 2.3). It is not only possible to export the transition system but also all generated states as images or Ecore models (`*.xmi` and `*.ecore` files).

5 Examples

This section provides some introductory examples on how to model specific problems using BDSL. Further it is shown how to verify pre-defined properties that must hold during the program's execution, or, more generally, the program's behavior in combination with the underlying bigraph theory.

Therefore, this report provides examples of different applications in the field of computer science to demonstrate the wide spectrum in which bigraphs may be employed. The first examples can be categorized roughly as optimization problems or logic games.

→ Section 5.3

Note that, however, even it is possible to express some computational problems in the field of theoretical computer science with BDSL, more specific algorithms may be much more efficient. Take for example the pathfinding problem, which is modeled in BDSL in Section 5.3. Other alternatives more suited to solves this problem are, for instance, the A* search algorithm or random trees, because there are designed to solve this specific problem smarter by exploiting various concepts. Therefore, these examples serve only for demonstration purposes in order to show the variability of BDSL's expressiveness.

5.1 Basic Mathematical Calculations the Bigraphical Way

BRSs can also be exploited to formulate simple computational expressions such as the sum of two integers, or a comparison operator.

Next, Listing 24 and Listing 25 show exactly how to implement these two operations for two integers in a bigraphical way using BDSL, all expressed with bigraphs.

Listing 24: Computing the inequality of two integers that are expressed as bigraphs.

```
1  signature SigLt {
2    active ctrl lessThenExpr arity 0
3    atomic ctrl true arity 0
4    atomic ctrl false arity 0
5    active ctrl s arity 0
6    atomic ctrl z arity 0
7    active ctrl left arity 0
8    active ctrl right arity 0
9  }
10
11 main = {
12   val numberExpr(SigLt):lessThenExpr = {
13     (left - s - s - s - z) | (right - s - s - s - s - z)
14   }
15   brs comp(SigLt) = {
16     agents = [$numberExpr],
17     rules = [$reduce, $isLes, $isGreater]
18   }
19   execute($comp) // perform computation
20 }
21
22 react reduce(SigLt) = { left - s - id(1) | right - s - id(1)},
```

```
23   { left - id(1) | right - id(1) }
24
25   react isLess(SigLt) = { left - z | right - s - id(1)},
26   { true }
27
28   react isGreater(SigLt) = { left - id(1) | right - z },
29   { false }
```

Equality of integers

The first example in Listing 24 basically evaluates, whether $3 < 4$ is true. The numbers and the comparison expression including the $<$ operator are expressed by purely using bigraphs (see `numberExpr` in Line 12), whereas the evaluation is performed by the BRS declaration `comp` defined in Line 15. The graphical representation of both the "number bigraph" and "less-than sign" is depicted in Figure 5. It can be seen that only three reaction rules are needed. The order of the rules in the BRS specification is not important as each is matched in the beginning against the original variable `number` (and afterwards, also the re-written variable state) until the rule `isLess` or `isGreater` matches.

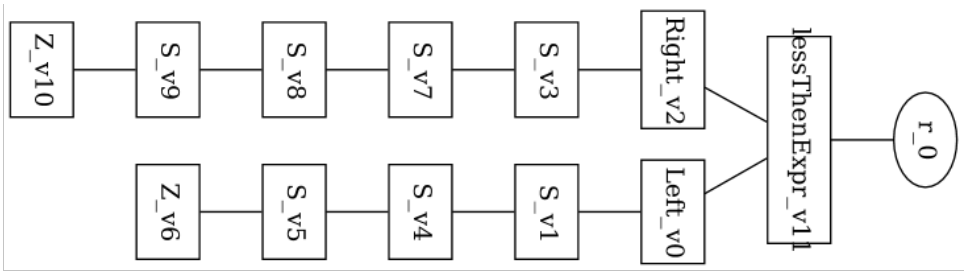

Figure 5: A bigraph representing a *less than expression* between two integers, namely, $3 < 4$.

Summation of integers

The second example in Listing 25 calculates the sum of two integers. Similar to the first example, the two numbers and the `+` operator are formulated by one bigraph variable `numberExpr` in Line 10. Here, the expression $3 + 4$ shall be calculated. The first rule `r1` describes how to move a node `s` from `left` under `right`. This is done as long as `s`-nodes are contained under `left`. Lastly, the second rule `r2` reduces the bigraph to a result containing only `s`-nodes that were previously collected under `right`. The execution itself is started by passing the BRS declaration `sum` (Line 13) to the method `execute()` (see Section 4.5).

→ Section 4.5

Listing 25: Summation.

```
1  signature SigSum {
2    active ctrl plusOp arity 0
3    active ctrl s arity 0
4    active ctrl z arity 0
5    active ctrl left arity 0
6    active ctrl right arity 0
7  }
8
9  main = {
10   val numberExpr(SigSum):plusOp = {
11     (left - s - s - s - z) | (right - s - s - s - s - z)
```

```
12   }
13   brs sum(SigSum) = {
14     agents = [$numberExpr],
15     rules = [$r1, $r2]
16   }
17   execute($sum) // perform computation
18 }
19
20 react r1(SigSum) = {left-s-id(1) | right-s-id(1)},
21   {Left | Right - s - id(1)}
22
23 react r2(SigSum) = {left-z | right-s-id(1)},
24   {s - id(1)}
```

5.2 Importing External Libraries

In this example, a BDSL program is given and further designated to be the *main program* that imports a *user-written library* with a specific namespace. This library itself is a BDSL program, usually without a `main` block.

Here, the operation that the main program executes is rather simple and serves only as demonstration for the *library approach* explained here. For instance, recall the two examples regarding bigraphical computations in Section 5.1. Here, the "number bigraphs" can be specified in a distinct *library file* as well as the BRS that performs the computation (e.g., summation). Then, a third program needs to import both programs by their respective namespace. Computing the sum for different "number bigraphs" can be accomplished by changing the bigraph variables in the first library only, or by passing a completely different library to the interpreter but using the same namespace.

→ Section 5.1

Listing 26: Common signature declaration.

```
1 namespace sum.types
2
3 signature SigSum {
4   active ctrl plusOp arity 0
5   active ctrl s arity 0
6   active ctrl z arity 0
7   active ctrl left arity 0
8   active ctrl right arity 0
9 }
```

Listing 27: Bigraphical number declarations.

```
1  namespace sum.numbers
2  import:bdsl sum.types.*
3
4  val numberLeft(SigSum): left = {
5    s - s - s - s - s - s - z
6  }
7
8  val numberRight(SigSum): right =
       {
9    s - s - s - s - z
10 }
11
12 val expression(SigSum): plusOp =
       {
13   $numberLeft | $numberRight
14 }
```

For the next example, the primary purpose of these libraries is to configure the main program without any modification afterwards. Therefore,

three additional libraries are implemented, one for the common signature definition (Listing 26), one that declares two "bigraphical numbers" (Listing 27), and one that specifies the actual summation operation in a bigraphical way (Listing 28). All three libaries are consolidated in Listing 29, where only the execution of the defined BRS is started.

Listing 28: Declaration of the summation operation.

```
namespace sum.operation
import:bdsl sum.types.*
import:bdsl sum.numbers.*

react r1(SigSum) = { (left - s - id(1)) | (right - id(1)) }, { (left - id(1)) | (right - s - id(1)) }
react r2(SigSum) = { (left - z) | (right - s - id(1)) }, { s - id(1) }

brs summation(SigSum) = {
  agents = [$expression],
  rules = [$r1, $r2]
}
```

Listing 29: Main program of the summation operation example.

```
namespace sum.program
import:bdsl sum.types.*
import:bdsl sum.numbers.*
import:bdsl sum.operation.*

main = {
  execute($summation)
}
```

To execute the whole program, assuming that all four files above are located in the same directory, the following command is used:

```
./bdsl --include=include-1.bdsl include-2.bdsl
    include-3.bdsl --main=main.bdsl
```

The first argument `--include` determines the additional libraries to include. The first one must be Listing 26, since Listings 27 and 28 are also importing the common signature declaration. The second argument `--main` specifies the BDSL program to be executed (more concretely, the respective `main` block contained therein). These arguments are discussed in Sections 2.2 and 2.3.3. The reaction rules in Listing 28 basically describe how to "transfer" a node, here `s`, from the left-hand side to the right-hand side until no `s`-nodes are available anymore under the node `left`.

→ Sections 2.2 and 2.3.3

Though, this simple example is deliberately over-complicated, it shows the full separation of different concerns, namely, from the specification of the data and the operational semantics to the actual execution instruction.

In contrast to hard-code different behaviors in a BDSL program that are switched based on external values (refer to Section 2.3), this form of library approach may provide also a mechanism for dynamic configuration of programs by simply swapping out the specific *library* that contains configurable data.

→ Section 2.3

5.3 Pathfinding: Naive Blind Search

Consider the following problem.[7] Given is an arbitrary *map* that consists of *places* and *roads*. A place comprises roads, and roads are connected to places or to other roads. It is possible that a place has roads which are not connected to other places or roads. Further, a car is given with a limited amount of fuel. The car starts at a random place called *A* and needs to travel to a destination called *target*. A car can only travel on connected roads. The starting situation is conceptually illustrated in Figure 6.

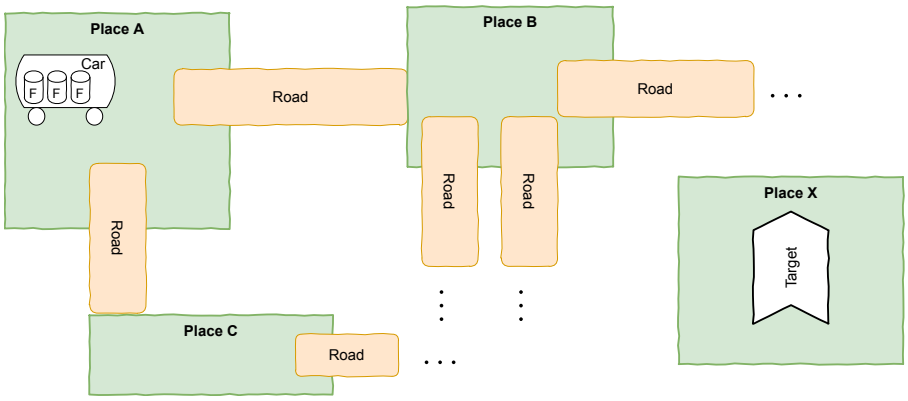

Figure 6: The initial situation of the pathfinding problem.

Several interesting questions can be posted now:

- Is the car able to reach the target with the given amount of fuel? If yes, how many roads and places must the car travel to reach the target?
- Are there any blind ends?
- At which states will the car get out of fuel so it never reaches the target?

Listing 30 shows an excerpt of the whole program on how to express this problem in BDSL by implementing a naive blind search in order to verify, whether the car will reach its target or not. (The full code is printed in Listing 40.) Here, verification is a means to answer the questions mentioned above, which will be apparent in a moment.

Listing 30: Pathfinding example (`bdsl-pathfinding.bdsl`, see Listing 40).

```
signature SigMap {
  Car: 1
  Fuel: 0
  Place: 1
  Road: 1
  Target: 1
}

```

[7] This particular example was inspired by jLibBig (`https://github.com/bigraphs/jlibbig`, last visited June 24, 2021), a Java library for BRSs. The concrete implementation in jLibBig is available from `https://github.com/EPresident/UniUdBig` (last visited June 24, 2021).

```
9  main = {
10   brs findPath(SigMap) = {
11     agents = [$map],
12     rules = [$moveCar],
13     preds = [$targetReached]
14   }
15   execute($findPath)
16 }
17
18 val map(SigMap) = {
19   // declaration omitted
20 }
21
22 react moveCar(SigMap) = {
23   // declaration omitted
24 }
25
26 pred targetReached(SigMap) = {
27   // declaration omitted
28 }
```

From Listing 30 it can be observed that only one rule and a predicate is needed to solve the stated problem above. The rule `moveCar` specifies when a car is able to move from a place to another place via a road, also reducing one amount of fuel, and, if possible, does so by rewriting the bigraph variable `map` (determined by its reactum, i.e., the second curly bracket clause of `moveCar`). The predicate specifies a state in which the car reaches the target. The input of this problem is a *map* declared in Line 22 which itself is a bigraph. This "bigraphical map" is visualized in Figure 7, whereas the full declaration is given in Listing 40.

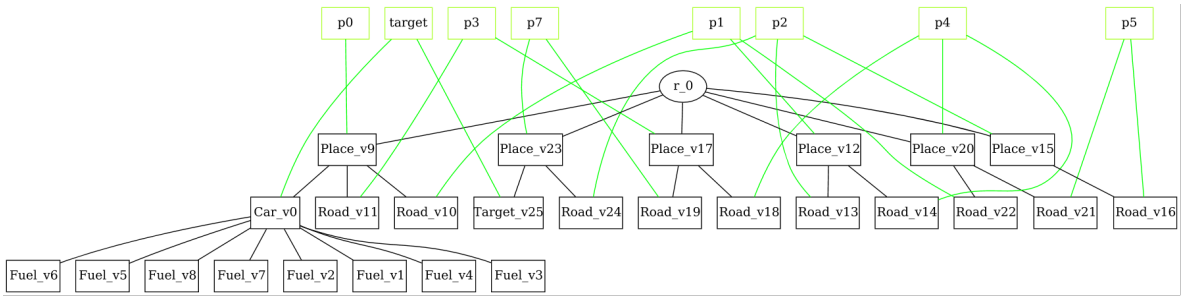

Figure 7: The map as a bigraphical structure, resembling the map in Figure 6.

The map, rule and predicate are supplied to the BRS variable declaration in Line 10, ready to be executed. The questions stated above can be answered by observing the synthesized transition system of the BDSL program after its execution in Line 15. The output of the transition system is depicted in Figure 8. This transition system was automatically exported by configuring the appropriate property (see Table 1).

Interpretation of the Result

The car reaches its target in two steps which is indicated by the green rectangle around a state due to the predicate `targetReached` in Line 37. BDSL automatically renders the file in this way. Moreover, it can be observed that four blind ends exist that do not have a road back to another place or road—the car gets stuck. Because the initial amount of

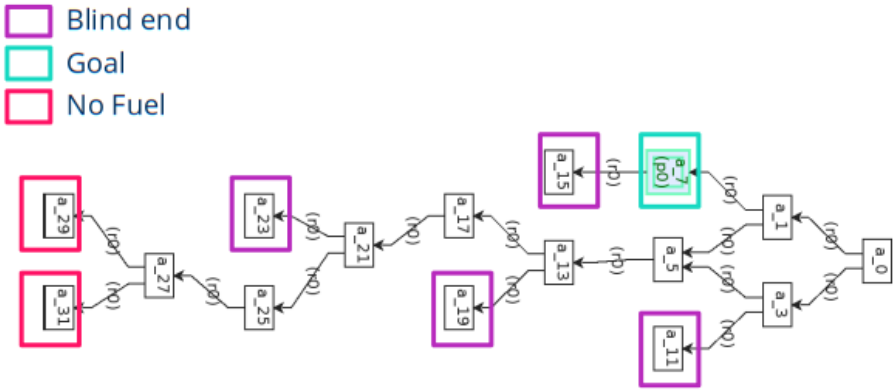

Figure 8: Generated transition system of the pathfinding problem as modeled in Listing 30.

fuel for the car was set to 8, it is logical, that the car has no fuel after 8 steps, which is indicated by a red rectangle in Figure 8.[8]

In this specific example the program was executed with the goal to explore all possible states. This may not be the desired goal, especially for the case in which the state-space is infinite. It is possible to add constraints to the execution by configuring the respective parameters (see Section 2.3).

→ Section 2.3

5.4 Mutual Exclusion Problem

This section deals with the well-known mutual exclusion problem in computer science. The aim of this example is to show how to model two processes that mutually access an exclusive resource using BDSL. Further, we show how to verify the correctness of the program. Therefore, we are going to specify some predicates that are checked at every state change. The resulting transition system used for verifying the behavior is exported for later inspection, or for further processing.

States of the Program Execution

Different approaches are possibly to model this kind of behavior. Here, we show only one of many in BDSL. The operation specification is always the same for every process: First, a process has to undergo a registration process by acquiring a *token*. After, the process may perform its work and is allowed to access the shared resource. Finally, the process has to release the access lock to make the shared resource available again for other processes. This operational behavior is expressed via three reaction rules (see Listing 31).

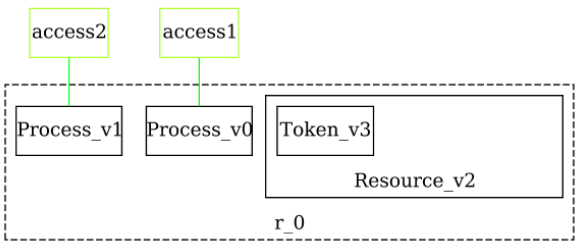

Figure 9: Initial state of the mutual exclusion problem. This state represents the variable `startingState` in Line 16 of Listing 31.

[8] These last two assertions are not given in the BDSL program in Listing 30, but were added later to the transition system in Figure 8.

Initial State of the Program

The initial situation is depicted in Figure 9. The figure shows two processes which have assigned the same control `Process` but are identified by two different links, namely, `access2` and `access1` for the "left" and "right" process, respectively. This fact can be exploited to add many more processes to the program without changing the logic (i.e., rules). Under the same environment a resource exists which contains an *access token*.

The complete BDSL program is presented in Listing 31.

Listing 31: Mutual exclusion problem in BDSL for two processes.

```
1  signature Sig1 {
2      Process: 1
3      Token: 1
4      Working: 1
5      Resource: 1
6  }
7
8  main = {
9      brs $mutual(Sig1) = {
10         agents = [$startingState],
11         rules = [$r0, $r1, $r2]
12     }
13     execute($mutual)
14 }
15
16 val startingState(Sig1): {
17     Process["access1"]
18     Process["access2"]
19     Resource - Token
20 }
21
22 react r0(Sig1) = { // acquire lock
23     Process["access"] || (Resource - Token)
24 }, {
25     Process["access"] || (Resource - Token["access"])
26 }
27
28 react r1(Sig1) = { // do work
29     Process["access"] || (Resource - Token["access"])
30 }, {
31     Process["access"] - Working || (Resource - Token["access"])
32 }
33
34 react r2(Sig1) = { // release lock
35     Process["access"] - Working || (Resource - Token["access"])
36 }, {
37     Process["access"] || (Resource - Token)
38 }
```

Description of the Program

→ Section 3.4.3

→ Section 3.4.5

The variable `startingState` in Line 16 declares the initial state with two processes. The declaration utilizes a feature described in Section 3.4.3 that automatically places the next nodes in parallel under one parent. (Here, it is the root node.) Another way would be to explicitly use the merge product operator `|` (Section 3.4.5). After the state declaration,

three rule definitions follow, see Line 22, Line 28 and Line 34. Basically, these rules implement the program execution behavior mentioned in the beginning of this section. They are generic and do not rely on the number of processes involved. This is exploited later at the end of this section.

Atfer the program's execution in Line 13, a transition system is generated which is depicted in Figure 10. This transition system was automatically exported by configuring the appropriate property (see Table 1). Figure 10 shows how each of the two processes are able to consecutively acquire a lock by connecting a link (either `access1` or `access2`) to a token under the resource `Resource` in order to access the resource and perform the actual work.

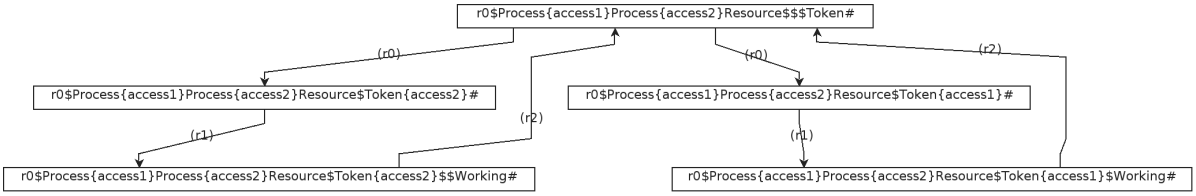

Figure 10: Generated transition system of the mutual exclusion problem as modeled in Listing 31. The states are bigraphs encoded as strings by using the *breadth-first string encoding* as described in [7].

The arc's labels and their direction of the transition system further show that the order of the reaction rules adhere to the operation specification, that is, first acquire lock (`r0`), do work (`r1`), finally release lock (`r2`). This is also apparent by the two cycles formed in the transition system. In other words, it is not possible for a process to acquire a lock, when the other process has access to the shared resource and is in the working state.

Extensions

The example presented exemplifies that the behavioral logic expressed in the program is very generic, as a result, the program can be extended with many more processes without any change of the core logic at all. Since *links* are used to distinguish between available processes, new ones can be easily added. For example: `val = startingState(Sig1): Process["a"] | Process["b"] | Process["c"] | ....`

6 Advanced Topics

6.1 User-defined Functions

User-defined functions (UDF) provide the basic functionality to externalize additional logic or behavior in form of arbitrary Java expressions. In other words, a BDSL program can be enriched by using the Java programming language, enabling to call specifically declared functions contained in a so-called *UDF archive* (see below). They are suited to be called after reactions occur in order to interact with the system's environment. For example, to communicate with OpenHAB to switch on a lamp in a smart home. This form of externalization facilitates strict separation of additional logic, and retains the core logic of a BDSL program. Moreover, UDFs can be tested independently by utilizing traditional test methods such as JUnit tests.

Collection of Functions ... ... UDF archive

Usually, a collection of UDFs are packaged within a `*.jar` archive allowing them to be conveniently imported in a BDSL program. These are usually called also *UDF archives*. This helps to organize UDFs that have similar concerns.

Importing an UDF archive

UDFs can be used within by importing their namespace via the `import:udf` statement. Their namespace resembles the Java `package` name under which the functions are located in the archive. A user can then refer to these classes by using its simple name, which represents the Java `class` name in the corresponding Java `package`. Without an `import:udf statement`, the *fully qualified name* must be used. This name corresponds to the Java `package`, under which the specific Java `class` is contained, and the name of the Java `class` itself.

! → Section 2.3

Logically, a BDSL program needs to know the location of UDF archives in order to be included first. The default here is the current directory from where the BDSL program is executed. To change the source locations from where to search for UDF archives and load them, see Section 2.3.[9]

Basic Usage

Three steps are necessary to utilize UDFs in a BDSL program, which are also explained in the following by a running example:

- UDFs must implement a specific interface from the *BDSL CE Framework* .
- A collection of UDFs must be packaged as a `*.jar` archive.
- They should be imported by their namespace (which corresponds to the Java `package` name under which they are placed).
- Functions defined in such a UDF archive can be directly called in a BDSL program by using their *simple name* or *full qualified name*.

The following explanation assumes basic knowledge on how to setup and package a Java project in combination with Maven, because further details are out of scope of this manual.

6.1.1 A Hello-World UDF Example

Implementing the Java Interface

→ Section 6.1.2

The *BDSL CE Framework* provides several Java interfaces depending on which kind of UDF one needs to implement (see Section 6.1.2). They differ from each other mainly in the type of the return value and type of arguments they accept.

[9] Therefore, the configurable property called `includeUdf` exists, which can be specified either via the command-line, or via an external configuration file.

For the first step, a new Maven project is created, with the dependency `bigraph-algebraic-interpreter-core` of the *BDSL Interpreter Framework* .[10] In this example, the UDF should print the character sequence "Hello, BDSL" to the console. Here, no argument is needed and no return value is necessary, thus, we use `ToVoidNoArgsFunction` from the `bigraph-algebraic-interpreter-core` dependency of the *BDSL CE Framework* . The content of that Java function is depicted in Listing 32. The interface `ToVoidNoArgsFunction` extends `BDSLUserDefinedConsumer<Void>` and provides a convenient way to implement UDFs that do not process a parameter and do not return anything.

Listing 32: A Java function showing the logic of the UDF for the running example.

```
1  package de.tudresden.inf.st.bigraphs.examples.interpreter.udf;
2  import de.tudresden.inf.st.bigraphs.dsl.udf.ToVoidNoArgsFunction;
3
4  public class HelloUdfFunction implements ToVoidNoArgsFunction {
5
6    @Override
7    public void accept(Void unused) {
8      System.out.println("Hello, BDSL");
9    }
10 }
```

After implementing the desired logic, the Java project needs to be compiled to produce the `*.jar` archive for the next step. For this example it is necessary to store the UDF archive next to the BDSL program which is presented in a moment.

! → If external dependencies are included beside using the standard Java functionality, it is necessary to build a so-called *fat jar* that includes all additional dependencies.

Importing the UDF Archive

The Java `package` name of the UDF archive (Line 1 in Listing 32) resembles the namespace to use in BDSL. To include the UDF archive, we write the following at the very top of a BDSL program (under the magic comment section): `import:udf de.tudresden.inf.st.bigraphs.examples.interpreter.udf.*`. Wildcards are allowed and work in the same way as for standard BDSL imports described in Section 3.3.3. Such an instruction is shown in Line 1 of Listing 33 that uses the special `import:udf` statement.

! → In case the UDF archive is not placed next to the BDSL program in the same folder, one can supply the command-line argument `--includeUDF` which accepts a comma separated list of filenames (see Section 2.3).

→ Section 2.3

Calling an UDF in BDSL

After importing the namespace of the corresponding UDF (i.e., the corresponding Java `package`), the next step is to call the implemented function (i.e., the corresponding Java class). The example in Listing 33 shows how to call the UDF `HelloUdfFunction` within the `main` block (Line 5) of a BDSL program.

[10] For more details, please refer to the documentation under this link `https://gitst.inf.tu-dresden.de/bigraphs/bigraph-algebraic-interpreter`.

Listing 33: Example on how to use UDFs in a BDSL program.

```
import:udf de.tudresden.inf.st.bigraphs.examples.interpreter.udf.*

main = {
  j {
    HelloUdfFunction()
  }
}
```

A Java expression, more specifically a BDSL UDF call, must be enclosed in a `j { ... }` block within curly brackets. Therein, multiple function calls are possible.

6.1.2 The Different Types of Functions

Return Values and Arguments

The *BDSL CE Framework* provides several Java interfaces to implement UDFs for different use cases, which vary with respect to their argument and return types. All BDSL-related UDF interface definitions are contained in the `de.tudresden.inf.st.bigraphs.dsl` module of the *BDSL CE Framework* under the Java package `de.tudresden.inf.st.bigraphs.dsl.udf`.

Basically, two core Java interfaces exist that can be implemented:

- `BDSLUserDefinedFunction<R>`: This interface specifies the method `R apply(UDFArgumentTypes var1)` which accepts one parameter and returns a value. The class `UDFArgumentTypes` represents, among others, all BDSL elements described in Section 3.4.

→ Section 3.4

- `BDSLUserDefinedConsumer<T>`: This interface specifies the method `void accept(T var1)` which accepts one parameter and does not return anything.

The type of the argument and return value can be freely specified. Note that only types inherent of BDSL can be used such as the variable types introduced in Section 3.4.

→ Section 3.4

! → An error will be thrown in case the number and types of arguments that need to be supplied to the UDF call in BDSL does not match with the Java implementation contained in the corresponding *UDF archive*.

Some Notes on the Execution

User-defined functions are usually executed in an asynchronous fashion in order to avoid blocking the program execution. Therefore, they shall only contain logic that is meant to be executed asynchronously.

6.2 Using the Interpreter Programmatically

The example presented here briefly outlines the programmatic integration of the BDSL interpreter in another application by using Java in combination with Maven as the build system. This programmatic approach is especially useful for developers who want to integrate the BDSL interpreter in custom applications without interacting with the command-line version of the interpreter through external scripts. The full build configuration (`pom.xml`), and both the Java and BDSL program are printed in Appendix C.

→ Appendix C

In light of the library case study described in Section 5.2, one could implement the following application. A BDSL program simply outputs

a bigraph variable to the console declared therein. The interpreter runs in a Java `for` loop and in every iteration a bigraph variable is randomly created that the main BDSL program uses.

The bigraph variable is created dynamically in Java using the *Bigraph Framework*. Afterwards it is exported to a file, more specifically, to an instance model file in the Ecore format (`*.xmi`). Then, the main BDSL program uses the `load()` method (Section 4.2) to load the bigraph model from the filesystem for assigning it to the variable declared in the BDSL program.

→ Section 4.2

Listing 34: Excerpt of the Java program showing on how to use the interpreter programmatically.

```
1  // This methods defines the core logic on how to call the BDSL
       interpreter
2  public void run(String... args) throws Exception {
3    CliExecutor cliExecutor = cliFactory.createCliExecutor();
4    CommandLineParser commandLineParser = cliFactory.
       getCommandLineParser();
5    CliOptionProcessor cliProcessor = cliFactory.
       createCliOptionProcessor();
6    DefaultDynamicSignature exampleSignature = createSignature();
7    for (int i = 0; i < 5; i++) {
8      // A random bigraph is created
9      PureBigraph generated = pureRandomBuilder(exampleSignature).
         generate(1, 6, 0.f);
10     // The instance and metamodel of the generated bigraph are saved
         on the filesystem
11     BigraphArtifacts.exportAsMetaModel(generated, dumpFolder.toPath())
         ;
12     BigraphArtifacts.exportAsInstanceModel(generated,
13       new FileOutputStream(
14         Paths.get(
15           dumpFolder.getAbsolutePath(),
16           "random-bigraph.xmi"
17         ).toString()
18     ));
19     // Command-line arguments are parsed ...
20     cliProcessor.process(commandLineParser, args);
21     // This statement actually calls the interpreter
22     cliExecutor.execute(cliProcessor.getProcessorResult());
23   }
24 }
```

→ Appendix C

Now to the Java program that is fully provided in Appendix C. An excerpt is shown in Listing 34 highlighting only the important parts with respect to the usage. The *BDSL Interpreter Framework* is implemented using the Spring Framework[11]. In this example, the program arguments are directly passed to the command-line processor object in Line 20 of Listing 34, which specifies the location of the BDSL program to parse. The `CliExecutor` object created in Line 3 is responsible for the actual execution, i.e., the interpretation, after the program was parsed. This is shown in Line 22.

[11] Spring Framework, `https://spring.io/` (last visited June 24, 2021).

6.3 IDE Support

The integrated development environment (IDE) specifically designed for BDSL provides a fast, platform-independent and flexible development environment for BDSL programs. The BDSL-IDE is available from `https://git-st.inf.tu-dresden.de/bigraphs/bdsl-textual-ide`, where all necessary installation instructions are provided as well.

Some of its features are:

- Platform-independent browser application
- Automatic syntax checking and early error reporting
- Auto-completion and syntax highlighting
- Outline of the program structure
- Creation of workspaces
- Git-like versioning support
- Execution of the interpreter in integrated terminal

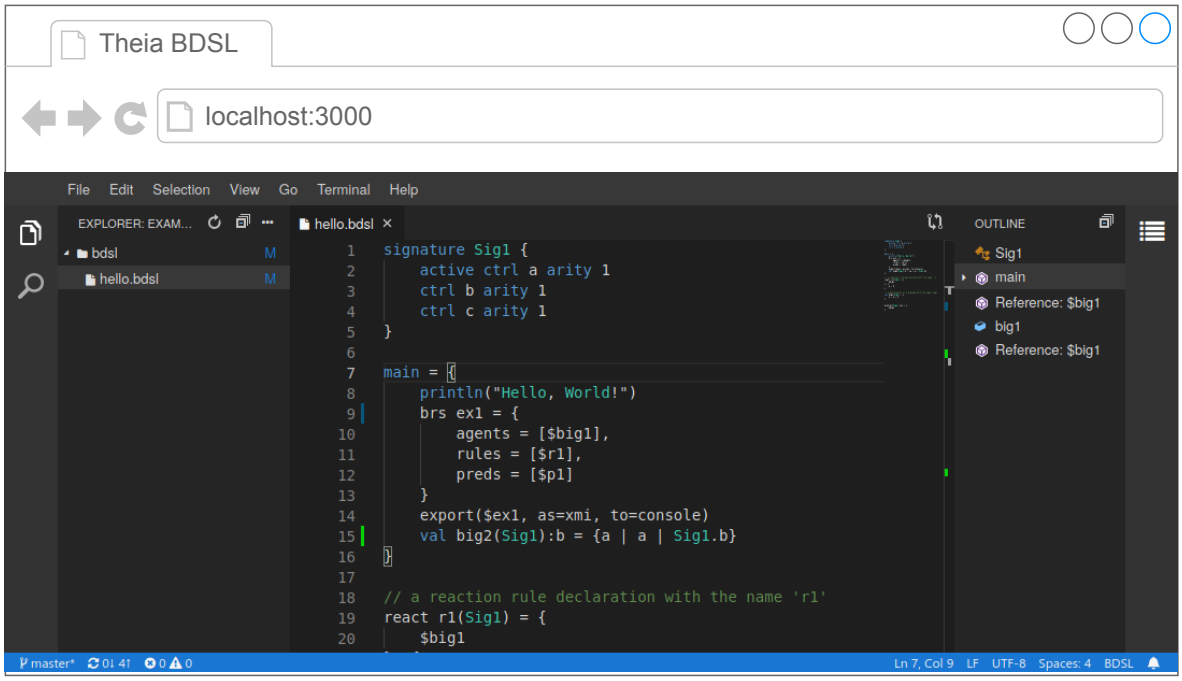

Figure 11: Screenshot of the BDSL-IDE that is based on Eclipse Theia.

7 Conclusion

Bigraphs allow to freely specify the syntax and semantic of reactive systems. One of the characteristics of bigraphs is the fact that they enable to change their structure by means of so-called reaction rules.

The intention of BDSL is the provision of a foundational framework to built upon other bigraphical model-driven domain-specific languages for the development of reactive systems.

The meta-modeling approach additionally facilitates the interoperability between different existing bigraph tools, thus, also enables the development of bigraphical tool chains [11].

Furthermore, software verification techniques can be employed that are going beyond traditional software testing methods such as functional unit test. Since bigraphs are a mathematical framework, it is possible to conduct mathematical reasoning and build programs based on a sound theory for reactive systems. Allowing to formally specify the semantic of a program and model checking it against some correctness properties, provides the formal means to design safe program.

7.1 Future Work

This section explores various directions for future work of BDSL. The main goals will be to add more language features for a greater support of a context-sensitive and rule-based programming approach, to extend the tool support of BDSL for end-users and developers alike, and finally improve the quality of the frameworks's code to enable more generalized and consistent grammar extensions for new domain-specific languages for reactive systems.

7.1.1 Language Features

One idea is to provide additional rule programming features and patterns such as priorities, re-invoking rules multiples times, rule inheritance (see [23]), or rule-based transformation of reaction rules (see [12]), to mention a few.

Another direction is to expand the possibilities that ease the construction of bigraphs. With regard to rule declarations it is planned to support direct loading of reaction rule files instead of individually assigning a bigraph to the redex and reactum of a RR (refer to Sections 3.4.6 and 4.2). To clarify, it is preferable to declare a rule like this: `react r = load(sig=Sig1, resourcePath="file:rule.xmi")`. Moreover, support for supplying instantiation maps for RR is planned.

→ Sections 3.4.6 and 4.2

The implementation of BDSL methods that accelerate the creation of bigraphs are also of focus for further work. One example could be the introduction of a `rep()` method that allows to conveniently create nodes in a specific way. To illustrate, `rep(a, op='|', times=3)` would be the equivalent of writing `a | a | a | a`.

Another emphasis lies on the declaration of bigraphical signatures. Signature inheritance is in some cases desirable, similar to rule inheritance. Therefore, the specification of an extension strategy must be supported that define whether duplicate controls should be overwritten, merged or deleted, for instance.

The *combination* of declared predicate variables (Section 3.4.7) is of in-

→ Section 3.4.7

terest. Predicates shall be connected via `and` and `or` operators to conveniently form more complex predicate expressions from simple ones.

7.1.2 Database Support

Currently only one option is available to supply a BDSL program to the interpreter. Namely, from the filesystem. Future work considers to pass a BDSL program also from a database. The same applies to the results that may be written into a database.

7.1.3 Interactive Shell

The functionality of the interpreter in *BDSL CE Framework* is designed to interpret any statement at any place in a BDSL program. The interpreter is able figure out the dependencies of the entities to evaluate next. In most cases, there is no extension necessary on prior or next statements to evaluate. This feature is independent of the integrated caching mechanism employed—omitting it would lead to the side-effect that the interpretation process will take more time.

REPL for Exploratory Programming and Debugging

→ Section 2

Thus, a so-called read-eval-print loop can be built easily. Having an interactive language shell for BDSL enables quick execution of single expressions and therefore facilitates experimentation[12] and debugging. In contrast to the interpreter presented in Section 2, where a BDSL document must be supplied, a REPL expects single expressions. A temporary environment is created to write the results and acquire current variables. When the shell is closed, the whole environment will be deleted.

[12] See `https://en.wikipedia.org/wiki/Exploratory_programming` (last visited June 24, 2021).

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Cambridge, Mass: The MIT Press, 2008. 975 pp. ISBN: 978-0-262-02649-9.

[2] L. Birkedal et al. "Bigraphical Models of Context-Aware Systems". In: *Foundations of Software Science and Computation Structures*. International Conference on Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar. 25, 2006, pp. 187–201. ISBN: 978-3-540-33045-5 978-3-540-33046-2. DOI: 10.1007/11690634_13. (Visited on 08/17/2018).

[3] Roberto Bruni et al. "On Hierarchical Graphs: Reconciling Bigraphs, Gs-Monoidal Theories and Gs-Graphs". In: *Fundamenta Informaticae* 134 (Jan. 1, 2014), pp. 287–317. DOI: 10.3233/FI-2014-1103.

[4] Muffy Calder et al. "Real-Time Verification of Wireless Home Networks Using Bigraphs with Sharing". In: *Science of Computer Programming* 80 (Feb. 1, 2014), pp. 288–310. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.08.004. (Visited on 11/23/2018).

[5] Ebbe Elsborg, Thomas T. Hildebrandt, and Davide Sangiorgi. "Type Systems for Bigraphs". In: *Trustworthy Global Computing*. Ed. by Christos Kaklamanis and Flemming Nielson. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 126–140. ISBN: 978-3-642-00945-7.

[6] Roberto Gorrieri. "Labeled Transition Systems". In: *Process Algebras for Petri Nets: The Alphabetization of Distributed Systems*. Ed. by Roberto Gorrieri. Monographs in Theoretical Computer Science. An EATCS Series. Cham: Springer International Publishing, 2017, pp. 15–34. ISBN: 978-3-319-55559-1. DOI: 10.1007/978-3-319-55559-1_2. (Visited on 12/05/2018).

[7] Dominik Grzelak and Uwe Aßmann. "A Canonical String Encoding for Pure Bigraphs". In: *SN Computer Science* X.X (2021), p. XXX. ISSN: 2661-8907. DOI: 10.1007/XXXXX.

[8] Dominik Grzelak, Barbara Priwitzer, and Uwe Aßmann. *Generating Random Bigraphs with Preferential Attachment*. Feb. 18, 2020. arXiv: 2002.07448 [cs]. (Visited on 02/19/2020).

[9] Thomas Hildebrandt, Henning Niss, and Martin Olsen. "Formalising Business Process Execution with Bigraphs and Reactive XML". In: *Coordination Models and Languages*. Ed. by Paolo Ciancarini and Herbert Wiklicky. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 113–129. ISBN: 978-3-540-34695-1.

[10] Dirk W. Hoffmann. "Software-Verifikation". In: *Software-Qualität*. Ed. by Dirk W. Hoffmann. eXamen.press. Berlin, Heidelberg: Springer, 2013, pp. 333–369. ISBN: 978-3-642-35700-8. DOI: 10.1007/978-3-642-35700-8_6. (Visited on 12/14/2019).

[11] Timo Kehrer, Christos Tsigkanos, and Carlo Ghezzi. "An EMOF-Compliant Abstract Syntax for Bigraphs". In: *Electronic Proceedings in Theoretical Computer Science* 231 (Dec. 4, 2016), pp. 16–30. ISSN: 2075-2180. DOI: 10.4204/EPTCS.231.2. arXiv: 1612.01638. (Visited on 11/23/2018).

[12] Rodrigo Machado, Leila Ribeiro, and Reiko Heckel. "Rule-Based Transformation of Graph Rewriting Rules: Towards Higher-Order Graph Grammars". In: *Theoretical Computer Science* 594 (Aug. 23, 2015), pp. 1–23. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2015.01.034. (Visited on 12/04/2019).

[13] Robin Milner. *The Space and Motion of Communicating Agents*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 978-0-521-73833-0.

[14] Rayene Moudjari, Zaidi Sahnoun, and Faiza Belala. "Towards a Fuzzy Bigraphical Multi Agent System for Cloud of Clouds Elasticity Management". In: *International Journal of Approximate Reasoning* 102 (Nov. 1, 2018), pp. 86–107. ISSN: 0888-613X. DOI: 10.1016/j.ijar.2018.07.012. (Visited on 02/04/2019).

[15] Gian Perrone, Søren Debois, and Thomas T. Hildebrandt. "A Model Checker for Bigraphs". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. New York, NY, USA: ACM, Mar. 26, 2012, pp. 1320–1325. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2231985. (Visited on 10/01/2018).

[16] J. Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning*. 2-Volume Set ed. edition. Amsterdam ; New York : Cambridge, Mass: The MIT Press, Sept. 1, 2001. 2150 pp. ISBN: 978-0-262-18223-2.

[17] Hamza Sahli, Faiza Belala, and Chafia Bouanaka. “Model-Checking Cloud Systems Using BigMC”. In: *CEUR Workshop Proceedings*. Vol. 1256. Sept. 29, 2014.

[18] M. Sevegnani et al. “Modelling and Verification of Large-Scale Sensor Network Infrastructures”. In: *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). Dec. 2018, pp. 71–81. DOI: 10.1109/ICECCS2018.2018.00016.

[19] Michele Sevegnani and Muffy Calder. “BigraphER: Rewriting and Analysis Engine for Bigraphs”. In: *28th International Conference on Computer Aided Verification*. CAV 2016. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Toronto, Canada: Springer International Publishing, July 17, 2016, pp. 494–501. ISBN: 978-3-319-41539-0 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_27. (Visited on 08/15/2018).

[20] John Stell et al. “Spatio-Temporal Evolution as Bigraph Dynamics”. In: *Spatial Information Theory*. Ed. by Max Egenhofer et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 148–167. ISBN: 978-3-642-23196-4.

[21] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013. 558 pp. ISBN: 978-1-4812-1858-0.

[22] Lisa A. Walton and Michael Worboys. “A Qualitative Bigraph Model for Indoor Space”. In: *Geographic Information Science*. Ed. by Ningchuan Xiao et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 226–240. ISBN: 978-3-642-33024-7.

[23] Manuel Wimmer et al. “A Comparison of Rule Inheritance in Model-to-Model Transformation Languages”. In: *Theory and Practice of Model Transformations*. Ed. by Jordi Cabot and Eelco Visser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 31–46. ISBN: 978-3-642-21732-6.

[24] Lars Wunderlich. *Java Rules Engines: Entwicklung von regelbasierten Systemen*. Frankfurt am Main: Entwickler.press, 2006. ISBN: 978-3-935042-75-8.

[25] Zhenchang Xing et al. “Differencing Labeled Transition Systems”. In: *Formal Methods and Software Engineering*. Ed. by Shengchao Qin and Zongyan Qiu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 537–552. ISBN: 978-3-642-24559-6. DOI: 10.1007/978-3-642-24559-6_36.

Appendix

A Configuration File for the BDSL Interpreter

Listing 35: A complete configuration file with all available configurable properties. Refer to Sections 2.1.1 and 2.3.2.

```
# BDSL - General
disableBanner=true # any value to disable the banner
main=main.bdsl # one filename
include=lib.bdsl,lib2.bdsl,lib3.bdsl # list of filenames
includeUdf=udf1.jar,../udf2.jar,config/udf3.jar # list of filenames
outputDir=./states/ # folder

# Model checking properties
model-checking.transitionOptions.maximumTransitions=1309 # Long value
model-checking.transitionOptions.maximumTime=1000 # Long value (
    milliseconds)
model-checking.exportOptions.printCanonicalStateLabel=false # "true"
    for simple labels for the transition system file below
model-checking.exportOptions.reactionGraphFile=./transitionSystem.png
model-checking.exportOptions.outputStatesFolder=./states/
model-checking.measure-time=false # for debugging

# Model properties
ns-uri=org.example
ns-prefix=sample
name=bigraph
encoding=UTF-8
schemaLocation=./metamodel.ecore # filename to the Ecore file
```

B BDSL Sample Programs

BDSL program examples that were abbreviated and mentioned in this manual are printed here in full length.

Listing 36: Source code of `test_bdsl_01.bdsl`.

```
// encoding: UTF-8
// ns-uri: http://www.example.org
// ns-prefix: sample
// name: F
// schemaLocation: test-1.ecore

signature Sig1 {
  atomic ctrl a arity 1
  atomic ctrl b arity 1
}

main = {
  println("Printing bigraph variable as Ecore model ...")
  println($bigVar)
```

```
15   brs example = {
16     agents = [$bigVar],
17     rules = [$testReact1],
18     preds = [$pred1]
19   }
20   println("Translating BRS to BigMC ...")
21   export($example, as=bigmc)
22 }
23
24 val bigVar(Sig1) = {a | b | b}
25 react testReact1(Sig1) = {$bigVar}, {a | b}
26 pred pred1(Sig1):iso = {a}
```

Listing 37: Source code of `bdsl_operators_01.bdsl`. Refer to Sections 3.4.4 and 3.4.5.

```
1  signature Sig1 {
2    active ctrl a arity 1
3    passive ctrl b arity 1
4    atomic ctrl c arity 1
5  }
6
7  // output all bigraphs defined below to the console
8  main = {
9    println($big1)
10   println($big2)
11   println($big3)
12   println($bigSubst1)
13 }
14
15 // creates three idle names "x", "y" and "z"
16 val big1(Sig1) = {
17   closure("x", "y", "z")
18 }
19
20 // closes the outer name and transforming it to an idle name, and
       removes the site of node a
21 val big2(Sig1): Sig1.a = {
22   closure("x") * (a["x"] - barren())
23 }
24
25 // creates for idle names "a", "x", "y", and "z"
26 val big3(Sig1) = {
27   closure("x") | clsre("z") | closure("y", "a") | clsre("z") | /("x")
28 }
29
30 // connects the outer name "a" with the two inner names "x" and "y"
31 val bigSubst1(Sig1) = {
32   subst("a", ["x", "y"])
33 }
```

Listing 38: Source code of `bdsl_operators_02.bdsl`. Refer to Sections 3.4.4 and 3.4.5.

```
signature Sig1 {
  active ctrl a arity 1
  passive ctrl b arity 1
}

main = {
  println($big1)
  println($big2)
}

val big1(Sig1) = {
  a | id(1) | b | id(2)
}

val big2(Sig1): Sig1.a = {
  a || id(1) || b || id(2)
}
```

Listing 39: Source code of `bigraph_innerVars_01.bdsl`. Refer to Section 3.4.3.

```
signature Sig1 {
  active ctrl a arity 1
  passive ctrl b arity 1
  atomic ctrl c arity 1
}

main = {
  println($big1)
  println($big2)
}

val big1(Sig1) = {
    val ex1:a = {
        a - b
        a - a
    }
    val ex2:a = {
        val foo:a = {
        }
    }
    val ex3:a = { $ex2 }
}

val big2(Sig1): a = {
    val ex1:a = {
        a - b
        a - a
    }
    val ex2:b = { }
    val ex3:c = { }
```

```
}
```

Listing 40: Source code of `bdsl-pathfinding.bdsl`. Refer to Section 5.3.

```
signature SigMap {
  Car: 1 // control status is "active" by default
  atomic Fuel: 0
  Place: 1
  Road: 1
  atomic Target: 1
}

main = {
  brs findPath(SigMap) = {
    agents = [$map],
    rules = [$moveCar],
    preds = [$targetReached]
  }
  execute($findPath)
}

val car(SigMap) = {
    Car["target"] - (Fuel | Fuel | Fuel | Fuel | Fuel | Fuel | Fuel |
       Fuel)
}

val map(SigMap) = {
  Place["p0"] - ($car | Road["p3"] - brn() | Road["p1"] - brn()) |
  Place["p7"] - (Target["target"] | Road["p2"] - brn()) |
  Place["p3"] - (Road["p7"] - brn() | Road["p4"] - brn()) |
  Place["p1"] - (Road["p2"] - brn() | Road["p4"] - brn()) |
  Place["p4"] - (Road["p1"] - brn() | Road["p5"] - brn()) |
  Place["p2"] - (Road["p5"] - brn())
}

react moveCar(SigMap) = {
  Place["fromD"] - id(1) | Place["fromS"] - (Road["fromD"] - brn() | (
     Car["target"] - (id(1) | Fuel)) | id(1))
}, {
  (Place["fromD"] - (id(1) | Car["target"] - id(1))) | (Place["fromS"]
      - (id(1) | Road["fromD"] - brn()))
}

pred targetReached(SigMap):partial = {
  Place["from"] - (id(1) | Target["target"] | Car["target"])
}
```

C Using the BDSL Interpreter Programmatically

The next listings are referring to the example presented in Section 6.2. The complete project is also available from `https://git-st.inf.tu-dresden.de/bigraphs/examples/bigraph-test-examples/-/tree/master/bdsl-interpreter-example`.

Listing 41: Main BDSL program called `example-01.bdsl` that is executed by the Java program shown in Listing 42.

```
signature Sig1 {
  atomic ctrl A arity 1
  atomic ctrl B arity 2
  atomic ctrl C arity 3
  atomic ctrl D arity 4
}

main = {
    println("<BDSL Program> \t Loading bigraph instance model now ..."
        )
  val bigvar(Sig1) = load(sig=Sig1, as=xmi, resourcePath="file:./dump/
      random-bigraph.xmi")
  println("<BDSL Program> \t Printing bigraph instance model now ...")
    println($bigvar)
}
```

Listing 42: The complete Java program is printed here showing how to implement the interpreter in order to run the script shown in Listing 41 .

```
package de.tudresden.inf.st.bigraphs.examples.interpreter;

import de.tudresden.inf.st.bigraphs.core.BigraphArtifacts;
import de.tudresden.inf.st.bigraphs.core.impl.DefaultDynamicSignature;
import de.tudresden.inf.st.bigraphs.core.impl.pure.PureBigraph;
import de.tudresden.inf.st.bigraphs.dsl.cli.CliExecutor;
import de.tudresden.inf.st.bigraphs.dsl.cli.CliFactory;
import de.tudresden.inf.st.bigraphs.dsl.cli.CliOptionProcessor;
import de.tudresden.inf.st.bigraphs.dsl.cli.configuration.
    BDSLExecutionProperties;
import de.tudresden.inf.st.bigraphs.dsl.cli.configuration.v1.
    BatchConfigurationV1;
import de.tudresden.inf.st.bigraphs.simulation.modelchecking.
    ModelCheckingOptions;
import org.apache.commons.cli.CommandLineParser;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.
    DataSourceAutoConfiguration;
import org.springframework.context.annotation.Import;

```

```
21 import java.io.File;
22 import java.io.FileOutputStream;
23 import java.io.IOException;
24 import java.io.InputStream;
25 import java.net.URL;
26 import java.nio.file.Files;
27 import java.nio.file.Paths;
28 import java.nio.file.StandardCopyOption;
29 import java.util.LinkedList;
30 import java.util.List;
31 import java.util.Objects;
32
33 import static de.tudresden.inf.st.bigraphs.core.factory.BigraphFactory
       .pureRandomBuilder;
34 import static de.tudresden.inf.st.bigraphs.core.factory.BigraphFactory
       .pureSignatureBuilder;
35
36 /**
37  * @author Dominik Grzelak
38  */
39 @SpringBootApplication( // Basic Spring annotations
40         scanBasePackageClasses = {ModelCheckingOptions.class,
               BDSLExecutionProperties.class},
41         exclude = {DataSourceAutoConfiguration.class}
42 )
43 @Import(BatchConfigurationV1.class) // necessary configuration for the
       interpreter
44 public class Main implements CommandLineRunner {
45     @Autowired
46     @Qualifier("cliFactory")
47     CliFactory cliFactory;
48
49     public static String dumpDir = "./dump/";
50     public static File dumpFolder = new File(dumpDir);
51
52     public static void main(String[] args) {
53         // Some preparation
54         String mainFile = "--main=" + getMainBDSLProgramPath("example
               -01.bdsl");
55         if (dumpFolder.mkdirs()) {
56             System.out.println("Output directory created: " +
                   dumpFolder.getAbsolutePath());
57         } else {
58             System.out.println("Output directory is: " + dumpFolder.
                   getAbsolutePath());
59         }
60
61         SpringApplication app = new SpringApplication(Main.class);
62         // The argument for the main program is configured directly
               here
63         List<String> argsNew = new LinkedList<>();
64         argsNew.add(mainFile);
65         app.run(argsNew.toArray(new String[0])); // and passed to the
               actual interpreter
```

```
66      }
67
68      // This methods defines the core logic on how to call the BDSL
            interpreter
69      public void run(String... args) throws Exception {
70          CliExecutor cliExecutor = cliFactory.createCliExecutor();
71          CommandLineParser commandLineParser = cliFactory.
                getCommandLineParser();
72          CliOptionProcessor cliProcessor = cliFactory.
                createCliOptionProcessor();
73          DefaultDynamicSignature exampleSignature = createSignature();
74          for (int i = 0; i < 5; i++) {
75              // A random bigraph is created
76              PureBigraph generated = pureRandomBuilder(exampleSignature
                    ).generate(1, 6, 0.f);
77              // The instance and metamodel of the generated bigraph are
                     saved on the filesystem
78              BigraphArtifacts.exportAsMetaModel(generated, dumpFolder.
                    toPath());
79              BigraphArtifacts.exportAsInstanceModel(generated,
80                      new FileOutputStream(
81                              Paths.get(
82                                      dumpFolder.getAbsolutePath(),
83                                      "random-bigraph.xmi"
84                              ).toString()
85                      ));
86              // Command-line arguments are parsed ...
87              cliProcessor.process(commandLineParser, args);
88              // This statement actually calls the interpreter
89              cliExecutor.execute(cliProcessor.getProcessorResult());
90          }
91      }
92
93      /**
94       * Helper method to resolve the real path of the BDSL program
             located in the resource directory of this project.
95       *
96       * @param programFilename the absolute path of the BDSL program to
              resolve
97       * @return the absolute path of the given BDSL program
98       */
99      private static String getMainBDSLProgramPath(String
            programFilename) {
100         URL resource = Main.class.getClassLoader().getResource(
                programFilename);
101         String mainBdslProgram = "src/main/resources/" +
                programFilename;
102         try {
103             File tempFile = File.createTempFile("bdsl-interpreter-
                    example_", ".bdsl");
104             InputStream inputStream = Objects.requireNonNull(resource)
                    .openStream();
105             Files.copy(inputStream, tempFile.toPath(),
                    StandardCopyOption.REPLACE_EXISTING);
```

```
106             mainBdslProgram = tempFile.getAbsolutePath();
107             return mainBdslProgram;
108         } catch (IOException e) {
109             e.printStackTrace();
110             return mainBdslProgram;
111         }
112     }
113
114     /**
115      * Must resemble the same signature as specified in the BDSL
             program
116      */
117     private DefaultDynamicSignature createSignature() {
118         return pureSignatureBuilder()
119                 .newControl("A", 1).assign()
120                 .newControl("B", 2).assign()
121                 .newControl("C", 3).assign()
122                 .newControl("D", 4).assign()
123                 .create();
124     }
125 }
```

Listing 43: Build configuration details of the `pom.xml` for the Maven-based Java program shown in Listing 42.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
                maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6      <groupId>de.tudresden.inf.st.bigraphs.examples.interpreter</
           groupId>
7      <artifactId>bdsl-interpreter-example</artifactId>
8      <name>bdsl-interpreter-example</name>
9      <version>1.0-SNAPSHOT</version>
10
11     <properties>
12         <java.version>11</java.version>
13         <maven.compiler.source>${java.version}</maven.compiler.source>
14         <maven.compiler.target>${java.version}</maven.compiler.target>
15         <!-- BDSL -->
16         <bdsl.interpreter.version>1.0.0-SNAPSHOT</bdsl.interpreter.
               version>
17     </properties>
18
19     <repositories>
20         <repository>
21             <snapshots>
22                 <enabled>true</enabled>
23             </snapshots>
24             <id>STFactory</id>
```

```
25              <name>st-tu-dresden-artifactory</name>
26              <url>https://stgroup.jfrog.io/artifactory/st-tu-dresden-
                    maven-repository/</url>
27          </repository>
28          <repository>
29              <id>sonatype-snapshots</id>
30              <url>https://oss.sonatype.org/content/repositories/
                    snapshots</url>
31          </repository>
32      </repositories>
33
34      <dependencies>
35          <dependency>
36              <groupId>de.tudresden.inf.st.bigraphs.dsl.interpreter</
                    groupId>
37              <artifactId>bdsl-interpreter-cli</artifactId>
38              <version>${bdsl.interpreter.version}</version>
39          </dependency>
40      </dependencies>
41
42      <build>
43          <plugins>
44              <plugin>
45                  <groupId>org.springframework.boot</groupId>
46                  <artifactId>spring-boot-maven-plugin</artifactId>
47                  <version>2.5.1</version>
48                  <executions>
49                      <execution>
50                          <goals>
51                              <goal>repackage</goal>
52                          </goals>
53                      </execution>
54                  </executions>
55              </plugin>
56          </plugins>
57      </build>
58  </project>
```