TRIK Studio: Technical Introduction

Dmitry Mordvinov
Department of Software Engineering
St.-Petersburg State University
St.Petersburg, Russia
Email: mordvinov.dmitry@gmail.com

Yurii Litvinov
Department of Software Engineering
St.-Petersburg State University
St.Petersburg, Russia
Email: y.litvinov@spbu.ru

***Аннотация*—This paper presents TRIK Studio — an environment for visual (and textual) programming of robotic kits, which is used in educational organizations across Russia and Europe. First part of the article provides overview of the system — its purpose, features, differences from similar programming environments, general difficulties of robot programming and solutions proposed by TRIK Studio. Second part presents implementation details of TRIK Studio and its most interesting components. This article combines five fields of study: robotics, domain-specific visual modelling, education, formal methods and methods of program analysis. Main contribution of this article is detailed technical description of TRIK Studio as a complex and successful open-source cross-platform robot programming environment written in C++/Qt, and first part of the article can also be interesting for teachers as it provides an overview of existing robot programming tools and related problems.**

I. Introduction

Current state of school education in computer science turned out to be quite like Seymour Papert predicted it to be. In 1967 he introduced a virtual Logo turtle, that is used to teach students programming at schools even nowdays. It is less known that Papert in his experiments also used a mechanical robot turtle, that was controlled from a computer [**?**], and it made educational process much more entertaining. Today Papert's ideas are widely spread, a lot of schools are using robots to teach programming (for instance, in Russia robotics is a part of compulsory education program within the Technology course [**?**], [**?**]). Several robotics educational kits are used, including Lego Mindstorms NXT, Lego Mindstorms EV3 [1], TRIK[2] etc.

The task of programming a robot is more complex than programming a virtual turtle: the program will be composed of manipulating motors power and sensor values instead of simple movements and turns. Therefore a lot of attention is paid to robot programming environments. Many of them are based on visual, diagram languages since they are more intuitive and easy to learn than textual ones. Programming in such visual environments is performed via drag-and-dropping blocks using mouse, and it makes programming available to small kids, that cannot even read yet.

Popularity of visual languages in educational robotics is proven by a number of diagram-based programming environments in this field. The most known are Robolab [**?**], NXT-G [**?**] and EV3-G [**?**], Scratch [**?**] and Scratch-based environments (S4A [**?**], mBlock [**?**], Enchanting [**?**], ScrathDuino [**?**], Blockly [**?**] and App Inventor [**?**], 12Blocks [**?**], Open Roberta [**?**], Ardublock [**?**]), less known are Robo PRO [**?**] for programming robots created with fischertechnik robotics kits or Program Maker for Scribbler robots and several tools for preschoolers: Lego WeDo Software [**?**], Create [**?**], Wonder [**?**].

For the last decade educational robotics has been and still is a very promising scientific field, so in 2010s almost every major university in the world invested in development of own projects in robotics. For example, in 2016 Harvard University presented Root platform [**?**], Carnegie Mellon University develops and promotes its Arts&Bots project [**?**], MIT contributed delivering Scratch environment [**?**] and so on. Detailed overview in Russian could be found in [**?**], and the author concludes there that despite the variety of tools in this field there is no single one that can satisfy the needs of all educational organizations. Vast majority of such tools implement only the basic functionality (visual editor for diagrams and an ability to run programs on a robot autonomously or in a controlled manner from the computer), but they lack more advanced tools for teaching programming. Examples of such tools could be generation of readable texual code from created diagrams (that will help students to migrate from diagram-based to code-based programs), tools for debugging the program using virtual robot simulator or embedded tools for checking correctness of created programs (which could help teachers checking students' tasks). Some tools do have some of these advanced features, but often only some of them, and most of such programming environments are proprietary (which puts their users into a vendor lock-in situation) and pretty expensive for lots of schools.

Almost all aforementioned visual languages are based on control-flow computational model. Such a model is easier to understand, so it suits the purpose of education well. Nevertheless this model is not the best for programming robots since they are highly reactive by their nature: the program controlling the robot is basically a transformation of sensor signals into motor impulses. Reactive models are best expressed in data-flow languages [**?**], where the program consists of a number of «black boxes», connected with data channels. Each such «box» (we will call it *block*) has a fixed number of inputs and outputs, and its job is to transform input data into output data (we will call data transfering through channels *tokens*>). For instance, each robot's sensor is represented by a single block, that simply sends tokens with sendors data to its output. Many researchers note usability of visual data-flow langiages comparing to texual ones [**?**], in particular because of clear visualization of data flows.

The idea of using data-flow programming languages in

[1]LEGO Mindstorms homepage, URL: http://www.lego.com/en-us/mindstorms (accessed: 07.02.2017)

[2]TRIK robotics platform homepage, URL: http://www.trikset.com/ (accessed: 07.02.2017)

robotics is not a novel one, they are implemented in almost every toolset for programming industrial and laboratory automation system. Among them are LabVIEW by National Instruments [**?**], Simulink [**?**], and Microsoft Robotics Developer Studio [**?**]. All of them solve the automation task very well and have very powerful tools inside (for example, Microsoft Robotics Developer Studio is run on MySpace social network servers [**?**]), but are very complex and hard to learn, so even if they are used in education, only in universities [**?**], [**?**]. School experiments with LabVIEW were popular in the late 1990s [**?**], [**?**], which resulted in adaptations of this environment (like Robolab), where data-flow model was replaced with control-flow model. So, a gap between simplified languages suitable for education and more elegant, but complex industrial languages exists. Some reseachers are trying to adapt the data-flow model for educational robotics, for example a research group from New Zealand introduced the RuRu [**?**] language. Nevertheless to our best knowledge there are still no production-ready data-flow based programming environments for educational purpose today.

Current paper describes TRIK Studio programming environment, an attempt to solve the problem mentioned above. The success in solving them is indirectly acknowledged by its applications: currently TRIK Studio is widely used in Russian education organizations (almost a hundred of schools and robotic clubs), several cases of TRIK Studio usage in organizations in Great Britain and France are known, there are TRIK Studio users on every inhabited continent. This paper presents technical overview of the programming environment, educational and methodical issues are not discussed in detail.

II. General description

TRIK Studio is an environment that allows to program robots using diagram-based and textual languages. It emerged as a further evolution of QReal:Robots [**?**] project, developed at Software Engineering chair of Saint-Petersburg State University. The official release supports Lego Mindstorms NXT, Lego Mindstorms EV3 and TRIK robotic kits. Each one of these kits can be programmed using one of two visual languages (the more simple control-flow and more complex data-flow language) or one of a number of texual ones. For Lego NXT the programmer can choose from NXT OSEK C and Russian version of C language (simplified for teaching textual programming), for TRIK — JavaScript, F# [**?**] or PascalABC.NET [**?**], for Lego EV3 the single official language for standard firmware, EV3 virtual byte code, is supported.

A program created using one of visual languages (*a visual program*) could be executed in one of three modes:

- debugging using virtual simulator,
- debugging on a PC while sending commands to a robot via USB, Bluetooth or Wi-Fi (see VI),
- textual code generation mode with subsequent upload and execution of the program on the robot.

The the first mode the program is being interpreted on a two-dimentional robot model (see Section X). Users are able to create 2D-model of the world surrounding the robot out of walls, colored elements and floor markup. According to TRIK Studio users this feature is very useful for initial program debugging, before any interaction with real robots. Our experience shows that this virtual model editor allows to recreate most of fields and obstacle courses used in competitions in robotics. Having such a simulation environment makes possible learning robotics and programming even without having the real robotic kits. There is also an experimental support for V-Rep 3D visual simulation environment [**?**].

Debugging using a PC (so called *interpretation mode*) is useful as a next step to see how the program behaves on a robot in real time. In this mode program variables' values can be observed in a special window (similar to how it is done in Watch list windows in almost all textual IDEs) and sensor data can be displayed in a form of graphs.

Code generation mode enables users to move from visual to textual programs. Gerenated code is displayed in an embedded QScintilla-based [3] text editor, that provides a set of common features of a code editor: syntax highlighting, code audocompletion, undo/redo, brackets highlighting etc. TRIK Studio installation contains everything neede for buiding and uploading programs into a robot (a number of cross-compilers, WinSCP, Putty, etc.), and it allows to make compilation process and communication with robots transparent to TRIK Studio users.

TRIK Studio's user interface is shown on Fig. 1. It displays process of debugging a program that passes the maze using virtual simulation model.

In 2D simulation mode the automatic task checking feature is available (see Section XI). Task checking program is written in internal event-based textual language, the file containing virtual world model and task checking program can be distributed as a task for students. Based on this feature a remote course has been launched on Stepik platform[4] containing video lectures on cybernetics and robotics, numerous small tests and more that 20 tasks on educational robotics. Each task can be downloaded, solved, checked within TRIK Studio on a student's computer, and uploaded to the server where the checking system will run it on its own tests (similar to ACM ICPC coding contests).

TRIK Studio is written in C++ using cross-platform Qt[5] framework, there are installers for Windows, Linux and Mac OS X operating systems. Because of the fact that official Lego NXT drivers are not available on Linux and Mac OS X x64, TRIK Studio contains our own implementation for them (see Section VI for details). The programming environment is completely free to use, open-source and is distributed under Apache License 2.0[6].

Aforementioned features distinguish TRIK Studio from all other such tools. In fact among all mentioned in Section I tools only 12Blocks have a comparable feature set, but it generates complex and unreadable code (for Lego NXT), does not have tools for checking tasks, is not free to use and does not have Russian localization in it. Among the TRIK Studio drawbacks we should mention weak methodological support, limited only

[3] https://riverbankcomputing.com/software/qscintilla/intro (accessed: 07.02.2017)

[4] https://stepik.org/s/7qe3xj4Z (accessed: 07.02.2017)

[5] https://www.qt.io/ru/ (accessed: 14.05.2016)

[6] http://www.apache.org/licenses/LICENSE-2.0 (accessed: 07.02.2017)

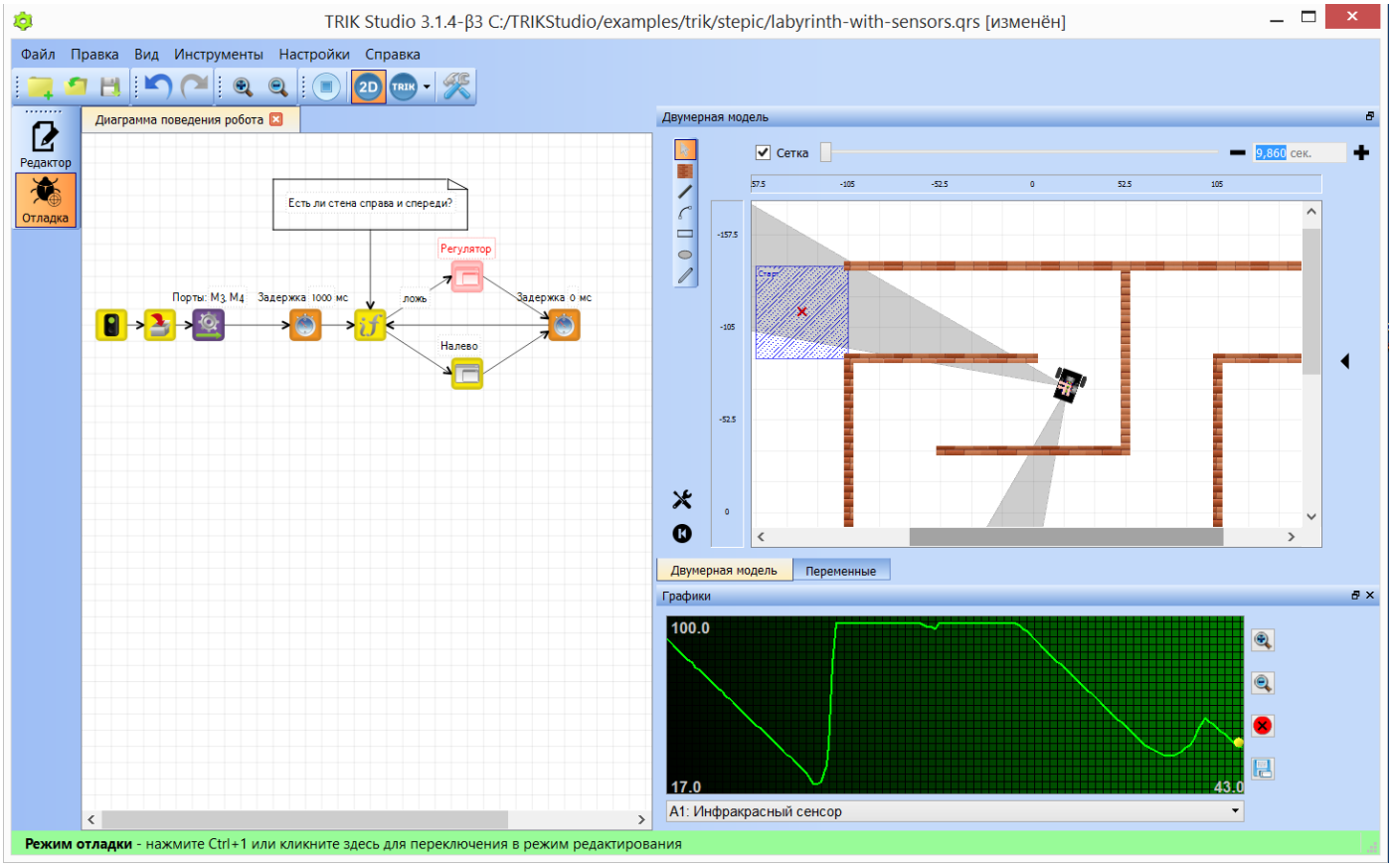

Рис. 1: TRIK Studio's user interface

by embedded reference guide in Russian, a set of examaples and the remote video course. Less important drawbacks will be mentioned in the following sections.

The rest of the paper is structured as follows. Sections III and IV briefly present TRIK Studio visual languages and their interpreters. Section V overviews the system architecture. Further sections describe separate TRIK Studio subsystems. Section VI presents robots communication infrastructure. Section VII describes visual language interpreters. The most interesting implementation details of code generators from control-flow language are provided in Section VIII. Section IX presents textual languages parsing subsystem. Section X presents implementation details of 2D simulation model sybsystem. Section XI describes task checking language and Section XI concludes the paper.

III. Visual language for beginners

The simplified control-flow based lanaguage (see Fig. 2) is the most often used one in TRIK Studio. It it a graph language, i.e. the program consists of nodes (*blocks*) and edges, organizing the nodes into a control flow. To create a program users drag-and-drop necesary blocks onto the editor's scene, sets their properties and connects them with arrows. While being run each block executes a sequence of basic commands and passed control to outgoing edges (to all or some of them depending on the block semantics).

All language blocks could be devided into four groups.

- The first group is for blocks supporting basic algotithmic expressions, like the start and the end of a program or a subprogram, conditions, switches, arithmetic loops, blocks for execution parallelization and for working with concurrent tasks (e.g. for merging and interruprting tasks), subprogram call and a block for textual programming.
- The second group combines blocks working with robot peripherals. These are actions that don't require waiting. For example, they are blocks for setting motor powers, playing sounds, accessing robot video processing capabilities, synthesizing speach by given text, controlling motor encoders and LEDs, sending messages to other robots (a part of multi-agent interaction support), working with robot file system, etc.
- The third group consists of blocks, that «freeze» the control flow. They are a timer block waiting for a given number of milliseconds (similar to `msleep` function), blocks waiting for a given value from a given sensor or from an operator gamepad, and a block waiting for receiving a message from another robot.
- And finally, the fourth group contains blocks that provide drawing functions for basic primitives (the drawing is being performed on a robot's screen). Such primitives are lines, rectangles, ellipses, arcs, text, images. There are parameters to change color and width of pen and brush used in drawing. There are also

special blocks that control robot model marker in a 2D simulation environment. It allows robot model to leave a trace on a ground while moving, similar to the Logo turtle.

Properties of each block can be set both within the diagram editor's scene and using special property editor window. All block properties (where applicable) support computable expressions written in TRIK Studio embedded textual language – a statically typed Lua[7] dialect. The parsing module for this parguage is written using a parser combinator library for C++11, also created within TRIK Studio project. Type inference of the resulting abstract syntax tree is performed using slightly simplified Hindley-Milner algorithm [**?**].

The domain-specific approach (*DSM-approach*) [**?**] was employed to create the described visual language. An editor was created using QReal DSM platform [**?**], [**?**]. Language metamodel was defined using QReal's metaeditor, and a plugin implementing the editor was automatically generated using QReal's tools. Plugging this module into the QReal's platform core a complete visual IDE based on the given language is obtained. This IDE «inherits» all tools and features of the DSM platform, including modern user interface, mouse gestures recognition support for creating diagram elements [**?**], [**?**], copy-pasting and undo/redo frameworks, zooming tools, tools for creating several types of edges on diagrams, model explorers, touch screens support and many more. According to its users TRIK Studio's user interface is much more usable and ergonomic than in any other such programming environment, and we spent about three man-days on it. We believe that it acknoledges the choice of QReal DSM platform as an underlying technology, but we have to note that while creating tool support for TRIK Studio numerous improvements were made to the QReal platform itself.

IV. Visual language for advanced users

Users that mastered the control-flow language can move on to more complex, but more convenient data-flow visual language. Unlike control-flow programs where control is passed according to edges between nodes, in data-flow programs blocks are executed simultaniously and communicate with each other by sending data tokens via channels. For instance, a data-flow program can have several entry points (most often they are robot sensors that send data each other for processing in a chain), while a control-flow program must always have a single entry point. This language also has block for basic algotithmic expressions, blocks for interaction with all robot devices, drawing blocks, etc. Almost always data-flow programs turn out to be more concise than their control-flow analogues. For example, Fig. 3 shows proportional controller implemented in control-flow language is compared with proportional-derivative controller implemented in data-flow language.

To make transition to this «advanced» language simpler it maintains some kind of conceptual compatibility with the control-flow language. Blocks can also be organized in a chain with explicit passing of control, to make it possible most blocks have a special activation port, that ignores all input data and just executes the block like it was in control-flow case. So users can write programs like they used to and move on to using smaller number of blocks as they gain more experience.

[7]http://www.lua.ru/ (accessed: 07.02.2017)

Expressive power of the language allows to use it for creating well-known complex robot control systems like Rodney Brooks's categorical architecture [**?**], Johnathan Connell's «colony» architecture [**?**], Ronald Arkin's behavioral navigation [**?**], or DAMN distributed navigation approach [**?**]. Proof of this statement worth a separate paper and is not given here. Another ideas on this matter could be found, for exmaple, in [**?**] or [**?**].

It worth noting that in the time of writing this paper TRIK Studio has only experimental support for data-flow language and it is not included in the official distributed package. The source code of the editor and all appropriate tool support is freely available[8] and could be compiled with the TRIK Studio's source code.

V. General architecture

This section aims to structure the information given previously describing TRIK Studio's architecture: most of the features described here were already mentioned above. All diagrams presented here are distilled of numerous architectural and implementation details that though could be interesting in a context of such a paper, still left aside limited by the paper size.

Fig. 4 presents high-level architecture of TRIK Studio. The system has a number of layers, each one implementing specific set of functionality and having strictly defined API. The lowest layer is libraries implementing interaction with real robots and and virtual robot models. They are used to implement a hierarchy of robot devices (sensors, motors, displays, speakers, gamepads, control buttons, etc.), which in their turn are used to specify high-level robot models for different robotic kits. Such robot model specifications are grouped into modules that are plugged into TRIK Studio's core, where they are used by other subsystems (e.g. intrepreters and code generators that are structured as plug-in modules too).

TRIK Studio core in its turn is a plug-in module for QReal DSM platform. It modifies QReal's user interface, adding several toolbars and windows, like 2D simulation window, sensor configuration window, variable watch list and graphs window, etc. The core load all robot model specifications and provides them for all other subsystems, provides user with information about interpretation and code generation processes, etc. Along with modules implementing TRIK Studio tools, visual language modules (that are created using the QReal platform itself) are also plugged into the QReal core. Implementation details of lower layers from Fig. 4 are provided in the subsequent sections.

Making the architecture highly modular allowed to provide flexible customization of the installation process: users can explicitly select which components they are interested in and install only them (for example, if someone has only Lego EV3 kit, he or she does not need support for Lego NXT and TRIK robots, and the target installation will not contain such files). Futhermore, such an approach is highly compatible with packet-based software install systems in Linux. The architecture also benefits from such separation since the components become very low coupled: TRIK Studio's core, for example, is very minimalistic, it does not «know» about all features that the IDE provides and contains only objects and interfaces common to all

[8]https://github.com/ZiminGrigory/qreal/tree/DFVPL (accessed: 07.02.2017)

Рис. 2: The program of traversing the maze using right-hand rule. The first four blocks initialize the program: set the wall proximity variable, turn the robot motors on, after that the robot moves for one second to enter the maze. The second four blocks define the main control loop. Each ten milliseconds the robot checks using its infrared sensors if it can move forward or right (the `sensorA2 <= threshold && sensorA1 <= threshold` condition). It it can move forward, the movement will be performed using proportional control over the right sensor data («Регулятор» subprogram). If there are walls in front of the robot and to the right, it turn 90 degrees left («Налево» subprogram)

.

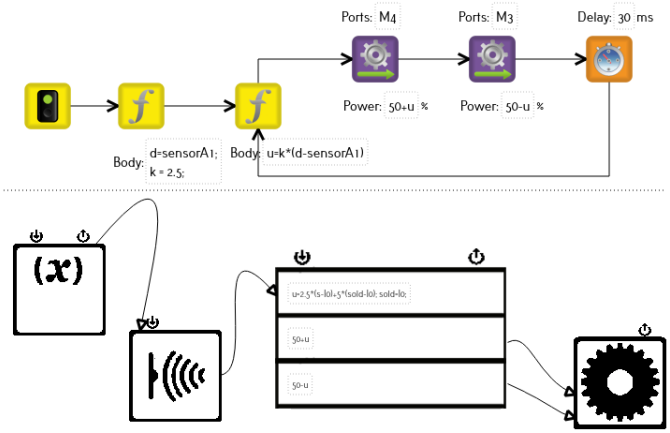

Рис. 3: Controllers implemented in different visual languages

robotic kits and models. API of each layer is fixed and well-documented, so it allows independent community developers to create extention modules supporting new languages, robotic kits and tools.

VI. Communications with robot controllers

Одна из важных подсистем TRIK Studio — это модуль взаимодействия с контроллерами роботов по тому или иному физическому протоколу. На рисунке 4 это один из самых низких «слоев» абстракции среды. API модуля коммуникаций предоставляет следующие операции:

- переключение текущего физического способа взаимодействия с целевым устройством (контроллером робота) между Bluetooth, USB и Wi-Fi,
- указание адреса устройства в терминах физического протокола (это может быть номер COM-порта в случае с взаимодействием по Bluetooth, IP-адрес или имя хоста в случае с взаимодействием по Wi-Fi),
- подключение-отключение к целевому устройству,
- посылка набора байтов на целевое устройство,
- события получения набора байтов с целевого устройства,
- события изменения статуса подключения (такие события вырабатываются, если связь с роботом установилась или разорвалась),
- события, оповещающие обо всех произошедших ошибках с локализованным их описанием.

Взаимодействие по USB происходит посредством библиотеки libusb[9], для реализации обмена по Bluetooth используется библиотека QextSerialPort[10], взаимодействие по Wi-Fi происходит поверх протоколов TCP и UDP посредством библиотеки QtNetwork, сложные протоколы коммуникации (такие как конфигурирование робота ТРИК перед запуском программы на исполнение) реализованы посредством Qt State Machine Framework. Процесс общения с целевым устройством происходит в отдельном потоке, чтобы

[9] http://libusb.org/ (дата обращения: 14.05.2016)
[10] https://github.com/qextserialport/ (дата обращения: 14.05.2016)

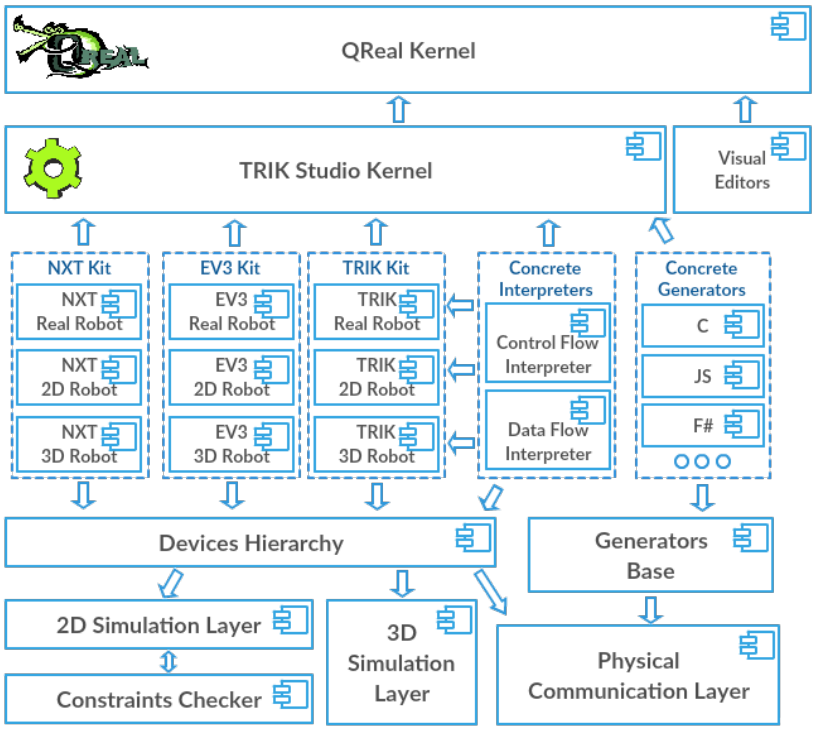

Рис. 4: High-level TRIK Studio architecture

не «замораживать» пользовательский интерфейс. Конкретные реализации механизмов коммуникации в подсистеме могут быть расширены и подменены другими компонентами «извне», это используется, например, для совместимости TRIK Studio со стандартным драйвером Lego NXT. Отметим, что взаимодействие через него — это не единственный способ общения с контроллером NXT по USB в TRIK Studio, в среде реализован собственный драйвер взаимодействия с Lego NXT, однако, в отличие от официального, работающий на всех поддерживаемых операционных системах через libusb.

VII. Interpreters

Интерпретатор преобразует визуальную диаграмму в последовательность команд на целевом устройстве (рис. 5). При этом иерархия устройств в системе построена таким образом, что не делается различий, реальное ли это устройство или симулируемое (рис. 6). Каждое устройство представляет собой класс C++, содержащий код взаимодействия с целевым механизмом. Реализации устройств какого-либо конструктора находятся в соответствующих библиотеках подключаемых модулей. Для общения по физическим каналам устройства используют подсистему коммуникаций (гл. VI), для передачи команд двумерному и трехмерному симуляторам — API этих симуляторов.

Интерпретаторы, также как и вся система в целом, написан на языке C++ с использованием инструментария Qt. На вход подсистеме интерпретации поступает описание созданной пользователем диаграммы в некотором внутреннем представлении. В среде реализовано два интерпретатора: интерпретатор языка для начинающих с передачей управления

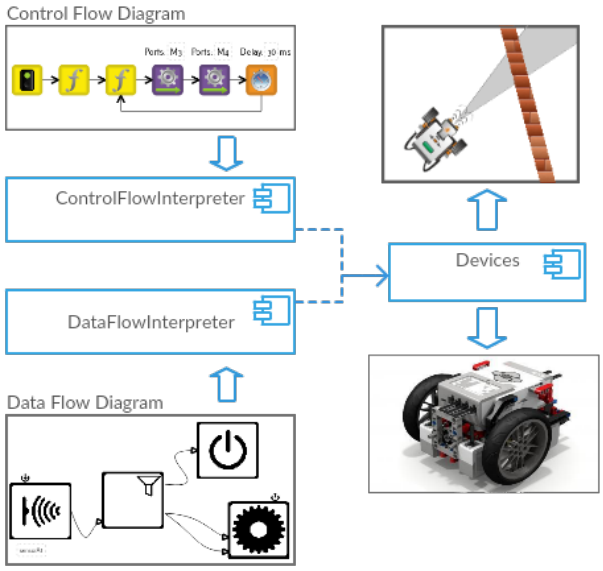

Рис. 5: Общий принцип работы систем интерпретации

и интерпретатор потокового языка.

Интерпретатор первого языка находит на диаграмме начальный блок, и далее трассирует поток управления диаграммы в том порядке, как он задан стрелками. Текущий блок исполнения подсвечивается на визуальной диаграмме, чтобы пользователю был виден прогресс исполнения. Для каждого посещенного блока специальные фабрики интерпретатора создают объекты, которые реализуют его поведение. В общем

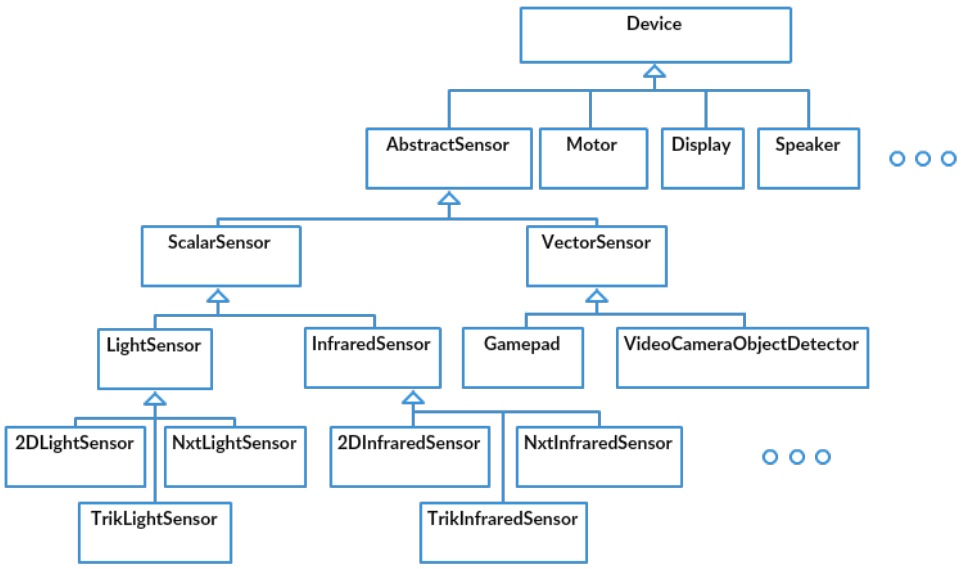

Рис. 6: Часть иерархии устройств в системе

случае, такой объект-реализация ищет среди готовых к работе устройств робота необходимые ему, вызывает соответствующие команды и передает управление следующим блокам по какой-либо исходящей ветке (которая определяется, опять-таки, самой реализацией блока). При этом не делается различий, является ли найденное устройство частью реального робота или симулируемым, таким образом, одна и та же подсистема интерпретации используется для исполнения диаграммы в двух режимах из трех (гл. II).

Если на каком-то шаге интерпретации повстречался блок распараллеливания, интерпретатор запускает новые потоки исполнения с соответствующих точек, создавая для каждого свой стек вызовов. Если встретился вызов подпрограммы, вычисляются ее параметры, укладываются на текущий стек вызовов, и далее процесс интерпретации повторяется рекурсивно. При достижении любого завершающего блока со стека снимается верхний фрейм, и исполнение продолжается с соответствующей точки на предыдущей диаграмме. Если при снятии очередного фрейма стек вызовов стал пустым, текущий поток исполнения считается завершенным. Таким образом, программа исполняется по мере того, как она «открывается» интерпретатору.

Отличительной чертой такой реализации является тот факт, что все изменения, вносимые пользователем на диаграмму во время процесса интерпретации, «подхватываются на лету». Это является удобным, например, в случае подбора коэффициентов пропорциональности какого-либо регулятора, реализованного в программе. При этом процесс исполнения не изменившихся частей диаграммы оптимизирован: объекты-реализации блоков кэшируются в отдельную таблицу, то же происходит с абстрактными синтаксическими деревьями и информацией о выводе типов выражений на текстовом языке. Таким образом, если содержимое программы не меняется во время процесса интерпретации, повторная передача управления в ветки программы повлечет использование уже созданных сущностей. Валидация диаграммы также осуществляется «на лету», в случае, если в диаграмме найдено некорректное место, пользователю отображается локализованное сообщение об ошибке. Последняя черта может рассматриваться как отрицательная, так как сообщение об ошибке появляется лишь в момент ее достижения в программе, что, впрочем, типично и для широко используемых текстовых интерпретируемых языков.

Интерпретатор потокового языка работает в два этапа. На первом, подготовительном этапе диаграмма проходит валидацию, создаются объекты-реализации элементарных блоков, преобразующие диаграмму в команды на устройствах роботов подобно тому, как это происходит в интерпретаторе языка, описанного в предыдущей главе. Объекты-реализации соединяются друг с другом посредством механизма сигналов и слотов инструментария Qt в соответствии с тем, как они соединены потоками данных на диаграмме; здесь оказался полезным паттерн проектирования «издатель-подписчик». На втором этапе интерпретатор запускает на автономное исполнение каждый из блоков, в который не входит ни один поток данных, каждый из которых, в свою очередь, будет активировать блоки, соединенные с ним выходными потоками данных при выработке токенов. Исполнение блоков происходит псевдопараллельно с централизованной очередью сообщений (рис. 7). Это решение отличает данный язык от всех его промышленных аналогов (к примеру, в Microsoft Robotics Developer Studio диаграмма разворачивается в набор независимых веб-сервисов [**?**]), его причины состоят в узкой направленности языка на «слабые» контроллеры учебных роботов, в которых параллельное исполнение большого количества блоков затруднительно или вовсе невозможно. Тем не менее, в языке все же присутствует блок распараллеливания, который полезен, например, для выражения вышеупомянутых подходов к проектированию сложных систем управления. Такой блок может рассматриваться как механизм низкоуровневого управления вычислительными ресурсами в языке.

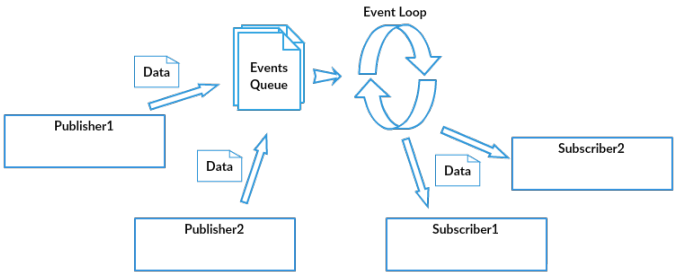

Рис. 7: Механизм исполнения потоковой программы в TRIK Studio

VIII. Generators

Одна из наиболее важных и востребованных функций среды TRIK Studio — генерация читаемого кода по визуальной диаграмме. По одной и той же диаграмме может быть сгенерирован код на любом поддерживаемом средой текстовом языке (C, JavaScript, F#, Pascal, РуСи [**?**], байткод BM EV3). Под визуальными диаграммами в данном разделе будем подразумевать программы на языке для начинающих (с передачей управления), под генераторами — модули генерации кода по визуальным диаграммам именно этого языка.

Визуальная диаграмма состоит из блоков и стрелок, таким образом, описывает *граф потока управления* программы [**?**]. Все основные алгоритмические конструкции (развилки, циклы, конструкции выбора и т.д.) складываются из стрелок. Например, чтобы задать бесконечный цикл, достаточно лишь провести стрелку от блока к одному из предыдущих, для описания циклов вида `while-do`, `do-while` или с инструкцией `break` внутри тела достаточно лишь провести одну из веток развилки из тела цикла к соответствующему блоку вне его. Очевидно, что одна стрелка на диаграмме соответствует инструкции `goto` в коде. Однако код, содержащий инструкции `goto` весьма труден для чтения человеком, тем более не подходит для обучения азам текстового программирования, поэтому следует избегать их появления в сгенерированном коде. Тем не менее, на TRIK Studio возможно написать программу, не выразимую общепринятыми алгоритмическими конструкциями, в таком случае появления инструкций `goto` в коде не избежать (существуют методы автоматического преобразования программы с goto в структурную программу, но результат их работы не лучше с точки зрения обучения текстовому программированию). С другой стороны, далеко не все современные текстовые языки поддерживают инструкцию `goto` (например, в языке JavaScript такой поддержки нет). Будем называть *приводимыми* диаграммы, которые можно выразить стандартными алгоритмическими блоками (`if-then`, `if-then-else`, `while-do loop`, `do-while loop`, `while-break loop`, `switch`), код на текстовом языке, соответствующий структурированной диаграмме также будем называть *структурированным*. В противном случае будем говорить, что диаграмма *неприводима*, а код будем называть *неструктурированным*. В случае, если диаграмму не удается сгенерировать в структурированный код, пользователю должно быть показано предупреждение.

Таким образом, реализация системы генераторов в TRIK Studio потребовала решения двух нетривиальных задач, которые сформулированы ниже в виде требований.

- Система генераторов должна быть организована таким образом, чтобы добавление в систему генератора в новый текстовый язык занимало минимальное количество времени и усилий. По возможности, это должно быть по силам даже людям, не знакомым с кодом системы.
- Для каждого языка (где это возможно) система должна поддерживать два режима генерации: приводимые диаграммы должны быть сгенерированы в структурированный код, неприводимые — в неструктурированный. При этом успешность структурирования диаграммы не должна зависеть от ее размера и сложности.

Первая задача является чисто архитектурной. Ее решение представлено на рисунке 8. Основная идея — разделение процесса генерации на два этапа. На первом диаграмма преобразуется в представление, независимое от целевого языка генерации — *семантическое дерево*. Семантическое дерево упорядочивает структуру графа потока управления до «древесной», в независимости от того, является ли диаграмма приводимой или нет. В случае, если диаграмма приводима, родительский узел семантического дерева всегда соответствует блоку вычислений в целевом коде, дети — инструкциям этого блока, среди которых нет `goto`. Опишем последовательность действий, отображенных на рисунке 8.

На вход генератору поступает визуальная диаграмма. На первом этапе диаграмма обходится в глубину компонентой-валидатором (построенной с применением шаблона проектирования «посетитель»). Для всех выражений на встроенном текстовом языке в свойствах блоков происходит их синтаксический разбор и вывод типов. В случае, если диаграмма или код в свойстве содержит ошибки, о них сообщается пользователю в локализованном виде и процесс завершается. В случае, если диаграмма синтаксически корректна, она поступает на вход анализатору потока управления, о котором будет рассказано ниже. В зависимости от того, может ли диаграмма быть сгенерирована в структурированный код, выбирается промежуточный генератор независимого представления, который извлекает семантическое дерева из потока управления диаграммы. Если целевой язык не поддерживает инструкцию `goto` или, наоборот, высокоуровневые конструкции, как, например, ассемблер-подобный байткод BM EV3 (что указывается в конфигурации генератора), соответствующему генератору будет предпочтена его альтернатива. Наконец, на последнем шаге процесса генерации семантическое дерево печатается в целевой текстовый язык. Для этого используется подход с заданием шаблонов генерации для целевого текстового языка. Все, что необходимо сделать для реализации нового генератора — переопределить набор этих шаблонов и указать конфигурацию генератора. К примеру, шаблон генерации условного оператора в язык C выглядит следующим образом:

```
if (@@CONDITION@@) {
    @@THEN_BODY@@
} else {
    @@ELSE_BODY@@
}
```

Вторая из решенных проблем — модуль анализатора по-

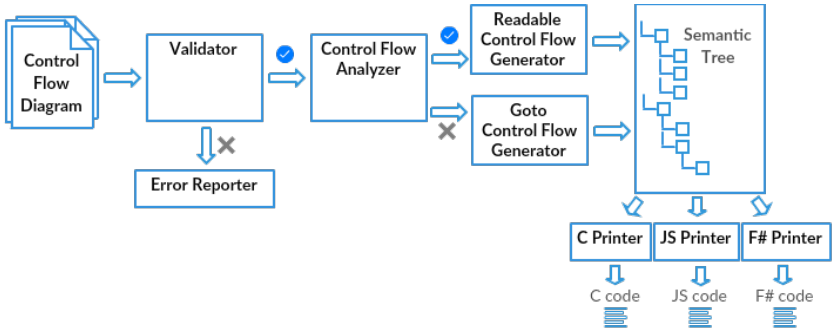

Рис. 8: Архитектура подсистемы генерации кода в TRIK Studio

тока управления. В терминах текстовых языков задача звучит так: по данной программе, поток управления которой описан исключительно инструкциями `goto`, требуется выдать эквивалентную структурированную программу. Данная задача решалась различными исследователями, работающими в области декомпиляции программного кода [**?**], [**?**]. Приемлемый способ ее решения по методу *интервального анализа* графа потока управления описан, к примеру, в монографии [**?**]. Алгоритм интервального анализа графа потока управления диаграммы с незначительными модификациями реализован в TRIK Studio. Коротко опишем его.

Неформально говоря, *интервал* — это участок графа потока управления с одним входом и одним выходом. Целью алгоритма является построение дерева вложенности интервалов графа потока управления программы, которое в нашем случае и является семантическим деревом. Алгоритм работает на рекурсивном подъеме поиска в глубину в графе потока управления. На каждом шаге алгоритм пробует сопоставить подграф, выходящий из текущей вершины, с каждым из шаблонов элементарных интервалов (алгоритмических конструкций, в терминах которых и будет структурирована программа). Если такой шаблон удается найти, весь подграф сворачивается в одну вершину, и процесс продолжается. Самый простой пример такого шаблона — два интервала, соединенные ровно одной дугой, что соответствует последовательной композиции. Если на каком-то шаге ни один шаблон не подошел, то диаграмма считается неприводимой.

Очевидным ограничениям на целевой язык в текущей реализации генераторов является его императивность: цепочка блоков на диаграмме должна преобразовываться в последовательное исполнение утверждений в текстовом языке, что проще всего достигается применением оператора последовательной композиции. Тем не менее, идея использования промежуточного шага генерации в независимое представление работает и здесь. К примеру, для поддержки генерации в чисто функциональные языки достаточно расширить элементы независимого представления конструкцией продолжения (continuation passing style, CPS) и добавить промежуточный генератор диаграммы в независимое представление в форме CPS. После этого достаточно переопределить шаблоны печати независимого представления во все необходимые языки.

IX. Textual language parser and interpreter

TRIK Studio uses visual languages for robots programming, but arithmetic expressions, intrinsic function calls and so on are better represented with textual strings (in fact, NXT-G tries to use visual blocks even for arithmetic expressions, representing them as syntax trees, it is very inconvenient). As a textual part of a language for both visual languages TRIK Studio uses a subset of Lua 5.3[11], customized for our needs. Decision to implement our own parser and interpreter of a textual language was made under these considerations:

- we needed a small textual language with lightweight syntax and without explicit typing since it shall be used by non-programmers;
- we needed explicit abstract syntax tree and the ability to run type inference on it since we were going to generate code on C or EV3 bytecode, which is strongly typed;
- we needed to be able to customize language syntax if the need will arise;
- all existing implementations for such languages had interpreter without access to a syntax tree or a parser that was not reusable from C++ code;
- since we needed only a subset of a language (arithmetic expressions, without statements, custom function and type definitions), writing custom parser was not a difficult task.

Parser and interpreter were implemented as a separate library in QReal core, so the resulting textual language is available not only in TRIK Studio, but in all other domain-specific solutions based on QReal. Parser implementation consists of two parts — general-purpose parser combinator library and Lua parser library implemented using parser combinators. Many projects use ANTLR[12], boost.spirit[13], yacc[14] as tools for parser development, but we once again created our custom solution, mainly to avoid additional external dependencies and complication of a build process — QReal and TRIK Studio are developed by a large community, and not everyone is happy to install additional tools and learn how to use them, especially if their work has nothing to do with syntax analysis and they are students who do not took formal languages course yet.

QReal parser combinator library supports recursive-descent parsers which are able to parse a subset of LL(1) grammars.

[11] https://www.lua.org/
[12] http://www.antlr.org/
[13] http://boost-spirit.com/home/
[14] http://dinosaur.compilertools.net/yacc/

FOLLOW(α) set is not calculated, so it limits the expressiveness of resulting grammars, but we were still able to use Lua 5.3 grammar almost as it is specified, with only minor modifications related to factorization and left recursion elimination, which shall be done anyway for LL parsers. For arithmetic expressions Precedence Climbing[15] algorithm was used, and it also required some minor alterations in Lua grammar. Custom modifications were made mainly on lexer level to allow, for example, to use '!=' for inequality in addition to '~=' used in Lua. Here is a quick example[16] of production written in C++ with our library ("'statement is ';' or a list of expressions, optionally followed by '=' and other list of expressions"'):

```
// stat ::= ';' | explist ['=' explist]
stat = (-LuaTokenTypes::semicolon
    | (explist & ~(-LuaTokenTypes::equals & explist)))
```

`LuaTokenTypes::semicolon` is a token corresponding to ';', operator '-' creates a simple parser that can parse only semicolons and excludes semicolon from AST, "'explist"' is a reference parser object, like "'stat"', defined elsewhere, '&' combines two parsers into a parser which accepts concatenation of their corresponding strings, 'l' combines two parsers into a parser that accepts alternative, '~' creates optional parser from its argument, which shall also be a parser. There are also operators for adding semantic actions to productions and assigning parser a name for debug purposes. When all parsers are combined in such a way, it is enough to call 'parse()' method of a resulting parser object, giving it a token stream. Parser combinator library was used for another language in QReal [**?**], so it is general enough to support not only Lua.

Parser returns abstract syntax tree on which type inference algorithm is executed, providing types for all variables and expressions. Type inference uses Hindley-Milner-style [**?**] algorithm, simplified for performance reasons and extended to support overloading and coercion. Type inference is also generalised and Lua type inferer only defines inference rules for Lua-specific AST nodes, core type inference functionality is available for all QReal textual languages.

After type inference is complete, expression is ready to be evaluated by an interpreter. Interpreter allows to register intrinsic functions, also it allows to add custom variables with their values, to support using current sensor values in calculations — robot communication subsystem receives telemetry data from a robot and injects sensor readings into interpreter, which uses them to calculate expressions.

Last notable feature of parsing/interpreting subsystem is extensive use of caching to avoid parsing or reevaluating expressions as much as possible. Program shall be interpreted in real-time, so reparsing and reevaluating expressions every several milliseconds, as required by many control algorithms, would be severe performance problem. But program can be changed by user during interpretation, values for some variables may be changed by external code, such as sensor readings changed by communication subsystem — so the interpreter keeps track of changes and uses previously calculated values if possible.

[15] http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

[16] full specification of our parser with many irrelevant technical details see at https://github.com/qreal/qreal/tree/master/qrtext/src/lua

X. Simulator

Особая часть среды TRIK Studio, отделимая от всей остальной системы — подсистема двумерного имитационного моделирования робота (двумерный симулятор). Окно симулятора встроено в интерфейс TRIK Studio, но может и использоваться отдельно. Основная часть симулятора — редактор модели мира. В специальном меню можно выбрать инструмент рисования (подобно тому, как это происходит в большинстве графических редакторов). Инструменты рисования делятся на стены (твердые предметы, на которые реагируют ультразвуковые и инфракрасные датчики роботов и сквозь которые робот не может проехать) и цветные маркеры на полу (элементы, на которые реагируют датчики цвета, освещенности и эмуляторы систем видео-зрения) — прямые линии и кубические кривые безье, прямоугольники, эллипсы и стилус (произвольная растровая фигура, рисующаяся мышью как карандашом). Для каждого маркера на полу может быть задана его толщина, цвет и заливка. Модель робота всегда присутствует на сцене редактора мира, на ней произвольным образом можно размещать виртуальные датчики. Робот представляет собой двухколесную тележку с дифференциальным приводом. Для удобства регионы сканирования датчиков расстояния подсвечиваются. На отдельной панели для каждой поддерживаемой модели робота (NXT, EV3 или TRIK) присутствует эмулятор панели контроллера: фронтальное изображение его лицевой панели с кнопками, нажатие мышью на которые эмулирует нажатие на кнопку реального контроллера, цветными светодиодами, меняющими свой цвет соответственно тому, как того требует написанная пользователем программа, и эмулятором дисплея, изображение на котором можно менять блоками из группы «Рисование».

Важной особенностью симулятора является то, что режим исполнения программы неотличим от режима создания модели мира. Модель мира может редактироваться даже в момент исполнения программы, на любое добавление пользователем стенок и цветных элементов во время исполнения программы датчики начнут реагировать немедленно; положение и направление робота и его датчиков в пространстве может быть изменено в любой момент, что соответствует в реальном мире физическому воздействию на корпус робота.

Симулятор построен на основе архитектуры Model-View-Controller. Модель содержит сериализуемое описание мира и робота, уведомляет о любом изменении любого свойства любого предмета. На события модели подписывается представление, все изменения в модели автоматически отображаются на сцене и панелях симулятора. Пользовательские действия в представлении выполняются через контролирующую компоненту, которая кладет очередное действие на вершину специального стека для возможности его отмены и повтора.

Важной частью модели является ее «физика». В симуляторе реализовано две физические модели поведения робота: идеальная и реалистичная. В идеальной физике игнорируются силы трения о пол и стены, импульсы моторов, при неизменных скоростях моторов робот будет двигаться равномерно (без ускорения и замедления). Любое, даже самое легкое столкновение со стеной в такой модели остановит робота. В реалистичной модели ведется полный учет сил тяги и трения робота о пол и стены, при столкновении со стеной робот поведет себя подобно тому, как это происходит в реальности.

Переключение между физическими моделями происходит при выставлении соответствующего флага на панели настроек симуляции и может произойти даже во время исполнения программы. Имеется возможность «зашумления» показаний датчиков и импульсов на моторах гауссовым шумом.

Время в двумерном симуляторе не соответствует процессорному: в нем существует централизованная временная прямая, темп хода которой может меняться на специальной панели (что не меняет поведения робота). Та же самая временная прямая используется в интерпретаторах визуальных языков TRIK Studio для соответствия блоков работы со временем модельному, а не процессорному времени.

XI. Automatic checking of solution correctness

Последняя важная подсистема, о которой будет рассказано в данной работе — механизм автоматической проверки заданий на TRIK Studio. Модель мира в двумерном симуляторе может быть превращена специальными средствами среды в упражнение, распространяемое между учениками. Для этого достаточно задать два набора описаний:

- описание того, какие части упражнения нельзя менять ученику (модель мира, начальное положение робота и его датчиков, сам набор датчиков, привязку моторов и физические настройки симуляции),
- программа проверки корректности решения задачи.

Программа проверки ограничений описывается на специальном текстовом XML-подобном языке. Данный язык интересен сам по себе и может рассматриваться как отдельный результат работы. Программа на таком языке представляет собой множество событий $\{e_1, e_2, ..., e_n\}$, где каждое событие e_i представляет собой тройку (id_i, c_i, T_i):

- id_i — идентификатор события: внутренняя метка, по которой другие события могут получать информацию о событии e_i, может быть пустым;
- c_i — условие срабатывания события, представляющее собой формулу логики первого порядка без кванторов, об интерпретации предикатных и функциональных символах которой будет рассказано ниже;
- T_i — упорядоченный список элементарных триггеров $[t_{i_1}, t_{i_2}, ..., t_{i_n}]$. Элементарный триггер задает действие, выполняемое в момент истинности условия срабатывания события c_i.

Одно событие задается одним элементом в XML-спецификации программы. Каждое событие в текстовом описании программы может быть представлено либо в *каноническом* виде, либо в виде *ограничения*. Событие в канонической форме — уже описанная тройка (id_i, c_i, T_i).

Событие в форме ограничения — это тройка $(id_j, c_j, message_j)$. Ограничение интерпретируется как событие $(id_i, \neg c_i, [fail(message_j)])$, где $fail(message_j)$ — триггер прекращения исполнения имитационной модели с выдачей сообщения об ошибке $message_j$. Другими словами, ограничение — это событие, которое срабатывает, когда нарушается заданное условие, и которое сообщает пользователю об этом нарушении. Описание события в форме ограничения введено в язык из чисто прагматических соображений, так как на практике удобно описывать утверждения вида «робот x не должен покидать пределы зоны z» или «к роботу x должен быть подключен набор датчиков s» именно в терминах ограничений, а не событий. Особый случай такого ограничения — лимит времени на выполнение программы. Лимит времени должен быть указан в любой программе проверки ограничений имитационной модели TRIK Studio, так как процесс проверки, очевидно, не может осуществляться бесконечное время. Наличие ограничения на лимит времени проверяется как часть синтаксиса языка.

Коротко опишем множество используемых в языке предикатных и функциональных символов и элементарных триггеров. Предикатные символы можно поделить на несколько групп:

- Предикаты сравнения значений термов $>$, $<$, $\leq$, $\geq$, $=$, $\neq$.
- Пространственные предикаты. Имеют вид «предмет x находится внутри области y».
- Предикаты состояния событий $settedUp(id_i)$ и $dropped(id_i)$, описывающие состояние события (активно/неактивно). В активном состоянии событие может быть выполнено, в неактивном оно не выполнит триггеры даже при выполнении условия срабатывания события.
- Предикат времени $timer(t)$, который становится истинным спустя t отсчетов модельного времени и остается истинным после, а до этого момента ложен.
- Прочие предикаты, выражаемые уже описанными и введенные исключительно в целях удобства.

Функциональные символы бывают следующих видов:

- Константы различных типов (целочисленные, с плавающей точкой, строковые, логические, цветовые, геометрические и пр.).
- Символ $variableValue(id)$ для получения значения переменной с идентификатором id. Переменные могут быть полезны для реализации сложной логики проверки, например, при подсчете числа проделанных роботом итераций.
- Арифметические и геометрические операции над значениями других термов, например, модуль числа, подсчет расстояния между двумя точками или выпуклая оболочка фиксированного множества точек.
- Символы сравнения формы двух объектов с использованием расстояния Левенштейна, полезные при проверке схожести фигур, нарисованных роботом и на его дисплее.
- Символ $objectState(path)$ — основное средство получения информации о состоянии устройств робота и свойствах предмета из внешнего мира. Аргумент $path$ — путь к желаемому свойству в иерархии объектов в модели мира. Такой путь будет преобразован системой в последовательность получений значений

свойств объектов C++, реализующих предметы в модели мира посредством механизма рефлексии Qt.

Наконец, элементарные триггеры делятся на следующие категории:

- $success$, $fail(message)$ — управление состоянием проверки. Первый помечает результат проверки как успешный, второй — как неудавшийся, отображая заданное сообщение об ошибке.
- Триггеры задания значения переменных и изменения значения свойств элементов имитационной модели мира.
- Триггеры управления состоянием событий. Каждое событие может активировать или деактивировать другое событие или себя; с помощью этого можно задавать сложную логику проверки.

Пример простейшей программы на языке задания ограничений имитационной модели мира в TRIK Studio приведён в листинге 1.

Архитектурно система проверки ограничений представляет собой отдельно стоящий модуль, использующий в качестве разделяемых библиотек ядро и имитационную модель системы TRIK Studio. Использование механизма рефлексии Qt для доступа к свойствам объектов имитационной модели позволяет намного проще масштабировать саму модель: при расширении возможностей системы или добавлении новых симулируемых устройств новые объекты и свойства будут немедленно доступны из языка описания ограничений, без дополнительной модификации кода проверяющей системы. Тестовая система подписывается на события имитационной модели подобно тому, как это делает представление симулятора (в смысле архитектуры MVC), и вызывает проверку ограничений на каждом отсчете модельного времени. При этом проверяются только активные ограничения, которых на практике в каждый конкретный момент времени немного, поэтому общая скорость работы симулятора при наличии проверяющей программы практически не меняется.

Описанный язык удобен для задания пространственных и временных ограничений на поведение системы. Он является значительно более удобным и выразительным, чем, к примеру, язык темпоральных логик или топологико-темпоральных логик, используемых для описания формальных ограничений на поведение робота в работах последних лет [**?**], [**?**], [**?**], [**?**]. При этом язык не тьюринг-полон (на нем, к примеру, нельзя выразить нормальные алгоритмы Маркова из-за отсутствия в языке средств работы с коллекциями произвольной длины, что делает невозможным реализацию на нем замены в строке), это говорит о возможности автоматического анализа интересных свойств программ на нем. Ко всему прочему, подмножество этого языка может использоваться для описания требований к программе для последующей ее формальной верификации. В будущих версиях TRIK Studio планируется реализация визуального конструктора ограничений на этом языке. Появление такого конструктора и интеграция мощного верификатора дадут возможность формально доказывать корректность визуальных программ, не написав ни одной формулы формальной логики. Подробное исследование этого вопроса — тема для будущих работ.

Система проверки ограничений используется для функционального тестирования среды на сервере системы непрерывной интеграции. Об автоматизированном тестировании среды TRIK Studio подробнее рассказано в работе [**?**].

This functionality is also used as an automatic checker of exercises for MOOC on Stepik platform[17]. Without such checker it would be impossible to provide feedback to possibly large number of students, making course much less interactive. From a technical point of view checker is a set of scripts which launch "“headless”" interpreter (i.e. interpreter without GUI) on a correct field with correct constraints. Checker is hooked up into Stepik infrastructure and runs in a separate Docker[18] container when new solution is submitted. Checker has several fields and corresponding constraint descriptions for each task (from one to five), each solution is checked against all those fields to test that it works in different situations and is not hardcoded. Fields are hidden from the user but error reporter output does gets displayed, so students can guess what went wrong (well, theoretically. But it's better than traditional ACM ICPC system, where only a number of incorrect test and a general type of error are reported).

Checker can also send a visual representation of a field with a trace of a robot to a client. Trace consists of points of a robot trajectory and values reported by its sensors, and this trace can be played back by a web application described in [**?**] (in Russian). This web application also allows to actually create solutions for almost all tasks from the course right in the browser, form a correct TRIK Studio save file and run it on a server on checker, playing back the result, so a student perceives this as "“TRIK Studio in a browser”". This application works standalone, but is not integrated into Stepik course yet due to technical difficulties related to Stepik infrastructure.

Conclusion

Technical description of TRIK Studio robots programming system was presented. TRIK Studio now has about ten thousand users across the globe, according to Google Analytics data. Now it supports English, Russian and French languages. It is open source[19] and has large and active community developing it. TRIK Studio has about 100K lines of code (excluding QReal core, which also has size of about 120K LOC), written in C++ with Qt.

Future work consists of two major development vectors: improvements of an existing system and new research projects based on it. Improvement tasks are gathered from teachers, pupils and students which use TRIK Studio, and there is already several hundreds of such tasks tracked. For example, support for new robotic platforms (such as Arduino, STM32), improvements for simulator (more precise physics simulation, multiagent systems simulation and so on), support for more textual languages (like Python, Pascal). Research projects that are currently using TRIK Studio as a technical base include research in domain-specific visual languages field and formal methods field of study. For example, we are creating technology to automatically generate visual domain-specific language using metainformation from packages of robotic middleware (ROS [**?**]

[17] https://stepik.org/s/7qe3xj4Z
[18] https://www.docker.com/
[19] https://github.com/qreal/qreal (accessed: 08.02.2017)

```
<!-- Корневой элемент, означающий начало программы проверки ограничений -->
<constraints>

    <!-- Обязательное в любой программе ограничение на время работы -->
    <timelimit value="2000"/>

    <!-- Ограничение на местоположение робота -->
    <constraint failMessage="Робот покинул допустимую область!">
        <inside objectId="robot1" regionId="myspace"/>
    </constraint>

    <!-- Условие успешности программы: робот должен сказать "Привет" с помощью
        встроенного механизма синтезирования речи и нарисовать улыбку на дисплее -->
    <event settedUpInitially="true">
        <conditions glue="and">
            <equals>
                <objectState object="robot1.shell.lastPhrase"/>
                <string value="Привет"/>
            </equals>
            <equals>
                <objectState object="robot1.display.smiles"/>
                <bool value="true"/>
            </equals>
        </conditions>
        <trigger>
            <success/>
        </trigger>
    </event>

</constraints>
```

Листинг 1: Пример программы проверки ограничений имитационной модели мира в TRIK Studio.

is currently considered), that visual language will be able to link middleware nodes together thus configuring environment for a particular robot. Research in formal methods aims to formalize semantics of used visual languages to be able to apply formal verification and synthesis methods to diagrams.