

ИСП

Институт Системного Программирования
Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

**Труды
Института Системного
Программирования РАН
Proceedings of the
Institute for System
Programming of the RAS**

Том 28, выпуск 2

Volume 28, issue 2

Москва 2016

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:



Редколлегия

Главный редактор - [Иванников Виктор Петрович](#), академик РАН, профессор, ИСП РАН (Москва, Российская Федерация).

Заместитель главного редактора - [Кузнецов Сергей Дмитриевич](#), д.т.н., профессор, ИСП РАН (Москва, Российская Федерация).

[Аветисян Арютюн Ишханович](#), д.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Бурдонов Игорь Борисович](#), д.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Воронков Андрей Анатольевич](#), д.ф.-м.н., профессор, Университет Манчестера (Манчестер, Великобритания).

[Вирбицкайте Ирина Бонавентуровна](#), профессор, д.ф.-м.н., Институт систем информатики им. академика А.П. Ершова СО РАН (Новосибирск, Россия).

[Гайсарян Сергей Суренович](#), к.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Евтушенко Нина Владимировна](#), профессор, д.т.н., ТГУ (Томск, Российская Федерация).

[Карпов Леонид Евгеньевич](#), д.т.н., ИСП РАН (Москва, Российская Федерация).

[Коннов Игорь Владимирович](#), к.ф.-м.н., Технический университет Вены (Вена, Австрия)

[Косачев Александр Сергеевич](#), к.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Кузюрин Николай Николаевич](#), д.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Ластовский Алексей Леонидович](#), д.ф.-м.н., профессор, Университет Дублина (Дублин, Ирландия).

[Ломазова Ирина Александровна](#), д.ф.-м.н., профессор, Национальный исследовательский университет «Высшая школа экономики» (Москва, Российская Федерация).

[Новиков Борис Асенович](#), д.ф.-м.н., профессор, Санкт-Петербургский государственный университет (Санкт-Петербург, Россия).

[Петренко Александр Константинович](#), д.ф.-м.н., ИСП РАН (Москва, Российская Федерация).

[Петренко Александр Федорович](#), д.ф.-м.н., Исследовательский институт Монреаль (Монреаль, Канада)

[Семенов Виталий Адольфович](#), д.ф.-м.н., профессор, ИСП РАН (Москва, Российская Федерация).

[Томилини Александр Николаевич](#), д.ф.-м.н., профессор, ИСП РАН (Москва, Российская Федерация).

[Черных Андрей](#), д.ф.-м.н., профессор, Научно-исследовательский центр CICESE (Энсенана, Нижняя Калифорния, Мексика).

[Шнитман Виктор Зиновьевич](#), д.т.н., ИСП РАН (Москва, Российская Федерация).

[Швустер Асаф](#), д.ф.-м.н., профессор, Технион — Израильский технологический институт Technion (Хайфа, Израиль)

Editorial Board

Editor-in-Chief - [Victor P. Ivannikov](#), Academician RAS, Professor, ISPS/ System Programming of the RAS (Moscow, Russian Federation).

Deputy Editor-in-Chief - [Sergey D. Kuznetsov](#), Dr. Sci. (Eng.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation).

[Arutyun I. Avetisyan](#), Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Igor B. Burdonov](#), Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Andrei Chernykh](#), Dr. Sci., Professor, CICESE Research Centre (Ensenada, Lower California, Mexico).

[Sergey S. Gaissaryan](#), PhD (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Leonid E. Karpov](#), Dr. Sci. (Eng.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Igor Konnov](#), PhD (Phys.–Math.), Vienna University of Technology (Vienna, Austria).

[Alexander S. Kossatchev](#), PhD (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Nikolay N. Kuzyurin](#), Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Alexey Lastovetsky](#), Dr. Sci. (Phys.–Math.), Professor, UCD School of Computer Science and Informatics (Dublin, Ireland).

[Irina A. Lomazova](#), Dr. Sci. (Phys.–Math.), Professor, National Research University Higher School of Economics (Moscow, Russian Federation).

[Boris A. Novikov](#), Dr. Sci. (Phys.–Math.), Professor, St. Petersburg University (St. Petersburg, Russia).

[Alexander K. Petrenko](#), Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Alexandre F. Petrenko](#), PhD, Computer Research Institute of Montreal (Montreal, Canada).

[Assaf Schuster](#), Ph.D., Professor, Technion - Israel Institute of Technology (Haifa, Israel)

[Vitaly A. Semenov](#), Dr. Sci. (Phys.–Math.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation).

[Victor Z. Shnitman](#), Dr. Sci. (Eng.), Institute for System Programming of the RAS (Moscow, Russian Federation).

[Alexander N. Tomilin](#), Dr. Sci. (Phys.–Math.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation).

[Irina B. Virbitskaite](#), Dr. Sci. (Phys.–Math.), The A.P. Ershov Institute of Informatics Systems, Siberian Branch of the RAS (Novosibirsk, Russian Federation).

[Andrey Voronkov](#), Dr. Sci. (Phys.–Math.), Professor, University of Manchester (Manchester, UK).

[Nina V. Yevtushenko](#), Dr. Sci. (Eng.), Tomsk State University (Tomsk, Russian Federation).

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.

Телефон: +7(495) 912-44-25

Е-mail: info-isp@ispras.ru

Сайт: <http://www.ispras.ru/proceedings/>

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Tel: +7(495) 912-44-25

E-mail: info-isp@ispras.ru

Web: <http://www.ispras.ru/en/proceedings/>

С о д е р ж а н и е

Дизайн средств обобщённого программирования в объектно-ориентированных языках: ключевые решения <i>Ю.В. Белякова</i>	5
Refinement типы для языка Jolie <i>Александр Чичигин, Лариса Сафина, Мохамед Эльвакиль, Мануэль Маццара, Фабрицио Монтези, Виктор Ривера</i>	33
Образовательный визуальный потоковый язык для программирования роботов <i>Г.А. Зимин, Д.А. Мордвинов</i>	45
Контекстно-ориентированная модель для разметки сквозной функциональности в исходном коде <i>М.С. Малеванный, С.С. Михалкович</i>	63
Подход к обнаружению анти-паттернов в сервис-ориентированных системах <i>А.С. Югов</i>	79
Технология создания семейства приложений на основе анализа предметной области <i>А.А. Гудошникова, Ю.В. Литвинов</i>	97
Применимость AutoProof: учебный пример верификации ПО <i>Мансур Хазеев, Виктор Ривера, Мануэль Маццара, Александр Чичигин</i>	111
Верификация преобразования грамматики в нормальную форму Хомского в F* <i>М.И. Полубелова, С.Н. Божко, С.В. Григорьев</i>	127
Исследование влияния использования параллелизма на производительность движка косимуляции в проекте INTO-CPS <i>Тул К., Ларсен П.Г.</i>	139
Способ статической оценки времени работы компонентов AADL-моделей <i>А.М. Троицкий, Д.В. Буздалов</i>	157
Практический опыт реализации подходов программной и системной инженерии для управления требованиями при разработке программного обеспечения в авиационной отрасли <i>И.В. Ковернинский, А.В. Кан, В.Б. Волков, Ю.С. Попов, Н.К. Горелиц</i>	173
Устройство и архитектура операционной системы реального времени <i>К.М. Маллачиев, Н.В. Пакулин, А.В. Хорошилов</i>	181
Разработка отладчика для операционной системы реального времени <i>А.Н. Емеленко, К.А. Маллачиев, Н.В. Пакулин</i>	193
Моделирование конвейера распознавания людей в системах контроля доступа. <i>Гёссен Ф., Маргариа Т., Гёке Т.</i>	205
Параллельная обработка и визуализация для результатов моделирования методом молекулярной динамики <i>Д.В. Пузырьков, В.О. Подрыга, С.В. Поляков</i>	221
Обзор предметной области и концепция фреймворка для разработки моделей мемристоров и мемристорных нейронных сетей <i>Д.Д. Кожевников, Н.В. Красилч</i>	243
Композиционная модель и способ построения функционально-ориентированных информационных ресурсов информационно-управляющих систем <i>И.И. Чуляев</i>	259

T a b l e o f C o n t e n t s

Language Support for Generic Programming in Object-Oriented Languages: Design Challenges <i>Julia Belyakova</i>	5
Refinement Types in Jolie <i>Alexander Tchitchigin, Larisa Safina, Mohamed Elwakil, Manuel Mazzara, Fabrizio Montesi, Victor Rivera</i>	33
Visual Dataflow Language for Educational Robots Programming <i>G.A. Zimin, D.A. Mordvinov</i>	45
Context-Based Model for Concern Markup of a Source Code <i>M.S. Malevanny, S.S. Mikhalkovich</i>	63
Approach to Anti-pattern detection in Service-oriented Software Systems <i>A.S. Yugov</i>	79
Technology for application family creation based on domain analysis <i>A.Gudoshnikova, Y.Litvinov</i>	97
Usability of AutoProof: a case study of software verification <i>Mansur Khazeev, Victor Rivera, Manuel Mazzara, Alexander Tchitchigin</i>	111
Certified Grammar Transformation to Chomsky Normal Form in F* <i>M.I. Polubelova, S.N. Bozhko, S.V. Grigorev</i>	127
Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS <i>C. Thule, P.G. Larsen</i>	139
A static approach to estimation of execution time of components in AADL models <i>A.M. Troitskiy, D.V. Buzdalov</i>	157
Practical experience of software and system engineering approaches in requirements management for software development in aviation industry <i>I.V. Koverninskiy, A.V. Kan, V.B. Volkov, Yu. S. Popov, N.K. Gorelits</i>	173
Design and architecture of real-time operating system <i>K.M. Mallachiev, N.V. Pakulin, A.V. Khoroshilov</i>	181
Developing a Debugger for Real-Time Operating System <i>A.N. Emelenko, K.A. Mallachiev, N.V. Pakulin</i>	193
Modelling the People Recognition Pipeline in Access Control Systems <i>F. Gossen, T. Margaria, T. Göke</i>	205
Parallel processing and visualization for results of molecular simulation problems <i>D.V. Puzyrkov, V.O. Podryga, S.V. Polyakov</i>	221
Memristor-based Hardware Neural Networks Modelling Review and Framework Concept <i>D.D. Kozhevnikov, N.V. Krasilich</i>	243
Composition model and method of creation of functionally-oriented information resources <i>I. Chucklyaev</i>	259

Language Support for Generic Programming in Object-Oriented Languages: Design Challenges¹

Julia Belyakova <julbel@sfedu.ru>

*I. I. Vorovich Institute of Mathematics, Mechanics and Computer Science,
Southern Federal University,
105/42, B. Sadovaya st., Rostov-on-Don, 344006, Russia*

Abstract. It is generally considered that object-oriented (OO) languages provide weaker support for generic programming (GP) as compared with functional languages such as Haskell or SML. There were several comparative studies which showed this. But many new object-oriented languages have appeared in recent years. Have they improved the support for generic programming? And if not, is there a reason why OO languages yield to functional ones in this respect? In the earlier comparative studies object-oriented languages were usually not treated in any special way. However, the OO features affect language facilities for GP and a style people write generic programs in such languages. In this paper we compare ten modern object-oriented languages and language extensions with respect to their support for generic programming. It has been discovered that every of these languages strictly follows one of the two approaches to constraining type parameters. So the first design challenge we consider is “which approach is better”. It turns out that most of the explored OO languages use the less powerful one. The second thing that has a big impact on the expressive power of a programming language is language support for multiple models. We discuss pros and cons of this feature and its relation to other language facilities for generic programming.

Keywords: object-oriented languages; generic programming; generics; types; constraints; concepts; interfaces; Concept pattern; multiple models; concept-parameters

DOI: 10.15514/ISPRAS-2016-28(2)-1

For citation: Belyakova Julia. Language Support for Generic Programming in Object-Oriented Languages: Design Challenges. *Trudy ISP RAN/Proc. ISP RAS*, Volume 28, Issue 2, 2016, pp. 5-32. DOI: 10.15514/ISPRAS-2016-28(2)-1

¹ This paper is the extended version of the conference paper [1] accepted for the XX Brazilian Symposium on Programming Languages.

1. Introduction

Almost all modern programming languages provide language support for generic programming (GP) [2]. Some languages do it better than others. For example, Haskell is generally considered to be one of the best languages for generic programming [3, 4], whereas mainstream object-oriented languages such as C# and Java are much less expressive and have many drawbacks. There were several studies that compared language support for generic programming in different languages [3–6]. However, these studies do not make any difference between object-oriented and functional languages. We argue that OO languages are to be treated separately, because they support the distinctive OO features that pure functional languages do not, such as inheritance, interfaces/traits, subtype polymorphism, etc. These features affect the language design and a way people write generic programs in object-oriented languages.

Several new object-oriented languages have appeared in recent years, for instance, Rust, Swift, Kotlin. At the same time, several independent extensions have been developed for the mainstream OO languages [7–10]. These new languages and extensions have many differences, but all of them tend to improve the support for generic programming. There is a lack of a careful comparison of the approaches and mechanisms for generic programming in *modern object-oriented languages*. This study is aimed to fill the gap: it gives a survey, analysis, and comparison of the facilities for generic programming that the chosen OO languages provide. We identify the dependencies between major language features, detect incompatible ones, and point the properties that a language design should satisfy to be effective for generic programming.

2. Main Ideas

Ten modern object-oriented languages and language extensions have been explored in this study with respect to generic programming. We have found out that in the case of OO languages there are exactly two approaches to a design of language constructs for generic programming. We call the first one “constraints-are-types”, because under this approach OO constructs such as interfaces or traits, which are usually used as types in object-oriented programs, are also used to constrain type parameters in generic programs. The second approach, “constraints-are-Not-types”, restricts OO constructs to be used as types only, and provides separate language constructs for constraining type parameters. Hence the first design challenge arises: is one of this approaches better than another? Or the same expressive power can be achieved using any of them? We answer these questions in Sec. 3. It turns out that the approaches cannot be integrated together, and the second one is more expressive.

The second point covered in the paper in detail (in Sec. 4) is language support for multiple models (by “model” we mean a way in which types satisfy constraints). There are several questions related to multiple models:

1. Is it desirable to have multiple models of a constraint?

2. How can support for multiple models be provided with the approaches discovered?
3. Why does not Haskell allow multiple models (instances of a type class)?
4. Is there a language design that reflects the support for multiple models better than the existing ones?

The short answers are:

1. Yes, it is desirable.
2. It can be naturally provided with the second approach but not with the first one.
3. Because of type inference.
4. Yes, there is.

In conclusion, we present a modified version of the well-known table [3, 5] showing the levels of language support for the features important for generic programming. Table 1 provides information on all of the object-oriented languages considered, introduces some new features, and demonstrates the relations between the features.

3. Two Approaches to Constraining Type Parameters

This section provides a survey of *language constructs for generic programming* in several modern *object-oriented* programming languages as well as some language extensions. All of the languages we explored adopt one of the two approaches:

1. Interface-like constructs, which are normally used as types in object-oriented programming, are also used to constrain type parameters. By “interface-like constructs” we mean, in particular, C#/Java interfaces, Scala traits, Swift protocols, Rust traits. Fig. 1 shows a corresponding example in C#: `IPrintable` interface acts as the type of `xs` in `PrintArr`, whereas in the function `InParens<T>` it is used to constrain the type parameter `T`.
2. For constraining type parameters a separate language construct is provided; such construct cannot be used as a type. We will see some examples in Sec. 3-2.

Sec. 3-1 analyses the languages of the first category; Sec. 3-2 is devoted to the second one. In Sec. 3-3 we compare both approaches and answer the question “Which one is better if any?”.

```
interface IPrintable { string Print(); }

void PrintArr(IPrintable[] xs)
{ foreach (var x in xs)
  Console.WriteLine("{0}\n", x.Print()); }

string InParens<T>(T x) where T : IPrintable
{ return "(" + x.Print() + " "; }
```

Fig. 1. An ambiguous role of C# interfaces

3.1 Languages with “Constraints-are-Types” Philosophy

C# and Java are probably the best-known programming languages in this category, with *interfaces* being used to constrain type parameters. In comparison with other languages that support generic programming, these ones are much less expressive and have several considerable drawbacks.

Lack of retroactive interface implementation. After a type had been defined, it cannot implement any new interface. A consequence is that generic code with constraints on type parameters can only be instantiated with types *originally* designed to satisfy these constraints. It is impossible to adapt types afterwards, even if they semantically conform the constraints.

```
interface IComparable<T> { int CompareTo(T other); }  
  
class SortedSet<T> where T : IComparable<T> { ... }
```

Fig. 2. The `IComparable<T>` interface in C#

Drawbacks of F-bounded polymorphism. F-bounded polymorphism [11] allows “recursive” constraints (F- constraints) on type parameters in the form `T : I<T>`, where `T` is a type parameter, `I<>` is a generic interface. Such kind of constraints solves the binary method problem [12]: Fig. 2 demonstrates a corresponding C# [13] example. The type parameter `T` in the interface `IComparable<T>` pretends to be a type that implements this interface. This is indeed the case for the class `SortedSet<T>` due to the constraint `T : IComparable<T>`, so the method `T.CompareTo(T)` is like a binary function for comparing elements of type `T`. But the semantics of `IComparable<T>` itself has nothing to do with binary methods. One could easily write some class `Foo` implementing `IComparable<Bar>`, and thus the semantics of comparing two `Bars` would be broken. Another shortcoming of the F-bounded polymorphism is that code with recursive constraints is rather cumbersome and difficult to understand. Yet, as we will see, the F-bounded polymorphism is not the only solution to the binary method problem. More detailed discussion on the pitfalls of the F-bounded polymorphism can be found in [9, 14].

Lack of associated types [14,15]. Types that are logically related to some entity are often called *associated types* of the entity. For instance, types of edges and vertices are associated types of a graph. There is no specific language support for associated types in C# and Java: such types are expressed in generic code in the form of extra type parameters.

Lack of constraints propagation [14,15]. Despite the fact that the definition of the class `SortedSet<T>` in Fig. 2 already contains a constraint on the type parameter `T`, in the `baz<T>` function defined below the constraint on `T` is to be placed as well.

```
void baz<T>(SortedSet<T> s) where T : IComparable<T> { ... }
```

Although `baz<T>` takes a value of type `SortedSet<T>`, so it is clear from the signature of the function that `T` must be comparable, the code would not compile without an

explicit constraint. In other words, a compiler does not propagate the constraints implied by formal parameters, this is a programmer's burden.

```
interface ITerm<Tm> { IEnumerable<Tm> Subterms(); ... }

interface IEquation<Tm, Eqtn, Subst> where Tm : ITerm<Tm>
where Eqtn : IEquation<Tm, Eqtn, Subst>
where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Subst Solve();
  IEnumerable<Eqtn> Split(); ... }

interface ISubstitution<Tm, Eqtn, Subst> where Tm : ITerm<Tm>
where Eqtn : IEquation<Tm, Eqtn, Subst>
where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Tm SubstituteTm(Tm);
  IEnumerable<Eqtn> SubstituteEq (IEnumerable<Eqtn>); ... }
```

Fig. 3. The C# interfaces for unification algorithm

Some of the drawbacks mentioned above have been successfully eliminated in the modern object-oriented languages. We briefly examine language facilities for generic programming in several OO languages with the “constraints-are-types” philosophy in the following subsections. But there is a problem common for all languages of this category, the problem of *multi-type constraints* (constraints on several types). Note that an interface (or a similar language construct) describes properties, an interface of a *single* type that implements/extends it. This has inevitable consequence: multi-type constraints cannot be expressed naturally. Consider a generic unification algorithm [16]: it takes a set of equations between terms (symbolic expressions), and returns the most general substitution which solves the equations. So the algorithm operates on three kinds of data: terms, equations, substitutions. A signature of the algorithm might be as follows:

Subst Unify<Tm, Eqtn, Subst>(IEnumerable<Eqtn>)

But a bunch of functions has to be provided to implement the algorithm: Subterms : Tm → IEnumerable<Tm>,

Solve : Eqtn → Subst, SubstituteTm : Subst × Tm → Tm,

SubstituteEq : Subst × IEnumerable<Eqtn> → IEnumerable<Eqtn>, and some others. All these functions are needed for unification at once, hence it would be convenient to have a single constraint that relates all the type parameters and provides the functions required.

Subst Unify<Tm, Eqtn, Subst>

(IEnumerable<Eqtn>) **where** <single constraint>

But in C#/Java the only thing one can do² is to define three different interfaces describing a term, equation and substitution, and then separately constrain every type parameter with a respective interface. Fig. 3 shows the interface definitions. To set

2 The Concept design pattern can also be used, but it has its own drawbacks. We will discuss concept pattern later, in Sec. 4-3-2.

up a relation between mutually dependent interfaces, three type parameters are used: `Tm` for terms, `Eqtn` for equations, and `Subst` for substitution. Moreover, the parameters are repeatedly constrained with the appropriate interfaces in every interface definition. These constraints are to be stated in a signature of the unification algorithm as well:

```
Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn>)
    where Tm : ITerm<Tm>
    where Eqtn : IEquation<Tm, Eqtn, Subst>
    where Subst : ISubstitution<Tm, Eqtn, Subst>
```

There is one more thing to notice here — interfaces are used in both roles in the same piece of code: the `IEnumerable<Eqtn>` interface is used as a type, whereas other interfaces in the `where` sections are used as constraints.

```
interface Equatable<T> { fun equal (other: T) : Boolean
                        fun notEqual (other: T) : Boolean
                        { return !this.equal(other) } }

class Ident (name : String) : Equatable<Ident> {
    val idname = name.toUpperCase()
    override fun equal (other: Ident) : Boolean
    { return idname == other.idname } }
```

Fig. 4. Interfaces and constraints in Kotlin

3.1.1 Interfaces in Ceylon and Kotlin

In contrast to C#, Ceylon [17] and Kotlin [18] *interfaces* support *default method implementation*, so Java 8 [19] interfaces do. This is a useful feature for generic programming. For instance, one can define an interface for equality that provides a default implementation for the inequality operation. Fig. 4 demonstrates corresponding Kotlin definitions: the `Ident` class implements the interface `Equatable<Ident>` that has two methods, `equal` and `notEqual`; as long as `notEqual` has a default implementation in the interface, there is no need to implement it again in the definition of the `Ident` class.

```
shared interface Comparable<Other> of Other
    given Other satisfies Comparable<Other>
{ shared formal Integer compareTo (Other other);
  shared Integer reverseCompareTo (Other other)
  { return other.compareTo(this); } }
```

Fig. 5. The use of “self type” in Ceylon interfaces

In addition to default method implementations, the Ceylon language also allows a type parameter to be declared as a *self type*. An example is shown in Fig. 5. In the definition of the `Comparable<Other>` interface the declaration of `Other` explicitly requires `Other` to be a self type of the interface, i. e. a type that implements this

interface. Because of this the `reverseCompareTo` method can be defined: both the other and this values are of type `Other`, with the `Other` implementing `Comparable<Other>`, so the call `other.compareTo(this)` is perfectly legal.

3.1.2 Scala Traits

Similarly to advanced interfaces in Java 8, Ceylon, and Kotlin, Scala *traits* [6,20] support *default method implementations*. They can also have *abstract type* members, which, in particular, can be used as associated types [21]. Just as in C#/Java/Ceylon/Kotlin, type parameters (and abstract types) in Scala can be constrained with traits and supertypes (upper bounds): the latter constraints are called *subtype constraints*. But, moreover, they can be constrained with subtypes (lower bounds), which are called *supertype constraints*. None of the languages we discussed so far support supertype constraints nor associated types. Another important Scala feature, *implicit*s [20], will be mentioned later in Sec. 4-1 with respect to the Concept design pattern.

```
struct Point { x: i32, y: i32, }
...
impl Point {
    fn moveOn(&self, dx: i32, dy: i32) -> Point
    { Point {x: self.x + dx, y: self.y + dy } } }
...
impl Point {
    fn reflect(&self) -> Point { Point {x: -self.x, y: -self.y} } }
...
let p1 = Point {x: 4, y: 3};
let p2 = p1.moveOn(1, 1);    let p3 = p1.reflect();
```

Fig. 6. Point struct and its methods in Rust

```
trait Eqtbl { fn equal(&self, that: &Self) -> bool;
    fn not_equal(&self, that: &Self) -> bool { !self.equal(that) } }
trait Printable { fn print(&self); }
...
impl Eqtbl for i32 {
    fn equal (&self, that: &i32) -> bool { *self == *that } }
...
struct Pair<S, T>{ fst: S, snd: T }
...
impl <S : Eqtbl, T : Eqtbl> Eqtbl for Pair<S, T> {
    fn equal (&self, that: &Pair<S, T>) -> bool
    { self.fst.equal(&that.fst) && self.snd.equal(&that.snd) } }
```

Fig. 7. An example of using Rust traits

3.1.2 Rust Traits

The Rust language [22] is quite different from other object-oriented languages. There is no traditional *class* construct in Rust, but instead it suggests *structs* that store the

data, and separate *method implementations* for structs. An example is shown in Fig. 6³: two `impl Point` blocks define method implementations for the `Point` struct. If a function takes the `&self`⁴ argument (as `moveOn`), it is treated as a method. There can be any number of implementation blocks, yet they can be defined at any point after the struct declaration (even in a different module). This gives a huge advantage with respect to generic programming: any struct can be *retroactively* adapted to satisfy constraints.

Constraints in Rust are expressed using *traits*. A trait defines which methods have to be implemented by a type similarly to Scala traits, Java 8 interfaces, and others. Traits can have *default method implementations* and *associated types*; besides that, the *self type* of the trait is directly available and can be used in method definitions. Fig. 7⁵ demonstrates an example: the `Eqtbl` trait defining the equality and inequality operations. Note how support for the self type solves the binary method problem (here `equal` is a binary method): there is no need in extra type parameter that “pretends” to be a self type, because the self type `Self` is already available.

Method implementations in Rust can be probably thought of similarly to .NET “extension methods”. But in contrast to .NET⁶, types in Rust also can *retroactively implement traits* in `impl` blocks as shown in Fig. 7: `Eqtbl` is implemented by `i32` and `Pair<S, T>`. The latter definition also demonstrates a so-called *type-conditional implementation*: pairs are equality comparable only if their elements are equality comparable. The constraint `<S : Eqtbl...>` is a shorthand, it can be declared in a `where` section as well.

There is no struct inheritance and subtype polymorphism in Rust. Nevertheless, as long as traits can be used not only as constraints but also as types, a dynamic dispatch is provided through a feature called trait objects. Suppose `i32` and `f64` implement the `Printable` trait from Fig. 7. Then the following code demonstrates creating and use of a polymorphic collection (the type of the `polyVec` elements is a reference type):

```
let pr1 = 3; let pr2 = 4.5; let pr3 = -10;
let polyVec: Vec<&Printable> = vec! [&pr1, &pr2, &pr3];
for v in polyVec { v.print(); }
```

3.1.3 Swift Protocols

Swift is a more conventional OO language than Rust: it has classes, inheritance, and subtype polymorphism. Classes can be extended with new methods using *extensions*

3 Some details were omitted for simplicity. To make the code correct, one has to add `#[derive(Debug, Copy, Clone)]` before the `Point` definition.

4 The “&” symbol means that an argument is passed by reference.

5 Some details were omitted for simplicity. The following declaration is to be provided to make the code correct: `#[derive(Copy, Clone)]` before the definition `struct Pair<S : Copy, T : Copy>`. Yet the type parameters of the `impl` for `pair` must be constrained with `Copy+Equtable`.

6 Similarly to .NET, Kotlin supports extending classes with methods and properties, but interface implementation in extensions is not allowed.

that are quite similar to Rust method implementations. Instead of interfaces and traits Swift provides *protocols*. They cannot be generic but support *associated types* and *same-type constraints*, *default method implementations* through protocol extensions, and explicit access to the *self type*; due to the mechanism of extensions, types can *retroactively* adopt protocols. Fig. 8 illustrates some examples: the Equatable protocol extended with a default implementation for notEqual (pay attention to the use of the Self type); the contains<T> generic function with a protocol constraint on the type parameter T; an extension of the type Int that enables its conformance to the Printable protocol; the Container protocol with the associated type ItemTy; the allItemsMatch generic function with the same-type constraint on types of elements of two containers, C1 and C2.

```
protocol Equatable { func equal(that: Self) -> Bool; }
extension Equatable { func notEqual(that: Self) -> Bool
    { return !self.equal(that) } }
func contains<T : Equatable> (values: [T], x: T) -> Bool { ... }

protocol Printable { func print(); }
extension Int : Printable { ... }

protocol Container { associatedtype ItemTy ... }
func allItemsMatch<C1: Container, C2: Container
    where C1.ItemTy == C2.ItemTy, C1.ItemTy: Equatable> ...
```

Fig. 8. Protocols and their use in Swift

3.2 Languages with “Constraints-are-Not-Types” Philosophy

Most of the languages in this category were to some extent inspired by the design of Haskell type classes [22]. For defining constraints these languages suggest *new language constructs*, which are usually second-class citizens⁷. These constructs have *no self types* and *cannot* be used as types, they describe requirements on type parameters in an external way; therefore, retroactive satisfaction of constraints (*retroactive modeling*) is automatically provided. Besides retroactive modeling, an integral advantage of such kind of constructs is that *multi-type constraints* can be easily and naturally expressed using them; yet there is no semantic ambiguity which arises when the same construct, such as C # interface, is used both as a type and constraint, as in the example below:

```
void Sort<T>(ICollection<T>) where T : IComparable<T>
```

Here ICollection<T> and IComparable<T> are generic interfaces, but the former one is used as a type whereas the latter one is used as a constraint.

```
interface EQ { boolean eq(This that);
```

⁷ Second-class citizens cannot be assigned to variables, passed as arguments, returned from functions.

```
        boolean notEq(This that); }
abstract implementation EQ [EQ] {
    boolean notEq(This that) { return !this.eq(that); } }

boolean contains<X>(List<X> list, X x) where X implements EQ { ... }

abstract class Expr {...}    class IntLit extends Expr {...}
class PlusExpr extends Expr { Expr left; Expr right; ... }
...
implementation EQ [Expr] { boolean eq(Expr that) { return false; } }
implementation EQ [PlusExpr]{ boolean eq(PlusExpr that) {...} }

interface UNIFY [Tm, Eqtn, Subst] {
    receiver Tm    { IEnumerable<Tm> Subterms(); ... }
    receiver Eqtn { IEnumerable<Eqtn> Split(); ... }
    receiver Subst { Tm SubstituteTm(Tm); ... } }
Subst Unify<Tm, Eqtn, Subst>(IEnumerable<Eqtn>)
where [Tm, Eqtn, Subst] implements UNIFY {...}
```

Fig. 9. Generalized interfaces in JavaGI

3.2.1 JavaGI Generalized Interfaces

JavaGI [7] *generalized interfaces* represent a kind of confluence of both “constraints-are-types” and “constraints-are-not-types” philosophies. Interfaces such as `PrettyPrintable` defined below are called single-parameter interfaces. They describe interfaces of a single type and can be used both as types and constraints.

```
interface PrettyPrintable { String prettyPrint(); }
```

Such interfaces have explicit access to the *self type* named `This`; an example is shown in Fig. 9, where the self type is used in the interface `EQ`. There is no direct support for default method implementations in JavaGI, but *abstract implementation definitions* can be used for this purpose⁸. For example, the `notEq` method of `EQ` (Fig. 9) is implemented in such a way. Generalized interfaces can be implemented *retroactively* in implementation blocks. They do not support associated types but can be generic; moreover, implementations can be generic as well, and the support for *type-conditional interface implementation* is provided:

```
implementation<S, T> EQ [Pair<S, T>] where S implements EQ
where T implements EQ { ... }
```

Besides single-parameter interfaces, there are *multi-headed* generalized interfaces that adopt several features from Haskell type classes [24] and describe interfaces of several types. There is no self type in a multi-headed interface; therefore, it cannot be used as a type, it is designed to be used as a *constraint only*. An example of multi-headed interface is shown in Fig. 9: the `UNIFY` interface contains all the functions required by the unification algorithm considered earlier; the requirements on three

⁸ The design of JavaGI we discuss here goes back to 2011 when default method implementations were not supported in Java. With Java 8 this task could probably be solved in a more elegant way.

types (term, equation, substitution) are defined at once in a single interface. Note how succinct is this definition as compared with the one in Fig. 3.

```
concept InputIterator<Iter> { type value; ... }
concept Monoid<T> { fun identity_elt() -> T;
                  fun binary_op(T, T) -> T; };
model Monoid<int>
{ fun identity_elt() -> int@ { return 0; } ... };

fun accumulate<Iter> where { InputIterator<Iter>,
                             Monoid<InputIterator<Iter>.value> }
(Iter first, Iter last) -> InputIterator<Iter>.value
{ let init = identity_elt(); ... }
```

Fig. 10. Concepts and their use in G

3.2.2 Language G and C++ concepts

Concept as an explicit language construct for defining constraints on type parameters was initially introduced in 2003 [25]. Several designs have been developed since that time [26–28]; in the large, the expressive power of concepts is rather close the Haskell type classes [4]. Concepts were designed to solve the problems of unconstrained C++ templates [14, 29]; they were expected to be included in C++0x standard, but this did not happen. A new version of concepts, Concepts Lite (C++1z) [30], is under way now. The language G declared as “a language for generic programming” [8] also provides concepts that are very similar to the C++0x concepts. G is a subset of C++ extended with several constructs for generic programming. For “C++ concepts” we use the G syntax in this paper.

Similarly to a type class, a concept defines a set of requirements on one or more type parameters. It can contain *function signatures* that may be accompanied with *default implementations*, *associated types*, nested *concept-requirements* on associated types, and *same-type* constraints. A concept can *refine* one or more concepts, it means that the refining concept includes all the requirements from the refined concepts. Refinement is very similar to multiple interface inheritance in C# or protocol inheritance in Swift. Due to the concept refinement, a so-called *concept-based overloading* is supported: one can define several versions of an algorithm/class that have different constraints, and then at compile time the most specialized version is chosen for the given instance. The C++ advance algorithm for iterators is a classic example of concept-based overloading application.

It is said that a type (or a set of types) *satisfies* a concept if an appropriate model of the concept is defined for this type (types). Model definitions are independent from type definitions, so the modeling relation is established *retroactively*; models can be generic and *type-conditional*. Fig. 10 illustrates some examples: the `InputIterator<Iter>` concept with the associated type of elements `value`; the `Monoid<T>` concept and its model for the type `int`; the `accumulate<Iter>` generic function with two constraints, on the type of the iterator and on the associated type of

this iterator. Note how `identity_elt` is called in `accumulate`: in contrast to the languages from the previous section, `identity_elt` is available in the body of `accumulate` at the top-level; this may lead to some inconvenience even if the autocomplete feature is supported in IDE.

3.2.3 C# with concepts

In the C#^{cp1} project [9] (C# with concepts) concept mechanism integrates with subtyping: type parameters and associated types can be constrained with *supertypes* (as in basic C#) and also with *subtypes* (as in Scala). In contrast to all of the languages we discussed earlier, C#^{cp1} allows *multiple models* of a concept in the same scope.

```
concept CEquatable[T]      { bool Equal(T x, T y);  
    bool NotEqual(T x, T y) { return !Equal(x, y); } }  
  
interface ISet<T> where CEquatable[T] { ... }  
bool Contains<T>(IEnumerable<T> values, T x)  
    where CEquatable[T] using CEq {... if (cEq.Equal(...) ...)}  
  
model default StringEqCaseS for CEquatable[String] { ... }  
model StringEqCaseIS for CEquatable[String] { ... }
```

Fig. 11. Concepts and models in C#^{cp1}

Some examples are shown in Fig. 11: the `CEquatable[T]` concept with the `Equal` signature and default implementation of `NotEqual`, the generic interface `ISet<T>` with the concept-requirement on the type parameter `T`, and two models of `CEquatable[]` for the type `String` — for case-sensitive and case-insensitive equality comparison. The first model is marked as a *default* model⁹: it means that this model is used if a model is not specified at the point of instantiation. For instance, in the following code `StringEqCaseS` is used to test equality of strings in `s1`.

```
ISet<String> s1 = ...;  
ISet<String>[using StringEqCaseIS] s2 = ...;  
s1 = s2; // Static ERROR, s1 and s2 have different types
```

Note that `s1` and `s2` have different types because they use different models of `CEquatable[String]`. This property is called “constraints-compatibility” in [9], but we will refer to it as “models-consistency”. One more interesting thing about C#^{cp1}: concept-requirements can be named. In the `Contains<T>` function (Fig. 11) the name `cEq` is given to the requirement on `T`; this name is used later in the body of `Contains<T>` to access the `Equal` function of the concept. It is also worth mention that the interface `IEnumerable<T>` is used as a type along with the concept `CEquatable[T]` being used as a constraint; thus, the role of interfaces is not ambiguous any more, interfaces and concepts are independently used for different purposes.

⁹ The default model can be generated automatically for a type if the type conforms to a concept, i.e. it provides methods required by the concept.

```
constraint Eq[T] { boolean T.equals(T other); }
constraint GraphLike[V, E] { V E.source(); ... }

interface ISet<T> where CEquatable[T] { ... }
bool Contains<T>(IEnumerable<T> values, T x)
  where CEquatable[T] using CEq { ... if (cEq.Equal(...) ... )

model default StringEqCaseS for CEquatable[String] { ... }
model StringEqCaseIS for CEquatable[String] { ... }
```

Fig. 12. Constraints and models in Genus

3.2.4 Constraints in Genus

Like G concepts and Haskell type classes, *constraints* in Genus [10] (an extension for Java) are used as constraints only. Fig. 12 demonstrates some examples: the Eq[T] constraint, which is used to constrain the T in the Set[T] interface; the model of Eq[String] for case-insensitive equality comparison; the multi-parameter constraint GraphLike[V,E], and the type-conditional generic model DualGraph[V,E]. Methods in Genus classes/interfaces can impose additional constraints:

```
interface List[E] { boolean remove(E e) where Eq[E]; ... }
```

Here the List[] interface can be instantiated by any type, but the remove method can be used only if type E of the elements satisfies the Eq[E] constraint. This feature is called *model genericity*.

Just as C#^{cpt}, Genus supports *multiple models* and automatic generation of the *natural* model, which is the same thing as the default model in C#^{cpt}. Due to this, the following code causes a static type error (we saw the same example in C#^{cpt}):

```
Set[String] s1 = ...;
Set[String with CIEq] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

In Genus this feature is called *model-dependent types*. An important note is to be made here: in contrast to true dependent types that depend on *values*, model-dependent types depend on models, which are compile-time artefacts. So the model-dependent types are just as dependent as generic types are type-dependent types.

As well as concept-requirements in C#^{cpt}, constraint-requirements in Genus can be named; the example is shown in Fig. 12: g is a name of the GraphLike[V,E] constraint required by the DualGraph[V,E] model. Because function signatures inside constraints are declared with an explicit receiver type (in a style close to JavaGI), such as the type T in the Eq[T] constraint, syntax of calls to functions in the case of named models is `_.(g.sink)()`, not `g.sink()`.

3.3 Which Philosophy Is Better If Any?

It is time to find out which approach is better. Taking into consideration what we explored in Sec. 3-1 and Sec. 3-2, we draw a conclusion that there are only two

language features important for generic programming that cannot be incorporated in a language *together*:

1. the use of a construct both as a type and constraint;
2. natural support for multi-type constraints.

Languages with “constraints-are-types” philosophy support the first feature but not the second, languages with “constraints-are-Not-types” philosophy vice versa¹⁰. Can we determine one feature that is more important?

It was shown in the study [31] that in practice interfaces that are used as constraints (such as `IComparable<T>` in C# or `Comparable<X>` in Java) are almost never used as types: authors had checked about 14 millions lines of Java code and found only one such example, which could be even rewritten and eliminated. According to [31], the same observation also holds for the code in Ceylon. It is hard to imagine any useful “constraint-and-type” example besides the `IPrintable` interface from Fig. 1. In those rare cases when this could happen, it is possible to provide a lightweight language mechanism for automatic generation of one construct from another. For example, single-parameter Genus constraints with some restrictions could be translated to Java interfaces, with the other direction being easier.

At the same time, multi-type constraints, which can be so naturally expressed under the “constraints-are-Not-types” approach, have rather awkward and cumbersome representation in the “constraints-are-types” approach as we have seen in Sec. 3-1. Language support for multiple models is also a problem in the latter approach: it is considered in detail in the next section. All other language facilities we discussed could be supported under any approach. Therefore, we claim that with respect to generic programming the “*constraints-are-Not-types*” approach is preferable. An additional benefit is that it eliminates the ambiguity in semantics of the interface-like constructs currently used for different purposes in OO languages.

4. Single Model versus Multiple Models

For simplicity, in this part of the paper we call “constraint” any language construct that is used to describe constraints, while a way in which types satisfy the constraints we call “model”. We have seen in the previous section that most of the languages allow having only one, unique model of a constraint for the given set of types; only C#^{opt} [9] and Genus [10] support multiple models¹¹. And indeed this makes sense for the languages with “constraints-are-types” philosophy, because it is not clear what to do with types that could implement interfaces (or any other similar constructs) in several ways. But how does this affect generic programming?

10 JavaGI seems to support both of them, but it actually provides different constructs for different purposes: single-parameter interfaces are more like Rust traits or Swift protocols, whereas multi-headed interfaces are similar to concepts and type classes; the latter cannot be used as types.

11 G [7] allows multiple models only in different lexical scopes.

It turns out that sometimes it is desirable to have multiple models of a constraint for the same set of types. The example of string sets with case-sensitive and case-insensitive equality comparisons we saw earlier is only one of such examples; another one is the use of different orderings on numbers, yet different graph implementations, and so on. Thus, in respect of generic programming, the absence of multiple models is rather a problem than a benefit. Without extending the language the problem of multiple models can be solved in two ways, and both of them have serious drawbacks.

1. Using the Adapter pattern. If one wants the type `Foo` to implement `IComparable<Foo>` in a different way, an adapter of `Foo`, the `Foo1` that implements `IComparable<Foo1>` can be created. This adapter then can be used instead of `Foo` whenever the `Foo1`-style comparison is required. An obvious shortcoming of this approach is the need to repeatedly wrap and unwrap `Foo` values; in addition, a code becomes cumbersome.
2. Using the Concept design pattern [20], which is considered in Sec. 4-1.

As we have discovered in Sec. 3-3, languages with the “constraints-are-types” philosophy are in the large less expressive than the ones with the “constraints-are-Not-types” philosophy. But may languages such as `C#cp` and `Genus`, which are in the “constraints-are-Not-types” category and support multiple models at the language level, be considered as the best languages for generic programming? Or we can imagine a language with a better design? We discuss this question in Sec. 4-3. And one more question: if language support for multiple models is a good idea, then why does not Haskell [24] allow multiple instances of a type class? This issue is considered in Sec. 4-2.

4.1 Concept Pattern

```
// F-bounded polymorphism
interface IComparable<T> { int CompareTo(T other); }
void Sort<T>(T[] values) where T : IComparable<T> { ... }
class SortedSet<T>      where T : IComparable<T> { ... }

// Concept Pattern
interface IComparer<T> { int Compare(T x, T y); }
void Sort<T>(T[] values, IComparer<T> cmp) { ... }
class SortedSet<T> { private IComparer<T> cmp; ...
    public SortedSet(IComparer<T> cmp) { ... } ... }
```

Fig. 13. The use of the Concept design pattern in `C#`

The Concept design pattern is suitable for programming languages with the “constraints-are-types” philosophy. It eliminates two problems:

1. First, it enables *retroactive modeling* of constraints, which is not supported in languages such as `C#`, `Java`, `Ceylon`, `Kotlin`, or `Scala`.
2. Second, it allows defining *multiple models* of a constraint for the same set of types.

The idea of the Concept pattern is as follows: instead of constraining type parameters, generic functions and classes take extra arguments that provide a required functionality — “concepts”. Fig. 13 shows an example: in the case of the Concept pattern the F-constraint $T : \text{Comparable}\langle T \rangle$ is replaced with an extra argument of the type $\text{IComparer}\langle T \rangle$. The $\text{IComparer}\langle T \rangle$ interface represents a concept of comparing: it describes interface of an object that can compare values of type T . As long as one can define several classes implementing the same interface, different “models” of the $\text{IComparer}\langle T \rangle$ “concept” can be passed into $\text{Sort}\langle T \rangle$ and $\text{SortedSet}\langle T \rangle$.

This pattern is widely used in generic libraries of mainstream object-oriented languages such as C# and Java; it is also used in Scala. Due to *implicit*s [6,20], the use of the Concept pattern in Scala is a bit easier: in most cases an appropriate “model” can be found by a compiler implicitly, so there is no need to explicitly pass it at a call site¹². Nevertheless, the pattern has two substantial drawbacks. First of all, it brings *run-time overhead*, because every object of a generic class with constraints has at least one extra field for the “concept”, while constrained generic functions take at least one extra argument. The second drawback, which we call *models-inconsistency*, is less obvious but may lead to very subtle errors. Suppose we have $s1$ of type $\text{HashSet}\langle \text{String} \rangle$ and $s2$ of the same type, provided that $s1$ uses case-sensitive equality comparison, $s2$ — the case-insensitive one. Thus, $s1$ and $s2$ use different, inconsistent models of comparison. Now consider the following function:

```
static HashSet<T> GetUnion<T>(HashSet<T> a, HashSet<T> b)
{
    var us = new HashSet<T>(a, a.Comparer);
    us.UnionWith(b);
    return us;
}
```

Unexpectedly, the result of $\text{GetUnion}(s1, s2)$ could differ from the result of $\text{GetUnion}(s2, s1)$. Despite the fact that $s1$ and $s2$ have the same type, they use different comparers, so the result depends on which comparer was chosen to build the union. Recall that in C#^{pt} and Genus models are part of types; therefore, a similar situation causes the static type error. But in the case of the Concept pattern models-consistency cannot be checked at *compile time*.

4.2 Instance Uniqueness in Haskell

Type classes in Haskell [23] provide the support for ad hoc polymorphism (function overloading). Like concepts and constraints, they define functions available for some types. For instance, a type class for equality comparison is defined in Haskell as follows:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

¹² Scala is often blamed for its complex rules of implicit resolution: sometimes it is not clear which implicit object is to be used.

```
x /= y = not (x == y)
```

It contains a function signature for the equality operator `==`, and provides a default implementation for the inequality operator `/=`. *Instances* (models) of this type class can be *retroactively* defined for types. For example, an instance for `Int`, a *type-conditional* instance for lists, and so on.

```
instance Eq Int where ...      -- (==) implementation
```

```
instance Eq a => Eq [a] where ... -- (==) implementation
```

As long as type classes support ad hoc polymorphism, they are “globally transparent”. If a function is a part of some type class, every time the name of this function is used, a compiler knows that an instance of the corresponding type class must be provided. Multiple instances of a type class for the same set of types are not allowed in Haskell, and there is a strong reason for that: *type inference*. Consider the following function definition:

```
foo xs ys = if xs == ys then xs else xs ++ ys
```

In Haskell such definition is valid and its type can be inferred. It is `Eq a => [a] → [a] → [a]`¹³. Inference succeeds, because a compiler knows the following facts:

- as long as `(++)` has the type `[a] → [a] → [a]`, `xs` and `ys` are lists;
- there is an instance of `Eq` for lists: `Eq a => Eq [a]`.

If there were no `Eq a => Eq [a]` instance available, type checking would fail.

Suppose that multiple instances of a type class are allowed. What to do with type inference of the `foo` in this case? To check whether there is at least one instance `Eq [a]` in the scope? But probably not all `Eq [a]` instances require `Eq a`, should not the type of the `foo` be changed in this case to the type `Eq [a] => [a] → [a] → [a]`?

Now look at the following code:

```
class Eq a => Baz a where
```

```
  bar :: a -> Int
```

```
useBar xs ys = if length xs > length ys then bar xs - bar ys  
              else bar ys - bar xs
```

If instances are uniquely defined, type checker just checks if there is an instance `Eq [a]` that implies `Baz [a]` (`xs` and `ys` are inferred to be lists because `length` has the type `[a] → Int`). But if there are multiple `Eq [a]` instances, then every `Baz [a]` instance must specify which `Eq [a]` instance it uses. It can even be the case that there is a `Baz [a]` instance for one `Eq [a]`, but not for another one. Therefore, at the point of the `useBar` definition a compiler has no idea whether there is an error of missed `Baz [a]` instance or not, because it knows nothing about the instance that might be used in a call to `useBar`. This information is available only at the point of the actual *call*, not the function definition.

Note that even with the *OverlappingInstances* extension for Haskell, multiple models in a sense we discuss in the paper are not supported. This extension indeed allows having in a scope several instances that match the constraints deduced for code. But

13 `[a]` is a type of generic list, it is a notation for `Data.List a`.

there must be only *one*, the most specialised instance among them that compiler can select unambiguously (according to some rules) at the point of the code *definition*. Again, not at the call site — at the point of definition. Thus, a user of the code still cannot choose between instances, an instance is already selected by a compiler. Thus, Haskell sacrifices language support for multiple models for the sake of type inference. It is a strong argument for Haskell users, but in the case of the most object-oriented programming languages, which usually do not permit omitting type annotations of function arguments as well as constraints on type parameters, there is *no need to prohibit multiple models* in OO languages.

4.3 Parameters versus Predicates

So far we have found out that languages with “constraints-are-Not-types” philosophy may potentially provide better support for generic programming compared to other languages, especially if they also allow multiple models definition. We have seen only two languages with such properties, C#^{opt} [9] and Genus [10], and there is an essential shortcoming in the design of both of them: constraints on type parameters are declared in “predicate-style” rather than “parameter-style”. For example, consider the following Genus definition [10]:

```
Map[V,W] SSSP[V,E,W] (V s)
where GraphLike[V,E], Weighted[E,W],
      OrdRing[W], Hashable[V] { ... }
```

SSSP[V,E,W] is a function for Dijkstras single-source shortest-path algorithm, with the GraphLike[V,E], Weighted[E,W], OrdRing[W], and Hashable[V] being constraints on type parameters. The constraints look as if they are predicates on types; and if they were predicates, this function would probably be well-designed. For example, in Haskell, G, C#, Java, Rust, and many other languages, where only one model of a constraint is allowed for the given set of types, constraints on type parameters are indeed predicates: types either satisfy the constraint (if they have a model that is unique) or not. But in Genus and C#^{opt} constraints are not predicates, they are actually parameters, as long as different models of a constraint can be used. In the worst case a call to the SSSP[V,E,W] function would be as follows:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
                 with MyGrLike with MyEdgeDW
                 with DescDOR with MyVerHash] (x);
```

Whereas in the best case:

```
...pathFromX = SSSP[MyVert, MyEdge, Double] (x);
```

Note that edge and weight types cannot be deduced, because they are determined by the models of the constraints, not by the vertex x itself. It is easy to imagine that the models of edge weighing (Weighted[E,W]) and its ordered ring (OrdRing[W]) would often vary, so in many cases a call to SSSP[V,E,W] is likely to look like this:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
                 with MyEdgeDW with DescDOR] (x);
```

This is not very bad but is also not good enough.

If look again at the SSSP algorithm one could notice that it really depends on three things: a source vertex, a model of a weighed graph which this vertex belongs to, and a model of hashing. Furthermore, at the level of the SSSP signature the type E of edges does not matter, we are interested in the model of weighed graph as a whole. Taking into account this ideas, we can rewrite the SSSP in the following way:

```
constraint WeighedGraph[V,E,W]
  extends GraphLike[V,E], Weighted[E,W], OrdRing[W] {}
Map[V,W] SSSP[V,E,W](V s)
  where WeighedGraph[V,E,W], Hashable[V] { ... }
```

Then a call to SSSP also becomes better:

```
...pathFromX = SSSP[MyVert, MyEdge, Double with MyWGr](x);
```

Nevertheless, we believe that in the case of multiple models the “predicate-style” syntax of constraints is misleading and makes it more difficult to write and call generic code. We suggest that the design of constraints has to be maintained in the “parameter-style”. One example of such design is provided by the extension for the OCaml language — *modular implicits* [32]; it is briefly discussed in Sec. 4-3-1. A sketch of the “parameter-style” design of constraints for object-oriented languages is presented in Sec. 4-3-2.

4.3.1 Modular Implicits in OCaml

In the “modular implicits” extension for the OCaml language [32] *module types* are used to describe constraints, *modules* represent models, with generic functions explicitly taking *module-parameters*. Fig. 14 demonstrates some examples. By contrast to concepts and genus constraints, module types and modules do not have type parameters, instead they have type members, such as the t in the Eq module type. Eq_int and Eq_list are the models of Eq for the int and generic list. Generic functions that need constraints, such as foo and foo', explicitly take the implicit module parameters EL and E. Notice that just as type parameters, EL and E are *compile-time* parameters, not run-time. They are called implicit because at a call to generic function actual models can be inferred, as in the x and y examples in Fig. 14. Note that in the foo function any model of comparison of lists is expected, whereas foo' expects a model of comparison of elements of lists and fixes the model Eq_list E for comparing lists.

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end

implicit module Eq_int = struct
  type t = int
  let equal x y = ...
end
```

```
implicit module Eq_list {E : Eq} = struct
  type t = Eq.t list
  let equal xs ys = ...
end

let foo {EL : Eq} xs ys = if EL.equal(xs, ys)
  then xs else xs @ ys
let foo' {E : Eq} xs ys = if (Eq_list E).equal(xs, ys)
  then xs else xs @ ys
let x = foo [1;2;3] [4;5]
let y = foo' [1;2;3] [4;5]
```

Fig. 14. OCaml modular implicits

4.3.2 Concept Parameters for C#

Fig. 15 shows some examples of generic code in the style of *concept-parameters*, which we call Cp# — C# with concept-Parameters. Concepts are the same as in C#^{cpt}, whereas constraints on type parameters are not predicates any more, they are explicitly stated as *parameters* in the angle brackets after the “|” sign. In the ICollection<T> interface the Remove method is obviously generic: it takes the concept-parameter eq for comparing values of type T. Note that concept-parameters can even be non-generic as in the MaxInt function.

If default models are supported, it must be possible to infer concept-arguments just in the same way as in C#^{cpt} or Genus, so that in common cases instances of generic functions and classes can be written in a usual way, without the need to specify the models required:

```
var ints = new ISet<int>(...);
var has5 = Contains(ints, 5);
var maxv = MaxInt(ints);
var minv = MaxInt<|IntOrdDesc>(ints);
ISet<String> s1 = ...;
ISet<String|StringEqCaseIS> s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

C#^{cpt} and Genus can easily be redesigned to follow the “concept-parameters” style presented here. With this style, the syntax of such languages would perfectly fit the semantics. On the other hand, the “concept-predicates” style used misleads a programmer and masks the fact that constraints can be satisfied non-uniquely.

```
concept Equality[T]{ bool Equal(T x, T y);
  bool NotEqual(T x, T y) {return !Equal(x, y);} }
concept Ordering[T] refines Equality[T] { int Compare(T x, T y); }

interface ISet<T | Equality[T] eq> { ... }
interface ICollection<T> { ... }
```

```

bool Remove<Equality[T] eq>(T x); ... }
bool Contains<T | Equality[T] eq>(IEnumerable<T> vs, T x)
{... if (eq.Equal(...)) ...}

int MaxInt<|Ordering[int] ord>(IEnumerable<int> vs) {...}
    
```

Fig. 15. The use of concept-parameters in C#

4. Single Model versus Multiple Models

Table 1 provides a summary on comparison of the languages: each row corresponds to one property important for generic programming, each column shows levels of support of the properties in one language. Black circle ● indicates full support of a property, ◐ — partial support, ○ means that a property is not supported at the language level, * means that a property is emulated using the Concept pattern, and the “-” sign indicates that a property is not applicable to a language. The “ModImpl” column corresponds to the Ocaml modular implicits. All the properties that appear in rows of Table 1 were discussed in Sec. 3 and Sec. 4. Related properties are grouped within horizontal lines; some of them are mutually exclusive. For example, as we saw earlier, the use of constraints as types and natural language support for multi-type constraints are mutually exclusive features. The major features analysed in the paper are highlighted in bold.

Table 1. The levels of support for generic programming on OO languages

	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C# ^{OP}	Genus	ModImpl
Constraints can be used as types	○	●	●	●	●	●	●	●	●	◐	○	○	○
Explicit self types	-	○	○	◐	◐	●	●	●	◐	-	-	-	-
Multi-type constraints	●	*	*	*	○	*	○	○	●	●	●	●	●
Retrospective type extension	○	●	○	○	○	○	●	●	○	○	○	○	○
Retrospective modeling	●	*	*	*	○	*	○	○	○	○	○	○	○
Type conditional models	●	○	○	○	○	○	●	○	●	●	●	●	●
Static methods	● ^a	○	●	○	●	●	●	●	●	● ^a	● ^a	● ^a	● ^a
Default method implementation	●	○	●	●	●	●	●	●	◐	●	●	○	○
Associated types	●	○	○	●	○	○	●	○	○	●	●	○	●
Constraints on associated types	◐	-	-	-	-	-	●	○	-	○	○	-	○
Same-type constraints	◐	-	-	●	-	-	●	○	-	●	●	-	●
Subtype constraints	-	●	●	●	●	●	-	●	○	○	●	○	-
Supertype constraints	-	○	○	●	○	○	-	○	○	○	○	○	-
Concept-based overloading	○	○	○	○	○	○	●	○	○	● ^d	○	○	○
Multiple models	○	*	*	*	*	*	○	○	○	● ^b	●	●	●
Models-consistency (model-dependent types)	- ^c	○	○	○	○	○	- ^c	- ^c	- ^c	- ^c	●	●	●
Model genericity	-	*	*	*	*	*	●	○	○	○	○	●	-

- a Constraints have no self types, therefore, any function member of a constraint can be treated as static function.
- b G supports lexically-scoped models but not really multiple models.
- c If multiple models are not supported, the notion of model-dependent types does not make sense.
- d C++0x concepts, in contrast to G concepts, provide full support for concept-based overloading.

The purpose of this table is not to determine the best language. The purpose is to show dependencies between different properties and to graphically demonstrate that the

“constraints-are-Not-types” approach is more powerful than the “constraints-are-types” one. It is also easy to see that there are features that can be expressed under any approach, such as static methods, default method implementations, associated types [15], and even type-conditional models.

It should be mentioned that the table is not exhaustive. There is a bunch of facilities that we did not discuss at all, although they can be considered independently of the study we made. Thus, for example, Genus [10] provides a support for such useful feature as *multiple dynamic dispatch*. Consider the following code:

```
constraint Intersectable[T] { T T.intersect(T that); }
model ShapeIntersect for Intersectable[Shape]
{   Shape Shape.intersect(Shape s) {...}
    // Rectangle and Circle are subclasses of Shape
    Rectangle Rectangle.intersect(Rectangle r) {...}
    Shape Circle.intersect(Rectangle r) {...}
    Shape Triangle.intersect(Circle c) {...}    ... }
```

It provides a subtype polymorphism on multiple arguments. So that in the call `s1.intersect(s2)` the most specific version of `intersect` would be used depending on the *dynamic* types of both `s1` and `s2`.

Another interesting feature is *concept variance*. For example, suppose we have the following C# definitions:

```
interface ISet<T | Equality[T] eq> { ... }
class B { ... }
class D : B { ... }
model EqB for Equality[B] { ... }
```

Should it be the case that `ISet<D, EqB>` is a legal instance? Under what conditions? It is also desirable to have the class `SortedSet<T | Ordering[T] ord>` implementing the interface `ISet<T|ord>`. Are there any problems here?

Now recall the `ICollection<T>` interface definition:

```
interface ICollection<T> { ...
    bool Remove<Equality[T] eq>(T x); ... }
```

The `SortedSet<T|ord>` class obviously implements the interface `ICollection<T>`. Should it be the case that the `ord` model of `Equality[T]` be used in place of `eq` in the `Remove` method? Or the `Remove` method has to remain model-generic?

And one more question. Consider the following function:

```
void foo<T | Equality[T] eq>(ISet<T|eq> s) { ... }
...
ISet<string | EqStringCaseS> s1 =
    new SortedSet<string | OrdStringCSAsc>(...);
foo(s1);
```

Which model of Equality[string] should be used inside the foo<>? The static EqStringCaseS or the dynamic OrdStringCSAsc one?

There are other questions similar to mentioned above that relate constraints on type parameters to usual features of object-oriented programming. Some of these questions require a careful type-theoretical investigation, so this is the subject for future work.

Acknowledgment

The author would like to thank Artem Pelenitsyn, Jeremy Siek, and Ross Tate for helpful discussions on generic programming.

References

- [1]. J. Belyakova. Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement. To appear in Lecture Notes in Computer Science, 2016.
- [2]. D. R. Musser, A. A. Stepanov. Generic Programming. Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, ISAAC '88, London, UK, UK: Springer-Verlag, 1989, pp. 13–25.
- [3]. R. Garcia et al. An Extended Comparative Study of Language Support for Generic Programming. *J. Funct. Program.*, Mar. 2007, 17(2), pp. 145–205.
- [4]. J.-P. Bernardy et al. A Comparison of C++ Concepts and Haskell Type Classes. Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP '08, Victoria, BC, Canada: ACM, 2008, pp. 37–48.
- [5]. R. Garcia et al. A Comparative Study of Language Support for Generic Programming. *SIGPLAN Not.*, Oct. 2003, 38(11), pp. 115–134.
- [6]. B. Oliveira, J. Gibbons. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. *J. Funct. Program.* July 2010, 20(3-4), pp. 303–352.
- [7]. S. Wehr, P. Thiemann. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Trans. Program. Lang. Syst.*, July 2011, 33(4), pp. 12:1–12:83.
- [8]. J. G. Siek, A. A. Lumsdaine. Language for Generic Programming in the Large. *Sci. Comput. Program.*, May 2011, 76(5), pp. 423–465.
- [9]. J. Belyakova, S. Mikhalkovich. Pitfalls of C# Generics and Their Solution Using Concepts. Proceedings of the Institute for System Programming, June 2015, 27(3), pp. 29–45.
- [10]. Y. Zhang et al. Lightweight, Flexible Object-oriented Generics. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, Portland, OR, USA: ACM, 2015, pp. 436–445.
- [11]. P. Canning et al. F-bounded Polymorphism for Object-oriented Programming. Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89, Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280.
- [12]. K. Bruce et al. On Binary Methods. *Theor. Pract. Object Syst.*, Dec. 1995, 1(3), pp. 221–242.
- [13]. A. Kennedy, D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. *SIGPLAN Not.*, May 2001, 36(5), pp. 1–12.

- [14]. J. Belyakova, S. Mikhalkovich. A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems. Transactions of Scientific School of I. B. Simonenko. Issue 2, 2015, no. 2, pp. 63–77 (in Russian).
- [15]. J. Järvi, J. Willcock, A. Lumsdaine. Associated Types and Constraint Propagation for Mainstream Object-oriented Generics. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, San Diego, CA, USA: ACM, 2005, pp. 1–19.
- [16]. A. Martelli, U. Montanari. An Efficient Unification Algorithm, *ACM Trans. Program. Lang. Syst.*, Apr. 1982, 4(2), pp. 258–282.
- [17]. The Ceylon Language Specification, version 1.2.2 (March 11, 2016). <http://ceylon-lang.org/documentation/1.2/spec/>
- [18]. The Kotlin Reference, version 1.0 (February 11, 2016). <https://kotlinlang.org/docs/reference/>
- [19]. Java Platform, Standard Edition (Java SE) 8. <http://docs.oracle.com/javase/8/>
- [20]. B. C. Oliveira, A. Moors, M. Odersky. Type Classes As Objects and Implicits. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.
- [21]. A. Pelenitsyn. Associated Types and Constraint Propagation for Generic Programming in Scala. *Programming and Computer Software*, 2015, 41(4), pp. 224–230.
- [22]. The Rust Reference, version 1.7.0 (March 3, 2016). <http://doc.rust-lang.org/stable/reference.html>
- [23]. C. V Hall. et al. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, Mar. 1996, 18(2), pp. 109–138.
- [24]. P. Wadler, S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, Austin, Texas, USA: ACM, 1989, pp. 60–76.
- [25]. B. Stroustrup. Concept Checking — A More Abstract Complement to Type Checking. Technical Report N1510=03-0093, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2003.
- [26]. B. Stroustrup, G. Dos Reis. Concepts — Design Choices for Template Argument Checking. Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2003.
- [27]. G. Dos Reis, B. Stroustrup. Specifying C++ Concepts. Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, Charleston, South Carolina, USA: ACM, 2006, pp. 295–308.
- [28]. B. Stroustrup, A. Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2012.
- [29]. A. A. Stepanov, M. Lee. The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories, 1995.
- [30]. A. Sutton. C++ Extensions for Concepts PPTS. Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2015.
- [31]. B. Greenman, F. Muehlboeck, R. Tate. Getting F-bounded Polymorphism into Shape. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 89–99.
- [32]. L. White, F. Bour, J. Yallop. Modular Implicits. ArXiv e-prints, Dec. 2015, arXiv: 1512.01895 [cs.PL].

Дизайн средств обобщённого программирования в объектно-ориентированных языках: ключевые решения*

Ю.В. Белякова <julbel@sfedu.ru>

*Институт математики, механики и компьютерных наук им. И.И. Воровича,
Южный федеральный университет,
344006, Россия, г. Ростов-на-Дону, ул. Б. Садовая, д. 105/42*

Аннотация. Принято считать, что объектно-ориентированные (ОО) языки программирования обеспечивают более слабую поддержку обобщённого программирования (ОП) по сравнению с такими функциональными языками как Haskell или SML. Это было показано в нескольких работах, посвящённых сравнительному анализу языков программирования. Но в последние годы появились новые объектно-ориентированные языки. Улучшили ли они поддержку обобщённого программирования? И если нет, есть ли причина, по которой ОО-языки до сих пор уступают функциональным языкам в этом отношении? В предыдущих исследованиях объектно-ориентированные языки не рассматривались специальным образом. Однако, возможности ОО-программирования влияют и на средства обобщённого программирования в языке, а также на сам стиль обобщённого программирования. В этой статье мы проводим сравнение средств обобщённого программирования в десяти современных объектно-ориентированных языках и их расширениях. В результате сравнительного анализа было обнаружено, что каждый из этих языков и расширений придерживается в точности одного из двух подходов к ограничению типовых параметров обобщённого кода. Таким образом, первый ключевой вопрос дизайна средств ОП, рассматриваемый в статье, это «какой подход лучше» (если он вообще есть). Оказывается, что большинство исследованных нами ОО-языков используют более ограниченный подход. Второй момент, который оказывает существенное влияние на выразительную мощь языка программирования, это поддержка множественных моделей. В статье рассматриваются преимущества и недостатки этой возможности, а также её связь с другими языковыми средствами поддержки обобщённого программирования.

Ключевые слова: объектно-ориентированные языки; обобщённое программирование; типы; ограничения; концепты; интерфейсы; концепт-паттерн; множественные модели; концепт-параметры.

DOI: 10.15514/ISPRAS-2016-28(2)-1

Для цитирования: Белякова Ю.В. Дизайн средств обобщённого программирования в объектно-ориентированных языках: ключевые решения. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 5-32 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-1

* Данная статья является расширенной версией статьи [1] принятой на конференцию «XX Бразильский симпозиум по языкам программирования».

References

- [1]. J. Belyakova. Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement. To appear in *Lecture Notes in Computer Science*, 2016.
- [2]. D. R. Musser, A. A. Stepanov. *Generic Programming*. Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, ISAAC '88, London, UK, UK: Springer-Verlag, 1989, pp. 13–25.
- [3]. R. Garcia et al. An Extended Comparative Study of Language Support for Generic Programming. *J. Funct. Program.*, Mar. 2007, 17(2), pp. 145–205.
- [4]. J.-P. Bernardy et al. A Comparison of C++ Concepts and Haskell Type Classes. Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP '08, Victoria, BC, Canada: ACM, 2008, pp. 37–48.
- [5]. R. Garcia et al. A Comparative Study of Language Support for Generic Programming. *SIGPLAN Not.*, Oct. 2003, 38(11), pp. 115–134.
- [6]. B. Oliveira, J. Gibbons. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. *J. Funct. Program.* July 2010, 20(3-4), pp. 303–352.
- [7]. S. Wehr, P. Thiemann. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Trans. Program. Lang. Syst.*, July 2011, 33(4), pp. 12:1–12:83.
- [8]. J. G. Siek, A. A. Lumsdaine. Language for Generic Programming in the Large. *Sci. Comput. Program.*, May 2011, 76(5), pp. 423–465.
- [9]. J. Belyakova, S. Mikhalkovich. Pitfalls of C# Generics and Their Solution Using Concepts. Proceedings of the Institute for System Programming, June 2015, 27(3), pp. 29–45.
- [10]. Y. Zhang et al. Lightweight, Flexible Object-oriented Generics. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, Portland, OR, USA: ACM, 2015, pp. 436–445.
- [11]. P. Canning et al. F-bounded Polymorphism for Object-oriented Programming, Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89, Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280.
- [12]. K. Bruce et al. On Binary Methods. *Theor. Pract. Object Syst.*, Dec. 1995, 1(3), pp. 221–242.
- [13]. A. Kennedy, D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. *SIGPLAN Not.*, May 2001, 36(5), pp. 1–12.
- [14]. Ю.В. Белякова, С.С. Михалкович. Средства обобщённого программирования в современных объектно-ориентированных языках. Часть 1. Анализ проблем. Труды научной школы И.Б. Симоненко. Выпуск 2, 2015, № 2, Ростов-на-Дону, стр. 63–77.
- [15]. J. Järvi, J. Willcock, A. Lumsdaine. Associated Types and Constraint Propagation for Mainstream Object-oriented Generics. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, San Diego, CA, USA: ACM, 2005, pp. 1–19.
- [16]. A. Martelli, U. Montanari. An Efficient Unification Algorithm, *ACM Trans. Program. Lang. Syst.*, Apr. 1982, 4(2), pp. 258–282.
- [17]. The Ceylon Language Specification, version 1.2.2 (March 11, 2016). <http://ceylon-lang.org/documentation/1.2/spec/>
- [18]. The Kotlin Reference, version 1.0 (February 11, 2016). <https://kotlinlang.org/docs/reference/>
- [19]. Java Platform, Standard Edition (Java SE) 8. <http://docs.oracle.com/javase/8/>

- [20]. B. C. Oliveira, A. Moors, M. Odersky. Type Classes As Objects and Implicits. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.
- [21]. A. Pelenitsyn. Associated Types and Constraint Propagation for Generic Programming in Scala. *Programming and Computer Software*, 2015, 41(4), pp. 224–230.
- [22]. The Rust Reference, version 1.7.0 (March 3, 2016).
<http://doc.rust-lang.org/stable/reference.html>
- [23]. C. V Hall. et al. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, Mar. 1996, 18(2), pp. 109–138.
- [24]. P. Wadler, S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, Austin, Texas, USA: ACM, 1989, pp. 60–76.
- [25]. B. Stroustrup. Concept Checking — A More Abstract Complement to Type Checking. Technical Report N1510=03-0093, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2003.
- [26]. B. Stroustrup, G. Dos Reis. Concepts — Design Choices for Template Argument Checking. Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2003.
- [27]. G. Dos Reis, B. Stroustrup. Specifying C++ Concepts. Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, Charleston, South Carolina, USA: ACM, 2006, pp. 295–308.
- [28]. B. Stroustrup, A. Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2012.
- [29]. A. A. Stepanov, M. Lee. The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories, 1995.
- [30]. A. Sutton. C++ Extensions for Concepts PDTS. Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2015.
- [31]. B. Greenman, F. Muehlboeck, R. Tate. Getting F-bounded Polymorphism into Shape. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 89–99.
- [32]. L. White, F. Bour, J. Yallop. Modular Implicits. ArXiv e-prints, Dec. 2015, arXiv: 1512.01895 [cs.PL].

Refinement Types in Jolie

Alexander Tchitchigin <a.chichigin@innopolis.ru>
Larisa Safina <l.safina@innopolis.ru>
Mohamed Elwakil <m.elwakil@innopolis.ru>
Manuel Mazzara <m.mazzara@innopolis.ru>
Fabrizio Montesi <fmontesi@imada.sdu.dk>
Victor Rivera <v.rivera@innopolis.ru>
Innopolis University, Software Engineering Lab.
420500, Russia, Innopolis, Universitetskaya Str. 1

Abstract. Jolie is the first language for microservices and it is currently dynamically type checked. This paper considers the opportunity to integrate dynamic and static type checking with the introduction of refinement types, verified via an SMT solver. The integration of the two aspects allows a scenario where the static verification of *internal* services and the dynamic verification of (potentially malicious) *external* services cooperate in order to reduce testing effort and enhance security.

Refinement types are well-known technique for numeric, array and algebraic data types. They rely on corresponding SMT-theories. Recently SMT solvers got support for a theory of strings and regular expressions. In the paper, we describe possible application of the theory to string refinement types. We use Jolie programming language to illustrate feasibility and usefulness of such extension. First, because Jolie already has syntax extension to support string refinements. We build on top of that extension to provide static type checking. Second, because in the realm of microservices the need for improved checking of string data is much higher as most of external communication goes through text-based protocols.

We present simplified but real-world example from the domain of web-development. We intentionally introduce a bug in the example demonstrating how easily it can slip a conventional type system. Proposed solution is feasible, as it do not accept program with the bug. Complete solution will need enhancements in precision and error reporting.

Keywords: Microservices, Jolie, Refinement Types, SMT, SAT, Z3

DOI: 10.15514/ISPRAS-2016-28(2)-2

For citation: Tchitchigin Alexander, Safina Larisa, Elwakil Mohamed, Mazzara Manuel, Montesi Fabrizio, Rivera Victor. Refinement Types in Jolie. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 33-44. DOI: 10.15514/ISPRAS-2016-28(2)-2

1. Introduction

“Stringly typed” is a new antipattern referring to an implementation that needlessly relies on strings, when other options are available. The problem of “string typing” appears often in service-oriented architecture and microservices on the border between a service and its clients (external interfaces) due to necessity to communicate over text-based protocols (like HTTP) and collaboration with clients written in dynamically-typed languages (like JavaScript). The solution to this problem can be found with refinement types, which are used to statically (or dynamically) check compatibility of a given value and refined type by means of predicates constraining the set of possible values. Though employment of numerical refinements is well-known in programming languages, string refinements are still rare.

In this paper, we introduce a design for extending the Jolie programming language [24,3] and its type system. On top of previous extensions with choice type [27] and regular expressions, we introduce here string refinement type and we motivate the reasons for such extension. Section 2 recalls the basic of the Jolie language and its type system while Section 3 describes the open problem this paper attacks with clarifying examples. Section 4 discusses related work in the context of using SMT solvers for static typing of refinement types.

2. Jolie programming language

Jolie [24] is the first programming language based on the paradigm of microservices [17]: all components are autonomous services that can be deployed independently and operate by running parallel processes, programmed following the workflow approach. Microservices can be composed to obtain, in turn, other microservices. The language was originally developed in the context of a major formalization effort for workflow and services composition languages, the EU Project SENSORIA [1], which spawned many models for reasoning on the composition of services (e.g., [19,20]). Jolie comes with a formally-specified semantics [16,15,23]; on the more practical side it is inspired by standards for Serviceoriented Computing such as WS-BPEL [4]. The combination of theoretical and practical aspects in Jolie enabled its usage in research on correct-by-construction software (see, e.g., [26,9,21]).

Microservices work together by exchanging messages. In Jolie, messages are structured as trees [23] (a variant of the structures that can be found in XML or JSON). Communications are type checked at runtime, when messages are sent or received. Type checking of incoming messages is especially relevant, since it mitigates the effect of ill-behaved clients. The work in [25] presents a first attempt at formalizing a static type checker for the core fragment of Jolie. However, for the time being, the language is still dynamically type checked.

3. Extension of Jolie Type System

Safina et al [28] extended the basic type system of Jolie with type choices. The work had been then continued with the addition of regular expression types, a special case

of refinement types. In refinement types, types are decorated with logical predicates which further constrain the set of values described by the type and therefore represent the specification of invariant on values. Here, we extend this with the possibility of expressing invariants on string values in form of regular expressions.

The integration of static and dynamic analysis allows considering “internal” services (native Jolie services) and calls from “external” services (potentially developed in other languages) in a complementary way. The first ones can be statically checked while the second ones, which could exhibit malicious behavior, still need a runtime validation.

The key idea behind service-oriented computing, and microservices in particular, is the ability to connect services developed in different programming languages and possibly running on different servers over standard communication protocols [18]. A common use case is the implementation of APIs for Web and mobile applications. In such scenarios, the de-facto standard communication protocol is HTTP(S), combined with standardized data formats (SOAP, JSON, etc.).

HTTP is a text-based protocol, where all data get serialized into strings¹. Moreover, clients of a service (an application or another service) may have been developed in a language that does not support particular datatypes (e.g., JavaScript does not have a datatype for calendar dates or time of day), therefore relying on string representation for internal processing too. The same issue arises with key-value storage systems (e.g., Memcache and Redis), which support only string keys and string values. These factors make string handling an important part of a service application, especially at the boundary with external systems.

Not all strings are made equal. For example, GUIDs are often used to identify records in a store. GUIDs are represented as strings of hexadecimal digits with a particular structure. Currently, developers have to manually check the conformance of received values to the expected format. In such a scenario, a developer has to find her way in a narrow stream between the Scylla of forgetting to insert necessary checks and the Charybdis of inserting too many checks for data that has been already validated².

Description of the shape of expected string data (like GUID or e-mail address) is natural with regular expressions. Adding the description of this shape to the datatype definition allows the compiler to automatically insert the necessary dynamic checks (for public functions) and statically validate the conformance (for internal calls). This is the extension of refinement type to string type. The same techniques and tools used for static verification of conformance for numerical refinements [17, 12] can be used for strings. For the purposes of this paper we will use Z3 SMT solver by Microsoft Research [6], which recently got support for theory of strings and regular expressions in its development branch.

¹ Jolie partially mitigates this aspect with automatic conversion of string serializations to structured data by following the interface definition of the service [23]. However, this does not solve the general problem addressed here.

² Scylla and Charybdis are monsters of Greek mythology living on the two sides of a narrow channel so that sailors trying to avoid one would have fallen into the other.

3.1 Example: a news board

The approach to static checking of string refinements using Z3 SMT solver is illustrated here by a simple example, i.e. a service using refined datatype for GUIDs and the SMT constraints generated for it.

A news board is a simple service in charge of retrieving posts composed by a particular user of the system. The service receives user information via HTTP in a string format. String refinement types allow the definition of constraints on user IDs as an alternative to the implementation of the logic checking the constraint inside the posts retrieving operation.

```
type guid: string (" [A-F\\d]{8,8} - [A-F\\d]{4,4} - [A-F\\d]{4,4} - [A-F\\d]{4,4} - [A-F\\d]{4,4} - [A-F\\d]{12,12} ")
```

Types for storing user and posts information are also necessary³.

```
type user: void {  
  .uid: guid  
  .name: string  
  .age: int (age > 18) }  
type post_type: void {  
  .pid: guid  
  .owner: guid  
  .content: string }  
type posts: void { .post*: post_type }
```

We leave service deployment information out of this paper due to its low relevance to the topic, the full code example can be found in [2]. The behavioral fragment of the *news board* demonstrates the post retrieval for a particular user. To get the information the right user has to be found (*find_user_by_name*) and pass the GUID to get all users posts.

There are two definitions of the operation in the following code fragment: *all_posts_by_user* and *all_posts_by_user2*. In the first one the correct data is passed to *get_all_users_posts*, i.e. *user.uid*; while in the second *user.name* is passed. Without string refinement a problem would arise. The code is syntactically correct. However, it's semantically incorrect since no information can be retrieved by user's name when user's ID is actually expected.

³ Please note that in Jolie we structure the variable's data as a tree, where the nodes contain values. Using the void type for the variable on the top of the tree, we show that it contains no data and is used as a container for its subtypes.

```
main {
  all_posts_by_user (name) {
    find_user_by_name@SelfOut (name) (user) ;
    get_all_users_posts@SelfOut (user . uid) (posts) };

  all_posts_by_user2 (name) {
    find_user_by_name@SelfOut (name) (user) ;
    //and here we pass the wrong field!
    get_all_users_posts@SelfOut (user . name) (posts) };

  //find_user_by_name definition
  //get_all_users_posts definition
}
```

Introducing string refinement allows Jolie to have both dynamic and static checking for strings. In case of dynamic checking, the string is verified at runtime when passed to the receiving service. The more interesting case is static checking by means of SMT. Here we present the most essential parts of the encoding, complete example can be found in [2].

```
; notions of types, terms and typing relation
(declare-sort Type)
(declare-sort Term)
(declare-fun HasType (Term Type) Bool)

; type of strings of a programming language
(declare-fun string () Type)
; translation from Z3 built-in String type to our string
  type and back
(declare-fun BoxString (String) Term)
(declare-fun string-term-val (Term) String)
(assert (forall ((str String))
  (= (string-term-val (BoxString str)) str)))
(assert (forall ((s String))
  (HasType (BoxString s) string)))

; guid type that refines string type
(declare-fun guid () Type)
(declare-fun guid-re () (Regex String))
; the construction of the regular expression is omitted
)
```

```
; refinement definition for guid type
(assert (forall ((x Term))
  (iff (HasType x guid)
    (and (HasType x string)
      (str.in.re (string-term-val x) guid-re))))))
; we define type 'user' through it's projections
(declare-fun user () Type)
(declare-fun user.uid (Term) Term)
(declare-fun user.name (Term) Term)
(declare-fun user.age (Term) Term)
; typing rules for projections
(assert (forall ((t Term))
  (implies (HasType t user)
    (and (HasType (user.uid t) guid)
      (HasType (user.name t) string)
      (HasType (user.age t) nat))))))

(declare-fun find_user_by_name (Term) Term)
; find_user_by_name : string -> user
(assert (forall ((name Term))
  (implies (HasType name string)
    (HasType (find_user_by_name name) user))))))

; type checking for all_posts_by_user
(assert (not (forall ((t Term))
  (implies (HasType t string)
    (HasType (user.uid (find_user_by_name t)) guid))))))
; type checking for all_posts_by_user2
(assert (not (forall ((t Term))
  (implies (HasType t string)
    (HasType (user.name (find_user_by_name t)) guid))))))
```

Type checking is based on proving a theorem stating that a function is correctly typed. Technically, the opposite proposition is actually stated and the SMT solver is put in charge of finding a counterexample. A failure in such an attempt leads to the conclusion that the original theorem has to be true (proof by contradiction).

The Z3 solver successfully proves the well-typedness theorem for the correct implementation of all posts by user, and fails to disprove the incorrect implementation (*all_posts_by_user2*) due to many simplifications to the presented SMT encoding for the sake of clarity and understandability. Employment of a more sophisticated encoding for the actual implementation of refinement constraints may mitigate this situation and is left as future work.

4. Related Work

Within the context of functional languages, type-checking of refined types by employing SMT solvers is not new. In [7], the authors present the design and implementation of the F7 enhanced type-checker for the functional language F# that verifies security properties of cryptographic protocols and access control mechanisms using Z3 [10]. The SAGE language [17] employs a hybrid approach [13] that performs both static and dynamic type-checking. During compilation time, the Simplify theorem prover [11] is used to check refinement types. If Simplify is not able to decide a particular subtyping relation, a proper type cast is inserted in the code and it is checked at runtime. If the type cast fails during runtime, this particular subtyping relation is inserted in a database of known failed casts. In contrast to checking syntactic subtyping as in F7 and SAGE, the authors of [8], introduce semantic subtyping checking for a subset of the M language [5] using the Z3 SMT solver.

5. Conclusions

The Jolie language is dynamically type-checked. This paper explores the possibility of integrated dynamic and static type checking with the introduction of refinement types, verified via an SMT solver. The integration of the two aspects allows a scenario where the static verification of internal services and the dynamic verification of (potentially malicious) external services cooperates in order to reduce testing effort and enhance security.

In this work, we motivate the usefulness and feasibility of string refinement types using an example. Naturally, we need to integrate this extension with an actual type-checker employing a more advanced SMT-encoding. Not only for strings but for numerical types too which is well-known and useful tool for correctness enhancement.

When we have a type-checker for refinement types, an interesting empirical study would be checking of existing programs augmented with refined types to discover whether this technique can uncover bugs caused by a developer's oversight.

References

- [1]. EU Project SENSORIA. Accessed April 2016. <http://www.sensoria-ist.eu/>.
- [2]. Gist of SMT constraints for the example. Accessed April 2016. <https://gist.github.com/gabriel-fallen/a04c33860e2157201fa8>.
- [3]. Jolie Programming Language. Accessed April 2016. <http://www.jolie-lang.org/>.
- [4]. WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html>.
- [5]. Power Query formula reference. Technical Report, August 2015.
- [6]. Microsoft Research. Accessed April 2016. Z3. <https://github.com/Z3Prover/z3>.

- [7]. Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffei. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011.
- [8]. Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10*, pages 105–116, New York, NY, USA, 2010. ACM.
- [9]. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
- [10]. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11]. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
- [12]. Joshua Dunfield. A unified system of type refinements. PhD thesis, Air Force Research Laboratory, 2007.
- [13]. Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06*, pages 245–256, New York, NY, USA, 2006. ACM.
- [14]. Tim Freeman and Frank Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991.
- [15]. Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
- [16]. Claudio Guidi, Roberto Lucchi, Gianluigi Zavattaro, Nadia Busi, and Roberto Gorrieri. Sock: a calculus for service oriented computing. In *ICSOC*, volume 4294 of LNCS, pages 327–338. Springer, 2006.
- [17]. Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report), 2006.
- [18]. James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. Accessed April 2016. <http://martinfowler.com/articles/microservices.htm>.
- [19]. Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
- [20]. Manuel Mazzara, Faisal Abouzaid, Nicola Dragoni, and Anirban Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations – A process algebra perspective. In *Web Services and Formal Methods - 8th International Workshop, WS-FM*, pages 64–78, 2011.
- [21]. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, September 1992.
- [22]. Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, 2010.
- [23]. Fabrizio Montesi. Process-aware web programming with Jolie. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 761–763, New York, NY, USA, 2013. ACM.
- [24]. Fabrizio Montesi and Marco Carbone. Programming Services with Correlation Sets. In *Proc. of Service-Oriented Computing - 9th International Conference, ICSOC*, pages 125–141, 2011.

- [25]. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.
- [26]. J. M. Nielsen. A Type System for the Jolie Language. Master’s thesis, Technical University of Denmark, 2013.
- [27]. Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbriellini. AIOGJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 161–170, 2014.
- [28]. Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.

Refinement типы для языка Jolie

Александр Чичигин <a.chichigin@innopolis.ru>

Лариса Сафина <l.safina@innopolis.ru>

Мохамед Эльвакиль <m.elwakil@innopolis.ru>

Мануэль Маццара <m.mazzara@innopolis.ru>

Фабрицио Монтези <fmontesi@imada.sdu.dk>

Виктор Ривера <v.rivera@innopolis.ru>

Университет Иннополис,

420500, Россия, респ. Татарстан, г. Иннополис, ул. Университетская, д.1.

Аннотация. Jolie — язык программирования для разработки микросервисов и на текущий момент является динамически проверяемым. В статье рассматривается возможность объединить динамическую и статическую проверку типов с помощью refinement типов, проверяемых SMT-решателем. Соединение этих двух аспектов делает возможным сценарий, когда статическая верификация *внутренних* сервисов и динамическая проверка (потенциально злонамеренных) *внешних* сервисов совместно снижают объёмы необходимого тестирования и увеличивают безопасность системы.

Refinement типы хорошо известны применительно к числовым типам данных, алгебраическим типам данных и массивам. Они основываются на соответствующих SMT теориях. Недавно SMT-решатели получили поддержку теории строк и регулярных выражений. В статье описывается возможность применения этой теории к строковым refinement типам. Мы используем язык программирования Jolie чтобы продемонстрировать целесообразность и полезность такого расширения. В первую очередь, потому что Jolie уже содержит синтаксическое расширение для строковых refinement типов. Мы развиваем указанное расширение, предоставляя статическую проверку типов. Во-вторых, поскольку в области микросервисов значение улучшенной проверки строковых данных гораздо выше, так как большинство коммуникаций с внешними системами происходит по текстовым протоколам.

Мы демонстрируем упрощённый, но реалистичный пример системы из области web-разработки. В пример преднамеренно внесена ошибка, показывая, как легко она ускользает от традиционной системы типов. Предложенное расширение целесообразно, поскольку оно не пропускает программу с ошибкой. Полноценное решение потребует доработки в части точности проверки и качества сообщений об ошибках.

Ключевые слова: Микросервисы, Jolie, Refinement типы, SMT, SAT, Z3

DOI: 10.15514/ISPRAS-2016-28(2)-2

Для цитирования: Чичигин Александр, Сафина Лариса, Эльвакиль Мохамед, Маццара Мануэль, Монтези Фабрицио, Ривера Виктор. Refinement типы для языка Jolie. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 33-44 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-2

Список литературы

- [1]. EU Project SENSORIA. Accessed April 2016. <http://www.sensoria-ist.eu/>.
- [2]. Gist of SMT constraints for the example. Accessed April 2016. <https://gist.github.com/gabriel-fallen/a04c33860e2157201fa8>.
- [3]. Jolie Programming Language. Accessed April 2016. <http://www.jolie-lang.org/>.
- [4]. WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html>.
- [5]. Power Query formula reference. Technical Report, August 2015.
- [6]. Microsoft Research. Accessed April 2016. Z3. <https://github.com/Z3Prover/z3>.
- [7]. Jesper Bengtson, Karthikeyan Bhargavan, Cedric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011.
- [8]. Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10*, pages 105–116, New York, NY, USA, 2010. ACM.
- [9]. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
- [10]. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11]. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
- [12]. Joshua Dunfield. A unified system of type refinements. PhD thesis, Air Force Research Laboratory, 2007.
- [13]. Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06*, pages 245–256, New York, NY, USA, 2006. ACM.
- [14]. Tim Freeman and Frank Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991.
- [15]. Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
- [16]. Claudio Guidi, Roberto Lucchi, Gianluigi Zavattaro, Nadia Busi, and Roberto Gorrieri. Sock: a calculus for service oriented computing. In *ICSOC*, volume 4294 of LNCS, pages 327–338. Springer, 2006.
- [17]. Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report), 2006.

- [18]. James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. Accessed April 2016. <http://martinfowler.com/articles/microservices.htm>.
- [19]. Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
- [20]. Manuel Mazzara, Faisal Abouzaid, Nicola Dragoni, and Anirban Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations – A process algebra perspective. In *Web Services and Formal Methods - 8th International Workshop, WS-FM*, pages 64–78, 2011.
- [21]. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, September 1992.
- [22]. Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master’s thesis, University of Bologna, 2010.
- [23]. Fabrizio Montesi. Process-aware web programming with Jolie. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC ’13*, pages 761–763, New York, NY, USA, 2013. ACM.
- [24]. Fabrizio Montesi and Marco Carbone. Programming Services with Correlation Sets. In *Proc. of Service-Oriented Computing - 9th International Conference, ICSOC*, pages 125–141, 2011.
- [25]. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.
- [26]. J. M. Nielsen. A Type System for the Jolie Language. Master’s thesis, Technical University of Denmark, 2013.
- [27]. Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. AIOGJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 161–170, 2014.
- [28]. Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2016.

Visual Dataflow Language for Educational Robots Programming

G.A. Zimin <zimin.grigory@gmail.com>

D.A. Mordvinov <mordvinov.dmitry@gmail.com>

*St Petersburg State University, Institute of Mathematics and Mechanics,
28 University Ave., 198504, Russia*

Abstract. Visual domain-specific languages usually have low entry barrier. Sometimes even children can program on such languages by working with visual representations. This is widely used in educational robotics domain, where most commonly used programming environments are visual. The paper describes a novel dataflow visual programming environment for embedded robotic platforms. Obviously, complex dataflow languages are not simple for understanding. The purpose of our tool is to "bridge" between lightweight educational robotic programming tools (commonly these tools provide languages which are based on control flow model) and complex industrial tools (which provide languages based on more complex dataflow execution model). We compare programming environments mostly used by robotics community with our tool. After brief review of behavioural robotic architectures, some thoughts on expressing them in terms of our dataflow language are given. Visual language, which is described here, provides opportunity to mix dataflow and control flow models for robotics programming. We believe that it is important for educational purposes. Program on our language consists of different blocks (visual representation of data transformation processes) and "links" which presents data flow between them. Domain-specific modelling approach was used to develop our language. Also, this paper provides the examples of solving two typical robot control tasks in our language.

Keywords: robotics, data flow, visual programming, educational robotics, domain-specific modelling, subsumption architecture.

DOI: 10.15514/ISPRAS-2016-282)-3

For citation: G.A. Zimin, D.A. Mordvinov. Visual Dataflow Language for Educational Robots Programming. *Trudy ISP RAN/Proc. ISP RAS*. vol. 28, issue 2, 2016, pp. 45-62 DOI: 10.15514/ISPRAS-2016-28(2)-3

1. Introduction

Programming languages for creating robotic controllers are actual topics of research oftenly discussed at major conferences, such as ICRA¹ or IROS². Visual programming languages (VPLs) are also actively discussed for the last three decades, the largest conferences are held annually, e.g. VL/HCC³. VPLs are oftenly applied in robotics domain [1–5] allowing to create and visualize robotic controllers. Robotic VPLs are commonly used for educational purposes, making possible for students of even junior schools to create robotic programs. For these aims there already exists a great number of educational robotic programming environments based on VPLs, e.g. NXT-G⁴, TRIK Studio⁵, ROBOLAB⁶, also there are some academic tools implementing interesting and novel approaches to educational robotics programming [1], [3], [5].

Robotic control programs are inherently reactive: they transform data which is continuously coming from multiple sensors into the impulses on actuators. For this reason dataflow languages (DFLs) are well-suitable for robotics programming. Many researchers denoted the convenience of dataflow visual programming languages (DFVPLs) [6], finding them more useful than textual DFLs, for example because data flows explicitly displayed on the diagram. There are large and complex general-purpose and domain-specific development environments such as LabVIEW⁷ and Simulink⁸ that provide a large (and sometimes even cumbersome) set of libraries for robotics programming. More detailed discussion of robotics VPLs will be provided in section 2.

There is a large number of robotic constructor kits for learning the basics of robotics and cybernetics, such as LEGO MINDSTORMS⁹, TRIK, ScratchDuino¹⁰. Modern programming languages, which are used for programming those kits, are based on the control flow model rather than on dataflow model. Control flow-based languages are good for solving scholar "toy" tasks, but may be inconvenient for programming more complex "real world" controllers that may be conveniently expressed on DFLs. The simple DFVPL may be considered as a useful step from educational VPLs to the programming languages, which are used in universities and industry.

This paper discusses a novel extensible tool for programming all popular educational robotic kits on dataflow visual programming language. It should be noted that, in distinction from other tools, our tool is focused on embedded systems (section 6).

¹IEEE International Conference on Robotics and Automation. Available: <http://www.icra2016.org/>

²International Conference on Intelligent Robots and Systems. Available: <http://www.iros2016.org/>

³IEEE Symposium on Visual Languages and Human-Centric Computing. Available: <https://sites.google.com/site/vl-hcc2016/>

⁴NXT-G quick programming guide. Available: <http://www.legoengineering.com/nxt-g-quick-guide/>

⁵All about TRIK: TRIK Studio. Available: <http://blog.trikset.com/p/trik-studio.html>

⁶ROBOLAB quick guide. Available: <http://www.legoengineering.com/robolab-quick-guide/>

⁷LabVIEW System Design Software - National Instruments. Available: <http://www.ni.com/labview/>

⁸Simulink - Simulation and Model-Based Design. Available: <http://www.mathworks.com/products/simulink/>

⁹MINDSTORMS EV3 – Products. Available: <http://www.lego.com/en-us/mindstorms/products/>

¹⁰ScratchDuino — Magnetic Robot Construction Kit. Available: <http://www.scratchduino.com/>

Another interesting detail of our work is the application of DSM-approach for implementation of visual editor: it is entirely generated by QReal DSM-platform [7], [8] without even a line of code written. We also take into consideration the popularity of Brooks' Subsumption Architecture [9] which is still mainstream approach to design of complex robotic controllers [1], [2], [4], [10] despite it was proposed 30 years ago. Brooks' Subsumption Architecture and some other are conveniently expressed in our language, they are discussed in section 3.

The remainder of a paper is organized as follows. An overview of robotics VPLs and DFVPLs is presented in section 2. Section 3 provides some general thoughts on how some widely used robotic behavioural architectures are expressed in our language. A detailed description of our language is given in section 4. Section 5 demonstrates two typical robotic controllers expressed in our language. The most important details of implementation are discussed in section 6. Finally, the last section concludes the paper and discusses possible directions for future work.

2. Similar Tools

Robot programming environments can be divided into three categories: educational, which allows to program small educational robotic kits; industrial, which have a rich toolkit for creating large and complex robotic controllers; academic, which implement new interesting ideas, however they are often unavailable for downloading or unusable.

Educational visual environments are for example NXT-G and ROBO LAB for LEGO MINDSTORMS NXT kit, EV3 Software for the Lego Mindstorms EV3 kit, TRIK Studio for NXT, EV3 and TRIK. Those environments simplify solving primitive robot control tasks like finding a way out of the maze and driving along the line using light sensors, which makes the process of learning the basics of programming and robot control easy. But their simplicity often bounds the flexibility of the language. Visual languages of all mentioned systems are based on control flow model.

There is also a number of well-known visual robotic programming environments of industrial level. For example, general-purpose LabVIEW from National Instruments with the DFVPL G, programming environment Simulink developed by MathWorks for modelling different dynamic models or control systems. Those products offer a huge set of models and libraries to create control systems, test benches, real-time systems of any complexity, using model-driven approach. LabVIEW provides opportunity for programming small robots. There are lots of examples of applying LabVIEW in education [11], [12], but much more often adaptations like Robolab are used in educational process. It should be noted that those environments are distributed under the commercial license.

Another example of an visual robotics industrial system is the Microsoft Robotics Developer Studio (MSRDS) [13], which is free for academic purposes and allow to create distributed robotic systems on DFVPL. MSRDS officially supports a large set of robotic platforms, LEGO NXT [14] in particular (however, the autonomous mode

for NXT is not supported). MSRDS has the ability of manual integration with custom robotic platforms, but unhappily is not maintained since 2014.

There is a lot of scientific research has done in this area, e.g., dissertation [1] describes a visual programming module for expressing robotic controllers in terms of extended Moore machines, [3], [4] describe visual environment for *occam- π* language and *Transterpreter* framework, and its usage in education and swarm robotics. Article [5] describes DFVPL for beginners, which is pretty close to a one we introduce here. However at the moment RuRu is under development, it has pretty limited functionality and even unavailable for download.

3. Robotic Behavioural Architectures

The task of creation complex and scalable robotic controller is indeed a non-trivial task. Starting from middle 80's many researchers have attempted to solve this problem and a number of behavioural robotic architectures were proposed [18]. Those approaches are quickly became popular in robotics community and they are still actual. For example, the original work that introduced Brooks' *Subsumption Architecture* [9] is one of the most cited works in the entire robotics domain. We believe that the description of modern language for programming robotic controllers should contain at least general thoughts on how those architectures may be expressed in it.

A controller built on Brooks' Subsumption Architecture is decomposed into a hierarchy of levels of competence where each new layer describes a new feature of robot's behaviour. Levels are "ordered" upside-down, the higher levels describe more "intelligent" behaviour of robot. Higher levels depend on lower ones but not vice versa, so failures of higher levels do not imply the failure of lower. This is important feature for mobile robotics, e.g. if robot's gripper was damaged the controller is still able to deliver robot to its base. Levels of responsibility are expressed as a set of "behaviours" running concurrently and interacting with each other via channels of *suppression* and *inhibition*. Using them, higher levels can suppress the activity of lower ones thus correcting the behaviour of the whole system.

Brooks' in his original work offered to express behaviours in terms of *state machines*. Each layer implements some simple logic of transformation sensor inputs into impulses on actuators. Dataflow languages are obviously as suitable as state machines for expressing such behaviours. In our language each behaviour can be represented as "black box" described by separate subprogram. Also, our language contains *Suppressor* and *Inhibitor* elements for layers communication. Levels can be invoked concurrently, so we can conclude that our language allows the convenient expression of controllers built with Subsumption Architecture. That is demonstrated by an example in section 5.

Connell's *Colony Architecture* [15] is a very similar to Brooks' one, but solves some scalability issues of Subsumption Architecture. It also decomposes the controller into a number of communicating concurrent levels, but they are unordered. The other difference is an absence of inhibition channel, data inhibition should be implicitly ex-

pressed by predicated in layers. Our language does not force any order between layers, predicative inhibition can be implemented simply with *Filter* block. So Colony Architecture is also well-expressed in our language.

There also exist Arkin's *Motor Schema* [16] and Rosenblatt's *Distributed Architecture for Mobile Navigation (DAMN)* [17] which are compatible with our language, but the detailed descriptions will be omitted here. General ideas on their implementation on *occam- π* language can be found in [18], we believe that those ideas will suffice in the context of this paper. The complete research of expressing behavioural architectures in our language is a topic for separate paper.

4. Language Description

Evolution of a domain-specific modeling (DSM) tools allows to quickly create a fairly sophisticated visual programming languages [19]. TRIK Studio programming environment is an example of a system that was created using DSM-based approach on QReal platform [7], [8]. Basing on an industrial experience of TRIK Studio developers we decided to create the visual editor of our language on QReal platform.

Program on DFVPL is a set of blocks and flows that connect blocks. DFVPL blocks process incoming tokens and emit resulting data into the output data flows. Blocks in our language can be divided into several groups that are described below. Some blocks require to specify information on textual language. The language we use is a statically typed dialect of Lua.

- *Control* blocks that implement basic algorithmic constructions (conditions, loops, etc).
 - *ConstValue* and *RandomValue* blocks that are responsible for generation of a random number or a predetermined value of any type.
 - *Loop*, *If*, *Switch*. These blocks implement general control flow algorithmic constructions in dataflow style. *Loop* is an entity which emits a sequence of numbers for a given amount of times. *If* checks the condition specified on a textual language and sends them to *True* or *False* channel. *Switch* successively checks guard conditions and if it is evaluated as *true* sends incoming data to corresponding channel.
 - *Function* block, which allows to process of the input data in a textual language. Most usually this block is used for mathematical processing of data.
 - *FinalBlock* stops the execution of program when receiving any data.
 - *Subprogram* for reusing the code. Double-click on subprogram block opens new visual editor tab with an implementation of this subprogram. Contents of that tab can be then edited by user in exactly the same way he edits the main diagram.
 - *GetSetVariable* – purely practical block for setting value of some global variable or emitting it into output flows.

- *Wait* block delays data processing.
- *DelayAndFilter* is the extension of the previous block adding the filtering condition and checking the amount of emitted data validated by condition.
- *Fork*, *EndFork* blocks that provide an ability of invoking code in platform-specific execution units. See section 6 for details.
- *Drawing*. Blocks for drawing on display of the robot and on the floor in simulator mode.
 - *PaintSettings* defines current background color, thickness and color of pen and color and style of the brush that draw graphical primitives.
 - *ShapePainter*, *SmilePainter*, *Text* are used for drawing some shape, text or smile on robot's display.
 - *Clear* block removes all graphics from robot's display when receiving any token.
 - *Pen* block puts down or raises the marker for drawing the robot's trace on the "floor" of 2D simulator.
- *Flow manipulation*. These elements provide opportunity to manipulate data, which flow between blocks.
 - *InPort*, *OutPort* emit tokens that come into some instance of *Subprogram* block into a diagram implementing it and similarly redirect data from subprogram implementation into output flows of active instance of *Subprogram* block.
 - *Supressor*, *Inhibitor* inhibit or replace token of some flow with tokens of another. These, *Subprogram* and *Fork* blocks provide a compatibility with the Brooks' Subsumption Architecture.
 - *Zip*, *Unzip* provide an opportunity to gather data from several Flows into one and vice versa.
- *Actions* provide an ability to query and modify state of robot's input and output devices.
 - *Sensor* continuously emits data from specified sensor, e.g. infrared, light, etc.
 - *Servo*, *Motors* process received data and send impulses to robot actuators.
 - *Encoders* block sets the motors tacho limit when receiving data and continuously emits encoder values into output flows.
 - *SendMessage*, *ReceiveMessage* responsible for the coordination of a group of robots.
 - *Say*, *PlayTone*, *LED* responsible for managing speakers and LED lights.
 - *RemoveFile*, *WriteToFile*, *ReadFile* implement working with file system.

- *InitCamera*, *DetectByVideo*, *StreamingNode* wrap some algorithms of computer vision.
- *PortBlock* provides an ability to write low-level to some port of the robot.
- *SystemCall* responsible for the command execution by command line interpreter, e.g. token "reboot" will reboot robot.
- *Gamepad* reads data from the operator's control device, e.g. gamepad, and emits it.

These blocks are enough to express a pretty wide range of the robotic controllers of varying complexity. If several blocks emitting data from one input device are met only one of them is active. That detail distinguishes our tool from other implementing data flow paradigm, for details see section 6. For example, figure 1 shows diagram with *Motors*, *ConstValue*, *Encoders*, *Flows* where *Encoders* block is presented twice. When interpretation started *ConstValue* emits data to *Motors* and *Encoders* (a) emits a value of a tacho counter. When block *Encoders* (b) receives some data and thus nullifies encoder value, at that moment *Encoders* (a) stops emitting tokens.

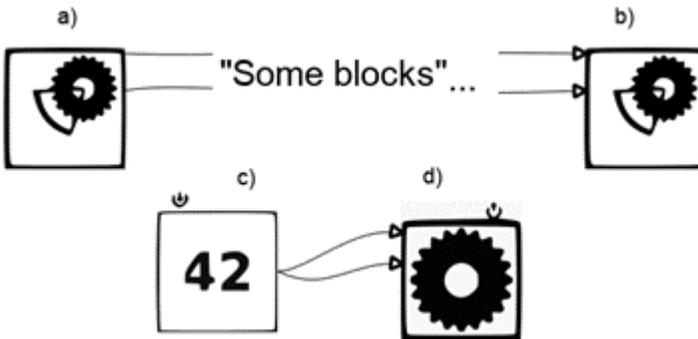


Fig. 1. Block with many representations but only one of them can be active. a,b – Encoders, c – ConstValue, d – Motors.

shows diagram with *Motors*, *ConstValue*, *Encoders*, *Flows* where *Encoders* block is presented twice. When interpretation started *ConstValue* emits data to *Motors* and *Encoders* (a) emits a value of a tacho counter. When block *Encoders* (b) receives some data and thus nullifies encoder value, at that moment *Encoders* (a) stops emitting tokens.

One important detail about our language is that it explicitly supports control flow model, that is important for educational goals. On figure 1 *ConstValue* and *Motors* have incoming and outgoing "arrows", which are used to connect control flow data. For example *Motors* block emits data to control flow channel when handle incoming data and *ConstValue* emits its value when receives control flow token.

Flows may be pinned to a block on left, right and bottom side, which are highlighted when user edits block (see Fig. 2). Also block may contain text fields, e.g. on figure 2 user entered textual condition.

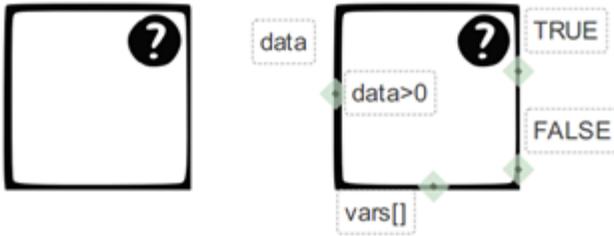


Fig. 2. Showing and editing of block.

5. Example

Figures 3, 4 show simple PD-regulator which keeps robot on a certain distance from a wall using infrared sensor.

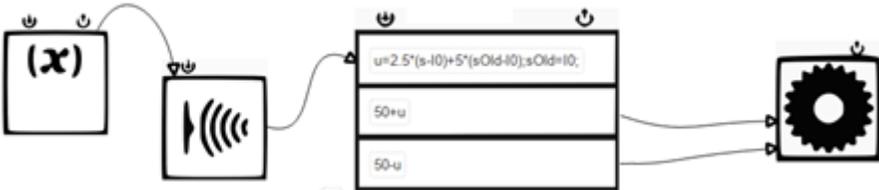


Fig. 3. Controller for the wall following.

Global variable is used for storing old sensor values. Expressions in *Function* block are calculated in upside-down order, results of previous expressions are available on lower levels. Each level emits resulting token into a corresponding flow, in our example two flows are connected directly to motors control block.

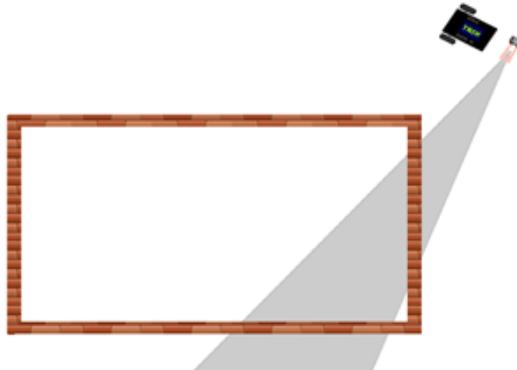


Fig. 4. Simulation process of the wall following.

Let's describe more complex robotic controller. We have the robot equipped with two power motors and two frontal infrared sensors positioned at an angle of 30 degrees on either side of the longitudinal line of symmetry of the robot. Let's consider the robot control system that manages robot wandering in space and avoiding frontal collisions. But at the same time it allows manual control with gamepad. We divide the problem into three levels responsibility using Subsumption Architecture. The first will be responsible for aimless movement of the robot. The second is responsible for collision avoidance: if the robot is too close to a collision, it must avoid the obstacles preventing robot wandering. The third will be responsible for maintenance of the user queries, the user obtains a full control, the previous levels are suppressed.

Figure 5 shows this decomposition. Each level represented as *Subprogram* and emits pulses to actuators. Execution begins with the launch of all levels concurrently. Robot wanders aimlessly. If the robot is close to the collision, the Collision avoidance level suppresses the flow with data emitted by Wandering level. If the user starts to manipulate with the gamepad, the data sent suppress levels described above.

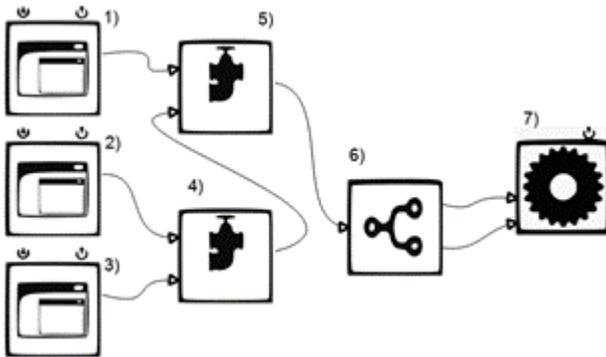


Fig. 5. Controller code with three competencies level. 1 – Human control level. 2 – Collision avoidance level. 3 – Wandering level. 4 – Suppressor block for levels 2,3. 5 – Suppressor block for levels 1 and 2,3. 6 – Unzip block. 7 – Motors block.

Each level is the simple robot controller without direct connection to actuators. Wandering (first level) continuously generates random number for each robot actuator, and sends its outside as array (see Fig. 6). The execution of this level starts with *InPort* which emits data to activate two *RandomValue* blocks. Each *RandomValue* generate random number and emits it to *Wait* block which after some predefined delay sends it to *Zip* block which produces an array storing output values.

The second level is needed to prevent collisions (see Fig. 7). It continuously gathers data by *Zip* from two infrared *Sensors* and checks if collision threatens (continuously after some delay by *DelayAndFilter*). If the collision can occur values sent for actuators to evade obstacles are calculated by *Function*. *Function* block emits it to *Zip* block, which produces an array storing output values.

The third level is responsible for gamepad control (see Fig. 8). *Gamepad* emits tokens describing current joystick and buttons state. For simplicity, we assume that pressing any button on gamepad will terminate the robot control program (by *FinalBlock*). The tokens are converted from the *Gamepad* to the array of pulses for actuators by *Function* block, which emits it through *OutPort* block.

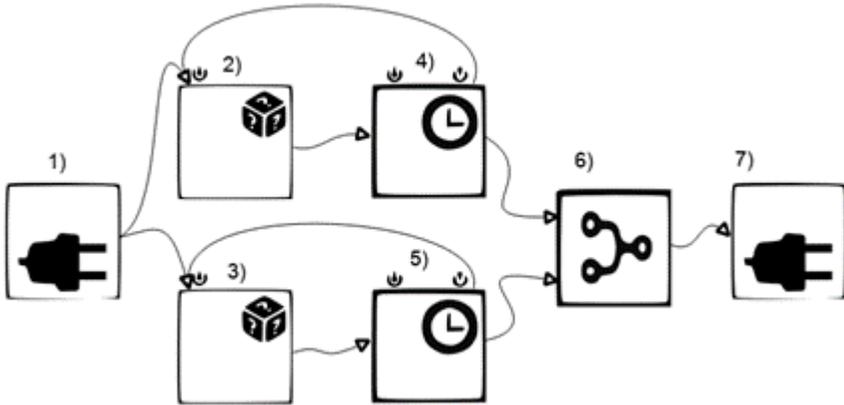


Fig. 6. *Walking*. 1 – InPort block. 2,3 – RandomValue blocks. 4,5 – Wait blocks. 6 – Zip block. 7 – OutPort block.

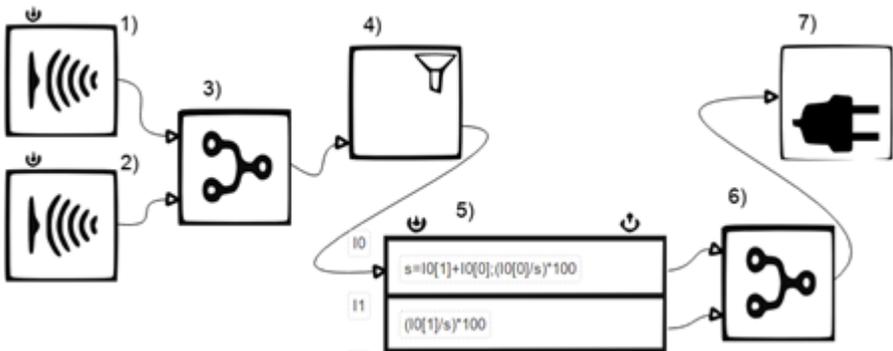


Fig. 7. *Collision avoidance*. 1,2 – Sensor blocks. 3,6 – Zip block. 4 – DelayAndFilter block. 5 – Function block. 7 – OutPort block. Human control. 1 – Gamepad block. 2 – FinalBlock. 3 – Function block. 4 – OutPort block.

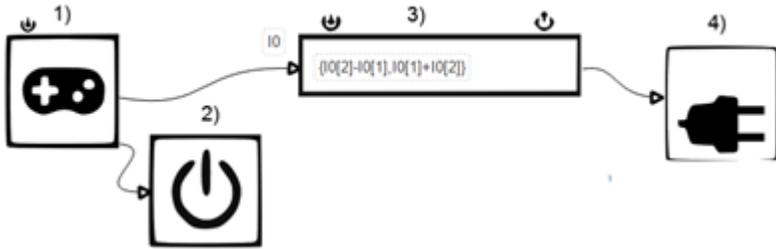


Fig. 8. The third level.

6. Implementation

The system is implemented as two plugins for TRIK Studio. The first one describes the visual language and provides visual editor for our system. It contains the meta-model of dataflow visual language and entirely generated by QReal DSM-platform. Plugged into TRIK Studio this module provides fully operational visual editor with all advantages of TRIK Studio control flow editor like modern-looking user interface, ability to create elements with mouse gestures, different appearances of links and so on. The time spent on the development of this plugin (not considering discussing and designing the prototype of visual language on paper) roughly equals three man-days. The benefit on exploiting the DSM-approach is obvious, the development of the similar editor from scratch would have been taken vastly more time.

The second plugin contains implementation of dataflow diagrams interpreter. Interpreter will transform given program, which is drawn in editor (provided by first plugin) into a sequence of the commands sent to a target robot (see Fig. 9). The target robot can be one of the supported in TRIK Studio infrastructure: Lego NXT or EV3 robot, TRIK robot, TRIK Studio 2D simulator or V-REP 3D simulator [20]. Commands are sent via high-level TRIK Studio devices API, a part of it presented at Fig. 10.

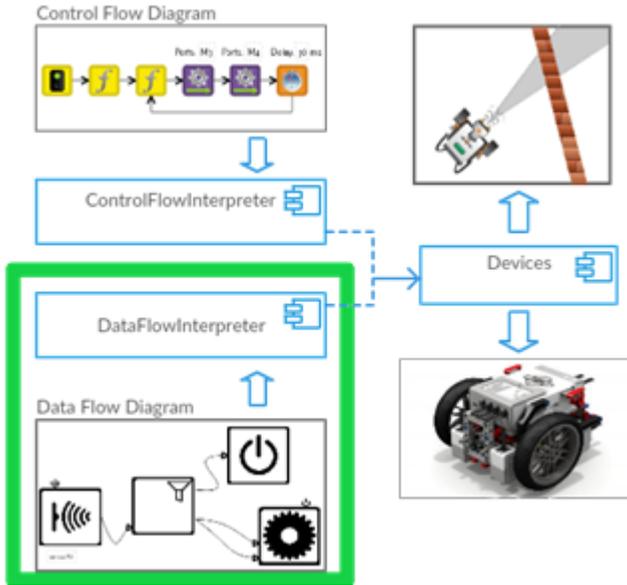


Fig. 9. The general architecture of the system.

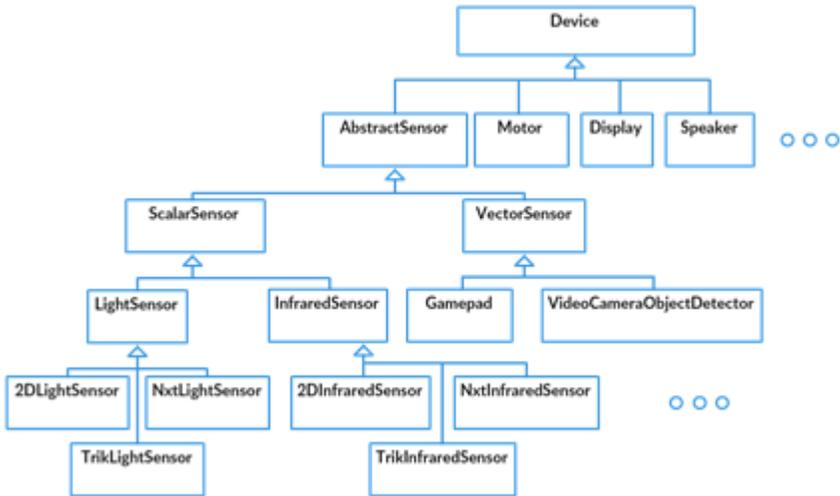


Fig. 10. Partial architecture of devices used in dataflow interpreter.

The general architecture of interpreter plugin is presented at Fig. 11. Interpreter traverses given dataflow diagram, validates and prepares it for interpretation process.

For each visited dataflow block implementation object is instantiated. Implementation objects are written in C++. Instantiation is performed by corresponding factory object. Implementation objects are then subscribed each to other like they are connected by flows on diagram, *publish-subscribe* pattern is used here. The set of initial blocks is determined next, those are blocks without incoming flows. After all that done preparation phase is complete and diagram starts being interpreted.

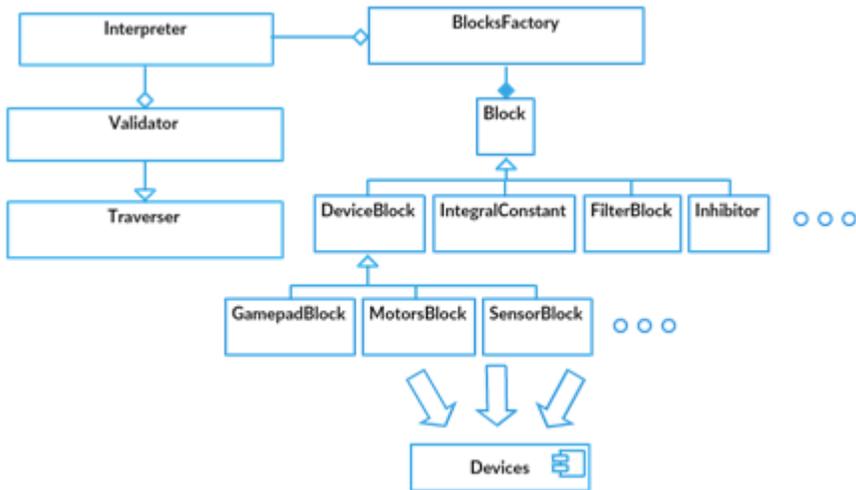


Fig. 11. The general architecture of dataflow interpreter plugin.

Interpretation process is not as straightforward as in most asynchronous dataflow environments. Usually components of dataflow diagram are executed concurrently, on different threads, processes or even machines (that is actively exploited, for example, by Microsoft Robotics Developer Studio where dataflow diagram is deployed into a number of web-services). That is a pretty convenient way to invoke dataflow diagrams on a powerful hardware, but not a case when we talk about embedded devices. In our case we deal exactly with embedded devices (Lego NXT, EV3, TRIK, Arduino controllers), so we propose here another way of executing dataflow diagrams. The main idea is to introduce global message queue and event loop for messages processing. When token is published by some block it is enqueued into messages queue and waits for its turn to be delivered to subscribers (Fig. 12). In fact thus we *flatten* the execution, convert concurrent way of dataflow interpretation to a pseudo-concurrent one where we schedule invocation order on our own. It must be noted that this mechanism is similar to events propagation system of Qt framework. That is actively exploited in our implementation, where message processing is completely performed by *QEventLoop* class and tokens delivering is done by Qt signal/slot system in *QueuedConnection* mode.

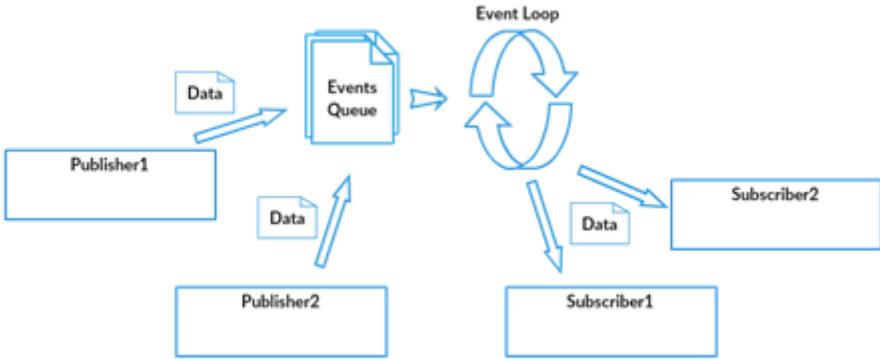


Fig. 12. Proposed mechanism of pseudo-concurrent dataflow interpretation.

Flat execution of dataflow diagram poses a number of small problems, one of them will be discussed here. Input device blocks (for example blocks publishing tokens from ultrasonic sensors) are constantly emitting tokens to subscribers. Subscribers transmit tokens to a next one (possibly in modified state) and so on. Thus there appears a chain of data processing. In our language that chain can activate control flow ports of blocks "reviving" them, so the control flow model is implicitly supported in our language (this is important in educational reasons). If later in this chain same input device block will be met then execution will come in a counter-intuitive way. Such conflicts are ruled out with a simple heuristic that among all the blocks sharing one physical device only one can be active and that is the last activated one. Thus when the execution token comes into some device block it immediately "deactivates" conflicting ones. Other problems like messages balancing (in case when some block "flooding" the whole messages queue) will not be discussed here.

The last thing we should remark here is the presence of *Fork* block in our language that usually is not provided by dataflow languages. Flattened model seems to work well on embedded devices, but sometimes users still need to use concurrent execution (for example for executing layers in Subsumption architecture). For that reason *Fork* block is introduced, it forks the execution into a number of platform-specific execution units (for example *threads* on UNIX or *tasks* on NXT OSEK). This block can be regarded as low-level control of execution process. It should be also marked that this block almost has no sense in interpretation mode (because execution itself is performed on desktop machine with only sending primitive commands to robot), but will be very useful in future works when autonomous mode will be introduced.

7. Conclusion and Discussion

In this work, we presented the prototype of dataflow language for programming different robotic kits (LEGO MINDSTORMS NXT, LEGO MINDSTORMS EV3, TRIK). The system provides ability to interpret diagrams on 2D- an 3D-simulators and real robotic devices. Here, we also propose an approach for executing dataflow

diagrams on embedded devices. The language implicitly supports control flow model for educational purposes. It is also convenient for expressing typical robotic controllers architectures which is demonstrated on example.

The implemented system can be regarded as a platform for future investigations. First of all, autonomous mode of work will be implemented. That will be done through code generation into a number of textual languages already supported by TRIK Studio (NXT OSEK C for Lego, bytecode for EV3, JavaScript, F# [21] and Kotlin for TRIK). We are also interested in academical research. First of all a formal semantics of our language should be expressed for applying various formal methods of program analysis. Another branch of research will be directed into a DSM-branch, here we want to consider an ability of dynamic language metamodel generation from specifications of available modules of robotics middleware (like ROS [22] or Player [23]).

References

- [1]. Banyasad, O. (2000). A Visual Programming Environment for Autonomous Robots.
- [2]. Simpson, J., Jacobsen, C. L., & Jadud, M. C. (2006). Mobile robot control. *Communicating Process Architectures*, 225.
- [3]. Simpson, J., & Jacobsen, C. L. (2008, September). Visual Process-Oriented Programming for Robotics. In *CPA* (pp. 365-380).
- [4]. Posso, J. C., Sampson, A. T., Simpson, J., & Timmis, J. (2011). Process-Oriented Subsumption Architectures in Swarm Robotic Systems. In *CPA* (pp. 303-316).
- [5]. Diprose, J. P., MacDonald, B. A., & Hosking, J. G. (2011, September). Ruru: A spatial and interactive visual programming language for novice robot programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on* (pp. 25-32). IEEE.
- [6]. Johnston, W. M., Hanna, J. R., & Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1), 1-34.
- [7]. A.S. Kuzenkova, A.O. Deripaska, K.S. Taran, A.V. Podkopaev, Y.V. Litvinov, T.A. Bryksin. [Tools for fast development of domain-specific solutions in metaCASE-platform Qreal] St. Petersburg State Polytechnical University Journal, p. 142, 2011 (in Russian).
- [8]. Kuzenkova A., Deripaska A., Bryksin T., Litvinov Y., Polyakov V. QReal DSM platform- An Environment for Creation of Specific Visual IDEs. In *ENASE* (pp. 205-211) 2013.
- [9]. Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1), 14-23.
- [10]. Proetzsch, Martin, Tobias Luksch, and Karsten Berns. "The behaviour-based control architecture iB2C for complex robotic systems." *KI 2007: Advances in Artificial Intelligence*. Springer Berlin Heidelberg, 2007. 494-497.
- [11]. Erwin, B., Cyr, M., & Rogers, C. (2000). Lego engineer and robolab: Teaching engineering with labview from kindergarten to graduate school. *International Journal of Engineering Education*, 16(3), 181-192.
- [12]. Gomez-de-Gabriel, J. M., Mandow, A., Fernandez-Lozano, J., & García-Cerezo, A. (2011). Using LEGO NXT mobile robots with LabVIEW for undergraduate courses on mechatronics. *Education, IEEE Transactions on*, 54(1), 41-47.

- [13]. Kuzenkova, A., Deripaska, A., Bryksin, T., Litvinov, Y., & Polyakov, V. (2013). QReal DSM platform-An Environment for Creation of Specific Visual IDEs. In *ENASE* (pp. 205-211)
- [14]. Kim, S. H., & Jeon, J. W. (2007, October). Programming LEGO Mindstorms NXT with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on* (pp. 2468-2472). IEEE.
- [15]. Connell, Jonathan H. *A colony architecture for an artificial creature*. No. AI-TR-1151. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1989.
- [16]. Arkin, Ronald C. Motor schema based navigation for a mobile robot: An approach to programming by behavior. *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*. Vol. 4. IEEE, 1987.
- [17]. Rosenblatt, Julio K. DAMN: A distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (1997): 339-360.
- [18]. Simpson, Jonathan, and Carl G. Ritson. *Toward Process Architectures for Behavioural Robotics*. CPA. 2009.
- [19]. D.V. Koznov. [Fundamentals of Visual Modeling] *Binom. Laboratorija znanij, Internet-universitet informacionnyh tehnologij*. 2008 (in Russian).
- [20]. Rohmer, Eric, Surya PN Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 2013.
- [21]. Kirsanov, Alexander, Iakov Kirilenko, and Kirill Melentyev. Robotics reactive programming with F#/Mono. *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. ACM, 2014.
- [22]. Quigley, Morgan, et al. ROS: an open-source Robot Operating System. *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.
- [23]. Gerkey, Brian, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th international conference on advanced robotics*. Vol. 1. 2003.

Образовательный визуальный потоковый язык для программирования роботов

Г.А. Зимин <zimin.grigory@gmail.com>

Д.А. Мордвинов <mordvinov.dmitry@gmail.com>

Санкт-Петербургский государственный университет,

математико-механический факультет,

Университетский пр-т. 28, 198504, Россия.

Abstract. Визуальные предметно-ориентированные языки зачастую имеют низкий порог вхождения: даже ученики школ и дошкольных учреждений могут программировать на таких языках, оперируя визуальными моделями. Этот факт нашел широкое применение в образовательной робототехнике, где большинство используемых сред разработки основано на визуальных языках. Данная работа описывает новый потоковый визуальный язык программирования роботов для распространенных встраиваемых робототехнических платформ. Очевидно, что сложные потоковые визуальные языки трудны для пони-

мания. Целью нашей работы было создание инструмента, представляющего собой переходную «ступень» между легковесными образовательными средами программирования, которые обычно предоставляют языки, основанные на модели потока управления, и сложными индустриальными средами, которые, в основном, предоставляют языки, основанные на модели потоков данных. В статье приводится сравнение широко распространенных сред программирования роботов с описанной в работе средой. Также в работе представлен краткий обзор популярных поведенческих архитектур для построения сложных систем управления роботами, таких как архитектура категорий Р. Брукса и «Колония» Д. Коннелла, и приведены идеи их выражения в новом языке программирования. Язык был создан с помощью предметно-ориентированного подхода. Он предоставляет возможность совмещать в себе две модели исполнения: пользователь может запрограммировать как в терминах потоков данных, так и в терминах потока управления. Мы считаем, что это важно в образовательных целях. Программы на нашем языке состоят из множества «блоков» – визуальных представлений процессов трансформации данных, и «связей», которые визуализируют потоки данных между ними. В качестве апробации среды созданы различные по сложности программы управления роботами.

Keywords: потоковые языки, потоки данных, визуальное программирование, образовательная робототехника, предметно-ориентированное моделирование, поведенческие архитектуры.

DOI: 10.15514/ISPRAS-2016-28(2)-3

Для цитирования: Зимин Г.А., Мордвинов Д.А. Образовательный визуальный потоковый язык для программирования роботов. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 45-62 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-3

Список литературы

- [1]. Banyasad, O. (2000). A Visual Programming Environment for Autonomous Robots.
- [2]. Simpson, J., Jacobsen, C. L., & Jadud, M. C. (2006). Mobile robot control. *Communicating Process Architectures*, 225.
- [3]. Simpson, J., & Jacobsen, C. L. (2008, September). Visual Process-Oriented Programming for Robotics. In *CPA* (pp. 365-380).
- [4]. Posso, J. C., Sampson, A. T., Simpson, J., & Timmis, J. (2011). Process-Oriented Subsumption Architectures in Swarm Robotic Systems. In *CPA* (pp. 303-316).
- [5]. Diprose, J. P., MacDonald, B. A., & Hosking, J. G. (2011, September). Ruru: A spatial and interactive visual programming language for novice robot programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on* (pp. 25-32). IEEE.
- [6]. Johnston, W. M., Hanna, J. R., & Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1), 1-34.
- [7]. Кузенкова А.С., Дерипаска А.О., Таран К.С., Подкопаев А.В., Литвинов Ю.В., Брыксин Т.А. Средства быстрой разработки предметно-ориентированных решений в metaCASE-средстве QReal. *Научно-технические ведомости СПбГПУ*, 142
- [8]. Kuzenkova A., Deripaska A., Bryksin T., Litvinov Y., Polyakov V. QReal DSM platform- An Environment for Creation of Specific Visual IDEs. In *ENASE* (pp. 205-211) 2013.
- [9]. Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1), 14-23.

- [10]. Proetzsch, Martin, Tobias Luksch, and Karsten Berns. "The behaviour-based control architecture iB2C for complex robotic systems." *KI 2007: Advances in Artificial Intelligence*. Springer Berlin Heidelberg, 2007. 494-497.
- [11]. Erwin, B., Cyr, M., & Rogers, C. (2000). Lego engineer and robolab: Teaching engineering with labview from kindergarten to graduate school. *International Journal of Engineering Education*, 16(3), 181-192.
- [12]. Gomez-de-Gabriel, J. M., Mandow, A., Fernandez-Lozano, J., & Garcia-Cerezo, A. (2011). Using LEGO NXT mobile robots with LabVIEW for undergraduate courses on mechatronics. *Education, IEEE Transactions on*, 54(1), 41-47.
- [13]. Kuzenkova, A., Deripaska, A., Bryksin, T., Litvinov, Y., & Polyakov, V. (2013). QReal DSM platform-An Environment for Creation of Specific Visual IDEs. In *ENASE* (pp. 205-211)
- [14]. Kim, S. H., & Jeon, J. W. (2007, October). Programming LEGO Mindstorms NXT with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on* (pp. 2468-2472). IEEE.
- [15]. Connell, Jonathan H. *A colony architecture for an artificial creature*. No. AI-TR-1151. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1989.
- [16]. Arkin, Ronald C. Motor schema based navigation for a mobile robot: An approach to programming by behavior. *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*. Vol. 4. IEEE, 1987.
- [17]. Rosenblatt, Julio K. DAMN: A distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (1997): 339-360.
- [18]. Simpson, Jonathan, and Carl G. Ritson. *Toward Process Architectures for Behavioural Robotics*. CPA. 2009.
- [19]. Кознов, Дмитрий Владимирович. *Основы визуального моделирования*. М.: Изд-во Интернет университета информационных технологий, ИНТУИТ.ру, БИНОМ, Лаборатория знаний. 2008.
- [20]. Rohmer, Eric, Surya PN Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 2013.
- [21]. Kirsanov, Alexander, Iakov Kirilenko, and Kirill Melentyev. Robotics reactive programming with F#/Mono. *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. ACM, 2014.
- [22]. Quigley, Morgan, et al. ROS: an open-source Robot Operating System. *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.
- [23]. Gerkey, Brian, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th international conference on advanced robotics*. Vol. 1. 2003.

Context-Based Model for Concern Markup of a Source Code

¹ M.S. Malevanny *<mmxforever@mail.ru>*

² S.S. Mikhalkovich *<miks@sfedu.ru>*

¹ Don State Technical University,
162, Socialisticheskaja st., Rostov-on-Don, 344022, Russia

² Southern Federal University,
8A, Mil'chakova, Rostov-on-Don, 344090, Russia

Abstract. In this paper we describe our approach to representing concerns in an interface of an IDE to make navigation across crosscutting concerns faster and easier. Concerns are represented as a tree of an arbitrary structure, each node of the tree can be bound to a fragment of code. It allows one to quickly locate fragments in the source code and makes switching between software development tasks easier. We describe a model which specifies data structures used to store the information about these code fragments and algorithms used to find the code fragment in original or modified source code. The model describes the information about code fragments as a set of contexts. Another important feature of the model is language independency. The model supports different programming, mark-up, DSL-languages and any structured text, such as a documentation. Main goal is to keep concern tree consistent with evolving source code. Search algorithm is designed to work with a modified source code, where the code fragment may change. The model is implemented as a tool, which supports different programming languages and integrates into different editors and integrated development environments. Source code analysis is performed by a set of lightweight parsers. In case of significant changes if the code fragment may be not found automatically the tool helps a programmer to find one by suggesting possible places in the source code based on the stored information.

Ключевые слова: Concerns; Separation of Concerns; Program Comprehension; Integrated Development Environments

DOI: 10.15514/ISPRAS-2016-28(2)-4

For citation: Malevanny M.S., Mikhalkovich S.S. Context-Based Model for Concern Markup of a Source Code. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 63-78. DOI: 10.15514/ISPRAS-2016-28(2)-4

1. Introduction

During software development and maintenance developers usually work with several code fragments related to their current task or *concern*. Most concerns are crosscutting[1], which means that code related to it tends to be scattered across a number of files, or different places within one file. Repeated navigation between these code fragments requires a considerable time and effort[2]. These fragments form a "working set". Switching to another task requires investigating the source code and locating all fragments relevant to the task. Returning to the task after working on another one may take significant time.

A number of techniques address to this problem, such as Aspect-Oriented Programming[3], Feature-Oriented Programming[4][5][6], Delta-Oriented Programming[7], Subject-Oriented Programming[8] and others. Most of them are intended to explicitly separate concerns into a number of modules and provide different mechanisms of composition of these modules. It often requires significant changes in the source code to use one of these techniques.

Other methods provide support of concerns by adding new tools to an IDE, such as virtual files[9][10][11] colour markup[12] without changing the source code. These tools are often designed for only one IDE and depend on its infrastructure and thus are limited to only few languages, supported by the IDE. Another common limitation is low tolerance of changes in the source code. When the code is modified some code fragments may be lost.

Many of these tools are limited to only one programming language, while large software projects are often developed in several languages, including DSL-languages and markup languages, and code fragments related to a concern may be scattered across files in different languages.

We are currently developing an approach[13] intended to mitigate the problems of navigation across the code and switching between different tasks. The approach doesn't require any changes to the source code. It defines a notion of a concern as a tree-like structure, consisting of sub-concerns and code fragments. Similarly to ConcernMapper[14] it displays a concern tree in an IDE as a toolbox and allows one to quickly locate fragments in the source code. Unlike most other tools it and may be used in different IDEs and allows one to work with code in different languages. Another goal is robustness, which allows working with the code being actively developed keeping concern tree consistent with the code.

2. Model

We present a model our approach is based on. It uses lightweight parsers to analyze source text and to create parse tree, which will be used later. The model defines the data being stored in the concern tree. And finally, it defines algorithms to search the code fragments in a modified source code.

2.1. Lightweight parsing

The model is common for different languages. To minimize dependency on IDE infrastructure we use lightweight parsing to analyze the source code and build parse tree, which contains information about significant entities in the code. Lightweight parsers can recover from errors and produce parse tree for code with errors or incomplete code, which is important while the code is being modified.

Adding support of another programming language requires development of a lightweight parser for this language. Lightweight parsers are simple and easy to develop using our DSL language LightParse. For most languages it takes only about 10-30 lines of text to express important language features and produce a lightweight parser. The parser is able to analyze source code and build a simple parse tree with only nodes, corresponding to these language features. Any other parts of source code (e.g. method bodies) are skipped. Saving information about an entity in the source code is available for all entities returned by the parser. The more detailed parse tree the parser produces — the more entities can be saved in the concern tree, however development of the parser may require more time.

Lightweight parsers produce a lightweight parse tree. Nodes of the tree have type, name and location in the source code. Node name consists of several tokens; one of them may be marked as important. For example method name consist not only of one identifier — name, which is marked as important, but also includes parameter names and types, access modifiers, return value type and so on.

An example of a lightweight parser is given in subsection 2.3. Lightweight parsing is described in our paper[15] in more detail. The paper provides examples of lightweight parser grammar. More examples may be found in GitHub repository of the tool¹ (files with extension ".lp").

2.2. Data

The approach is not limited to any specific programming language and therefore the information in the concern tree should be sufficient to support different languages. Also, we assume that the source code may change and the concern tree should possibly store some redundant data to find the code fragment after the code has changed.

Each code fragment in the concern tree stores next 5 items:

- Type.
- Header context. It may include entity name and any number of additional tokens.
- Outer context. It includes names and types of all parent nodes from the immediate parent to the root of the parse tree.

¹ <https://github.com/MikhailoMMX/AspectMarkup/tree/master/Parsers>

- Horizontal context. It consists of two subsets of names and types of preceding and subsequent sibling nodes.
- Inner context. It includes a subset of subnodes of current code fragment.

These items form *Context* of the node. Except for type, any other item may be empty.

Type is used to filter non-relevant nodes when searching for the code fragments. If a concern tree item is bound to a method only methods should be considered, other nodes, e.g. classes, fields may be ignored.

Header context represents entity name and several additional tokens. In the following C# code example

```
public void visit(TreeNode t)
public void visit(Expression e)
```

both methods are named `visit`, but have different parameter types and names. Header context makes possible distinguishing overloaded methods and other entities with same names. Header context is represented as a list of tokens, where one token may be marked as important and it is considered as the **name** of the entity. Header context as well as name may be empty.

Outer context stores enclosing entities for the code fragment, such as classes and namespaces. In many languages there may be variables and methods with exactly same names, but defined in different classes or namespaces. An example is the implementation of one interface by different classes. In this case it's necessary to save not only the name of the entity, but also the name of enclosing entities. In the following example

```
namespace N
{
    class C1 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
    class C2 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
}
```

both methods have same names and header contexts, but are defined in different classes. For example, outer context for the first method will include name and type of class `C1` and namespace `N`. Outer context for an entity is a list of Header contexts and Types for each enclosing entity starting from the immediate parent to the topmost entity in the source file.

Header context and outer context are sufficient for most programming languages, where all names are unique, at least in a certain scope. However, there is another class of languages, such as Yacc (grammar definition language), or markup languages, such

as XML. In these languages there may be two entities with same name in same scope. Without additional information binding concern tree nodes to such entities is ambiguous. To handle these cases two different kinds of context were added to the model.

Horizontal context keeps nearest neighbors before and after the node. It consists of two sets of pairs (Header context + Type), one for preceding entities and one for subsequent entities. Following example is an excerpt from ANSI C grammar[16]:

```
selection_statement
    : IF '(' expression ')'
      statement ELSE statement
    ...
    ;
```

There are two occurrences of `statement` in a subrule of a rule `selection_statement`. Their horizontal contexts are different: token `ELSE` and another non-terminal `statement` are located *after* the first occurrence of `statement` and *before* the second one. This information makes it possible to distinguish similar entities by their location among their neighbor entities.

It could have been achieved by saving an *index* of the entity. For example, first `statement` gets index 1 and second one gets index 2, but saving indexes is less tolerant to changes in the source text. Adding or removing entities in the beginning of a subrule invalidates indexes of all subsequent entities, but has almost no effect on horizontal context.

Inner context is intended to store subnodes of an entity. In some cases an entity can have empty name and may be distinguished from another one only by its content. For example, variable declaration sections in such language as PascalABC.NET[17] are unnamed, but they have different variables:

```
var
    X, Y : Double;
var
    Name, Address : string;
    Age : integer;
```

In this example, there are two sections. It may be necessary to bind a concern tree node to a whole section. Horizontal context cannot be reliable in this case because it keeps only type and name, which is empty — changing their order will lead to incorrect result of the search. Inner context is a set of Header contexts and Types for some subnodes. In the example above saving only one subnode (i.e. variable name) is enough to distinguish these sections. Amount of subnodes to be saved as the inner context may vary.

Inner context for leaves of a parse tree may contain lines of source code. This may apply if the entity spans multiple lines in the source code (e.g. methods).

Inner and horizontal contexts may be empty if the entity has no neighbor nodes or subnodes. Otherwise, it may be not necessary to store all neighbors or subnodes. Usually, a small amount of unique nodes is enough to distinguish similar entities. In many languages horizontal and inner contexts are a redundant information. However, using horizontal and inner contexts increases reliability of the search even with a code on a programming languages that normally don't need these two kinds of context. When the code has changed this information may be useful.

Let T is a parse tree node. $Context(T) = (Name_T, Type_T, N_T, O_T, H_T, I_T)$ is a tuple of node Name, Type and its Header, Outer, Horizontal and Inner contexts described above. When a binding to the node T is added to the concern tree, $Context(T)$ is saved.

Name and *Type* are strings. Header context $N_T = (S_1, S_2, \dots, S_n)$ is a list of strings. Outer context $O_T = ((N_1, T_1), (N_2, T_2), \dots, (N_n, T_n))$ is a list of pairs, where N_i is a Header Context and T_i is a type of an enclosing entity. Inner Context $I_T = \{(N_i, T_i)\}$ is a set of pairs: header contexts and type of an entity. And Horizontal context $H_T = (\{(N_i, T_i)\}, \{(N_j, T_j)\})$ is a pair of sets of header contexts and types of entities.

2.3. Additional markup

Our approach is focused on finding code fragments without using any modifications of source code. Additional markup, such as comments with special keywords clutters the code if used frequently. However, in some cases it might be feasible to mark some places in the code with comments. First scenario is binding to code fragments in a file, which contains a lot of very similar entities. Some XML files may have such structure. In this example:



There are two nodes C , with equal contexts. Despite being subnodes of different parent nodes, their outer contexts are equal, because both parent nodes have same name. To handle this case it might require to save horizontal context for each parent node, which is not implemented in the model.

Another scenario is binding to code fragments in frequently modified code, where entities may undergo significant changes.

This kind of markup requires a lightweight parser, which builds parse tree based on comments. Comments may define points and spans in the source code.

```
// ConcernBegin Serialization
```

```
...
```

```
// Concern SomePoint
```

...

```
// ConcernEnd Serialization
```

The code above shows an example of a markup with comments. *Concern* *Serialization* is a span and *SomePoint* is a single line marked with a comment.

Lightweight parser for this markup is simple and may work with source code in many languages. The only modification it may require to adapt the parser to a different language is changing comment start symbols. Here is a grammar of the lightweight parser written in *LightParse*:

```
%Extension "*"
Token Tk [[:IsLetterOrDigit:]]*|
        [[:IsPunctuation:]][:IsSymbol:]]
Token NewLine \r|\n|\r\n
Rule Program :   [#Comment|Other]*
Rule Comment :  "//" @CTk? @Tk+
Rule CTk:       @"ConcernBegin"
                | @"ConcernEnd"
                | @"Concern"
Rule Other :    Tk
                | NewLine
                | #error
```

3. Algorithms

There are two aspects of working with the concern tree: adding a node to the tree and searching the code fragment, related to the node. Both actions require a parse tree, which is provided by a lightweight parser. In the following part of the section we take into consideration only a subset of parse tree nodes whose type is equal to the type of an entity being saved or the one being searched. Given the T is a parse tree node to be saved in the concern tree, we consider a set $Tree = \{T_i \mid Type_{T_i} = Type_T\}$.

Next step is calculating a distance between T and every item $T_i \in Tree$.

3.1. Calculating distances

Distance two tree nodes is a vector of distances between each component of a context for a given pair of nodes.

$$Distance(T, T_i) = \overline{D}_i = (DName, DType, DN, DO, DH, DI)$$

where:

$$DType = \begin{cases} 1, & \text{if } Type_T \neq Type_{T_i} \\ 0, & \text{if } Type_T = Type_{T_i} \end{cases}$$

Distance for other part of context is calculated with functions $LDistance$ and $SDistance$, described further below:

- $DName = LDistance(Name_T, Name_{T_i})$
- $DN = LDistance(N_T, N_{T_i})$
- $DO = LDistance(O_T, O_{T_i})$
- $DH = LDistance(H_T, H_{T_i})$
- $DI = LDistance(I_T, I_{T_i})$

Zero in each component of a vector \bar{D} means equality of corresponding parts of contexts of T and T_i . The higher these values — the less similar two parts of contexts are.

Calculating the distance for Name, Header context and outer context is based on a Levenshtein metric [18]. Levenshtein distance for two strings reflects the number of edits (insertions, deletions and substitutions) required to change one string into the other. *Names* of entities are just strings, however Header contexts are lists of strings. Levenshtein distance in this case is calculated similarly, but each edit is a deletion, insertion or substitution of a token. Weight of a substitution in this case depends on similarity of tokens and ranges between 0 (tokens are equal) to 2 (weight of insertion + weight of deletion) if two tokens have maximum possible edit distance between them. Distance between two outer contexts is calculated similarly. Each item of an outer context is a pair (Type, Header Context) and the weight of substitution depends on distance between to header contexts.

Calculation of edit distance is performed by overloaded functions *LDistance*.

Horizontal and inner contexts contain a subset of nodes and the distance is calculated as a number of subnodes present in T and absent in T_i .

Calculation of distance between sets is performed by function *SDistance*:

$$SDistance(I, I_i) = |I \setminus I_i|$$
$$SDistance(H, H_i) = |H_L \setminus H_{i_L}| + |H_R \setminus H_{i_R}|$$

3.2. Saving information

Name, Type, Header and Outer contexts are required parts of a context and are saved always. Inner and Horizontal contexts are optional in some cases. To determine should they be saved or not and how much nodes they should contain we are looking for other nodes in the parse tree with similar Header Contexts.

Given the T is the parse tree node to be saved we define two sets of parse tree nodes:

$$TreeL = \{T_i \mid O_{T_i} = O_T\}$$

$$TreeG = \{T_i \mid O_{T_i} \neq O_T\}$$

In other words, one subset consists of all neighbour nodes for T (**L**ocal scope) and other one - of all other nodes (**G**lobal scope).

After that, we calculate two values: *NearL* and *NearG*.

$$\begin{aligned} \text{NearL} &= \text{LDistance}(N_T, N_{T_i}) : T_i \in \text{TreeL}; \forall T_j \in \text{TreeL}, \text{LDistance}(N_T, N_{T_j}) \\ &\geq \text{LDistance}(N_T, N_{T_i}) \end{aligned}$$

In other words, we find a distance between header contexts of T and the most similar node *within* the scope of a node T .

$$\begin{aligned} \text{NearG} &= \text{LDistance}(N_T, N_{T_i}) : T_i \in \text{TreeG}; \forall T_j \in \text{TreeG}, \text{LDistance}(N_T, N_{T_j}) \\ &\geq \text{LDistance}(N_T, N_{T_i}) \end{aligned}$$

similar to NearL , but *outside* of the scope of T .

When $\text{NearG} > 0$, $\text{NearL} > 0$ there are no other nodes with same header. In this case Inner and Horizontal contexts are optional and may be omitted. If $\text{NearG} = 0$, $\text{NearL} > 0$ there are similar nodes with different outer context. Again, saving Inner and Horizontal contexts is optional, but may improve search results if the source file is modified. In case of $\text{NearL} = 0$ saving inner and horizontal context is required.

The values NearL and NearG are saved within the concern tree and will be used for the search.

3.3. Searching

A node in the concern tree keeps Context of some node T .

$$\text{Context}(T) = (\text{Name}_T, \text{Type}_T, N_T, O_T, H_T, I_T)$$

After some modifications were applied to the source file, target node may change as well. In some cases target node may be absent in the parse tree, if the code fragment related to the concern was removed. We do not address this case in our research and the tool is designed to always try to find target node or suggest a list of most similar entities.

The search begins with parsing a file and calculating edit distance $\overline{D}_i = \text{Distance}(T, T_i) \forall T_i \in \text{Tree}$

Next step — checking if there is only one node in the tree, which is similar to the target node and therefore considered as the result of the search. It depends on values NearG and NearL .

If $\text{NearL} > 0$, then there was only one entity in the source file with Header context H_T . In this case if there is only one node T_i with similar Header context in the tree — it is returned as the result:

$$\begin{aligned} \text{Result} &= T_i \in \text{Tree} : \text{LDistance}(N_T, N_{T_i}) < \frac{\text{Min}(\text{NearG}, \text{NearL})}{2}; \\ &\forall T_j \neq T_i \text{LDistance}(N_T, N_{T_j}) > \frac{\text{Min}(\text{NearG}, \text{NearL})}{2} \end{aligned}$$

If $\text{NearL} = 0$, then there were other entities in the source tree, but only in the same scope as T . In addition to the condition above we can return T_i if it has minimal distance for Header, Inner and Horizontal contexts among all other nodes:

$$\begin{aligned} \text{Result} = T_i \in \text{Tree} : \forall T_j \neq T_i : & \text{LDistance}(N_T, N_{T_i}) \leq \text{LDistance}(N_T, N_{T_j}) \\ & \text{SDistance}(I_T, I_{T_i}) \ll \text{SDistance}(I_T, I_{T_j}) \\ & \text{SDistance}(H_T, H_{T_i}) \ll \text{SDistance}(H_T, H_{T_j}) \end{aligned}$$

These conditions are correct if $\text{NearG} > 0$. Otherwise there were other entities in the source file with same Header Context outside of the scope of T . In this case we add requirements $\text{LDistance}(O_T, O_{T_i}) = 0$ and $\text{LDistance}(O_T, O_{T_j}) = 0$ to both conditions.

If there are no exactly one node T_i , which satisfies the requirements above we consider the search result as ambiguous and cannot return only one node as the result. It may occur when the source code was modified significantly, the target entity was changed or removed and there are 0 or 2 or more nodes in the parse tree, similar to the target node. In this case the set of all nodes is sorted according to the product of $\overline{D}_i \cdot \overline{W}$, where vector \overline{W} defines weights of parts of contexts.

3.4. Complexity

Wagner-Fischer algorithm[19] is used to calculate edit distances. It has a time complexity of $O(NM)$ where N and M are lengths of two strings. Calculating edit distance of Header Contexts requires calculating edit distance between two strings at each step. For simplicity, we assume that all tokens and all header contexts have similar length. It gives a time complexity of $O(N^2M^2)$, where N is the length of Header contexts (in tokens) and M is length of tokens.

Calculating edit distance between two Outer Contexts has a time complexity of $O(N^2M^2K^2)$, where K is a length of Outer Context (depth of the parse tree).

In most cases values N , M and K are relatively small. Length of separate tokens usually ranges between 1 and 10–15, longer identifiers are rare. Header Context contains usually not more than 10–15 tokens. Outer context in case of most programming languages contains 1–3 items (e.g. a namespace and a class).

Calculating edit distance is performed for each item in set *Tree*.

Other operations have a time complexity between $O(N)$ (calculating NearG and NearL , finding exact match) and $O(N \log N)$ (sorting), where N is a number of items in set *Tree*.

4. Tool

he tool² based on the model was designed to be easily integrated into different integrated developer environments and text editors, such as Microsoft Visual Studio and Notepad++.

² Available at <https://github.com/MikhailoMMX/AspectMarkup>

4.1. Architecture

The tool is separated into 3 main parts:

- A collection of lightweight parsers and a parser generator. A parser analyzes source files written in a specific language and provides a parse tree which is then used by the core. To make development of new parsers easier a DSL-language `{\em LightParse}` was implemented along with an utility which generates `lex/yacc` and `C\#` code of the parser from an input `LightParse` file.
- Core. It implements the model with algorithms. It loads and runs parsers to get a parse tree when it's necessary for saving or searching for a code fragment. A visual component with user interface ready to be integrated into different IDEs is also implemented.
- A collection of plug-ins for integrated development environments or text editors. Since the tool relies on lightweight parsers rather than on a specific IDE, and the visual part of the tool along with algorithms is provided by the core, the tool can be very easily integrated into different IDEs. A plug-in for an IDE should only display the UI component and implement simple interface, which defines 10 methods, such as getting and setting cursor position, accessing the text of currently open files and event handlers for opening and closing the IDE.

At this moment implemented lightweight parsers include: `C#`, `Lex` and `Yacc`, `Java`, `XML`, `PascalABC.NET` and a parser for our own language `LightParse`. Plug-ins for `Microsoft Visual Studio`, `Notepad++` and `PascalABC.NET`[20] are developed and the tool is also integrated into a grammar editor `Yacc MC`.

4.2. Functionality

The tool adds a concern tree to the interface of a IDE. Concern tree may have arbitrary structure and is created by a developer. Each tree node has title and optional description and subnodes. Description length is not limited. It's displayed as a tooltip and may be edited in a separate window.

Each node may be bound to a fragment of code. In this case the node is marked with an arrow. Double click performs navigation to the code fragment if the code fragment may be identified unambiguously. Otherwise, the tool suggests several most similar code fragments. Each code fragment may be navigated to in one click and if the code fragment is found, double click updates the information in the concern tree, so next navigation will not require any additional actions.

A reverse search is also possible. The tool can find a node in the concern tree by cursor position in a current file. Along with the descriptions for tree nodes it may be used to extract some long comments from the code into the concern tree and still be able to easily find and read them.

There are several scenarios of using the concern tree. First, it may be used to maintain a "working set" of fragments, related to a current task. Concern tree is relatively small and finding the node in the tree may be much faster than finding the code fragment in one of currently open files manually.

Concern tree significantly simplifies re-creating working set when returning to a task. Instead of recalling class and method names, performing cross-reference search it's only necessary to expand a subnode in the concern tree related to the task.

Concern tree is very helpful when a new developer starts working with unfamiliar project. Concern tree resembles a table of contents, it's easy to find concerns in it and each concern contains all code fragments related to it with descriptions. Reading description and navigating across the code helps to understand how the code is organized and how it works.

The functionality, concern tree examples and the tool usage scenarios were presented at CEE-SEC(R) 2015 Conference³.

5. Conclusion

We propose an approach to working with crosscutting concerns. Concerns are organized in a tree-like structure and tree nodes are bound to code fragments scattered across the project. Concern tree is added to the interface of IDE as a toolbox. Concern tree simplifies navigating across scattered fragments and is helpful for investigating and re-investigating a concern. We describe a model our approach is based on. A metrics of distance between entities in a code is defined. A description of data, stored in a concern tree is given. Algorithms of identifying a minimal amount of data to store and searching an entity in a modified source code are provided.

The model is implemented in a tool, which supports different programming languages and integrates into different editors and integrated development environments. It performs either navigation to a saved code fragment if it can be determined precisely, or shows most similar code fragments otherwise. The concern markup tool is used in development of PascalABC.NET and the tool itself.

At this moment some features of the model are not implemented yet, such as horizontal context.

We are currently collecting statistical data and enhancing algorithms to better handle most frequent changes in the source code. Some parameters, such as weights of operations need adjustments.

References

- [1]. M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns" in Proceedings of the First International Workshop on Assessment

³ <http://2015.secr.ru/lang/ru/program/submitted-presentations/aspect-markup-of-a-source-code-for-quick-navigating-a-project>

- of Contemporary Modularization Techniques, ser. ACoM '07. Washington, DC, USA: IEEE Computer Society, 2007, p. 2. DOI: 10.1109/ACOM.2007.4.
- [2]. A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks" *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006. DOI: 10.1109/TSE.2006.116.
- [3]. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ" in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP'01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353. (<http://dl.acm.org/citation.cfm?id=646158.680006>)
- [4]. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The genvoca model of software-system generators" *IEEE Softw.*, vol. 11, no. 5, pp. 89–94, Sep. 1994. DOI: /10.1109/52.311067.
- [5]. D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement" in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197. (<http://dl.acm.org/citation.cfm?id=776816.776839>).
- [6]. S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language independent, automated software composition" in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.
- [7]. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines" in *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91. (<http://dl.acm.org/citation.cfm?id=1885639.1885647>).
- [8]. W. Harrison and H. Ossher, "Subject-oriented programming: A critique of pure objects" in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 411–428. DOI: 10.1145/165854.165932.
- [9]. M. C. Chu-Carroll, J. Wright, and A. T. T. Ying, "Visual separation of concerns through multidimensional program storage" in *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, ser. AOSD '03. New York, NY, USA: ACM, 2003, pp. 188–197. DOI: 10.1145/643603.643623.
- [10]. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. LaViola, Jr., "Code bubbles: A working set-based interface for code understanding and maintenance" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512. DOI: 10.1145/1753326.1753706.
- [11]. S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, "Do we really need to extend syntax for advanced modularity?" in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 95–106. DOI: 10.1145/2162049.2162061.
- [12]. C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines" in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.
- [13]. M. Malevanny and S. Mikhalkovich, [Implementation of support of aspects in integrated development environments], in *Sovremennyye informatsionnyye tekhnologii: tendentsii i*

- perspektivy razvitiya: materialy konferentsii [Modern information technologies: tendencies and perspectives of evolution], 2015, pp. 351–353 (in Russian).
- [14]. M. P. Robillard and F. Weigand-Warr, “Concernmapper: Simple view-based separation of scattered concerns” in Proceedings of the 2005OOPSLA Workshop on Eclipse Technology eXchange, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 65–69. DOI: 10.1145/1117696.1117710.
- [15]. M. Malevanny, [Lightweight parsing and its application in development environment]. Informatizatsiya i svyaz' [Informatization and communication], vol. 3, pp. 89–94, 2015, (in Russian).
- [16]. ANSIC C grammar. (<http://www.quut.com/c/ANSIC-grammar-y.html>)
- [17]. PascalABC.NET. (in Russian). <http://pascalabc.net/>
- [18]. V. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals” *Soviet Physics – Doklady*, vol. 10, no. 8, pp. 707–710, 1965, (in Russian).
- [19]. R. A. Wagner and M. J. Fischer, “The string-to-string correction problem” *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974. DOI: 10.1145/321796.321811.
- [20]. I. V. Bondarev, Y. V. Belyakova, and S. S. Mikhalkovich, [System pascalabc.net 10 years of evolution], in ”XX Nauchnaya konferentsiya Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiya. Materialy konferentsii [XX Scientific conference Modern information technologies: tendencies and perspectives of evolution”], 2013, pp.69–71, (in Russian).

Контекстно-ориентированная модель для разметки сквозной функциональности в исходном коде

¹ М.С. Малеванный <mtxforever@mail.ru>

² С.С. Михалкович <miks@sfedu.ru>

¹ Донской Государственный Технический Университет,
344022, Россия, г. Ростов-на-Дону, ул. Социалистическая, д. 162.

² Южный Федеральный Университет,
344090, Россия, Ростов-на-Дону, Мильчакова, д. 8А.

Аннотация. В данной статье описывается подход к упрощению работы со сквозной функциональностью в исходном коде за счет добавления к среде разработки средств разметки сквозной функциональности. Разметка представлена в виде дерева, отдельные узлы которого могут быть привязаны к блокам кода, обеспечивая быструю навигацию по фрагментам кода, реализующим сквозную функциональность. Привязка узлов дерева к коду осуществляется за счет сохранения в дереве набора информации о фрагментах кода. Сохраняемая информация содержит имя и тип фрагмента кода, а также несколько видов контекстов, которые позволяют однозначно найти фрагмент в коде. Эти контексты позволяют в рамках одной модели работать с кодом на различных языках, как программирования, так и языках разметки, DSL-языках, а также с любым структурированным текстом, например, документацией. Реализация алгоритмов поиска фрагмента по сохраненной информации учитывает возможность внесения изменений в код в процессе разработки, что обеспечивает *устойчивость* привязки. При небольших изменениях исходного кода фрагмент может быть найден автоматически. В случае более серьезных изменений реализован полуавтоматический поиск при минимальном участии

программиста. Исходный код анализируется легковесными парсерами, не полагаясь на инфраструктуру среды разработки. За счет этого достигается возможность работать с широким спектром языков, а также интеграция инструмента в различные среды разработки с минимальными усилиями. В статье представлена модель хранения данных, алгоритмы поиска, а также обзор инструмента, реализующего данную модель.

Ключевые слова: разделение ответственностей; аспекты; языки программирования; среды разработки

DOI: 10.15514/ISPRAS-2016-28(2)-4

Для цитирования: Малеваный М.С., Михалкович С.С. Контекстно-ориентированная модель для разметки сквозной функциональности в исходном коде. *Труды ИСП РАН*, том 28, вып. 2, 2016 г., стр.63-78 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-4

Список литературы

- [1]. M. Eaddy, A. Aho, and G. C. Murphy, “Identifying, assigning, and quantifying crosscutting concerns” in Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques, ser. ACoM '07. Washington, DC, USA: IEEE Computer Society, 2007, p. 2. DOI: 10.1109/ACOM.2007.4.
- [2]. A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks” *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006. DOI: 10.1109/TSE.2006.116.
- [3]. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ” in Proceedings of the 15th European Conference on Object-Oriented Programming, ser. ECOOP'01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353. (<http://dl.acm.org/citation.cfm?id=646158.680006>)
- [4]. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, “The genvoca model of software-system generators” *IEEE Softw.*, vol. 11, no. 5, pp. 89–94, Sep. 1994. DOI: /10.1109/52.311067.
- [5]. D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement” in Proceedings of the 25th International Conference on Software Engineering, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197. (<http://dl.acm.org/citation.cfm?id=776816.776839>).
- [6]. S. Apel, C. Kastner, and C. Lengauer, “Featurehouse: Language independent, automated software composition” in Proceedings of the 31st International Conference on Software Engineering, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.
- [7]. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, “Delta-oriented programming of software product lines” in Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91. (<http://dl.acm.org/citation.cfm?id=1885639.1885647>).
- [8]. W. Harrison and H. Ossher, “Subject-oriented programming: A critique of pure objects” in Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 411–428. DOI: 10.1145/165854.165932.

- [9]. M. C. Chu-Carroll, J. Wright, and A. T. T. Ying, “Visual separation of concerns through multidimensional program storage” in Proceedings of the 2nd International Conference on Aspect-oriented Software Development, ser. AOSD '03. New York, NY, USA: ACM, 2003, pp. 188–197. DOI: 10.1145/643603.643623.
- [10]. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., “Code bubbles: A working set-based interface for code understanding and maintenance” in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512. DOI: 10.1145/1753326.1753706.
- [11]. S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, “Do we really need to extend syntax for advanced modularity?” in Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 95–106. DOI: 10.1145/2162049.2162061.
- [12]. C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines” in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.
- [13]. М.С. Малеванный, С.С. Михалкович, Реализация поддержки аспектов программного кода в интегрированных средах разработки. Современные информационные технологии: тенденции и перспективы развития, 2015, стр. 351–353.
- [14]. M. P. Robillard and F. Weigand-Warr, “Concernmapper: Simple view-based separation of scattered concerns” in Proceedings of the 2005OOPSLA Workshop on Eclipse Technology eXchange, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 65–69. DOI: 10.1145/1117696.1117710.
- [15]. М.С. Малеванный, Легковесный парсинг и его использование для функций среды разработки. Информатизация и связь, том 3, стр. 89–94, 2015.
- [16]. ANSI C grammar. (<http://www.quut.com/c/ANSIC-grammar-y.html>)
- [17]. PascalABC.NET. <http://pascalabc.net/>
- [18]. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академии Наук СССР, 1965. 163.4, стр. 845–848.
- [19]. R. A. Wagner and M. J. Fischer, “The string-to-string correction problem” J. ACM, vol. 21, no. 1, pp. 168–173, Jan. 1974. DOI: 10.1145/321796.321811.
- [20]. Бондарев И. В., Белякова Ю. В., Михалкович С. С. Система программирования PascalABC.NET — 10 лет развития // XX Научная конференция «Современные информационные технологии: тенденции и перспективы развития». Материалы конференции. Ростов н/Д, 2013. С. 69–71.

Approach to Anti-pattern detection in Service-oriented Software Systems

A.S. Yugov <yugovas@live.ru>

*National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.*

Abstract. A service-based approach is a method to develop and integrate program products in a modular manner where each component is available through any net and has the possibility of being dynamically collaborated with other services of the system at run time. That approach is becoming widely adopted in industry of software engineering because it allows the implementation of distributed systems characterized by high quality. Quality attributes can be about the system (e.g., availability, modifiability), business-related aspects (e.g., time to market) or about the architecture (e.g., correctness, consistency). Maintaining quality-attributes on a high level is critical issue because service-based systems lack central control and authority, have limited end-to-end visibility of services, are subject to unpredictable usage scenarios and support dynamic system composition. The constant evolution in systems can easily deteriorate the overall architecture of the system and thus bad design choices, known as anti-patterns, may appear. These are the patterns to be avoided. If we study them and are able to recognize them, then we should be able to avoid them. Knowing bad practices is perhaps as valuable as knowing good practices. With this knowledge, we can re-factor the solution in case we are heading towards an anti-pattern. As with patterns, anti-pattern catalogues are also available. In case of continues evolution of systems, it is metric-based techniques that can be applied to obtain valuable, factual insights into how services work. Given the clear negative impact of anti-patterns, there is a clear and urgent need for techniques and tools to detect them. The article will focus on rules to recognize symptoms of anti-patterns in service-based environment, automated approaches to detection and applying metric-based techniques to this analysis.

Ключевые слова: service based systems; anti-patterns; specification and detection; software quality; quality of service (QoS)

DOI: 10.15514/ISPRAS-2016-28(2)-5

For citation: Yugov A.S., Approach to Anti-pattern detection in Service-oriented Software Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 79-96. DOI: 10.15514/ISPRAS-2016-28(2)-5

1. Introduction

Service-based style of software systems is very widely spread at the industrial development because it allows implementing flexible and scalable distributed systems at a competitive price. The result of development are autonomous, reusable,

and independent units of a platform – services – that can be consumed via any network including the Internet [1].

Traditional approaches to software delivery are based on life cycle phases of the system, when in the development process became involved various teams inside a company or even by different companies [2]. Moreover, in classical approach, the focus is on one vendor supplying the entire system or subsystem. The emergence of service-oriented architecture approach introduces a model divided into levels. It enables the existence of different design approaches; whereby different parties deliver service layers as separate elements. Experience in development of joint projects, divided into separate services, shows that errors may appear in potentially dangerous areas. As part of this work, we will call these areas as anti-patterns.

Anti-patterns in software systems based on services are “bad” solutions recurring design problems. In contrast to design patterns, anti-patterns are well-proven solutions that engineers should avoid. Anti-patterns can also be introduced as a consequence of various changes, such as new user requirements or operating environment changes.

This paper presents an introduction to the anti-pattern detection domain and describes proposed approach for the automated detection of anti-patterns.

2. Examples of Anti-patterns in Service-based Systems

Design (architecture) quality is vitally important for building a well thought-out, easy to maintain and evolving systems. The presence of patterns as antipattern in the system design was recognized as one of the most effective ways to express architectural problems and their solutions, and hence higher quality criterion among different systems [3].

A number of efforts have been taken to formalize the properties of the concept of "bad" practices, i.e., decisions that adversely affect the quality of the system. Despite the emerging interest to service based systems, the literature is not really consistent with respect to pattern and anti-pattern definition and specification in this area. Indeed, the available catalogs use different classification, either based on their nature, scope or objectives.

Some completely new approaches were introduced to identify and detect code vulnerabilities and anti-patterns [4], [5]. The methods used in these campaigns were very diverse: completely manual, based on the research guidelines; metrics based on heuristic methods using rules and thresholds for various metrics; or Bayesian networks. Some approaches [6] are applicable to the application level and can be applied to initial stages of the software life cycle.

Quite a large number of methodologies and tools exist for the detection of anti-patterns, in particular, in object-oriented (OO) systems [7], [8]. However, the detection of anti-patterns in service-based systems, in contrast to the OO systems is still in its infancy. One of the last works by detecting of antipattern in service-oriented architectures (SOA) has been proposed in Moha et al. In 2012 [4].

The authors proposed an approach to the determination and detection of an extensive set of SOA anti-patterns operating such concepts as granularity, cohesion and duplication. Their instrument is able to detect the most popular SOA anti-patterns, defined in literature. In addition to these antipatterns, authors identified three antipatterns, namely: bottleneck service, service chain and data services. Bottleneck is a service that is used by many other components of the system, and as a result, is characterized by high incoming and outgoing connections affecting the response time of service. Chains of services occur when a business object is achieved by a long chain of successive calls. Data service is a service that performs a simple operations of information search or data access, which may affect the connectivity of the component.

In 2012, Rotem-Gal-Oz [9] identified the “knot” antipattern, a small set of connected services, which, however, is closely dependent on each other. Anti-pattern, thus, may reduce the ease of use and response time.

Another example of anti-pattern is “sand pile” defined by Kr'al et al [10]. It appears when many small services use shared data, which can be accessed through the service, which represent the “data service” anti-pattern.

In the paper of Scherbakov et al. proposed “duplicate service” antipattern [11] that affects sharing services that contain similar functions, causing problems in the support process.

In 2003 Dudney et al. [12] have identified a set of anti-patterns for the J2EE applications. “Multi service” anti-pattern stands out, among others, a “tiny service” and “chatty service”. Multi service is a service that provides a variety of business operations, which have no practical similarity (for example, belong to different subsystems) that can affect service availability and response time. Tiny service is a small service with few methods, which are always used together. This can lead to the inability of reuse. Finally, an anti-pattern “chatty service” represents such services that constantly call each other, passing small amount of information.

3. Metric-based Approach to the Detection of Anti-patterns

As DeMarco noted [13], in order to control the quality of development, correct quantitative methods are needed. Already in 1990 Card emphasized that metrics should be used to assess the development of software in terms of quality [14]. But what should be measured? In the above context of design rules, principles and heuristics, this question should be rephrased as follows: is it possible to express the principles of “good design” in a measurable way?

The main goal of this approach is to provide a mechanism for engineers, which will allow them to work with metrics on a more abstract level, which is conceptually much closer to real conditions of applying numerical characteristics. Mechanism defined for this purpose is called a discovery strategy:

Detection strategy is a quantitative expression of the rules by which specific pieces of software (architectural elements), corresponding to this rule, can be found in the source code.

By this reason, the detection strategy is a common approach to analysis of the source code model using metrics. It should be noted that in the context of the above definition, "quantitative expression of the rule" means that the rule should be properly expressible using metrics. The use of metrics in detection strategies grounded filtering mechanisms and composition. In the following subsections, these two mechanisms will be considered more detailed.

The key problem in data filtering is reducing the initial collection of information, so that there remain only those values that are of particular value. This is commonly referred to as data reduction [15]. The aim is to detect those elements of the system, which have special properties. Limits (boundaries) of the subset are determined on the basis of the type of filter. In the context of the measurement process with respect to the software, we usually try to find the extreme (abnormal) values or those values that lay within a certain range. Therefore, distinguish types of filters [16]:

- Marginal filter is a data filter, in which one limit (border) in the result set is clearly identified with a corresponding restriction of the original data set.
- Interval filter is a data filter, in which the lower and upper limits of the resulting subset are explicitly specified in the definition of the data set.

Marginal filters consist of two depending on how we specify the borders, resulting dataset limiting filters may be semantical or statistical.

- Semantical. For these filters two parameters must be specified: a threshold value that indicates a limit value (to be explicitly indicated); and the direction that determines whether the threshold upper or lower limit of the filtered data set. This category of filters is called semantical as the choice of options is based on the semantics of specific metrics in the framework of the model chosen for the interpretation of this metric.
- Statistical. Unlike semantical filters, statistical ones do not require explicit specifications for the threshold, as it is defined directly from the original data set using statistical methods (e.g., scatter plot). However, the direction is still to be specified. Statistical filters are based on the assumption that all the measured entities of the system are designed using the same style, and therefore, the measurement results are comparable.

In this paper, a set of specific data filters of the two previous categories were used. Basing on practical use and interpretation of the selected models, these filters may be grouped as follows:

- Absolute semantic filters: *HigherThan* and *LowerThan*. These filtering mechanisms are parameterized by a numerical value representing the border. We will only use data filters are to express "clear" design rules or heuristics, such as "class should not be associated with more than 6 other classes." It

should be noted that the threshold is specified as a parameter of the filter, while the two possible directions are defining by two particular filters.

- Relative semantic filters: *TopValues* and *BottomValues*. These filters differentiate the filtered data set according to the parameter that determines the number of objects to be recovered, and do not indicate the value of the maximum (or minimum) values are permitted in the result set. Thus, the values in the result set will be considered with respect to the original data set. The parameters used may be absolute (for example, "select 20 objects with the highest values") or percentile (for example, "to remove 10% of the measured objects with the lowest values"). This type of filter is useful in situations where it is necessary to consider the highest (or lowest) values of a given data set, rather than indicating the exact thresholds.
- Statistics: scatter plots. Scatter diagram is a statistical method that can be used to detect outliers in the data set [17]. Data filters based on these statistical techniques, which, of course, not limited to only the scatter diagrams, are useful in the quantification of rules. Again, we need to specify the direction of the deviation of adjacent values based on design rules of semantics.
- Interval Filters. Obviously, for the data interval it is necessary to define two thresholds. However, in the context of the detection strategies, where, in addition to the mechanism of filtering, the composition mechanism exists, filter interval is defined by composition of two semantic absolute filters of opposite directions.

Unlike simple metrics and interpretation models of it, detection strategy should be able to draw conclusions on the basis of a number of rules. Consequently, in addition to the filtering mechanism, which supports the interpretation of the particular metric results, we need a second mechanism for comparing the results of calculations of a number of metrics – a mechanism of composition. Composition mechanism is a rule combining the results of calculating several metric values. In the literature three composition operators were observed: “and”, “or” and “butnot” [16].

These operators can be discussed from two different perspectives:

- From a logical point of view. These three operators are a reflection of rules to combine multiple detection strategies, where operands are descriptions of the design characteristics (symptoms). They facilitate reading and understanding of the detection strategy, because operators of composition are generally expressed in the form of quantitative characteristics, so it is similar to the original wording of the informal thoughts. From this point of view, for example, the operator «and» presupposes that the investigated object has both symptoms that are combined by the operator.
- From the point of sets. This view helps to understand how to build the ultimate result of the detection strategy. The initial set of calculation results on each of the metrics is carried out through the filtering mechanism. Then

remains limited set of system elements (and calculated metrics for these elements), which are interesting for further investigation. The resultant plurality of filtered sets should be merged with the operators using the formulation. Thus, in terms of operations on sets, the operator "and" will correspond to the operation of intersection (\cap), the operator "or" to reunion operation, and the operator "butnot" to minus operation.

4. Definition of detection strategy

This section will be written in the formation of a strategy on the example of the detection of a particular anti-pattern "God Object" [18]. The starting point is the presence of one (or more) of the informal rules that describes the problem situation. In this example, we will proceed from the three heuristics found in the book of Riel [18]:

- The top-level services should share equally the responsibility.
- Services should not contain large amounts of semantically separate functions.
- Services should not have access to fields or properties of other services.

The initial step to create a detection strategy is to translate the set of informal rules into symptoms that can be evaluated by a particular metric. In the case of God Object anti-pattern, the first rule refers to an equal sharing of responsibilities among services, and therefore it refers to service complexity. The second rule tells us about the intensity of communications among this service and all other services; thus, it refers to the low cohesion of services. The third heuristic describes a special coupling i.e., the direct access to data items manipulated by other services. In this case, the symptom is access to "foreign" data.

The second step is to find appropriate metrics, which evaluate more precisely every of the discovered properties. For the God Service anti-pattern, these properties are complexity of the service, cohesion of the service and access to data from other services. Therefore, we found the following set of metrics:

- Weighted Method Count (WMC) is the sum of the static complexity of all methods in a class [19]. We considered the McCabe's approach as a complexity measure [20].
- Tight Class Cohesion (TCC) is the relative number of directly connected methods [21].
- Access to Foreign Data (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [22].

The next step is to select an appropriate filtering scheme that should be applied to all metrics. This step is mainly done basing on the rules described earlier. Therefore, as the first symptom is a "high service complexity" the *TopValues* relative semantical filter was chosen for the WMC metric. For the "low cohesion" symptom it was also chosen a relative semantical filter, but now the *BottomValues* one. For the third

symptom, an absolute filter was selected as we need to catch any try to access a “foreign” data; thus, we the *HigherThan* filter will be used.

One of vital issues in creating a detection strategy is to choose proper parameters (i.e., threshold values) for all data filters. Several approaches exist to do this, but now we just take a 25% value for both the *TopValues* filter for to the WMC metric and to the *BottomValues* filter for the TCC metric. As for filter boundary for the ATFD metric, the decision is pretty simple: no direct access to the data of other services should be allowed, therefore, the threshold value is 1.

The final step is to join all the symptoms, with applying of the special operators described before. From the unstructured heuristics as presented in [18], it was inferred that all three symptoms should be combined if a service is supposed to be a behavioral God Object.

The intention of this work is to use detection strategies in rule definitions in order to facilitate detection of anti-patterns in service based software systems i.e., to select such areas of the system (subsystem) that are participated in a particular anti-pattern. From this point of view, it should be emphasized that the detection strategy approach and the whole method is not limited by finding problems, but it also can facilitate completely different objectives too. For instance, different investigation purposes could be in reverse engineering [23], design pattern detection [22], identification of components in legacy systems [24], etc.

5. Implementing a Tool for Detection of Anti-patterns in Service-based Systems

5.1. Description of Metrics

Calculations intended to detect antipatterns is conducted basing on several basic metrics:

- incoming call rate;
- outgoing call rate;
- response time;
- number of service connections;
- cohesion with other services;
- etc.

Each metric has its specific model and its specific algorithm to calculate. Values of this metric have decisive influence on detection of services participating in antipatterns.

In calculation of metrics, objective measures of occurrence pattern interestingness of data mining like confidence and support are used. These are based on the structure of discovered patterns and the statistics underlying them.

A measure for association rules of the form $X \rightarrow Y$ is called support, representing the percentage of transactions from a log database that the given rule satisfies. This is

intended to be the probability $P(X \cup Y)$, where $X \cup Y$ indicates that a transaction contains both X and Y , that is, the union of item sets X and Y .

Another objective measure for association rules from data mining is confidence, which addresses the degree of certainty of the detected association. In classical data mining this is taken to be the conditional probability $P(X \cap Y)$, that is, the probability that a transaction containing X also contains Y . More formally, confidence and support are defined as

$$\begin{aligned} \text{Support}(X \rightarrow Y) &= P(X \cup Y), \\ \text{Confidence}(X \rightarrow Y) &= P(X \cap Y). \end{aligned}$$

In general, each measure of interestingness is associated with a threshold, which may be controlled. For calculation of metrics each final value of metric is confidence (which is calculated not as in classical data mining but more complexly) divided by support measure (which is calculated in the same manner as in classical data mining). Further, each metric is described in more details.

Incoming and Outcoming Call Rates. The model for calculation of *IncomingCallRate* metric is call matrix. This matrix represents calls services make to each other. For building this matrix and some other models, we need to identify the order of calls. This information is not stored in logs, therefore, the first task is to mine service calls from log. Procedure of mining calls consists of several main steps. The first is ordering log events by traces. This is necessary because occurrence of events in particular order in boundaries of one trace gives us evidence of one particular service call. To mine all the service calls properly it is needed to sort events in the log chronologically within every trace. Once ordering on both levels (trace and timestamp) is finished, we can go through the log and reconstruct service calls. Received values in mined matrix will represent generalized number of calls among services for as *IncomingCallRate* as *OutcomingCallRate*.

Response Time. Response time metric represent general bandwidth of a particular service. This parameter is crucial for systems having high load. Calculation of this metric uses assumptions made for both metrics *IncomingCallRate* and *OutcomingCallRate* but with some modifications. As we are aimed here at measure of time characteristic, the object to explore will be time stamp parameter of the log. Given defined algorithm for incoming and outcoming call rates, we modify it with calculation of time prospect. Instead of just number of calls, we calculate general length of service response. In such a way the summarized time while service was busy is calculated. As a result of precious calculations, the matrix of general time every service spent on work was obtained. Following step is to normalize real values, i.e. to measure not in absolute number but in relative number. This relative number will show percentage of time where the service was working on processing calls. This metric can be used for detection of both highly loaded services and rarely used services.

Cohesion with Other Services. For calculation of this metric classical data mining rules are implemented. For this the conditional probability $P(X \cap Y)$ is taken. That

is, the probability that a transaction containing X also contains Y. Additionally, the special rule for ordering is added. This means that $X \rightarrow Y$ and $Y \rightarrow X$ is different relations. I.e. we observe not only occurrence at one trace but also the order of occurrences. High rate of confidence of this metric is evaluated as high cohesion of several services and, therefore, high behavioral dependency.

Number of Service Connections. All the previous metrics were dynamic characteristics of a system under consideration while number of service connections is a static property of the system. For mining this property, it is enough to have the incidence matrix of service calls. If one service called once another service, we do not consider the same connections in future. Obtained incidence matrix allows us to calculate all existing connections in the system.

The basic model to calculate each of metrics is Graph model (fig. 1) which is extended in each particular metric calculation algorithm with specific attributes. As part of this work, it is assumed that each object, once appeared in the system, initiates a sequence of operations to be performed on the object. This sequence of operations is called workflow. It is worth noting that not every service-oriented system is based on this principle, but we will consider only such systems.

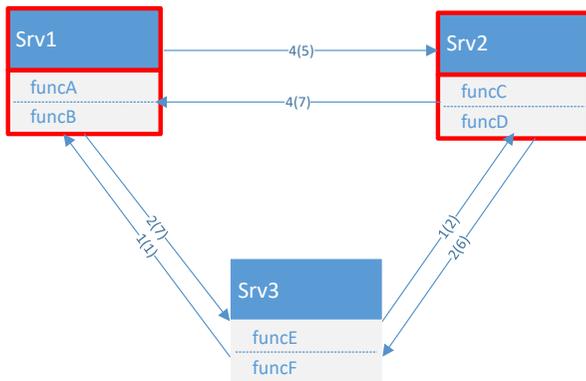


Fig. 1. Base graph model for calculation of metrics.

In this model, services of a software system are presented by graph nodes. Arcs of the graph represent the call ratio, i.e., oriented arc from *Srv1* to *Srv2* shows that *Srv1* in the process of operation calls one of *Srv2* functions. Depending on what metric should be calculated, edges of the graph are marked by specific values. For example, on fig. 1 arcs are labeled by amount of calls in a particular direction and, in parentheses, some weighted value of the transmitted data.

5.2. Extracting Data from Event Logs

The main weakness of previously observed works was the necessity to modify source code of a particular system in order to evaluate concrete metrics. In this work we use

event logs to create a process model of the system and calculate metrics basing on this model. To apply these, it is assumed that the information system records data of events. These logs also contain unstructured and irrelevant data, e.g. information on hardware components, errors and recovery information, and system internal temporary variables. Therefore, extraction of data from log files is a non-trivial task and a necessary pre-processing step for analysis. Business processes and their executions related data are extracted from these log files. Such data are called process trace data. For example, typical process trace data would include process instance id, activity name, activity originator, time stamps, and data about involved elements. Extracted data are converted into the required format.

To be able to analyze log content, the log should have specified structure. In our case the minimal requirements for log is as follows:

- TraceID: shows the identity for a particular trace;
- ServiceID: shows the identity for a particular service;
- FunctionID: shows the identity attribute for a particular function in the service;
- Timestamp: shows the time of occurrence of the event.

The log sample is presented in table 1.

Table 1. Source log sample

TraceId	Service	Function	TimeStamp
1	Srv2	C	2015-06-15 00:25:20
1	Srv1	A	2015-06-15 00:33:24
2	Srv4	F	2015-06-15 00:32:25
3	Srv3	E	2015-06-15 00:24:13
1	Srv2	C	2015-06-15 00:31:52
3	Srv1	B	2015-06-15 00:34:05
4	Srv4	G	2015-06-15 00:25:12
3	Srv3	E	2015-06-15 00:26:28
4	Srv1	A	2015-06-15 00:28:21
4	Srv2	C	2015-06-15 00:30:32
2	Srv1	A	2015-06-15 00:29:48
2	Srv2	C	2015-06-15 00:29:51

Each field included in log has its own purpose in future usage. TraceID is needed for distinguishing events among execution sequences, i.e. for majority of metrics it is necessary to connect events in boundaries of one trace. Moreover, inside traces events appears in chronological order. That is why timestamp is included in log format.

ServiceID and FunctionID describe source of each event. In addition, dimensions of functions and services are main structural units in analysis and creation of models.

5.3. Specification of Rule Cards

The rule cards are storing in XML format. The structure of XML represents scheme of rule card structure. The scheme of XML is presented in fig. 2 in graphical mode. In fig. 3 for more detailed view in XSD standard. The XML should have specialized namespace: “RuleCardNS”. The root element is “RuleCard”. It has name element called “AntipatternName”. This also plays the role of identification attribute.

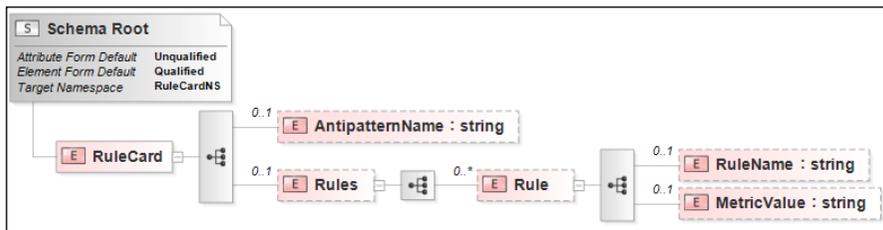


Fig. 2. Structure of antipattern XML

Each Rule is defined through type attribute, metric value and its own name. The type attribute describes what metric (from a set of available metrics) should be calculated. Metric value refers to specific value of calculated metric, which shows whether the service under analysis has a particular symptom or not. Finally, rule name is an identification property for rule.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="unqualified"
elementFormDefault="qualified"
targetNamespace="RuleCardNS">
  <xsd:element name="RuleCard">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="AntipatternName" type="xsd:string" />
        <xsd:element name="Rules">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="unbounded" minOccurs="1"
name="Rule">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="RuleName" type="xsd:string" />
                    <xsd:element name="MetricValue" type="xsd:string" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xs:schema>
```

Fig. 3. Detailed structure of antipattern XML.

5.4. Description of Research Prototype of Analytic System

To automate process of anti-pattern detection the research prototype of information system, which implements the described approach, has been developed. The scheme of the proposed approach is shown in fig. 4. The workflow of the software system consists of several steps. At the point of entry, the program takes log, which is reading from the relational database implemented in SQL Server, and rule card describing rules to detect particular antipattern.

General workflow structure is presented in fig. 5. It starts with reading input data, which are:

- log from some software system implemented according to SOA principles;
- rule card describing all the rules and metrics needed for detection of each particular antipattern.

Once the XML with antipattern description is read the system starts calculation of metrics. Each metric is calculated against its specific algorithm. So for each rule the process of metric calculation has been launching. First, the special model used for

analysis of a particular metric is build. All the models were defined previously. Then with use of received model metrics are calculated. As a result of this process, services suspected in participation in the antipattern are selected.

Next step is to integrate results received in threads of calculation of metrics. The integration is conducted as intersection of result sets from previous threads. Finally, we obtain set of suspicious services, which are parts of antipattern. Commonly there are several services, but is always can be that just one service represents antipatterns or no such services at all were discovered.

Results of analysis is depicting in general graph representation (fig. 4).

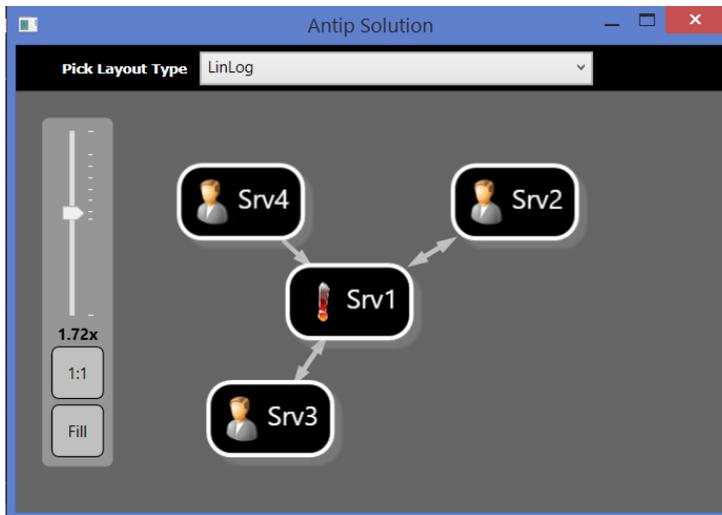


Fig. 4. Graphical representation of results.

Nodes in this graph are services and edges in this graph are direct references among services. Each node represents one service observed in the system whose log has been observed. As for example on fig. 5, nodes such as for services Srv2, Srv3, and Srv4 represent proper developed services, i.e. they are not participated in antipatterns. Suspicious services are marked with “!” sign, that means that this particular service is a part (or is whole) of antipattern. In our example this is service number 1 (Srv1).

Edges represent calls made of one service to another one. Concerning example from fig. 2.6, Srv4 calls Srv1 therefore one edge directed from Srv4 to Srv1 is depicted. Srv1 and Srv2 calls functions of each other therefore the edge is bidirectional.

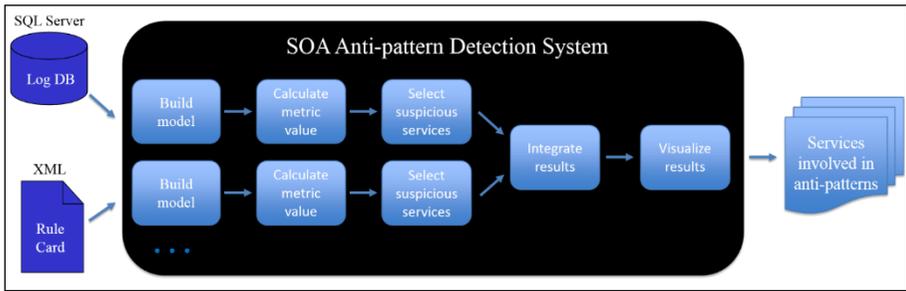


Fig. 5. Base graph model for calculation of metrics.

6. Conclusion

This work addresses the issue of necessity of monitoring circumstance of software systems implemented through service-based approach in conditions of continuous development and enhancement when number and complexity of systems is expanding faster than a human being can handle.

During the exploration of process mining and data mining domains the general service-based specific antipattern detection rules were invented. All rules consist of several metrics and its specific values, describing symptoms of antipatterns. At the time, five metrics are available for usage in detection rules: incoming call rate, outgoing call rate, response time, cohesion, and number of service connections. With applying these metrics several antipatterns was specified and algorithms for it detection were introduced.

Algorithms of antipattern detection based on metric calculation were implemented as a software tool (research prototype), which allows by specifying rule cards in XML format and log in SQL Server database to detect antipatterns. The software tool is developed with usage of Windows Presentation Foundation framework.

It is planned that in future the information system will be refined according to analysis of real life logs from and number of available metrics and possible to detect antipatterns will be significantly greater. The following step will be introducing dynamic analysis of system behavior in addition to implemented analysis of static footprints. Furthermore, some fuzziness can be introduced for the evaluation of the threshold values thus to make antipattern detection rules more flexible.

References

- [1]. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.
- [2]. G. Farrow. SOA antipatterns: When the SOA paradigm breaks // IBM Developer Works [Online]. Available: http://www.ibm.com/developerworks/library/wa-soa_antipattern/
- [3]. M. Nayrolles; N. Moha; P. Valtchev. Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces in Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), pp. 321–330, IEEE, 2013.

- [4]. N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéheneuc, B. Baudry, J.-M. Jezequel. Specification and Detection of SOA Antipatterns. In International Conference on Service-Oriented Computing (ICSOC), pp. 1–16, 2012
- [5]. F. Khomh, M. D. Penta, Y.-G. Guéheneuc, G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17(3):243–275, 2012.
- [6]. D. Arcelli, V. Cortellessa, C. Trubiani. Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns. *ECEASST* 59 (2013).
- [7]. M. Kessentini, S. Vaucher, and H. Sahraoui. “Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code” in Proceedings of the IEEE/ACM ASE. ACM, 2010, pp. 113–122.
- [8]. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
- [9]. A. Rotem-Gal-Oz, *SOA Patterns*, 1st ed. Manning Publications, 2012.
- [10]. J. Kr’al and M. Zemlicka, “The most important service-oriented antipatterns,” in ICSEA, 2007, p. 29.
- [11]. L. Cherbakov, M. Ibrahim, and J. Ang, “Soa antipatterns: the obstacles to the adoption and successful realization of service-oriented architecture”.
- [12]. B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2002
- [13]. T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation*. Yourdan Press, New Jersey, 1982.
- [14]. D. Card and R. Glass. *Measure Software Design Quality*. Prentice-Hall, NJ, 1990.
- [15]. P.G. Hoel. *Introduction to Mathematical Statistics*. Wiley, 1954.
- [16]. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM’04). Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 350–359.
- [17]. N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.
- [18]. A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [19]. S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [20]. T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, dec 1976.
- [21]. J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. Proc. ACM Symposium on Software Reusability, apr. 1995.
- [22]. R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In Proceedings of TOOLS USA 2001, pages 103–116. IEEE Computer Society, 2001.
- [23]. E. Casais. State-of-the-art in Re-engineering Methods. Achievement report SOAMET-A1.3.1, FAMOOS, October 1996.
- [24]. A. Trifu. Using Cluster Analysis in the Architecture Recovery of OO Legacy Systems. Diploma Thesis, Karlsruhe and the “Politehnica” University Timisoara, 2001

Подход к обнаружению анти-паттернов в сервис-ориентированных системах

А.С. Югов <yugovas@live.ru>

*Национальный Исследовательский Университет «Высшая Школа
Экономики»,*

101000, Россия, Москва, ул. Мясницкая, д.20.

Аннотация. Сервис-ориентированные системы, как стиль в архитектуре приложений, широко принят в промышленной разработке программного обеспечения, потому что это позволяет разрабатывать гибкие и масштабируемые распределенные системы по более выгодной цене. Результатом разработки становятся автономные, многоразовые и независимые от платформы использования функционала единицы – сервисы. Сервис-ориентированные системы, как и любые другие программные системы, развиваются с течением времени, независимо от того, какими были предпосылки изменений: новые требования, изменение среды функционирования, и т.п. Эта эволюция может усложнить только что измененные системы, и тем самым увеличить трудность их технической поддержки и дальнейшего развития. Постоянные изменения могут привести к появлению в системе «плохих» решений – анти-паттернов, что, в свою очередь, снижает качество программной системы и требует большего внимания разработчиков на всех этапах жизненного цикла системы. Анти-паттерны в процессе эксплуатации систем на базе сервисов представляют собой «плохие» решения повторяющихся проблем проектирования. В противоположность паттернам проектирования, которые являются хорошими проверенными решениями, анти-паттерны инженерам следует избегать. Анти-паттерны также могут быть введены как следствие различных изменений, таких как, например, новые требования пользователей или изменения среды функционирования. Знание анти-паттернов является таким же важным, как и знание анти-паттернов, поэтому анти-паттерны описываются специалистами ИТ области, а сами описания собираются в каталоги. И чаще всего именно метрико-ориентированный подход может быть применен для получения ценной, основанной на фактах, информации о том, как работают сервисы. В данной статье рассматриваются примеры анти-паттернов и методов их автоматического обнаружения. Все методы будут сосредоточены на метрико-ориентированном подходе к анализу программных систем.

Ключевые слова: сервис-ориентированные системы, анти-паттерны, спецификация и обнаружение, качество программного обеспечения.

DOI: 10.15514/ISPRAS-2016-28(2)-5

Для цитирования: Югов А.С. Подход к обнаружению анти-паттернов в сервис-ориентированных системах. Труды ИСП РАН, том 28, вып. 2, 2016 г. стр. 79-96 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-5

Список литературы

- [1]. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.

- [2]. G. Farrow. SOA antipatterns: When the SOA paradigm breaks // IBM Developer Works [Online]. Available: http://www.ibm.com/developerworks/library/wa-soa_antipattern/
- [3]. M. Nayrolles; N. Moha; P. Valtchev. Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces in Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), pp. 321–330, IEEE, 2013.
- [4]. N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Gúeheneuc, B. Baudry, J.-M. Jézequel. Specification and Detection of SOA Antipatterns. In International Conference on Service-Oriented Computing (ICSOC), pp. 1–16, 2012
- [5]. F. Khomh, M. D. Penta, Y.-G. Gúeheneuc, G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17(3):243–275, 2012.
- [6]. D. Arcelli, V. Cortellessa, C. Trubiani. Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns. *ECEASST* 59 (2013).
- [7]. M. Kessentini, S. Vaucher, and H. Sahraoui. “Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code” in Proceedings of the IEEE/ACM ASE. ACM, 2010, pp. 113–122.
- [8]. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
- [9]. A. Rotem-Gal-Oz, *SOA Patterns*, 1st ed. Manning Publications, 2012.
- [10]. J. Kr’al and M. Zemlicka, “The most important service-oriented antipatterns,” in ICSEA, 2007, p. 29.
- [11]. L. Cherbakov, M. Ibrahim, and J. Ang, “Soa antipatterns: the obstacles to the adoption and successful re-alization of service-oriented architecture”.
- [12]. B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2002
- [13]. T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation*. Yourdan Press, New Jersey, 1982.
- [14]. D. Card and R. Glass. *Measure Software Design Quality*. Prentice-Hall, NJ, 1990.
- [15]. P.G. Hoel. *Introduction to Mathematical Statistics*. Wiley, 1954.
- [16]. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM’04). Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 350–359.
- [17]. N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.
- [18]. A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [19]. S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [20]. T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, dec 1976.
- [21]. J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, apr. 1995.
- [22]. R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In Proceedings of TOOLS USA 2001, pages 103–116. IEEE Computer Society, 2001.
- [23]. E. Casais. State-of-the-art in Re-engineering Methods. Achievement report SOAMET-A1.3.1, FAMOOS, October 1996.
- [24]. A. Trifu. Using Cluster Analysis in the Architecture Recovery of OO Legacy Systems. Diploma Thesis, Karlsruhe and the “Politehnica” University Timisoara, 2001

Technology for application family creation based on domain analysis

A. Gudoshnikova <gudoshnikova.anna@gmail.com>

Y. Litvinov <y.litvinov@spbu.ru>

Software Engineering chair,

St.Petersburg State University,

198504, Russia, St.Petersburg, Peterhof, Universitetsky prospekt, 28

Abstract. The theme of code reuse in software development is still important. Sometimes it is hard to find out what exactly we need to reuse in isolation of context. However, there is an opportunity to narrow the context problem, if applications in one given domain are considered. Same features in different applications in one domain have the same context respectively so the common part must be reused. Hence, the problem of domain analysis arises. On the other hand, there is metaCASE-technology that allows to generate code of an application using diagrams, which describe this application. The main objective of this article is to present the technology for application family creation which connects the metaCASE-technology and results of domain analysis activity. We propose to use some ideas of FODA (feature-oriented domain analysis) approach for domain analysis and use feature diagrams for describing of variability in a domain. Then we suggest to generate metamodel of the domain-specific visual language, based on feature diagram. After that based on generated metamodel domain-specific visual language editor is generated with the aid of metaCASE-tool. With this language user can connect and configure existing feature implementations thus producing an application. This technology supposed to be especially useful for software product lines.

Keywords: domain analysis; metaCASEtechnology; domain-specific language; application family

DOI: 10.15514/ISPRAS-2016-28(2)-6

For citation: Gudoshnikova A.A., Litvinov Y.V. Technology for application family creation based on domain analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue. 2, 2016, pp. 97-110
DOI: 10.15514/ISPRAS-2016-28(2)-6

1. Introduction

The term “reuse” in software engineering is closely associated with context. Reuse objects can be programs, parts of programs, specifications, requirements, architectures, test plans, etc. Reuse of one object leads to reuse of another object. This means, there is a need to reuse something more than just code, i.e. there is a call for

increasing the abstraction level. It is commonly supposed that reuse, as some kind of activity, can be divided into groups according to what should be reused: components, process for gaining the product, technology or knowledge. At all accounts any reuse object cannot be discussed without environment, where the given object exists. Hence, the context problem still remains. However, if we reuse objects in one domain, the context issue may be narrowed. The product line implies that there is a common part, it can be: (1) architecture, (2) components, (3) algorithms, (4) methods, etc. — and this part exists in the same context. This fact facilitates the reuse problem. Consequently, the common part must be reused.

Gathering information about the domain is the crucial step in the whole process of software development. Nowadays applications in one domain are often designed independently; this approach leads to increase of development time and cost. Usually such applications have similar functionality, so the reuse problem moves to the forefront in an attempt to speed up the development and to decrease the cost for systems in one domain. The reuse process in one domain supposes the necessity of the domain analysis activity. At present domain analysis in software life cycle is performed in informal way. There are some domain analysis tools, but such tools are not integrated with development tools. As the result of the domain analysis activity some diagrams just are put up on the board, and do not take part in following process of software design. The risk of incorrect understanding of domain-dependent knowledge increases. Therefore, many peculiarities of the domain may be missed in development process because of the factor of human error. This fact may lead to development of the product, which does not satisfy requirements at all. Hence, there emerged a need for a tool in which domain analysis activity would play a vital role in software development process, i.e. based on this activity would be possible to generate some design model, so developers and other process actors could rely on this model. At the present day there is no tool that could allow to solve this problem.

One possible solution for this problem is the use of domain analysis tool in model-driven development, or, more precisely, domain-specific modeling. Domain-specific approach uses visual languages to specify system under development, but, contrary to general model-driven approach, which uses general-purpose visual languages like UML, domain-specific languages are tailored specifically for given domain or a set of problems. Existing studies [1-4] show that due to closeness to a problem domain and the ability to generate complete application by visual models domain-specific languages boost development productivity by 3 to 10 times compared to general-purpose languages. It is clear that developing a tool for domain-specific language “from scratch” for each domain will be prohibitively costly, so special systems are used that allow to declaratively specify syntax of a language and to automatically generate such tools as visual editor, source code generators, constraints checkers and so on. Such systems are called DSM platforms, most known of these is MetaEdit+ [5-7], Eclipse GMP [8, 9], Microsoft Modeling SDK [10].

Main idea of domain-specific modeling is to use a number of visual languages in one tool to develop a complete system. Every language can provide a different point of

view on a system. We propose to exploit this idea to automatically produce useful artifacts from the results of domain analysis thus seamlessly integrating this phase into development process (such as [11]). For that, we will use specific visual language to perform domain analysis and to build domain model, language simple enough to be useful to analysts and domain experts who do not necessarily possess programming skills. Then, using this domain model, we will generate actual domain-specific language that will allow to configure various existing pre-built components and integrate them to generate a working application. As we will see, this language will also typically be very simple so that non-programmers can use it. The only real coding in the proposed approach occurs when creating components from which applications will be built, but for product lines these components will already exist anyway, as they will in a case when a team develops many applications in one domain for some time. Not all steps in proposed approach are fully automatic, as a visual language needs tailoring after generation from domain model — we still need to manually specify shapes of its elements (to be familiar for domain experts) and configure properties which depend on existing components and cannot be derived from domain model. It is also possible that generated application will need tailoring by hand, but generation can significantly lower the effort needed to create application.

Main contribution of this research-in-progress paper is a novel approach to product line development and assets reuse. Also an implementation of technology which uses this approach is presented. Our technology is based on QReal DSM platform [12], an open source tool developed by Software Engineering chair of St. Petersburg State University. An evaluation of proposed approach is also presented, but on a rather simple problem, so a much wider evaluation is needed for this study to be considered complete, such as the applicability of this approach to complex real-life situations and determining actual productivity boost on real-life problems.

The rest of this paper is structured as follows: in section 2 most important terminology for domain analysis is given, also related works are considered. In section 3 we present our method and its implementation as development platform, in section 4 an example of application of our approach is given, we will consider a family of Android gamepads for remote control of various robot models. Section 5 concludes the paper.

2. Domain analysis approaches

There is no any clear and long-standing definition of the term “domain analysis”. Almost all papers, in which this term is considered, go back to 80s-90s of the twentieth century. It was then that scientists, taking into account rapidly growing technologies, were thinking about global reuse. Always projects are developing for concrete user needs, so then the term “domain” took the definition. Domain is the field of expertise, problems in which the software intends to solve. According to Rugaber [13], the domain is described in terms of glossary, some assumptions, architecture approach and literature.

Then the question arise, how we need to analyze the domain for acquiring the necessary information. At present, the information gathering into knowledge bases is

understood under the term “domain analysis”. Although, Prieto-Diaz [14] confirms that domain analysis is an activity, which is held before system analysis and its output is used for system analysis to the same degree as system analysis’s output is used for system design. There are other definitions of the term “domain analysis”. Ferre [15] has presented definitions, such as: (1) the process of identification, organization and presenting the relevant information of a given domain, (2) the process, in which the customer’s knowledge are identified, concretized and systemized. The relevant information of the domain should be presented in objective, readily available way, such way is called “domain model”. Mernik [16] specifies that the domain model includes not only glossary, but also must describe commonalities and variabilities of terms. Such model should precisely set bounds of the domain, i.e. clear and exact characterize a range of questions, which are considered in the domain. Term variabilities allow to define exactly, what information must be specified in concrete system implementation. Term commonalities are used for defining a set of shared operations between different applications. Implementing commonalities and adding the gained model with information, which can be specified in instance of the concrete system, a set of different systems can be obtained based on one common model. In such manner, based on one domain model, the set of different systems in given domain can be implemented. Taking into account definitions above mentioned, we can conclude that domain analysis is the activity of forward system analysis, which goal is to provide the domain model.

As stated above, at present in many software companies the term “domain analysis” is understood as information gathering into some knowledge bases, but it is obvious that there are disadvantages of this approach. It may lead to incomplete glossary, absence of agreements about understanding some terms in the domain, so any misunderstanding of domain can result in an improper product. Therefore, several dozens of years ago were introduced some formal approaches for domain analysis. Here will be mentioned some of them. Main objective any domain analysis approach is to gain the domain model.

Despite different understanding of the term “domain analysis”, Arango [17] showed that all formal domain analysis methods follow the general process for obtaining the domain model. This process includes next stages: (1) domain characterization, (2) data collection, (3) data analysis, (4) classification and finally (5) evaluation of domain model. There are following domain analysis approaches: 1) DARE (Domain Analysis and Reuse Environment) [18]. The crucial idea of this method is to create the domain book, that will include the universal architecture and library of reusable components. 2) DSSA (Domain-Specific Software Architectures) [19]. Given approach allows to create a domain glossary with the aid of use case analysis. 3) ODE (Ontology-based Domain Engineering) [20]. This approach connects the ontology idea with object-oriented approach. Ontology includes terms and their connections, definitions, properties and constraints. Library of objects is built based on mapping ontology with object-oriented entities. 4) FODA (Feature-Oriented Domain Analysis) [21]. This method has get popularity among scientists in the research area because of its simplicity for non-programmers. The main idea of this approach is creating feature

100

model. This model describes functionality, which the future product should possess. Such model must note what features are compulsory for implement in any instance of application in a given domain, what features must be implemented but there is some alternative between them, and present features, which may be implemented but not compulsory. This model can be easily built by expert in the domain.

Concerning product line creating with the aid of using domain model, Estublier [22] presented approach, which is based on some aspects and requirements. These entities were proposed by authors. Such approach based on MDE methodology. Domain model is considered as metamodel, which is described on MOF or UML. There is an interpreter, which translates each term in metamodel into Java class, and concrete models — into instances of these classes. Domain model is accompanied with feature model, which include some external behavior of the system. Authors use aspect-oriented techniques for feature implementing and following their mapping with terms in domain model. Consequently, there is a close interaction between domain modeling and feature modeling. It seems that such approach is a bit complicated for non-programmers. In addition, there is no any industrial use of this method, but it is worth noting that authors describe appliance in this article [23].

3. Proposed approach

In our approach we will use some ideas of Feature- Oriented Domain Analysis (FODA) method to perform domain analysis and to create feature models. For this we will use visual editor that implements feature diagrams and is easy enough for domain experts. Then, when feature models are ready, each feature is implemented as reusable and configurable component on selected implementation language (C#, C++, Java and so on) and feature library is formed as a collection of such components. This process requires qualified programmers and requires more effort than to simply create one application, but it allows to reuse features from feature library to create as many applications as needed. Also, this process is scalable, so we may add new features into feature library later, thus allowing to create more complex applications. At this stage of development domain experts shall work with programmers, and they shall use feature diagrams as an input for creation of feature library to simplify matching between features and components in feature library.

Next step is to create domain-specific language that allows to combine and configure features from feature library to implement applications in given domain. This is where our approach differs from common reuse strategies. Naïve approach would be to generate an application directly from feature diagram, somehow marking features that shall be included into application, and it actually works fine when domain variability is low [24]. But more common is the situation when features themselves have properties that allow to configure them, those properties can have different types. Also, components may be related to each other in different ways, be used in configuration of one another, or some of their properties may be meaningless in absence or presence of other feature. Those rules may be implemented implicitly in

application generator and require that programmers will always observe them, but we propose that these rules will be captured explicitly by dedicated domain-specific language. Such language may make models that do not observe those rules syntactically incorrect, and it will greatly reduce the possibility of human error and reduce knowledge required to efficiently use programming system. By using DSM platforms one can relatively quickly create domain-specific language that will capture domain knowledge, but we already have feature diagram, so we actually can generate the language using it. Generator takes feature diagram as input and produces metamodel of a language. Metamodel is a visual model of a language syntax, that can be opened and edited in yet another visual editor that is part of DSM platform, this editor is called metaeditor. Features from feature diagram become entities in metamodel, this metamodel is then edited to provide shape and a list of properties for each entity. Any vector image can play the role of shape, so the best practice is to select shape that is similar to a feature it depicts. For example, if an application can have buttons, “button” becomes entity in domain-specific language and looks like a button on a diagram. The same happens with properties — for each feature they are added in metaeditor to corresponding language entity with respect to feature library that actually implements this feature and uses the property to configure it. Properties have name, type and default value. On this step it is also possible to define some constraints on a metamodel that will be checked when model will be edited. If some constraints are violated, user will immediately receive warning, which makes errors in a target application even less likely to occur.

On a next step we use editor generator of the DSM platform to create visual editor for our newly created language. This step is fully automated, and when an editor is generated and loaded into DSM platform, we can use it to create diagrams that specify target applications.

The next thing we need is to generate actual code on target textual language that will call feature library and glue features together. For this we shall return to metamodel level and define generation rules for metamodel. This step is performed only once for a given domain after the feature library and metamodel are finished, and then the same generator is used for each application created by using of the technology. Recommendations for development of domain-specific generator are well-known in DSM literature (for example, [7]): it is the best to write first application by hand, then draw a model that is supposed to be generated into this application, then find the places in handwritten application that shall be parameterised by information from model and let the generator replace such handwritten parts with data from model. This process is continued until handwritten application becomes a template that is filled by generator with information taken from model. Handwritten application and, consequently, a generator shall extensively use feature library to minimize the amount of code that is generated directly, in ideal case generator shall produce merely a glue code that binds components from feature library together.

After all steps above are finished we have feature library, visual editor for simple domain-specific language that allows to describe how features are combined and configured in a concrete application, and a generator that automatically produces

complete application by a model in domain-specific language using feature library as domain-specific runtime [7]. Now we may create as many applications as we wish by just drawing models and automatically generate complete executable code. Theoretically. Of course, in practice there will always be a need to modify feature diagram, to extend feature library and, consequently, domain-specific language metamodel, modify generator and even to make some changes in generated code, there is no silver bullet. But we believe that our approach can provide better separation of concerns, provides better utilization of domain experts knowledge and expertise among a team. Summary of a process described above and relation between various tools and roles of developers is provided on fig. 1.

This approach was implemented in a technology based on QReal DSM platform. QReal became an enabler technology because it provides easy and effective way to create visual editor for domain-specific languages that allows to create fully functional editor in less than an hour. It has visual metaeditor, visual constraints definition tool, visual shape editor and a C++ library that allows to quickly specify generation rules. Feature diagram editor and generator that creates metamodel by feature diagrams were both implemented as plugins to QReal core. Note that feature diagram language is itself domain-specific language for the domain of domain analysis, so it was implemented using QReal metaeditor. The same metaeditor (including shape editor and constraints editor) is then used to tailor the generated metamodel of domain-specific language. Then metaeditor generator is used to generate yet another plugin to QReal that provides visual editor for created language. Then the generator is implemented by hand on C++ with Qt library using generator creation library included in QReal. Then it is possible to create special distribution of QReal (using Qt Installer framework) that includes only QReal core, editors for feature diagrams (at this point they are needed only as reference) and domain-specific language, generator and feature library, thus forming a complete technology that can be used to generate target applications.

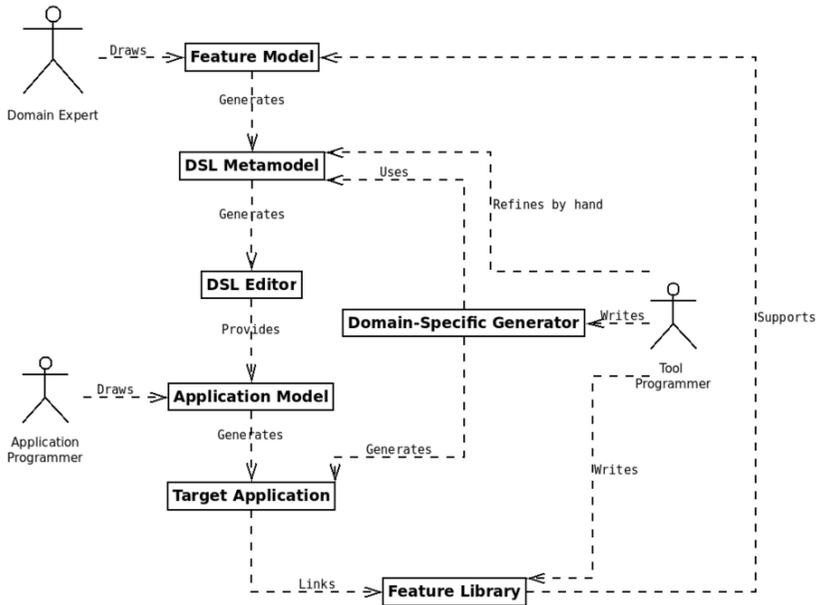


Fig. 1. Relations between artifacts and roles in proposed approach to domain components reuse.

4. Evaluation

For demonstration of the efficiency of proposed above approach there was implemented a model application for remote control of various robot models — “Joystick”. The main substantiation for implementing such application is that controlling different robot models requires different control elements. For example, one model can be controlled with only two pads, but another — with one pad and two buttons. Such application was implemented in C# for Windows Phone platform. Screenshots of this simple application are presented on fig.2.



Fig. 2. Screenshots of “Joystick” application.

As mentioned above, it was used QReal as DSM tool. A visual language was implemented there for describing feature models. Appropriate feature model for “Joystick” application family is proposed on fig.3. This feature model presents explicit features, which are labeled as green, and some unite feature groups, which are labeled as blue. Type of arrow shows which feature is compulsory (shown as solid line with arrow on the end), which compulsory but there is some alternative between them (shown as dash line with an arrow on the end), and optional features, which may be implemented but not compulsory (shown as dash line with a circle on the end).

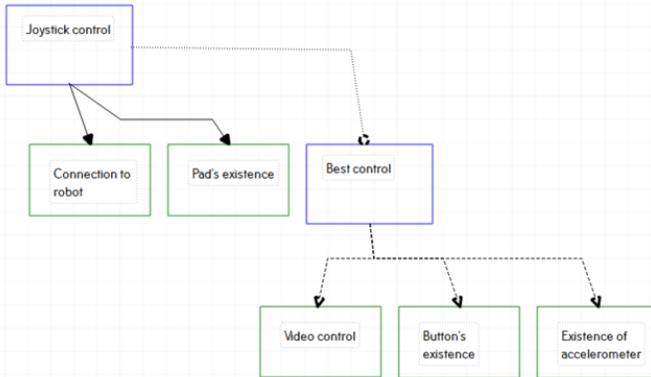


Fig. 3. Feature model for “Joystick” application family.

Based on this feature model a metamodel for future visual language was generated, which is required for building different models for different configurations. Generated metamodel is presented on fig. 4. As it can be seen, metamodel is very simple. At this stage we can propose that entities, such as “buttons” and “pads”, may have a property “Quantity”. In addition, we can specify images for these entities, which will be shown in visual language.

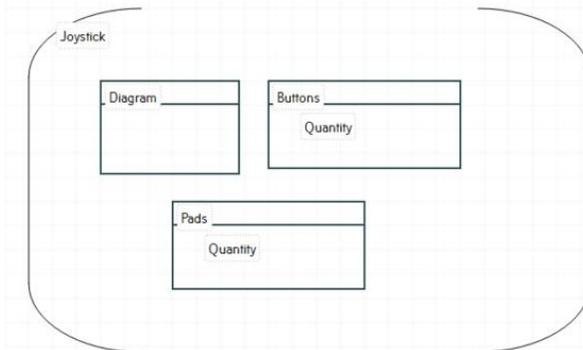


Fig. 4. Metamodel of visual language for “Joystick” application family.

Then with the aid of QReal tool a visual language was generated. Example of generated visual language is demonstrated on Fig. 5. It can be seen that in visual language editor can be specified property “Quantity”, explicitly noting the concrete number of pads. As can be seen, example is quite simple for demonstrating extensive possibilities of the approach proposed above. At present there is no rigorous evaluation of the proposed process. Also, cohesive and consistent technology for creating application family based on domain analysis is not implemented yet, here we have described a conceptproof prototype. Therefore, this work requires more detailed explorations.

5. Conclusion

The problem of not using domain analysis result for further generation of some entities for software development process was stated. There were considered some formal domain analysis approaches and we concluded that creation of feature diagrams is the most elegant decision for domain analysis that can be conducted by domain expert, i.e. non-programmer, maybe in collaboration with system analysts. Moreover, there was discussed one of the possible solutions, which is presented by Estublier, we specify some problems of such method. We suggested our own approach for creation of application family in one domain based on domain analysis. Thus, some target applications can be implemented even by non-programmers using domain-specific language with configuring features from library. Also, there was some evaluation of this approach, where we pointed out that this example remains many questions because of its simplicity.

References

- [1]. Tolvanen J.-p., Kelly S. Model-Driven Development Challenges and Solutions // *Modelsward*, 2016, pp. 711–719.
- [2]. Baker P., Loh S., Weil F. Model-driven engineering in a large industrial context — Motorola case study // *MoDELS’05: Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*. Berlin: Springer, 2005, pp. 476–491.
- [3]. A software engineering experiment in software component generation / R. Kieburtz, L. McKinney, J. Bell et al. // *Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 542–552.
- [4]. Kelly S., Tolvanen J.-P. Visual domain-specific modeling: Benefits and experiences of using metaCASE tools // *International Workshop on Model Engineering*, at ECOOP. 2000. URL: http://dsmforum.org/papers/Visual_domain-specific_modelling.pdf.
- [5]. Tolvanen J.-P., Pohjonen R., Kelly S. Advanced tooling for domain-specific modeling: MetaEdit+ // *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM’07)*, 2007. URL: <http://www.dsmforum.org/events/DSM07/papers/tolvanen.pdf>.
- [6]. Tolvanen J.-P. and Kelly S. MetaEdit+: defining and using integrated domain-specific modeling languages // *Proceedings of the 24th ACM SIGPLAN conference companion*

- on Object oriented programming systems languages and applications / ACM. New York, NY, USA: ACM, 2009, pp. 819–820.
- [7]. Kelly S., Tolvanen J.-P. Domain-specific modeling: enabling full code generation. Hoboken, New Jersey, USA: Wiley-IEEE Computer Society Press, 2008, p. 444.
- [8]. Gronback R. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Stoughton, Massachusetts, USA: Addison-Wesley, 2009, p. 736.
- [9]. Viyovic V., Maksimovic M., Perisic B. Sirius: A rapid development of DSM graphical editor // IEEE 18th International Conference on Intelligent Engineering Systems INES 2014. Los Alamitos, CA, USA: IEEE Computer Society, 2014, pp. 233–238.
- [10]. Domain-specific development with Visual Studio DSL Tools / S. Cook, G. Jones, S. Kent et al. Crawfordsville, Indiana, USA: Addison-Wesley, 2007, p. 576.
- [11]. Koznov D. Process Model of DSM Solution Development and Evolution for Small and Medium-Sized Software Companies // Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International / IEEE. 2011, pp. 85–92.
- [12]. QReal DSM platform—An Environment for Creation of Specific Visual IDEs / A. Kuzenkova, A. Deripaska, T. Bryksin et al. // ENASE 2013—Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering. Setubal, Portugal: SciTePress, 2013, pp. 205–211.
- [13]. Rugaber S. Domain analysis and reverse engineering // White Paper, January. 1994.
- [14]. Prieto-Diaz R. Domain analysis for reusability // Software reuse: emerging technology / IEEE Computer Society Press. 1988, pp. 347–353.
- [15]. Ferre X., Vegas S. An evaluation of domain analysis methods // 4th CASE/IFIP8 International Workshop in Evaluation of Modeling in System Analysis and Design / Citeseer. 1999, pp. 2–6.
- [16]. Mernik M., Heering J., Sloane A. M. When and how to develop domain-specific languages // ACM computing surveys (CSUR). 2005. Vol. 37, no. 4. P. 316–344.
- [17]. Arango G. Domain analysis methods // Software Reusability. 1994, pp. 17–49.
- [18]. DARE: Domain analysis and reuse environment / W. Frakes, R. Prieto, C. Fox et al. // Annals of Software Engineering. 1998, Vol. 5, no. 1, pp. 125–141.
- [19]. Taylor R. N., Tracz W., Coglianese L. Software development using domain-specific software architectures // ACM SIGSOFT Software Engineering Notes. 1995, vol. 20, no. 5, pp. 27–38.
- [20]. Falbo R. d. A., Guizzardi G., Duarte K. C. An ontological approach to domain engineering // Proceedings of the 14th international conference on Software engineering and knowledge engineering / ACM. 2002, pp. 351–358.
- [21]. Feature-oriented domain analysis (FODA): Tech. Rep.: / K. C. Kang, S. G. Cohen, J. A. Hess et al.: DTIC Document, 1990.
- [22]. Estublier J., Vega G. Reuse and variability in large software applications // ACM SIGSOFT Software Engineering Notes. 2005, vol. 30, no. 5, pp. 316–325.
- [23]. An approach and framework for extensible process support system / J. Estublier, J. Villalobos, L. Anh-Tuyet et al. // Software Process Technology. Springer, 2003, pp. 46–61.
- [24]. The Variability Model of the Linux Kernel / S. She, R. Lotufo, T. Berger et al. // VaMoS. 2010, vol. 10, pp. 45–51.

Технология создания семейства приложений на основе анализа предметной области

А.А.Гудошникова <gudoshnikova.anna@gmail.com>

Ю.В. Литвинов <y.litvinov@spbu.ru>

Кафедра системного программирования,

Санкт-Петербургский государственный университет,

198504, Россия, Санкт-Петербург, Старый Петергоф, Университетский проспект, д. 28

Аннотация. Тема переиспользования кода при разработке программного обеспечения до сих пор актуальна. Иногда трудно понять, что нужно переиспользовать в изоляции от контекста, в частности переиспользование одного объекта влечет за собой переиспользование другого. Однако есть возможность сузить проблему контекста, если рассматривать приложения в одной предметной области. Одни и те же характеристики в разных приложениях, но которые относятся к одной предметной области, имеют один и тот же контекст, поэтому важно и нужно переиспользовать эту общую часть. Таким образом, на первый план выходит задача анализа предметной области. С другой стороны, в настоящее время активно развиваются metaCASE-технологии, которые позволяют сгенерировать код целевого приложения, основываясь на диаграммах, описывающие это приложение. Главной целью данной статьи является представление технологии для создания семейств приложений в одной предметной области, которая соединяет деятельность по анализу предметной области и metaCASE-технологии. Мы используем некоторые идеи метода для анализа предметной области FODA (от англ. “Feature-Oriented Domain Analysis”), а именно создаем диаграмму характеристик для описания предметной области. Затем на основе такой диаграммы предлагаем генерировать метамодель предметно-ориентированного визуального языка. После этого средствами metaCASE-инструмента генерируем редактор предметно-ориентированного визуального языка. С помощью такого языка пользователь может соединять и конфигурировать существующие заранее реализованные характеристики, таким образом создавая целевое приложение. Полагается, что такая технология будет полезна при создании линейки продуктов.

Ключевые слова: анализ предметной области; metaCASE-технология; предметно-ориентированный язык; семейство приложений.

DOI: 10.15514/ISPRAS-2016-28(2)-6

Для цитирования: Гудошникова А.А., Литвинов Ю.В. Технология создания семейства приложений на основе анализа предметной области. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 97-110 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-6

Список литературы

- [1]. Tolvanen J.-p., Kelly S. Model-Driven Development Challenges and Solutions // *Modelsward*, 2016, pp. 711–719.

- [2]. Baker P., Loh S., Weil F. Model-driven engineering in a large industrial context — Motorola case study // *MoDELS'05: Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*. Berlin: Springer, 2005, pp. 476–491.
- [3]. A software engineering experiment in software component generation / R. Kiebertz, L. McKinney, J. Bell et al. // *Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 542–552.
- [4]. Kelly S., Tolvanen J.-P. Visual domain-specific modeling: Benefits and experiences of using metaCASE tools // *International Workshop on Model Engineering*, at ECOOP. 2000. URL: http://dsmforum.org/papers/Visual_domain-specific_modelling.pdf.
- [5]. Tolvanen J.-P., Pohjonen R., Kelly S. Advanced tooling for domain-specific modeling: MetaEdit+ // *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*. 2007. URL: <http://www.dsmforum.org/events/DSM07/papers/tolvanen.pdf>.
- [6]. Tolvanen J.-P. and Kelly S. MetaEdit+: defining and using integrated domain-specific modeling languages // *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications / ACM*. New York, NY, USA: ACM, 2009, pp. 819–820.
- [7]. Kelly S., Tolvanen J.-P. *Domain-specific modeling: enabling full code generation*. Hoboken, New Jersey, USA: Wiley-IEEE Computer Society Press, 2008, p. 444.
- [8]. Gronback R. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Stoughton, Massachusetts, USA: Addison-Wesley, 2009, p. 736.
- [9]. Viyovic V., Maksimovic M., Perisic B. Sirius: A rapid development of DSM graphical editor // *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. Los Alamitos, CA, USA: IEEE Computer Society, 2014, pp. 233–238.
- [10]. *Domain-specific development with Visual Studio DSL Tools / S. Cook, G. Jones, S. Kent et al.* Crawfordsville, Indiana, USA: Addison-Wesley, 2007, p. 576.
- [11]. Koznov D. *Process Model of DSM Solution Development and Evolution for Small and Medium-Sized Software Companies // Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International / IEEE*. 2011, pp. 85–92.
- [12]. QReal DSM platform-An Environment for Creation of Specific Visual IDEs / A. Kuzenkova, A. Deripaska, T. Bryksin et al. // *ENASE 2013—Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*. Setubal, Portugal: SciTePress, 2013, pp. 205–211.
- [13]. Rugaber S. *Domain analysis and reverse engineering // White Paper*, January. 1994.
- [14]. Prieto-Diaz R. *Domain analysis for reusability // Software reuse: emerging technology / IEEE Computer Society Press*. 1988, pp. 347–353.
- [15]. Ferre X., Vegas S. An evaluation of domain analysis methods // *4th CASE/IFIP8 International Workshop in Evaluation of Modeling in System Analysis and Design / Citeseer*. 1999, pp. 2–6.
- [16]. Mernik M., Heering J., Sloane A. M. When and how to develop domain-specific languages // *ACM computing surveys (CSUR)*. 2005. Vol. 37, no. 4. P. 316–344.
- [17]. Arango G. *Domain analysis methods // Software Reusability*. 1994, pp. 17–49.
- [18]. DARE: Domain analysis and reuse environment / W. Frakes, R. Prieto, C. Fox et al. // *Annals of Software Engineering*. 1998, Vol. 5, no. 1, pp. 125–141.
- [19]. Taylor R. N., Tracz W., Coglianese L. *Software development using domain-specific software architectures // ACM SIGSOFT Software Engineering Notes*. 1995, vol. 20, no. 5, pp. 27–38.

- [20]. Falbo R. d. A., Guizzardi G., Duarte K. C. An ontological approach to domain engineering // Proceedings of the 14th international conference on Software engineering and knowledge engineering / ACM. 2002, pp. 351–358.
- [21]. Feature-oriented domain analysis (FODA): Tech. Rep.: / K. C. Kang, S. G. Cohen, J. A. Hess et al.: DTIC Document, 1990.
- [22]. Estublier J., Vega G. Reuse and variability in large software applications // ACM SIGSOFT Software Engineering Notes. 2005, vol. 30, no. 5, pp. 316–325.
- [23]. An approach and framework for extensible process support system / J. Estublier, J. Villalobos, L. Anh-Tuyet et al. // Software Process Technology. Springer, 2003, pp. 46–61.
- [24]. The Variability Model of the Linux Kernel / S. She, R. Lotufo, T. Berger et al. // VaMoS. 2010, vol. 10, pp. 45–51.

Usability of AutoProof: a case study of software verification

*Mansur Khazeev <m.khazeev@innopolis.ru>
Victor Rivera <v.rivera@innopolis.ru>
Manuel Mazzara <m.mazzara@innopolis.ru>
Alexander Tchitchigin <a.chichigin@innopolis.ru>
Innopolis University, Software Engineering Lab.
420500, Russia, Innopolis, Universitetskaya Str. 1*

Abstract. Verification tools are often the result of several years of research effort. The development happens as a distributed effort inside academic institutes relying on the ability of senior investigators to ensure continuity. Quality attributes such as usability are unlikely to be targeted with the same accuracy required for commercial software where those factors make a financial difference. In order for such tools to become of widespread use, it is therefore necessary to spend an extra effort and attention on users' experience, and allow software engineers to benefit out of them without the necessity of understanding the mathematical machinery in full detail. In order to put the spotlight on usability of verification tools we chose an automated verifier for the Eiffel programming language, AutoProof, and a well-known benchmark, the Tokeneer problem. The tool is used to verify parts of the implementation of the Tokeneer so to identify AutoProof's strengths and weaknesses, and finally propose the necessary updates. The results show the efficacy of the tool in verifying a real piece of software and automatically discharging nearly two thirds of verification conditions. At the same time, the case study shows the demand for improved documentation and emphasizes the need for improvement in the tool itself and in the Eiffel IDE.

Keywords: static verification; formal specification; Eiffel, Autoproof; Design by Contract

DOI: 10.15514/ISPRAS-2016-28(2)-7

For citation: Khazeev Mansur, Rivera Victor, Mazzara Manuel, Tchitchigin Alexander. Usability of AutoProof: a case study of software verification. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 111-126. DOI: 10.15514/ISPRAS-2016-28(2)-7

1. Introduction

Tools for software verification allow the application of theoretical principles in practice, in order to ensure that nothing bad will ever happen (safety). The extra effort required by the use of these tools is certainly not for free and comes with increased development costs [1]. There is a common belief in industry that developing software

with high level of assurance is too expensive, therefore not acceptable, especially for non safety-critical or financially-critical applications.

Tools and techniques for the formal development of software have played a key role on demystifying this belief. There are several approaches, for instances abstract interpretation and model checking [2], [3] that seek the automation to formally proving certain conditions of systems. However, these techniques tend to verify simple properties only. On the other end of the spectrum, there are interactive techniques for verification such theorem provers [4]. These techniques aim at more complex properties but demand the interaction of users to help the verification.

Nowadays, there are new approaches that aim at finding a good trade-off between both techniques, e.g. auto-active: users are not needed during the verification process (it is automatically performed); they are required instead to provide guidance to the proof using annotations. AutoProof [5], is a static auto-active verifier for functional properties of object-oriented programs. Using AutoProof, users write code and equip classes with contracts and annotations to help the tool to prove certain properties.

The main goal resented in this paper is to provide insights on how easy/difficult is for users (mainly engineers without deep knowledge of formal verification) to use current methodologies and tools for the development of software with high level of assurance, in particular on the use of the AutoProof tool.

Generally, to prove the correctness of a program one needs some mechanisms to express what the program is supposed to do and clearly state it in the specifications that are used later to verify the program. Eiffel programming language natively supports these mechanisms by means of contracts. Eiffel is an object-oriented programming language, which directly implements the concepts of Design-by-Contract (DbC) [1], [6]. The key concept is viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. This is realized equipping methods with pre- and post-conditions, and classes with invariants. The key feature of the Eiffel language is indeed the idea that all the methods might and should contain contracts.

Contracts and annotations used in Eiffel are used by AutoProof to statically verify the consistency of the classes. To demonstrate the usability of the tool, the Tokeneer project [7] was implemented in Eiffel and AutoProof was used to verify the consistency of the code. The Tokeneer project is a system specified and implemented by National Security Agency (NSA). Initially, NSA carried out this challenge to prove that it is possible to develop secure systems rigorously in a cost effective manner. Since its development, it became a testing range for different software development methodologies and verification tools. Results of the project are publicly available. This paper reports on the use of AutoProof to verify an Eiffel implementation of Tokeneer and also reports on how easy/difficult is for users to use the tool, e.g. the burden of helping the tool by means of annotations in the code.

The rest of the paper is organized as follows: Section II introduces the Tokeneer project, Eiffel and the AutoProof tool. Section III describes the methodology used to verify the implementation of the Tokeneer project. Section IV presents empirical

results helping to draw conclusions. Section V is devoted to related work and Section VI concludes and mentions future work.

2. Preliminaries

2.1 The Tokeneer Project

In 2002, with the aim to prove/disprove the common believe in industry that development of software of high level of assurance is too expensive and therefore not feasible, the National Security Agency (NSA) asked Altran to undertake a research project to develop part of an existing secure system, the Tokeneer System, in accordance with Altran's Correctness by Construction development process. The system was specified using Z notation [8] and implemented in Ada [9]. The project was successfully delivered in 2003 within 260 days of effort, and later, in 2008, all the results were made available by NSA to the software development and security communities in order to demonstrate the possibility to develop secure systems in a cost effective manner. It includes the ``Core" Tokeneer ID System Software, test cases derived from the system test specification, ``Support" Tokeneer ID System Software and test tokens and biometric data, project documents. Since the delivery, the Tokeneer project has become a milestone point and a testing range for different verification tools before applying them in industrial projects. Despite the fact that after delivery 4 bugs¹ were found, the system is still deemed to be very secure.

Tokeneer is a secure enclave consisting of a set of system components, some housed inside the enclave and some outside, as depicted in Figure 1.

¹ According to [7]

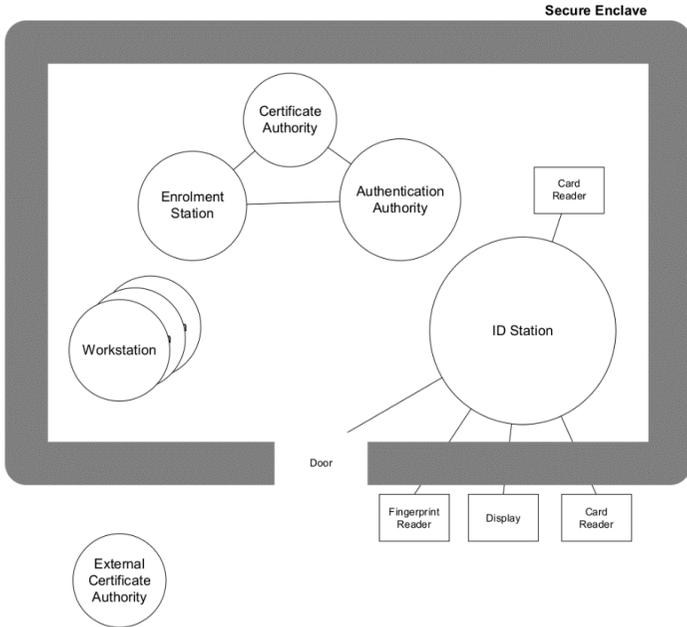


Fig. 1. The Tokeneer System.

The ID Station (TIS) is part of the larger Tokeneer system. It has four connected peripherals, namely, a fingerprint reader, a smartcard reader (users use Tokens - smartcards- as identification), a door and visual display. The objective of the enclave is to ensure that anyone who enters the enclave has a proper access, and no one else can access to the enclave.

In order to ensure the entrance of users to the enclave, TIS implements a series of protocols and checks (the use of smart cards and biometrics) to grant or deny the entrance to it. This paper discusses one of these protocols: the enrollment to the ID Station. The protocol starts in a state where the user is not enrolled. Users can request enrollment and then insert a **FLOPPY** (it retains an internal view of the last data written) for the system to proceed. The system reads the information in the floppy and either fails the enrollment process, in which case takes the process to the initial state, or correctly validates the data in the floppy.

2.2 Eiffel

Eiffel is a real complex object oriented programming language that natively supports Design-by-Contract methodology. Users can specify the behavior of Eiffel classes by equipping them with contracts: pre- and post-conditions and class invariants that are represented as assertions.

```
class
  ACCOUNT
  create make

  feature -- Initialization
    make -- Initialize empty account
    do
      balance := 0
    ensure
      balance_set: balance = 0
    end

  feature -- Access
    balance : INTEGER -- Balance of account

  feature -- Element change
    deposit (amount : INTEGER) -- Deposit 'amount' on account
    require
      amount not negative : amount >= 0
    do
      balance := balance + amount
    ensure
      balance_increased : balance = old balance + amount
    end

    withdraw (amount : INTEGER) --Withdraw 'amount' from account
    require
      enough_balance : amount <= balance
    do
      balance := balance - amount
    ensure
      balance_decreased : balance = old balance - amount
    end

  invariant
    non_negative_balance : balance >= 0
end
```

Fig.2 ACCOUNT Eiffel class.

Figure 2 depicts a reduced implementation of a Bank Account. In Eiffel, creation procedures are listed under the keyword `create`, for class **ACCOUNT**, routine `make` is used as a creation procedure. The class defines a class attribute `balance` to represent the current balance of the account. It also defines two routines (methods), `deposit` and `withdraw`. `deposit` implements a deposit of amount of money to the account and `withdraw` implements withdrawing money. Eiffel encourages software developers to express formal properties of classes by writing assertions. Routine pre-conditions express the requirements that clients must satisfy whenever they call a routine. They are introduced in Eiffel by the keyword `require`. Routine `deposit` imposes a pre-condition on the call, the client must pass as an argument a non-negative number (i.e. **amount_not_negative: amount >= 0**) for the routine to work correctly: a negative value might invalidate the invariant of the class. Routine post-conditions, introduced in Eiffel by the keyword `ensure`, express conditions that the routine (the supplier) guarantees on method exit, assuming the pre-condition. Routine `deposit` guarantees that the balance of the account will be the previous value of the balance (expressed in Eiffel by the keyword `old`: the value on entrance of the routine) plus the amount being deposited. Routine `withdraw` imposes the constraint to the caller that the argument must be less than or equal to the current balance of the account to avoid having

negative value in the balance. The routine ensures that, after execution, the new value of balance will be the value on routine entry minus the amount withdrawn.

A class invariant must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class (public routines). The invariant appears in a clause introduced by the keyword `invariant`. Class **ACCOUNT**'s invariant imposes the restriction that class attribute `balance` can never be negative (i.e. **`non_negative_balance: balance >= 0`**).

2.3 AutoProof

AutoProof [5] is a static verifier of contracts for Eiffel programs. It follows the auto-active paradigm where verification is done completely automated, similar to model checking [3], but users are expected to feed the classes providing additional information in the form of annotations to help the proof. AutoProof identifies software issues without the need of executing the code, therefore opening a new frontier for "static debugging", software verification and reliability, and in general for software quality.

AutoProof verifies the functional correctness of Eiffel classes. It translates Eiffel code to Boogie programs [10] and calls the Boogie tool to generate verification conditions: logic formulas whose validity entails correctness of the input programs. Finally, retrieves the answer back to Eiffel. AutoProof verifies that routines satisfy pre- and post-conditions, maintenance of class invariants, loops and recursive calls termination, integer overflow and non-Void (*null* in other programming languages) references calls. The tool also supports most of the Eiffel language constructs: in-lined assertions such as `check` (*assert* in other programming languages), types, multi-inheritance, polymorphism.

3. Verification of Tokeneer using AutoProof

The Tokeneer project was implemented in Eiffel following the specifications file 41_2.pdf (see [7]) of the Tokeneer System and equipping classes with contracts. This research work encompasses only the enrolment process of the whole Tokeneer System therefore it implements only the entities involved in this process.

One of the main parts of TIS is the **ID_STATION** (see Figure 8) – it describes how all components of the system are related to each other: one of the components is implemented in class **INTERNAL_S** (not shown here) whose responsibility is to keep knowledge of the status of user entry and the enclave and to hold a timeout when relevant; another component is implemented on class **FLOPPY** (not shown here) that retains an internal view of the last data written to the floppy as well as the current data on the floppy. **ID_STATION** displays the configuration data on the screen which is implemented in **SCREEN_DISPLAY**. There are a number of messages that may appear on the TIS screen. The Real World types (described in [7] Specification document, section 2.7.1) of the system such as messages that appear on the display and screen, were implemented all together in class **CONST** which implements the

constants used in the TIS. And finally, a number of interactions between all these entities within the enclave are implemented in **ENCLAVE_OPERS**.

AutoProof does not make any assumptions out of box therefore users are expected to feed the Eiffel classes for a succeed verification.

```
class
  ID_STATION
  -- Some lines were omitted--

create
  make

feature --Initialization
  make
    note
      status : creator
    do
      -- Some lines were omitted --
    end
```

Fig. 3. Initialization of ID STATION Eiffel class.

This is expressed by means of Eiffel's **note** clause. **note** clause enables users to attach addition information to the class that is ignored by the Eiffel's compiler. AutoProof uses this information to succeed in the verification. For instance, AutoProof's annotation **status** defines which procedure is used to initialize newly created objects: Figure 3 depicts procedure `make` with annotation **note** (e.g. **note** `status: creator`) to help Autoproof to discharge the corresponding proof obligations related to creation procedures: the procedure will be called only when an object of this class is being created, AutoProof needs to verify a creation routine only once.

note clause is also used to define models queries to express the abstract state space of a classes. Model queries are part of model-based contracts to help users to write abstract and concise specifications [11], they are used to specify the behavior of the class. In Eiffel, this is specified by adding a **note** clause at the beginning of the class followed with a keyword `model`: and listing one or more attributes of the class. Model queries are also used to describe frame conditions: which allocations are allowed to be modified by procedures.

In Eiffel, frame conditions are listed using the **modify** clause, which lists the model queries that the feature is allowed to modify, as shown in Figure 7 (i.e. **modify_model("current_display", Current)**).

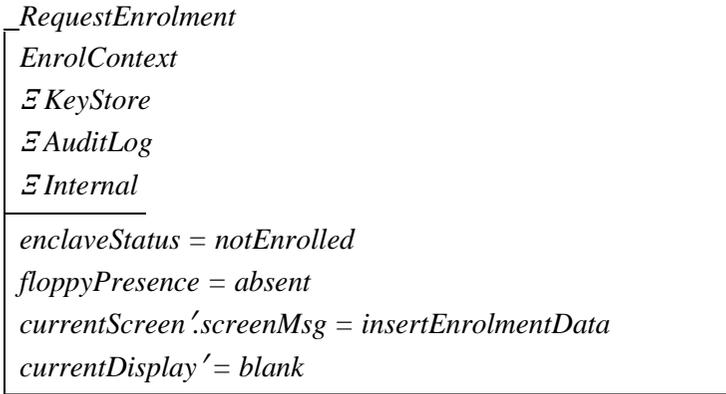


Fig. 4. Z schema of **RequestEnrolment**.

According to **RequestEnrolment** (a Z-schema that is a part of the formal specification of the project Tokeneer), which is presented in Figure 4, requesting enrolment involves **EnrolContext**, **KeyStore**, **AuditLog**, **Internal**. Schemas in Z consist of an upper part, in which some variables are declared, and a lower part, which describes the relationship between values and variables. The notation Ξ indicates an operation in which the state does not change, and the apostrophe indicates the state of the variable after the change [12]. **RequestEnrolment** specifies that the ID station will request enrolment by displaying a request string on the screen and keeping the display blank. This is only possible while there is no Floppy present. Therefore, initially **floppyPresence** = **absent** and **enclaveStatus** set to **notEnrolled**. An ensure clause was used in the creation procedure to guarantee this after the initialization of **ID_STATION** object:

```
make
    -- Some lines were omitted --
ensure
    enclave status = cons floppy.not enrolled
    floppy presence = cons internal.absent
    token removal timeout = 0
end
```

Fig. 5. ensure clause in feature make.

Figure 6 depicts the class invariant for class **ID_STATION**. It states that a message displayed on the display outside the enclave is one of the available from the list of messages (i.e. **constants.display_message.has(current_display)**) and that class attribute constants is attached to an object (i.e. **constants != Void**).

```
invariant
  constants.display message.has(current display)
  constants /= Void
```

Fig. 6. Invariants of ID STATION Eiffel class.

Figure 7 shows the implementation of procedure `set_current_display`. Its first pre-condition was added to satisfy the invariant ensuring that argument `v` belongs to the allowed displayed messages. The second pre-condition restricts the procedure to change values only to model query `current_display`.

```
feature -- Element Change
  set_current_display (v: STRING)
  require
    constants.display message.has(v)
    modify model("current display", Current)
  do
    current display := v
  ensure
    current display = v
  end
```

Fig. 7. Feature equipped with modify clause.

Figure 8 shows the final version of class `ID_STATION`: with the respective annotations for AutoProof to successfully verify the class. In class `ID_STATION`, class attributes `current_screen` and `current_display` implements the physical screen and display, respectively, of the enclave.

```
class
  ID STATION
  -- Some lines were omitted --
create
  make

feature -- Initialization
  make
  note
    status: creator
  do
    create constants
    current display := constants.blank
    create current screen.make

    create cons floppy
    enclave status := cons floppy.not enrolled
    token removal timeout := 0
```

```
        create cons internal
        floppy presence := cons internal.absent
    ensure
        enclave status = cons floppy.not enrolled
        floppy presence = cons internal.absent
        token removal timeout = 0
    end

feature    -- Element Change
    set_current_display(v: STRING)
        require
            constants.display_message.has(v)
            modify model("current display", Current)
        do
            current display := v
        ensure
            current display = v
        end

feature    -- Access
    constants : CONST
    current screen : SCREEN DISPLAY
    current display : STRING

invariant
    constants.display_message.has(current display)
    constants /= Void
end
```

Fig. 8. Verified ID STATION Eiffel class.

4. Empirical Results

The usability of a verification tool cannot be considered in isolation and, in particular, cannot be hived off by the effectiveness of the tool itself. First, as a general observation, the cost of using an instrument can only be justified by its return, which can ultimately be linked to financial consideration by top management. Second, and this aspect is less general and more peculiar to the auto-active verification approach, a tool like AutoProof is as much effective and usable as is its ability to discharge verification conditions completely automatically, without feeding the code of annotation overhead or requiring particular tweaking. Finally, the necessity for users to add further annotations and dedicate extra effort (and considerable time) is, by itself, an obstacle to adoption and (technically) a usability issue. Verification tools should require minimal annotational effort and give valuable feedback when verification fails.

The case study analyzed in this paper presented good results in term of automatic discharge of verification conditions, though not comparable to others seen in literature [13].

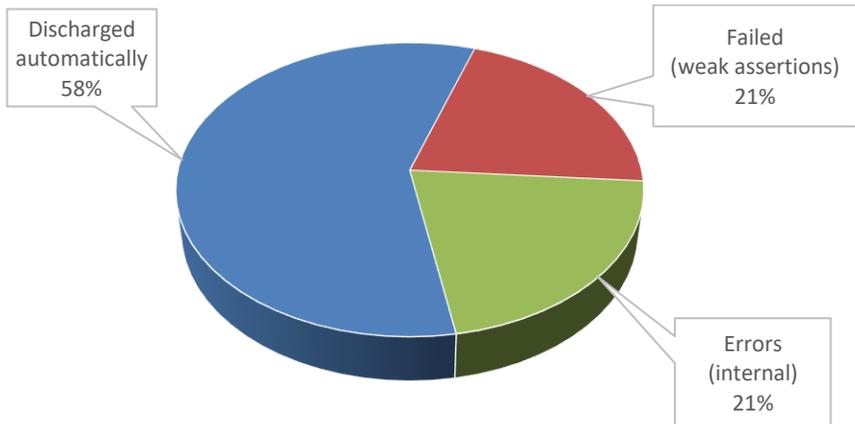


Fig. 9. Verifications results.

In total there were 38 generated proofs. Of these, 22 (58%) were discharged automatically (see figure 9), 8 (21%) could not be satisfied, and the rest (21%) failed due to internal errors, which in our case were basically caused by the attempt to create objects in the contract, and that is not allowed by the tool. As observed before, the success of verification is unsurprisingly linked to the complexity of programs [13]. Previous literature mostly dealt with students users and university projects. The use of Tokeneer as a benchmark demands for detailed comparisons with different verification efforts (for example, [14]).

5. Related Work

Formal/mathematical notations have existed for a long time and have been used to specify and verify systems. Examples are process algebras [15], specification languages like Z [16], B [17] and Event-B [18]. The Vienna Development Method (VDM) is one of the earliest attempts to establish a formal method for the development of computer systems [19]. A survey of these (and others) formalisms can be found in [20] while a discussion on the methodological issues of a number of formal methods is presented in [21].

All these approaches (and others described in the literature) still leave an open issue, i.e., they are built around strict formal notations which affect the development process from the very beginning. These approaches demonstrate a low level of flexibility. To overcome this problem, a seamless methodological connection built on top of a portfolio of diverse notations and methods is presented in [22]. Another approach is presented in [14], [23] using [24], where users start the development of system from a strict formal notation (i.e. Event-B), to then automatically translate it to Java code with JML [25] specifications embedded (following Design-by-Contract methodology). Even though this approach enables users with less mathematical

expertise to work on formal development, it does not give a seamlessly methodology for the development as presented in this paper.

On the other side, Design-by-contract [6] when combined with AutoProof technology offers the pros of both rigorous methodologies and supporting tools able to semi-automate the process. Before this to be available for the average developer it is however necessary to improve the users' experience. A comparison between different approaches (for example Event-b/Rodin and Design-by-contract/AutoProof) is beyond the scope of this paper and it is left as future work.

6. Conclusion

AutoProof allows for “static debugging”, i.e. debugging becomes possible without the need of executing the program. The most effective way to release correct software is a combination of static debugging and traditional run-time debugging. Being all human activities (therefore including programming and testing itself) error-prone, there is no magic or free lunches out there. Abandoning testing and adopting a proof-oriented approach does not make miracles, debugging remains a trial-and-error long and laborious process. AutoProof does not change the rules of the game: developers will have to try, observe the results and make changes as a consequence. A proof-oriented approach does not make the process smoother and necessarily simpler. However, it makes it more accurate and robust, therefore effective. Adjustment can be now focused on the implementation side (possibly sinergically with run-time debugging), on the specification side (the contracts used to annotate the code as integral part of the code itself), or in the proof itself (fine-tuning may be necessary for AutoProof and its behind-the-curtains machinery to be able to prove correctly). All this comes with a cost: the willingness and ability of the user to use extra tools and being able to master them, and possibly invest extra time in the process. On the other side, it is necessary for the tools to be simple to master and to provide intelligible feedback.

The Tokeneer project case study showed the efficacy of AutoProof in verifying a real piece of software, the complexity of which can be compared not only with most of the commercial Off-the-Shelf software, but also with safety and financial-critical applications, both in terms of computational logic and architectural organization. AutoProof is capable to verify industrial software and may well be adopted in commercial companies and its use injected into the development process. However, some obstacles have been identified that could prevent its broader adoption.

As result of an academic effort, documentation is not at par with commercial software, in particular for what concerns the size of the library of correctly verified examples: tutorials on the official website are quite useful, but not enough. On top of this, the tool itself has limitations. First, existing implementations need to be modified in order to be verified. This would represent an unsurmountable obstacle in most institutions since the overall cost of code adaptation may overrun the saves occurring to the testing phase. This consideration may be different, however, for safety-critical

systems. Second, the Eiffel IDE - necessary for functioning - calls for increased stability and robustness.

7. Acknowledgments

We would like to thank Innopolis University for logistic and financial support, and the laboratories of Software Engineering (SE) and Service Science and Engineering (SSE) for the intellectual engagement and vivid discussions.

References

- [1]. B. Meyer, *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Publishing Company, Incorporated, 1 ed., 2009.
- [2]. P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’77*, (New York, NY, USA), pp. 238–252, ACM, 1977.
- [3]. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [4]. D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
- [5]. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, “AutoProof: Auto-active functional verification of object-oriented programs,” in *21st International Conference, TACAS 2015, London, UK, April 11-18, 2015*. Proceedings, pp. 566–580, 2015.
- [6]. B. Meyer, *Object-oriented software construction, ch. 11: Design by Contract: building reliable software*. Prentice Hall PTR, 1997.
- [7]. AdaCore, “Tokeneer.” <http://www.adacore.com/sparkpro/tokeneer/download>, accessed in April 2016.
- [8]. J.-R. Abrial, S. Schuman, and B. Meyer, “Specification Language,” in *On the Construction of Programs*, R. M. McKeag and A. M. Macnaghten, editors, pp. 343–410, Cambridge University Press, 1980.
- [9]. J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [10]. K. R. M. Leino, “This is boogie 2,” tech. rep., June 2008.
- [11]. N. Polikarpova, C. A. Furia, and B. Meyer, “Specifying reusable components,” in *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE’10)* (G. T. Leavens, P. O’Hearn, and S. Rajamani, eds.), vol. 6217 of *Lecture Notes in Computer Science*, pp. 127–141, Springer, August 2010.
- [12]. J. Spivey, “An introduction to Z and formal specifications,” *Software Engineering Journal*, 1989.
- [13]. C. A. Furia, C. M. Poskitt, and J. Tschannen, “The AutoProof verifier: Usability by non-experts and on standard code,” in *Proc. Formal Integrated Development Environment (F-IDE 2015)*, vol. 187, pp. 42–55, *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2015.
- [14]. V. Rivera, S. Bhattacharya, and N. Cataño, “Undertaking the tokeneer challenge in Event-B,” To appear in *4th FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2016.

- [15]. J. C. M. Baeten, "A brief history of process algebra," *Theor. Comput. Sci.*, vol. 335, no. 2-3, pp. 131–146, 2005.
- [16]. J. Abrial, S. A. Schuman, and B. Meyer, "Specification language," in *On the Construction of Programs*, pp. 343–410, 1980.
- [17]. J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
- [18]. J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
- [19]. C. B. Jones, *Software Development: A Rigorous Approach*. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980.
- [20]. "On modelling and analysis of dynamic reconfiguration of dependable real-time systems," in *Proceedings of the 2010 Third International Conference on Dependability, DEPEND '10*, (Washington, DC, USA), pp. 173–181, IEEE Computer Society, 2010.
- [21]. M. Mazzara, "Deriving specifications of dependable systems: toward a method," in *Proceedings of the 12th European Workshop on Dependable Computing, EWDC, 2009*.
- [22]. R. Gmehlich, K. Grau, A. Iliassov, M. Jackson, F. Loesch, and M. Mazzara, "Towards a formalism-based toolkit for automotive applications," *1st FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2013.
- [23]. V. Rivera, N. Cataño, T. Wahls, and C. Rueda, "Code generation for Event-B." To appear in *International Journal on STTT*, 2016.
- [24]. V. Rivera and N. Cataño, "Translating Event-B to JML-Specified Java programs," in *29th ACM SAC*, (Gyeongju, South Korea), March 24-28, 2014.
- [25]. G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006.

Применимость AutoProof: учебный пример верификации ПО

Мансур Хазеев <m.khazeev@innopolis.ru>

Виктор Ривера <v.rivera@innopolis.ru>

Мануэль Маццара <m.mazzara@innopolis.ru>

Александр Чичигин <a.chichigin@innopolis.ru>

Университет Иннополис,

420500, Россия, респ. Татарстан, г. Иннополис, ул. Университетская, д.1.

Аннотация. Очень часто инструменты статической верификации являются результатом многолетних научно-исследовательских работ. По этой причине разработки ведутся с распределением задач внутри учебных заведений и с расчетом на способность старших исследователей обеспечивать её непрерывность. В такой ситуации некоторые атрибуты качества, такие как удобство и простота использования программного обеспечения, чаще всего, не рассматриваются на должном уровне, что плохо сказывается на возможности дальнейшей коммерциализации продукта. Для того, чтобы данные инструменты получили широкое применение необходимо обратить внимание и направить усилия при дальнейшей доработке на упрощение механизма взаимодействия пользователей с приложением, для того, чтобы дать инженерам программного

обеспечения возможность пользоваться инструментом без необходимости полного понимания всех математических механизмов во всех деталях. Для того, чтобы обратить внимание общественности на важность удобства использования инструментов верификации, мы применили инструмент AutoProof к хорошо известному проекту Tokeneer. Данный инструмент использовался для верификации части имплементации реального проекта Tokeneer, в ходе чего были выявлены сильные и слабые стороны AutoProof, и, как результат, был составлен список необходимых улучшений. Результат данной работы иллюстрирует эффективность инструмента при верификации фрагмента реального программного обеспечения: он позволил автоматически проверить практически две трети всех свойств. В то же время, данное исследование показало потребность в доработке документации к данному инструменту и подчеркнуло необходимость улучшения как самого инструмента, так и среды Eiffel IDE.

Ключевые слова: статическая верификация, формальная спецификация, Eiffel, Autorproof, контрактное программирование

DOI: 10.15514/ISPRAS-2016-28(2)-7

Для цитирования: Хазеев Мансур, Ривера Виктор, Маццара Мануэль, Чичигин Александр. Применимость AutoProof: учебный пример верификации ПО. *Труды ИСП РАН*, том 28, вып. 2, 2016 г., стр. 111-126 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-7

Список литературы

- [1]. B. Meyer, *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Publishing Company, Incorporated, 1 ed., 2009.
- [2]. P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’77*, (New York, NY, USA), pp. 238–252, ACM, 1977.
- [3]. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [4]. D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
- [5]. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, “AutoProof: Auto-active functional verification of object-oriented programs,” in *21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 566–580, 2015.
- [6]. B. Meyer, *Object-oriented software construction*, ch. 11: Design by Contract: building reliable software. Prentice Hall PTR, 1997.
- [7]. AdaCore, “Tokeneer.” <http://www.adacore.com/sparkpro/tokeneer/download>, accessed in April 2016.
- [8]. J.-R. Abrial, S. Schuman, and B. Meyer, “Specification Language,” in *On the Construction of Programs*, R. M. McKeag and A. M. Macnaghten, editors, pp. 343–410, Cambridge University Press, 1980.
- [9]. J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [10]. K. R. M. Leino, “This is boogie 2,” tech. rep., June 2008.

- [11]. N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10) (G. T. Leavens, P. O'Hearn, and S. Rajamani, eds.), vol. 6217 of Lecture Notes in Computer Science, pp. 127–141, Springer, August 2010.
- [12]. J. Spivey, "An introduction to Z and formal specifications," *Software Engineering Journal*, 1989.
- [13]. C. A. Furia, C. M. Poskitt, and J. Tschannen, "The AutoProof verifier: Usability by non-experts and on standard code," in Proc. Formal Integrated Development Environment (FIDE 2015), vol. 187, pp. 42–55, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.
- [14]. V. Rivera, S. Bhattacharya, and N. Cata˜ no, "Undertaking the tokeneer challenge in Event-B," To appear in 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2016.
- [15]. J. C. M. Baeten, "A brief history of process algebra," *Theor. Comput. Sci.*, vol. 335, no. 2-3, pp. 131–146, 2005.
- [16]. J. Abrial, S. A. Schuman, and B. Meyer, "Specification language," in *On the Construction of Programs*, pp. 343–410, 1980.
- [17]. J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
- [18]. J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
- [19]. C. B. Jones, *Software Development: A Rigorous Approach*. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980.
- [20]. "On modelling and analysis of dynamic reconfiguration of dependable real-time systems," in Proceedings of the 2010 Third International Conference on Dependability, DEPEND '10, (Washington, DC, USA), pp. 173–181, IEEE Computer Society, 2010.
- [21]. M. Mazzara, "Deriving specifications of dependable systems: toward a method," in Proceedings of the 12th European Workshop on Dependable Computing, EWDC, 2009.
- [22]. R. Gmehlich, K. Grau, A. Iliasov, M. Jackson, F. Loesch, and M. Mazzara, "Towards a formalism-based toolkit for automotive applications," 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2013.
- [23]. V. Rivera, N. Cata˜ no, T. Wahls, and C. Rueda, "Code generation for Event-B." To appear in *International Journal on STTT*, 2016.
- [24]. V. Rivera and N. Cata˜ no, "Translating Event-B to JML-Specified Java programs," in 29th ACM SAC, (Gyeongju, South Korea), March 24-28, 2014.
- [25]. G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006.

Certified Grammar Transformation to Chomsky Normal Form in F*

M.I. Polubelova <polubelovam@gmail.com>

S.N. Bozhko <gkerfimf@gmail.com>

S.V. Grigorev <Semen.Grigorev@jetbrains.com>

Saint Petersburg State University,

7/9, Universitetskaya Nab., St. Petersburg, 199034, Russia

Abstract. Certified programming allows to prove that the program meets its specification. The check of correctness of a program is performed at compile time, which guarantees that the program always runs as specified. Hence, there is no need to test certified programs to ensure they work correctly. There are numerous toolchains designed for certified programming, but F* is the only language that support both general-purpose programming and semi-automated proving. The latter means that F* infers proofs when it is possible and a user can specify more complex proofs if necessary. We work on the application of this technique to a grammarware research and development project YaccConstructor. We present a work in progress verified implementation of transformation of Context-free grammar to Chomsky normal form, that is making progress toward the certification of the entire project. Among other features, F* system allows to extract code in F# or OCaml languages from a program written in F*. YaccConstructor project is mostly written in F#, so this feature of F* is of particular importance because it allows to maintain compatibility between certified modules and those existing in the project which are not certified yet. We also discuss advantages and disadvantages of such approach and formulate topics for further research.

Keywords: certified programming; F*; program verification; context-free grammar; Chomsky normal form; grammar transformation; dependent type; refinement type

DOI: 10.15514/ISPRAS-2016-28(2)-8

For citation: Polubelova M.I., Bozhko S.N., Grigorev S.V. Certified Grammar Transformation to Chomsky Normal Form in F*. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 127-138. DOI: 10.15514/ISPRAS-2016-28(2)-8

1. Introduction

Certified programming is designed for proving that a program meets its specification. For this technique, proof assistants or interactive theorem prover are used [1], what allows to check correctness of the program at compile time and guarantees that the program always works according to its specification. Classical fields of application

of certified programming are the formalization of mathematics, security of cryptographic protocols and the certification of properties of programming languages. There are two approaches to certified programming [2]. In the classical approach the program, its specification, and the proof that the program meets its specification are written separately, as different modules. Such technique costs too much to be applied in software development. More effective approach is to combine program, its specification, and the proof in one module by means of dependent types [3], [4]. The most well-known toolchains for program verification are Coq [5], Agda [6], F* [7] and Idris [8]. Among them, F* is the only language which supports semi-automated proving and general-purpose programming [9].

As a proof assistant, F* allows to formulate and prove properties of programs by using lemmas and enriching types. F* not only infers types of functions, but also the properties of its computations such as purity, statefulness, divergence. For example, consider the following function:

```
val f : (int -> Tot int) -> int -> Tot int
let f g x = g x
```

The keyword `val` indicates that we declare a function `f` and its type signature. The function `f` takes a function `g` and an integer value, as arguments. The effect of computation `Tot t` is used for total expression, which always evaluates to a `t`-typed result without entering an infinitive loop, throwing exception or other side effects. Hence, one can prove for some programs not only their properties and restrictions on the types, but also guarantee their termination and that a result has assigned type.

We apply certified programming using F* to a grammarware research and development project YaccConstructor (YC) [10], [11]. YC is a tool for parser construction and grammar processing. Also it is a framework for research and development of lexer and parser generators and other grammarware for .NET platform. The verification of its programs covers the topic of parser correctness: how to obtain formal evidence that a parser is correct with respect to its specification [12]. In this article, we consider only one algorithm implemented in YC, namely the transformation of context free grammar to Chomsky normal form, that is a small step towards the certification of entire project. The algorithm of grammar normalization consists of four transformations. We prove totality of each of them and establish an order of their application to the input grammar. In addition, we describe the peculiarities of evaluation F* as a proof assistant and formulate topics for further research.

2. Overview of F*

We use a functional programming language F* [7] for program verification. It is the only language that support semi-automated proving and general-purpose programming [9]. The main goal of this tool is to span the capabilities of interactive proof assistants like Coq [5] and Agda [6], general-purpose programming languages like OCaml and Haskell, and SMT-backed semi-automated program verification tools like Dafny [13] and WhyML [14].

Type system of F* includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus [15], [16]. These features allow expressing precise and compact specification for programs [7].

Dependent function type has the following form $x_1:t_1 \rightarrow \dots \rightarrow x_n:t_n[x_1..x_{n-1}] \rightarrow E t [x_1..x_n]$. Each of a function's formal parameters are named x_i and each of these names in the scope to the right of the first arrow that follows it. The notation $t[x_1..x_m]$ indicates that the variables $x_1..x_m$ may appear free in t .

Refinement type has a form $x:t\{\text{phi}(x)\}$. It is a sub-type of t restricted to those expressions of type t that satisfy a predicate $\text{phi}(e)$.

In addition to inferring a type, F* also infers side effects of an expression such as exceptions and state. The following are the most significant monadic effects.

- **Tot** t – the effect of a computation that guarantees evaluation to a t -typed result, without entering an infinite loop, throwing an exception, reading or writing the program's state.
- **ML** t – the effect of a computation that may have arbitrary effects, but if some result is computed, then it is always of type t .
- **Dv** t – the effect of a computation that may diverge.
- **ST** t – the effect of a computation that may diverge, read, write or allocate on a heap.
- **Exn** t – the effect of a computation that may diverge or raise an exception.

The effects $\{\text{Tot}, \text{Dv}, \text{ST}, \text{Exn}, \text{ML}\}$ are arranged in a lattice, with **Tot** at the bottom, **ML** at the top, and with **ST** unrelated to **Exn**.

There are two main approaches to prove properties: either by enriching the type of a function (intrinsic style) or by writing a separate lemma about it (extrinsic style). You can see an example of the first approach below; keyword `val` indicates declaration of a value and its type signature.

```
val append: l1:list 'a -> l2:list 'a
    -> Tot (l:list 'a{length l=length l1+length l2})
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | hd :: tl -> hd :: append tl l2
```

The following example demonstrates extrinsic style, in which the formula after keyword `requires` is the pre-condition of the lemma, while the one after keyword `ensures` is its post-condition.

```
val append_len: l1:list 'a -> l2:list 'a
    -> Lemma (requires True)
    (ensures (length (append l1 l2)=
    length l1 + length l2)))
let rec append_len l1 l2 =
```

```
match l1 with
| [] -> ()
| hd::tl -> append_len tl l2
```

There is no general rule which style of proving to use, but in some cases it is impossible to prove a property of a function directly in its types and one has to use a lemma.

When defining lemmas or expressions that are total, F* automatically proves their termination. The termination check is based on a well-founded relation. For natural numbers, F* uses classical decreasing metric, for inductive types – the sub-term ordering, for recursive function, it requires the tuple of parameters to be in decreasing lexicographic ordering. The last case can be overridden with using clause `decreases % $[x_1..x_n]$` , which explicitly chooses a lexicographic ordering on arguments.

To conclude, one can use F* to write effectful programs, specify them using dependent and refinement types, verify them using an SMT solver or providing interactive proofs. Programs written in F* can be translated to OCaml or F# for further execution.

3. Verification of transformation of CFG to CNF

In this section we briefly describe some necessary aspects of the theory of formal languages, sketch a totality proof for one of grammar transformations to Chomsky normal form in F*, and formulate some advantages and disadvantages of this approach.

3.1 Context-free grammar and Chomsky normal form

In this section we give basic definitions and formulate a theorem that helps us to verify the implemented algorithm of a transformation of context-free grammar to Chomsky normal form.

In formal language theory, a **context-free grammar** (CFG) is a formal grammar in which every production rule is of the form $A \rightarrow \alpha$, where A is single nonterminal symbol and α is a string of terminals and/or nonterminals (α can be empty).

Context-free grammar is said to be in **Chomsky normal form** (CNF) if all of its production rules are of the form:

- $A \rightarrow BC$
- $A \rightarrow a$
- $S \rightarrow \varepsilon$,

where A , B and C are nonterminal symbols, a is a terminal symbol, S is the start nonterminal, and ε denotes the empty string. Also, neither B nor C may be the start symbol, and the third production rule can only appear if ε is in $L(G)$, namely, the language produced by the context-free grammar G .

Context-free grammars given in Chomsky normal form are very convenient to use. It is often assumed that either CFGs are given in CNF from the beginning or there is an

intermediate step of normalization. Having a certified implementation of normalization for CFGs enables us to stop thinking in terms of CFG and consider grammar in CNF without losing guarantees of correctness.

CFG normalization theorem: There is an algorithm which converts any CFG into an equivalent one in Chomsky normal form.

The full normalization transformation for a CFG is a composition of the following constituent transformations.

- Replace all rules $A \rightarrow X_1X_2..X_k$, where $k \geq 3$ with rules $A \rightarrow X_1A_1$, $A_1 \rightarrow X_2A_2$, ..., $A_{k-2} \rightarrow X_{k-1}X_k$, where A_i are "fresh" nonterminals.
- Eliminate all ε -rules.
- Eliminate all chain rules.
- For each terminal a , add a new rule $A \rightarrow a$, where A is a "fresh" nonterminal and replacing a in the right-hand sides of all rules with length at least two with A .

3.2 Verification with F*

Our purpose is to verify a core YaccConstructor (YC) using F*. YC is an open source modular tool for research in lexical and syntax analysis and its main development language is F# [17]. In this paper we consider only a verification of normalization grammar algorithm [18] which is defined in a following way:

```
let toCNF (ruleList: Rule.t<_,_>list) =
    ruleList
        |> splitLongRules
        |> deleteEpsRules
        |> deleteChainRules
        |> renameTerm
```

The function `toCNF` is a composition of the four transformations mentioned. Notice that the order of rules execution is important. The first rule must be executed before the second, otherwise normalization time may increase to $O(2^n)$. The third rule follows the second, because elimination of ε -rules may produce new chain rules. Also, the fourth rule must be executed after the second and the third as they can generate useless symbols.

F*, as a proof assistant, allows to formulate and prove properties of function of interest using lemmas or enriching types. For example, in F# function $(f (x:int) = 2*x)$ is inferred to have type $(int \rightarrow int)$, while in F* we infer $(int \rightarrow \text{Tot } int)$. This indicates that $(f (x:int) = 2*x)$ is a pure total function which always evaluates to `int`. A lemma is a ghost total function that always returns the single unit value `()`. When we specify a total function, we have to prove totality of every nested function, because F* supports only high-level annotations. In others words, we cannot add annotation for a nested function. Therefore, to prove totality of a function

containing nested functions, we need to lift all nested functions up and explicitly prove totality of these functions.

We describe each function of interest in an individual module to avoid namespace collision. We use module architecture similar to YC architecture. Module `IL` contains type constructors for describing productions of a grammar. Module `Namer` contains a function to generate new names. Finally, we created individual modules for each transformation and a separate, main, module which contains the definition of `toCNF` transformation.

We implemented all the transformations in F* [19], but in this paper we consider only one of them, namely `SplitLongRules`, which eliminates long rules. Firstly, we describe all the helpers we need, prove their totality and other necessary properties, and then explain why this transformation is correct.

In the first transformation, it is necessary to create new nonterminals, so we need a function to supply them. The function `Namer.newSource` defined below is used.

```
val newSource: n:int -> oldSource:Source -> Tot Source
let newSource n old =
    (old with text = old.text^(string_of_int n))
```

Integer `n` is equal to the size of the list of rules which we have at the moment of function `Namer.newSource` call. Obviously, function `Namer.newSource` is injective. In other words, unique rule names remain unique after application `splitLongRules`.

Some necessary helpers are grouped in `TransformAux` module: for example, functions `createRule` and `createDefaultElem`, which take some arguments and return `Rule` and `Elem` respectively. `Elem` is the right part of the rule if the latter is a sequence. Also, we define follow one simple function which returns the length of the right part of the rule.

```
val lengthBodyRule: Rule 'a 'b -> Tot int
let lengthBodyRule rule =
    List.length (match rule.body with
        | PSeq(e, a, l) -> e
        | _ -> [ ])
```

The most interesting function is `cutRule`. It takes a rule and a list-accumulator as an input. If the length of the right-hand side of the rule is less or equal to 2, `cutRule` only renames a nonterminal to avoid name collision. Otherwise, it is necessary to create new nonterminal B_{k-2} , cut off last two elements $X_{k-1}X_k$, pack them into a new rule $B_{k-2} \rightarrow X_{k-1}X_k$, and then add the nonterminal to the end of the long rule. Then the new rule is added to the accumulator and the function `cutRule` is recursively called on the new rule and accumulator. This way, we reduce our rule by one. Function signature is the following.

```
val cutRule: rule:(Rule 'a 'b)
-> resRuleList:(list (Rule 'a 'b))
-> Tot (list (Rule 'a 'b))
```

```
(decreases %[lengthBodyRule rule])
```

There are some peculiarities in our implementation, which are worth mentioning. One of them is the representation of the right-hand side of the rules by lists. In the algorithm, we need to cut off two last elements of a rule, so we carry out the following steps.

```
let revEls = List.rev elements
let cutOffEls = [List.Tot.hd revEls;
                 List.Tot.hd (List.Tot.tl revEls)]
```

Functions `List.hd` and `List.tl` from a standard library are not defined for an empty list, so they cannot be considered total, which limits their usage in our code. In F* there is a module `List.Tot` which provide proper total analogs of the functions mentioned. We only provide their signature here.

```
val hd: l:list 'a{is_Cons l} -> Tot 'a
val tl: l:list 'a{is_Cons l} -> Tot (list 'a)
```

Predicate `is_Cons` takes a list as an input and returns `false` if it is empty, otherwise it returns `true`.

If function `List.Tot.hd` is applied to a list, nonemptiness of which is not clear from the context, F* reports a type mismatch. A pleasant peculiarity of F* is that in some rare cases it can derive necessary properties. In our implementation of the transformation, only the rules which have more than two symbols in the right-hand side are split. In this case F* is able to automatically derive required type, so we can choose two elements. It can be illustrated with the following example.

```
// lst has type list int and can be empty
assume val lst: list int
// f takes only nonempty lists
assume val f: lst:(list int){is_Cons lst} -> Tot int
assume val g: lst:(list int) -> Tot int

//Ok
let test1 (lst:list int) =
  if List.length lst >= 1
  then f lst
  else g lst

//Fail: subtyping check failed
let test2 (lst:list int) =
  if List.length lst >= 0
  then f lst
  else g lst
```

At the same time, we have to prove and explicitly add even simple lemmas for functions. For example, if list `lst` has type `(list 'a){is_Cons lst}`, then F* can only infer that `(List.rev lst)` has type `(list 'a)`. This can be easily fixed

with the instruction `SMTPat`. In addition, we should formulate the following lemma which proof can be derived automatically by F*. The following code makes `List.rev` preserve information about the length:

```
val rev_length: l:(list 'a)
  -> Lemma (requires true)
  (ensures (List.length (List.rev l) = List.length l))
  [SMTPat (List.rev l)]
```

We proved totality of all the nested functions. Now we want to prove termination of the general one. In our case, it is sufficient that the length of the rule strictly decreases on each recursive call and we are not interested in the length of the accumulator. To prove this we must explicitly specify that after applying `List.Tot.tl` to a list, its length reduces by 1. So, we must use the same method as we used before.

```
val tail_length : l:(list 'a){is_Cons l}
  -> Lemma (requires True)
  (ensures(List.length (List.Tot.tl l)=(List.length l)-1))
  [SMTPat (List.Tot.tl l)]
```

With this sufficient information F* has to conclude that `cutRule` is total.

Function `splitLongRules` takes a list of rules and applies `cutRule` to each rule, then concatenates all the results and returns the combined list.

```
val splitLongRules: list (Rule 'a 'b)
  -> Tot (list (Rule 'a 'b))
let splitLongRules ruleList =
  List.Tot.collect
    (fun rule -> cutRule rule [ ]) ruleList
```

Totality is proved automatically by F*.

Previously we proved totality of our transformation, but we had not mentioned properties of the rules we get after applying `splitLongRules`. We add restriction on the type of function, which guarantees the necessary property of the result, instead of proving the lemma about these properties. The function signature now look like this.

```
val cutRule: rule:(Rule 'a 'b)
  -> acc:(list (Rule 'a 'b))
  {List.Tot.for_all (fun x->lengthBodyRule x<=2) acc}
  -> Tot (res:(list (Rule 'a 'b))
  {List.Tot.for_all (fun x->lengthBodyRule x<=2) res})
  (decreases %[lengthBodyRule rule])

val splitLongRules:list (Rule 'a 'b)
  -> Tot (res:list (Rule 'a 'b)
  {List.Tot.for_all (fun x->lengthBodyRule x<=2) res})
```

Now we have almost everything we need to prove such properties. We have to provide some additional information so that F* could check arguments type when `collect`

is recursively called. At the moment of cutting the rule off, we should fix the length in the type of the cut part. For this purpose we have to define a function to take our list and return part with that type. Further, we have to prove lemma that states that concatenation of two lists with short rules is the list with short rules. After that F* accepts type correctness.

3.3 Advantages and disadvantages of F*

In this section we want to outline some advantages and disadvantages of F* programming language. In F#, even if there is no doubt that some functions are correct, an incorrect result may still be obtained by applying them in a wrong order. F* can prevent such situations, if a programmer specifies the properties demanded from an input data in a function type. For instance, `deleteChainRules` should only be applied after deleting epsilon rules. This can be ensured by specifying the following signature of `deleteChainRules` function (where predicate `has_no_eps_rules` checks that there are no epsilon rules).

```
val deleteChainRules:
  ruleList:(list (Rule 'a 'b))
  {has_no_eps_rules ruleList}
  -> Tot (list (Rule 'a 'b))
```

Unfortunately, there are some disadvantages of F* which we want to emphasize. First of all, it does not provide any – even primitive – support for object-oriented features. One can use structures instead of classes, but it complicates development. For example, we had to explicitly create functions for constructing elements of types. In other words, rather than create class `Person` with constructors and methods:

```
let person = new Person("Nick", 27)
```

One has to write code in a rather cumbersome manner:

```
let new_Person name age = {name=name; age=age}
```

There is a special construct in many functional languages which checks whether some property holds for a value. Such construct is called `guard` in Haskell and `when` in OCaml and F# and is often used in pattern matching to simplify code. Unfortunately, it is not supported in F* and one can only hope that it will be supported in the latter language versions.

Lastly, we can notice poor quality of error reporting in F* which sometimes makes it hard to understand why proofs do not pass correctness tests.

4. Conclusion and future work

We presented a verification of one of transformations of context-free grammar to the Chomsky normal form. We proved totality of each function implemented, as this property guarantees that computations always terminate and do not have side effects, which is useful in practice. Although for a complete proof of the correctness of the grammar transformation we still need to prove the equivalence of the original and

the resulting grammar, we have already obtained interesting results. We can specify an input and an output of functions – using refinement and dependent types – that allows us to establish application order of the four transformations, by means of which correctness of the whole transformation is guaranteed.

We use programming language F* to verify the implementation, but to be able to execute it one needs to extract it to OCaml or F# and then compile it using the OCaml or F# compiler respectively. At the moment, the mechanism of extraction code from F* to F# omits casts, erases dependent types, higher rank polymorphism and ghost computation [9]. These features are very important and lack of them breaks the consistency and correctness of programs within the target language. F* is currently under active development, and implementation of the extraction mechanism which copes with the above shortcoming is actual topic of our further research.

References

- [1]. H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 2009, vol. 34, issue 1. pp. 3–25. DOI: 10.1007/s12046-009-0001-5.
- [2]. A. Chlipala. *Certified programming with dependent types*. MIT Press, 2013, 440 p.
- [3]. T. Sheard, A. Stump, S. Weirich. Language-based verification will change the world. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 343–348. DOI: 10.1145/1882362.1882432.
- [4]. E. Tanter, N. Tabareau. Gradual certified programming in Coq. *Proceedings of the 11th Symposium on Dynamic Languages*, 2015. pp. 26–40. DOI: 10.1145/2816707.2816710.
- [5]. The Coq proof assistant. June 2016. <https://coq.inria.fr/>
- [6]. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
- [7]. The F* homepage. June 2016. <https://www.fstar-lang.org/>
- [8]. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 2013, vol. 23, n. 5, pp. 552–593. DOI: 10.1017/S095679681300018X.
- [9]. N. Swamy, C. Hritcu, C. Keller. Dependent Types and Multi-Monadic Effects in F*. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655
- [10]. The YaccConstructor homepage. June 2016. <https://github.com/YaccConstructor/>
- [11]. I. Kirilenko, S. Grigorev, D. Avdiukhin. Syntax analyzers development in automated reengineering of informational system. *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, 2013, no. 174, pp. 94 – 98.
- [12]. J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR (1) parsers. *Programming Languages and Systems*, 2012, pp. 397–416.
- [13]. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010, pp. 348–370.
- [14]. J.-C. Filliatre, A. Paskevich. Why3: Where Programs Meet Provers. *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

- [15]. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002, 645 p.
- [16]. F* tutorial. June 2016. <https://www.fstar-lang.org/tutorial/>
- [17]. D. Syme, A. Granicz, A. Cisternino. *Expert F#*. Apress, 2012, 609 p.
- [18]. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. *Proceedings of the 2015 Conference on Certified Programs and Proofs*, 2015, pp. 167–174. DOI: 10.1145/2676724.2693177.
- [19]. Verification of grammar transformation to Chomsky normal form in F*. June 2016. https://github.com/YaccConstructor/YC_FStar.

Верификация преобразования грамматики в нормальную форму Хомского в F*

М.И. Полубелова <polubelovam@gmail.com>

С.Н. Божко <gkerfjmf@gmail.com>

С.В. Григорьев <Semen.Grigorev@jetbrains.com>

*Санкт-Петербургский государственный университет,
199034, Россия, Санкт-Петербург, Университетская наб., д.7/9*

Аннотация. Сертификационное программирование позволяет доказывать, что программа соответствует своему формальному описанию. Проверка корректности производится статически, благодаря чему становится возможным отказаться от дальнейшего тестирования верифицированных программ. Среди инструментов, предназначенных для сертификационного программирования, только инструмент F* позволяет реализовывать программы на языке общего назначения и автоматизирует доказательство их корректности. Последнее означает, что инструмент F* автоматически выведет доказательство корректности, где это возможно, при этом пользователь может специфицировать более сложные доказательства, если это необходимо. Мы работаем над применением данного подхода к проекту YaccConstructor – платформе для исследования и разработки генераторов лексических и синтаксических анализаторов и других алгоритмов для работы с грамматиками. В данной статье рассматривается верификация реализации одного из таких алгоритмов – преобразования грамматики в нормальную форму Хомского – что является первой задачей на пути к верификации всего проекта YaccConstructor. Для программы, реализующей данное преобразование, доказаны завершаемость и тотальность, а также установлен порядок применения используемых в ней основных преобразований с использованием зависимых и уточняющих типов. Следующим важным направлением данной работы является доказательство эквивалентности исходной и преобразованной грамматики. Инструмент F* позволяет извлекать код, написанный на F*, как программу на языке программирования F# или OCaml. Так как F# является основным языком разработки проекта YaccConstructor, это позволит сохранить совместимость верифицированных программ с существующими в проекте. В статье сформулированы преимущества и недостатки применения инструмента F*.

Ключевые слова: сертификационное программирование; F*; верификация программ; контекстно-свободная грамматика; нормальная форма Хомского; преобразование грамматики; dependent type; refinement type.

DOI: 10.15514/ISPRAS-2016-28(2)-8

Для цитирования: Полубелова М.И., Божко С.Н., Григорьев С.В. Верификация преобразования грамматики в нормальную форму Хомского в F*. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 127-138 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-8

Список литературы

- [1]. H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 2009, vol. 34, issue 1. pp. 3–25. DOI: 10.1007/s12046-009-0001-5.
- [2]. A. Chlipala. *Certified programming with dependent types*. MIT Press, 2013, 440 p.
- [3]. T. Sheard, A. Stump, S. Weirich. Language-based verification will change the world. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 343–348. DOI: 10.1145/1882362.1882432.
- [4]. E. Tanter, N. Tabareau. Gradual certified programming in Coq. *Proceedings of the 11th Symposium on Dynamic Languages*, 2015. pp. 26–40. DOI: 10.1145/2816707.2816710.
- [5]. The Coq proof assistant. June 2016. <https://coq.inria.fr/>
- [6]. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
- [7]. The F* homepage. June 2016. <https://www.fstar-lang.org/>
- [8]. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 2013, vol. 23, n. 5, pp. 552–593. DOI: 10.1017/S095679681300018X.
- [9]. N. Swamy, C. Hritcu, C. Keller. Dependent Types and Multi-Monadic Effects in F*. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655
- [10]. The YaccConstructor homepage. June 2016. <https://github.com/YaccConstructor/>
- [11]. I. Kirilenko, S. Grigorev, D. Avdiukhin. Syntax analyzers development in automated reengineering of informational system. *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, 2013, no. 174, pp. 94 – 98.
- [12]. J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR (1) parsers. *Programming Languages and Systems*, 2012, pp. 397–416.
- [13]. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010, pp. 348–370.
- [14]. J.-C. Filliatre, A. Paskevich. Why3: Where Programs Meet Provers. *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.
- [15]. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002, 645 p.
- [16]. F* tutorial. June 2016. <https://www.fstar-lang.org/tutorial/>
- [17]. D. Syme, A. Granicz, A. Cisternino. *Expert F#*. Apress, 2012, 609 p.
- [18]. D. Firsov, T. Uustalu. Certified normalization of context-free grammars. *Proceedings of the 2015 Conference on Certified Programs and Proofs*, 2015, pp. 167–174. DOI: 10.1145/2676724.2693177.
- [19]. Verification of grammar transformation to Chomsky normal form in F*. June 2016. https://github.com/YaccConstructor/YC_FStar.

Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS

C. Thule <casper.thule@eng.au.dk>

P. G. Larsen <pgl@eng.au.dk>

*Aarhus University, Department of Engineering,
Inge Lehmanns Gade 10, 8000 Aarhus C, Denmark*

Abstract. The development of Cyber-Physical Systems often involves cyber elements controlling physical entities, and this interaction is challenging because of the multi-disciplinary nature of such systems. It can be useful to create models of the constituent components and simulate these in what is called a co-simulation, as it can help to identify undesired behaviour. The Functional Mock-up Interface describes a tool-independent standard for constituent components participating in such a co-simulation and can support different formalisms. This paper describes an exploration of whether different concurrency features in Scala (actors, parallel collections, and futures) increase the performance of an existing application called the Co-Simulation Orchestration Engine performing co-simulations. The investigation was conducted by refactoring the existing application to make it suitable for implementing functionality that takes advantage of the concurrency features. In order to compare the different implementations testing was carried out using four test co-simulations. These test co-simulations were executed using the concurrent implementations and the original sequential implementation, verifying the simulation results, and retrieving the execution times of the simulations. The analysis showed that concurrency can be used to increase the performance in terms of execution time in some cases, but in order to achieve optimal performance, it is necessary to combine different strategies. Based on these results, future work tasks has been proposed.

Keywords: Co-simulation; concurrency; INTO-CPS; cyber-physical systems; FMI.

DOI: 10.15514/ISPRAS-2016-28(2)-9

For citation: Thule C., Larsen P.G. Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS. *Trudy ISP RAN/Proc. ISP RAS*], vol. 28, issue 2, 2016, pp. 139-156. DOI: 10.15514/ISPRAS-2016-28(2)-9

1. Introduction

Cyber-Physical Systems (CPSs) need to have close interaction between computer-based cyber parts controlling physical artefacts in a dependable way. In order to develop CPSs in a dependable manner it can be useful to create models of constituent

components that jointly form the system. A constituent model is an abstract description of a constituent, where the irrelevant details are abstracted away. Constituent models can be described in very different forms depending upon their nature, but here we will restrict ourselves to Discrete Event (DE) and Continuous-Time (CT) models representing very different disciplines. Such constituent models can then be used in a collaborative simulation (a co-simulation), which is able to couple models created in different formalisms. Thereby it is possible to simulate the entire system by simulating the components and exchange data as the common simulated time is progressing.

Typically such co-simulations are organised with a master-slave architecture where a Master Algorithm (MA) is used to manage the simulation. Fig 1 shows an example of four slaves, their dependencies, and input/output ports. It is the responsibility of the MA and thereby the master to orchestrate the simulation. This means to allow the different slaves to progress for determined time steps and resolve the dependencies between steps. A co-simulation often consists of three phases: Initialisation, simulation, and tear down. In the initialisation phase the master gets the properties of the slaves, chooses an MA, initialises the slaves, and establishes the communication channels. Next, in the simulation phase the master retrieves output values from the slaves, sets input values on the slaves, and invokes them to run a simulation step with a specific time step size. The slaves must respond with a status whether the step was accepted. In this phase, it can be necessary to perform a rollback¹ (if possible) for the relevant slave and run the simulation again with a different step size. Lastly, the outputs from the slaves are retrieved and the process repeats until a configured end time is reached. The final phase is tear down, where the slaves are shut down, memory is released, results are reported, and so forth.

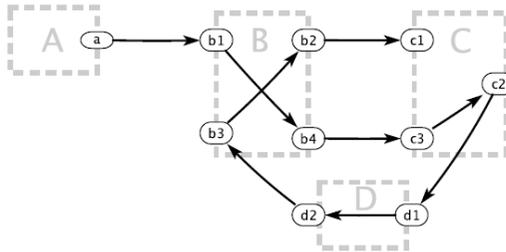


Fig. 1. Example of a simulated CPS with dependencies between slaves (the grey boxes) via their respective ports (the black ellipses) [1].

A challenge in using co-simulation as part of developing CPSs is that many complex multi-disciplinary systems cannot be modelled naturally in one simulation tool alone, but require several specialised simulation tools, that each do their part [2]. This makes it necessary to develop solutions tailored for a specific purpose instead of generalised solutions, which is expensive. The Functional Mock-up

¹ A rollback can be necessary e.g. if a slave rejects a step size.

Interface (FMI) was created to solve these challenges, as it is a tool-independent standard for co-simulation [3]. The standard provides and describes C interfaces that can be partly or fully implemented by a component, which is then called a Functional Mock-up Unit (FMU). This makes it possible to create generalised solutions, as the components can contain their own solvers, and still adhere to FMI. The INTO-CPS project² [4] makes use of FMI for a simulation kernel of a tool suite ranging from original requirements expressed in SysML over heterogeneous constituent models that can be co-simulated and gradually moved down to their corresponding realisations. When developing CPSs using co-simulation, it is desirable to execute the simulations as fast as possible to enable the use of increasingly complex models and try a greater range of test scenarios. As many processors today have multiple cores [5] concurrency may increase the performance of an application, but it also introduces overhead. It is therefore of keen interest to determine, how concurrency can be used to potentially improve the performance. The performance in this context is considered to be how fast a co-simulation is performed, and is therefore measured in terms of time. This paper describes how the usage of concurrency was implemented in an existing application called the Co-Simulation Orchestration Engine (COE), which orchestrates co-simulations using FMI. Different implementations were performed in Scala using three different concurrency features: Akka Actors [6], futures [7], and parallel collections [8]. These were chosen because they offer different capabilities that can be taken advantage of in the COE, and therefore the trade-off between features and performance is interesting. One of the most important capabilities is composability, because FMUs can have different step sizes and rollbacks can be necessary, which can lead to complicated scenarios. Following is a short description of the concurrency features:

Parallel Collections: The motivation behind adding parallel collections to Scala was to provide a familiar and simple high-level abstraction to parallel programming [8]. Parallel collections are conceptually simple to use, as a regular collection can be converted to a parallel collection by invoking the function “par”. Once it is a parallel collection, functions such as map and filter are executed concurrently. Parallel collections are considered less composable than the other implementations, as the results are gathered in a blocking fashion.

Futures: A future is a placeholder for a value, that is the result of some concurrent calculation, and it can be accessed synchronously or asynchronously. The term “future” was originally proposed by Baker and Hewitt [9] in the context of garbage collection of processes. As opposed to parallel collections, it is possible to chain futures, such that when a future has been computed, the computed value is passed to the chained future.

² Public deliverables and more information regarding the INTO-CPS project can be retrieved from <http://into-cps.au.dk>.

Actors: The Actor Model was introduced as an architecture to efficiently run programs with a high degree of parallelism without the need for semaphores [10]. An actor is an autonomous object that encapsulates data, methods, a thread, a mailbox, and an address [11]. Actor methods can return futures, and therefore offer the same composability as futures in this regard. Actors also provide additional composable features, such as hierarchical structures, remote capabilities, message parsing, and so on.

The paper is structured as follows: Section 2 describes the initial implementation and the implementations using concurrency. Afterwards, Section 3 describes how the implementations were tested and presents the results. Then related work is treated in Section 4. Lastly, the work is summarised in Section 5 and future work is outlined in Section 6.

2. Co-Simulation Orchestration Engine Implementations

This section concerns the implementations of the COE application³. It focuses on the MA part of the implementations, as the initialisation and tear down phases are unaltered for the implementations described below.

The COE application runs as a web server using HTTP. The following HTTP requests are performed in the given order to run a simulation:

1. **Initialise:** A configuration file is sent to the web server. The configuration file contains the FMUs to be used in the simulation, the mapping between input and output values, and whether to use a fixed or variable step size.
2. **Simulate:** This request starts a simulation.
3. **Results:** This request returns the result and duration of a given simulation.

There are different implementations of the MA in the COE: A sequential implementation, and three implementations that execute concurrently, following the principles described above. These different implementations were developed in order to test and compare the performance of the COE in a sequential/concurrent setting and determine whether using concurrency could improve the performance.

2.1 Sequential Implementation

The sequential implementation of the MA consists of the following steps in the given order:

4. **Resolve inputs:** This step consists of mapping the outputs of the FMUs to the inputs of the other FMUs.
5. **Set inputs:** The input values determined in the previous step are passed to the FMU instances in this step.
6. **Serialize state:** In this step the states of the FMUs are serialized, so it is possible to perform a rollback in case of an error.

³ See [12] for further details on the implementation.

7. **Get step size:** If variable step size is supported by the FMUs, then the maximum step size is retrieved in this step. Otherwise a configured fixed step size is used.
8. **Do step:** The FMU instances are invoked to perform a step with the step size determined in the previous step. This function contains the most extensive calculations performed by the FMUs.
9. **Process result:** The return values from the previous invocations are analysed and in case of any errors a rollback is performed or the simulation is terminated.
10. **Get state:** The state in terms of output values is retrieved in this step, and thereby the next iteration can begin.

In the sequential implementation a mapping operation is performed over the FMU instances in every step except the “Process result” step, where it depends on whether errors are encountered and if so which errors. This sums to six, possibly seven, mapping operations over the FMU instances.

2.2 Implementations with Concurrency

When implementing concurrency in the COE it is desirable that as much work as possible is performed in every concurrent invocation. To allow for a better usage of concurrency some functions should be grouped, such that a group of functions can be invoked concurrently. If concurrency was used in the sequential implementation to invoke the FMUs without refactoring the implementation, it would be necessary to invoke every step in different concurrent invocations. This would result in several thread initialisation and synchronizations per simulation step, where a synchronization is a waiting operation until all threads have finished computing. An example of this is shown in Fig. 2. The figure shows a possible usage of concurrency based on the sequential implementation with four FMUs (horizontal frames), where the functions “Set inputs”, “Serialize state”, “Do step”, and “Get state” are invoked in different concurrent invocations. The realised implementation (vertical frame) invokes the functions using the same concurrent invocation for a given FMU. This will be described further below.

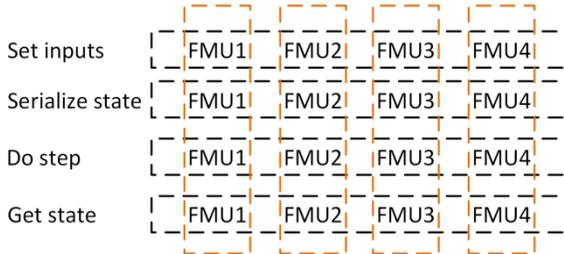


Fig. 2. The horizontal frames represent a possible usage of concurrency based on the sequential implementation. The vertical frames represent the usage of concurrency based on the implementations.

By refactoring and grouping these functions, it is possible to reduce the thread initialisations and synchronizations. This leads to more work performed by every spawned thread and fewer synchronizations, which minimizes the overhead of using concurrency. It is not possible to eliminate synchronization completely, because it is necessary to resolve the inputs for the FMUs before progressing, which requires retrieving the outputs from other FMUs, and therefore the simulation cannot continue until this has been performed. Besides minimizing the overhead of using concurrency, this grouping will also help to minimize the number of mapping operations performed in the steps in the sequential implementation, which is desirable to improve the performance.

The grouping and flow of a simulation step for the implementation using concurrency is shown in Fig. 3. The grouping was implemented in a separate and encapsulated function that exhibits referential transparency to prevent the necessity of locking mechanisms. This grouping will be referred to as the concurrent entity below.

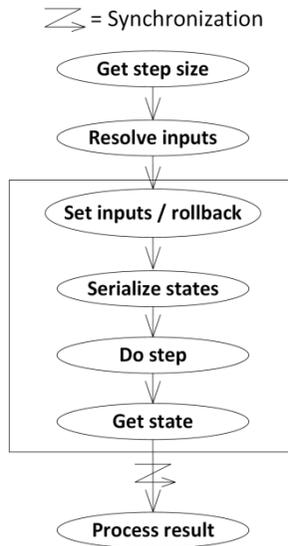


Fig. 3. Simulation step flow in the implementation using concurrency. The box represents the functions grouped together.

By creating these concurrent entities, it was a conceptually simple task to take advantage of the concurrency features. Furthermore, it effectively reduced the mapping operations from six, possibly seven depending on the step “Process result”, to three. This implementation also makes it possible to include “assignment functions” such as “Set inputs” in the concurrent entity without lowering performance. Including “Set inputs” as its own concurrent invocation (as shown in the horizontal frames in Fig. 2) would lower the performance, because the overhead of using concurrency is too high compared to invoking the function sequentially.

Using the grouping (the vertical frame in Fig. 2) it improves performance to include “Set inputs”, because it can be grouped with the other functions, e.g. “Do step”, without additional overhead. However, the grouping also came with a trade-off: In the sequential implementation, the state would not be retrieved, if one or more FMUs fail in the step called “Do step”, because it would be wasteful due to the error(s). But in the implementation using grouped functions, the state of the FMUs not failing in the step “Do step” would still be retrieved, because the entities responsible for the FMU simulation step are unaware of the state of other entities until the synchronization phase⁴. This can therefore lead to unnecessary retrieval of states.

In the sequential implementation, the flow is to calculate the parameters necessary for the next immediate function to be invoked on the FMUs, and then calculate the parameters again. In the implementations with concurrency this is changed to calculate the parameters necessary for an entire simulation step, and invoke the concurrent entity for each FMU concurrently. This makes it possible to maximize the workload for each concurrent invocation.

3. Testing

This section presents the evaluation of the COE described in Section II. The purpose is to gain data that can be used to compare performance of the sequential implementation and the implementations using concurrency. Furthermore, as concurrency can lead to non-determinism, it is important to verify the simulation results, which are the output values of the FMUs at different points in time relative to the step sizes. For this purpose, the sequential implementation was considered an oracle, and therefore simulation results of the concurrent implementations were compared against simulation results from the sequential implementation. In the longer term the plan is to use a representation of the FMI semantics as the ultimate oracle [13]. Here semantics is provided using the Communicating Sequential Processes [14] and this has been used to model check FMI for deadlock and livelock properties using the FDR tool [15].

The following test principles were followed during testing:

Test environment: A test consisting of multiple simulations should be performed on the same hardware with approximately the same processes running during the test. The reason for stating “approximately the same processes” is that the tests were run in a Windows environment, where it is not possible to completely control the running processes from the Operating System. All processes irrelevant to the execution of tests should be disabled during the tests.

Test functions: To limit inconsistencies in the processes running between simulations, each test should be implemented as a single test function. This means that a test performing simulations using the sequential implementation and the three

⁴ Several programming languages offer the possibility to abort threads in a case like this. However, that increases the complexity and is not considered applicable in general.

concurrent implementations should be implemented in one test function to avoid undesirable interaction required to start other tests. To further ensure usable results the COE application should be restarted for every simulation.

Correct simulation results: The sequential implementation is considered to be an oracle and it is assumed that it calculates the “correct” simulation results. It should be verified that the concurrent implementations calculate the same simulation results as the sequential implementation.

Automation: The tests should be automated so they are easy to replicate and less prone to manual errors. This will also make them usable in the future development of the COE.

3.1 Test Setup

To enable automatic testing a framework was developed. This enabled testing of different concurrent implementations, evaluation of performance, and verification of consistency between the sequential simulation results and the concurrent simulation results. Implementation-wise this required support for launching the different implementations with different arguments, invoking the web servers using HTTP requests along with gathering, and verifying the consistency of results. To verify the consistency of results, the simulation results of the implementations using concurrency are automatically compared to the simulation results of the sequential implementation, as this is considered an oracle.

Different FMUs were used in the tests to investigate the performance, including a configurable FMU that was developed to control the level of computations, which will be described below. The tests and their corresponding FMUs are the following:

Heating, Ventilation, and Air Conditioning (HVAC) test: This test uses FMUs that perform the most extensive computations available in the project. The simulation consists of five FMUs: one controller FMU and four Fan Coil Unit FMUs. A test, which will be referred to as HVAC #1, was set up with an end time of 1000 seconds and a step size of 0.1 seconds. This is inspired by the case study undertaken by Unified Technologies Research Center inside the INTO-CPS project [16].

Sine Integrate Wait tests: These tests consist of three different FMUs, that perform limited computations, and therefore one has been modified. The FMUs are: a sine FMU generating a sine wave, an integrate FMU that integrates the sine values, and a modified integrate FMU. It is possible to configure the modified integrate FMU, such that it performs busy waiting in the “Do step” function for a given number of microseconds. It makes use of “QueryPerformanceCounter” recommended by Microsoft to use when high-resolution time stamps are required with microsecond precision [17]. The configuration of the busy wait does not have any impact on the performance of the FMU, because it happens in the initialisation phase, which is not part of the performance measurement. These FMUs were used to set up three tests, referred to as SI #1/2/3, where each simulation in the tests have an end time of 100 seconds and time step size of 0.1 seconds. The tests are the following:

SI #1 consists of one sine FMU, one modified integrate FMU, and three simulations: In the first simulation, the modified integrate FMU has a wait time of zero milliseconds, then 0.5 milliseconds, and lastly 1 millisecond.

SI #2 uses one sine FMU and five modified integrate FMUs with the same simulation setup as SI #1.

SI #3 uses one sine FMU and 100 integrate FMUs.

3.2 Test Results

This section contains the results of the tests described in Section 3.1. The results are presented in tables, where the unit of the numbers is milliseconds, and the table columns represent the following: Sequential refers to the sequential implementation, “Future” refers to the concurrent implementation using futures, “Par” refers to the concurrent implementation using parallel collections, and “Actor” refers to the concurrent implementation using actors. The result for the HVAC test is presented in Table. 1, and the results for the SI tests are presented in Table. 2, 3, and 4.

Based on these tests it is possible to draw some conclusions:

Executing simulations concurrently can be faster than executing them sequentially: The results for HVAC #1, SI #2, and SI #3 show that concurrent execution can be faster than sequential execution.

Executing simulations sequentially can be faster than executing them concurrently: The results for HVAC #1, SI #1, SI #2 and SI #3 show, that sequential execution can be faster than concurrent execution. Some of these test results contradict the previous conclusion, and therefore it is necessary to pay attention to the concurrency feature used.

Trade-off: An interesting discovery is that parallel collections perform worse than futures and actors. This indicates that even though parallel collections offer fewer capabilities than the other concurrency features, it does not perform faster.

Table. 1. Results from HVAC #1.

Sequential	Future	Par	Actor
31256	29822	31980	30919

Table. 2. Results from SI #1.

Wait	Sequential	Future	Par	Actor
0.0	195	330	656	374
0.5	4468	4635	5161	4715
1.0	8758	8938	9545	9032

Table 3. Results from SI #2.

Wait	Sequential	Future	Par	Actor
0.0	355	434	834	622
0.5	21904	4679	5042	4746
1.0	43356	8970	9348	9184

Table 4. Results from SI #3.

Sequential	Future	Par	Actor
355	434	834	622

4. Related Work

In order to make use of the improvements in hardware, it is necessary to improve the software. An adage known by “Wirth’s law” goes: “Software is getting slower more rapidly than hardware becomes faster” [18]⁵. He argues that methodologies are important in order to take full advantage of the improvements in hardware. Sutter urges application developers to take a hard look at the design of their applications and identify places that could benefit from concurrency [20]. This is necessary to exploit hardware capabilities, as processor manufacturers are turning to multicore processors. Harper et. al. conducted a study on a large-scale Publish/Subscribe bus system, and found an overall performance of 80 percent based on concurrency experiments [21]. Additionally, they surveyed concurrency design patterns with the purpose of helping developers towards the “right” patterns.

As mentioned previously, it is important to reduce communication and synchronization overhead between processes to achieve a fast simulation. Agrawal et. al. have implemented and evaluated three communications primitives for hardware/software co-simulation and found that a message-queue based communication backplane is preferable [22]. The other two primitives evaluated were shared memory and file-based sockets. Strategies that address the issue of synchronization are also introduced by Bishop et. al., and these strategies also deal with time management [23]. They conclude that using the design strategies discussed can enable the development of high-performance application-specific co-simulations. Kim et. al. consider synchronization between components simulators as the main reason for poor performance of HW/SW co-simulation [24]. They propose a novel technique based on virtual synchronization, which improves the simulator speed and minimizes the synchronization overhead. Becker et. al. describes an approach, where distributed communicating processes are used for the interaction between software and hardware using Unix interprocess communication mechanisms [25]. The

⁵ Wirth attributes this to a different saying by Reiser [19].

approach does not accurately simulate the relative speeds of the hardware and software components, but the authors found this to be acceptable in their case.

The articles above consider synchronization, communication between simulators, and concurrency as a bottleneck in achieving fast co-simulations. It is therefore of keen interest to minimize the communication and synchronization along with taking advantage of concurrency. This work addresses these issues as well, as it is an attempt to limit synchronization and take advantage of concurrency. Furthermore, it is an attempt to avoid unnecessary inter-thread/inter-process.

5. Conclusion

Using FMI it is possible to develop a generalised application capable of performing co-simulation, thereby avoiding the need for tailored solutions developed to support the co-simulation of specific systems. It is desirable to perform a co-simulation as fast as possible, as it can help to verify the behaviour of systems or lead to the discovery of undesired behaviour. It was therefore investigated whether concurrency could be used to improve the performance of an application performing co-simulation. In some cases the usage of concurrency resulted in faster co-simulations, whereas in other cases sequential computation offered better performance. Because of this it is reasonable to conclude, that it is necessary to allow for different simulation strategies to achieve the fastest simulation. These strategies should support running simulations sequentially, concurrently, or a mix of these. For example, if an FMU that performs long-lasting computations is to be simulated with three FMUs that perform fast computations, then it could be optimal to run this simulation in a hierarchical structure using two threads as shown in Fig. 4.

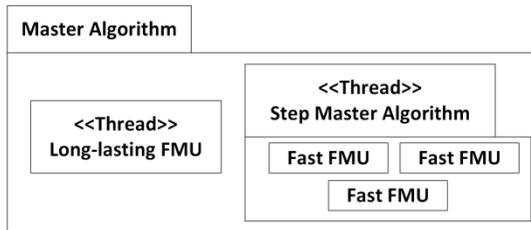


Fig. 4. Master Algorithm simulating four FMUs using an additional step Master Algorithm

Allowing for different strategies inevitably involves computing which strategies to use. A way of assisting the choice of strategy is to include a measure of how long-lasting the computations performed by an FMU are within the properties of the given FMU. However, this might be difficult to realise in a practical manner, where different hardware is used. An alternative approach is to use meta data for a given simulation. This can be configured beforehand, or the COE can determine it, when running the first co-simulation using the given FMUs.

6 Future Work

In order to improve the performance of the COE and choose when to use concurrency, there are several tasks to undertake:

Testability: Currently, the COE supports reporting the duration of an entire simulation without initialisation and reporting of results. As these steps inevitably are part of a simulation, they should be part of the performance tests. Additionally, the COE should offer better granularity for performance measurements. Better granularity will make it possible to examine the performance of different parts of the application, which can aid in finding bottlenecks and help target the development effort.

Investigate concurrency: Besides concluding that concurrency can/cannot improve the performance of the application in some cases, it is interesting to investigate when concurrency can improve the performance. Part of this investigation is to determine, whether an increase of performance is achievable by enabling sequential, concurrent, and mixed processing, as mentioned in the previous section. The approach is to implement nested COEs that appears as FMUs externally as shown in Fig. 5.

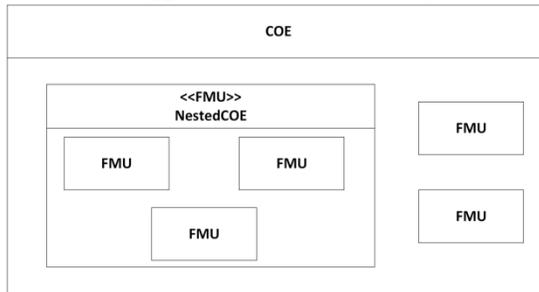


Fig. 5. Example of a nested COE participating in a co-simulation

This approach would allow for compositional co-simulation, as the nested COE will exhibit the same behaviour as an FMU externally and therefore be closed under composition. This allows for an elegant representation of complex systems, as it can be considered a co-simulation of co-simulations. Furthermore, it allows for hierarchical co-simulations, which can contribute to the reusability of co-simulations. The implementation will also support distributed scenarios, where nested COEs can be executed on different machines/operating systems and therefore allow for a greater range of co-simulation scenarios.

Guidelines: Since the future work concerns investigation of concurrency, it is compelling to attempt to generalise the lessons that will be learned and apply them on different case studies. The hope is that this can contribute to existing methodologies and guidelines on using concurrency efficiently.

Semantics alignment: The continuation of the FMI semantics work referred to above will also involve theorem proving using the Isabelle theorem prover [26] and we hope that it will be possible to align that with the COE work in order to use the semantics

directly as an oracle of checking conformance. This also involves examining the semantic properties of the concurrency features.

Graphical user interface: A Graphical User Interface (GUI) for the tools in the INTO-CPS project is currently being developed. This contains functionality to configure and interact with the COE. Furthermore, it will be possible to configure which simulation strategies to use and how the FMUs should be organised. The GUI application is being developed as a desktop application, as some of the tools in the tool chain do not support a distributed approach. However, by using Electron [27] it is possible to use web technologies for desktop applications. Therefore the GUI application is cross platform and supports a possible transition to being hosted on a web server in a distributed fashion.

Acknowledgement

The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. Furthermore, the authors would like to thank Nick Battle for reviewing and providing input to this paper. Finally, thanks to Alexander Petrenko for translating parts of the paper into Russian.

References

- [1]. D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *2013 Proceedings of the International Conference on Embedded Software*, Sept 2013, pp. 1–12. ISBN: 978-1-4799-1443-2.
- [2]. J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *Proceedings of the 8th International Modelica Conference*, 2011, pp. 115-120. DOI: 10.3384/ecp11063115.
- [3]. FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014.
- [4]. J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains," in *FormaliSE: FME Workshop on Formal Methods in Software Engineering*. Florence, Italy: ICSE 2015, May 2015, pp. 40-46. DOI: 10.1109/FormaliSE.2015.14.
- [5]. D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005. DOI: 10.1109/MC.2005.160.
- [6]. Typesafe Inc, "Akka scala documentation," <http://akka.io/docs/>, Akka, September 2015, Release 2.4.0.
- [7]. P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," <http://docs.scala-lang.org/overviews/core/futures.html>, (Visited on 05/03/2016).
- [8]. A. Prokopec and H. Miller, "Parallel collections – overview - scala documentation," <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>, 2015, (Visited on 05/03/2015).
- [9]. H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming*

- Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: <http://doi.acm.org/10.1145/800228.806932>. DOI: 10.1145/800228.806932.
- [10]. C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: <http://worrydream.com/refs/Hewitt-ActorModel.pdf>.
- [11]. G. A. Agha and W. Kim, “Actors: A unifying model for parallel and distributed computing,” *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1383762198000678>.
- [12]. C. Thule, “Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS,” Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep. ECE-TR-26, May 2016. [Online]. Available: <http://ojs.statsbiblioteket.dk/index.php/ece/issue/archive>.
- [13]. N. Amalio, A. Cavalcanti, C. König, and J. Woodcock, “Foundations for FMI Co-Modelling,” INTO-CPS Deliverable, D2.1d, Tech. Rep., December 2015.
- [14]. T. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.
- [15]. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, “FDR3 — A Modern Refinement Checker for CSP,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 8413, 2014, pp. 187–201.
- [16]. J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E. D. Mady, “Collaborative Model-based Systems Engineering for Cyber-Physical Systems - a Case Study in Building Automation”. *INCOSE*. Edinburgh, Scotland. July 2016.
- [17]. Microsoft, “Acquiring high-resolution time stamps (windows),” [https://msdn.microsoft.com/en-us/library/dn553408\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx), 2015, (Visited on 05/03/2016).
- [18]. N. Wirth, “A plea for lean software,” *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995.
- [19]. M. Reiser, *The Oberon System: User Guide and Programmer’s Manual*. New York, NY, USA: ACM, 1991.
- [20]. H. Sutter, “A fundamental turn toward concurrency in software,” *Dr. Dobbs’s Journal*, vol. 30, no. 3, pp. 16–23, 2005.
- [21]. K. E. Harper, J. Zheng, and S. Mahate, “Experiences in initiating concurrency software research efforts,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 139–148. [Online]. Available: <http://doi.acm.org/10.1145/1810295.1810316>. DOI: 10.1145/1810295.1810316.
- [22]. B. Agrawal, T. Sherwood, C. Shin, and S. Yoon, “Addressing the challenges of synchronization/communication and debugging support in hardware/software cosimulation,” in *VLSI Design, 2008. VLSID 2008. 21st International Conference on VLSI Design (VLSID 2008)*, Jan 2008, pp. 354–361.
- [23]. W. Bishop and W. Loucks, “A heterogeneous environment for hardware/ software cosimulation,” in *Simulation Symposium, 1997. Proceedings., 30th Annual*, Apr 1997, pp. 14–22.
- [24]. D. Kim, Y. Yi, and S. Ha, “Trace-driven hw/sw cosimulation using virtual synchronization technique,” in *Design Automation Conference, 2005. Proceedings. 42nd*, June 2005, pp. 345–348.

- [25]. D. Becker, R. K. Singh, and S. G. Tell, “An engineering environment for hardware/software co-simulation,” in *In 29th ACM/IEEE Design Automation Conference*, 1992, pp. 129–134.
- [26]. T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.
- [27]. Github, “Electron – Build cross platform desktop apps with JavaScript, HTML, and CSS”, version 1.2.2, <http://electron.atom.io/>, (Visited on 10/06/2016).

Исследование влияния использования параллелизма на производительность движка косимуляции в проекте INTO-CPS

К. Тул <casper.thule@eng.au.dk>

П. Г. Ларсен <pgl@eng.au.dk>

*Орхусский университет, Технический факультет,
ул. Инге Лехманс, 10, 8000 Орхус С, Дания*

Аннотация. Кибер-физические системы часто включают в себя управляющие кибер-элементы, контролирующие физические объекты и взаимодействующие с ними. Анализ процессов в таких системах является сложной задачей из-за междисциплинарного характера этой области исследований. Моделирование и симуляция поведения составляющих систему компонентов, так называемая косимуляция, позволяет выявлять возможность нежелательного поведения. Интерфейс FMI (Functional Mock-up Interface) описывает стандартный интерфейс взаимодействия с составляющими компонентами, участвующими в такой косимуляции, и может поддерживать различные формализмы. Статья описывает исследование того, насколько различные возможности параллелизма в Scala (акторы, параллельные коллекции и футуры) увеличивают производительность существующего движка Co-Simulation Orchestration Engine, выполняющего косимуляцию. Исследование сопровождалось рефакторингом имеющегося кода с тем, чтобы реализация могла использовать преимущества параллельных возможностей. Для того, чтобы сравнить различные варианты реализации выполнялось по четыре тестовых косимуляции. В тестовых косимуляциях сравнивались параллельные реализации и исходная последовательная реализация, верифицировались результаты моделирования и получались оценки времени моделирования. Анализ показал, что в некоторых случаях параллелизм может использоваться для повышения производительности, но для того, чтобы достичь оптимальной производительности, необходимо комбинировать различные стратегии. На основе полученных результатов предлагаются будущие направления исследований.

Ключевые слова: Косимуляция; параллелизм; INTO-CPS; кибер-физические системы; FMI

DOI: 10.15514/ISPRAS-2016-28(2)-9

Для цитирования: Тул С., Ларсен П.Г. Исследование влияния использования параллелизма на производительность движка косимуляции в проекте INTO-CPS. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 139-156. DOI: 10.15514/ISPRAS-2016-28(2)-9

Список литературы

- [1]. D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *2013 Proceedings of the International Conference on Embedded Software*, Sept 2013, pp. 1–12. ISBN: 978-1-4799-1443-2.
- [2]. J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *Proceedings of the 8th International Modelica Conference*, 2011, pp. 115-120. DOI: 10.3384/ecp11063115.
- [3]. FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014.
- [4]. J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains," in *FormaliSE: FME Workshop on Formal Methods in Software Engineering*. Florence, Italy: ICSE 2015, May 2015, pp. 40-46. DOI: 10.1109/FormaliSE.2015.14.
- [5]. D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005. DOI: 10.1109/MC.2005.160.
- [6]. Typesafe Inc, "Akka scala documentation," <http://akka.io/docs/>, Akka, September 2015, Release 2.4.0.
- [7]. P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," <http://docs.scala-lang.org/overviews/core/futures.html>, (Visited on 05/03/2016).
- [8]. A. Prokopec and H. Miller, "Parallel collections – overview - scala documentation," <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>, 2015, (Visited on 05/03/2015).
- [9]. H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: <http://doi.acm.org/10.1145/800228.806932>. DOI: 10.1145/800228.806932.
- [10]. C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: <http://worrydream.com/refs/Hewitt-ActorModel.pdf>.
- [11]. G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1383762198000678>.
- [12]. C. Thule, "Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS," Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep. ECE-TR-26, May 2016. [Online]. Available: <http://ojs.statsbiblioteket.dk/index.php/ece/issue/archive>.
- [13]. N. Amalio, A. Cavalcanti, C. König, and J. Woodcock, "Foundations for FMI Co-Modelling," INTO-CPS Deliverable, D2.1d, Tech. Rep., December 2015.

- [14]. T. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.
- [15]. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, “FDR3 — A Modern Refinement Checker for CSP,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 8413, 2014, pp. 187–201.
- [16]. J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E. D. Mady, “Collaborative Model-based Systems Engineering for Cyber-Physical Systems - a Case Study in Building Automation”. *INCOSE*. Edinburgh, Scotland. July 2016.
- [17]. Microsoft, “Acquiring high-resolution time stamps (windows),” [https://msdn.microsoft.com/en-us/library/dn553408\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx), 2015, (Visited on 05/03/2016).
- [18]. N. Wirth, “A plea for lean software,” *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995.
- [19]. M. Reiser, *The Oberon System: User Guide and Programmer’s Manual*. New York, NY, USA: ACM, 1991.
- [20]. H. Sutter, “A fundamental turn toward concurrency in software,” *Dr. Dobbs’s Journal*, vol. 30, no. 3, pp. 16–23, 2005.
- [21]. K. E. Harper, J. Zheng, and S. Mahate, “Experiences in initiating concurrency software research efforts,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering – Volume 2*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 139–148. [Online]. Available: <http://doi.acm.org/10.1145/1810295.1810316>. DOI: 10.1145/1810295.1810316.
- [22]. B. Agrawal, T. Sherwood, C. Shin, and S. Yoon, “Addressing the challenges of synchronization/communication and debugging support in hardware/software cosimulation,” in *VLSI Design, 2008. VLSID 2008. 21st International Conference on VLSI Design (VLSID 2008)*, Jan 2008, pp. 354–361.
- [23]. W. Bishop and W. Loucks, “A heterogeneous environment for hardware/ software cosimulation,” in *Simulation Symposium, 1997. Proceedings., 30th Annual*, Apr 1997, pp. 14–22.
- [24]. D. Kim, Y. Yi, and S. Ha, “Trace-driven hw/sw cosimulation using virtual synchronization technique,” in *Design Automation Conference, 2005. Proceedings. 42nd*, June 2005, pp. 345–348.
- [25]. D. Becker, R. K. Singh, and S. G. Tell, “An engineering environment for hardware/software co-simulation,” in *In 29th ACM/IEEE Design Automation Conference*, 1992, pp. 129–134.
- [26]. T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.
- [27]. Github, “Electron – Build cross platform desktop apps with JavaScript, HTML, and CSS”, version 1.2.2, <http://electron.atom.io/>, (Visited on 10/06/2016).

A static approach to estimation of execution time of components in AADL models

A.M. Troitskiy <troitskiy@ispras.ru>

D.V. Buzdalov <buzdalov@ispras.ru>

*Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*

Abstract. During development of modern avionics systems and other mission-critical systems modelling is vitally used. Models can be used for checking and validation of developed system, including early validation. Early validation is very important because the cost of errors is raising exponentially depending on the development stage. For modelling of such systems, Architecture Analysis and Design Language (AADL) is widely used. It allows to model both architecture of a developed system and some of behavioral characteristics of its components. In the paper, the task of automated model checking for consistency of some behavioral properties is considered. In particular, we focus on the problem of estimation of working time of model components and corresponding between this time and other properties in a model. This problem is close to the worst-case execution time problem (WCET) but it has its own specific in this application. We considered a static approach allowing to work with standard specification of components behaviour in AADL-models with specialized extended finite automata. In the paper, peculiarities of used behaviour model (specialized finite automata) were considered including work with time and external events. We considered the problem of working time estimation for such models connected with non-local characteristic of this property. We propose an algorithm for time estimation for such behaviour models. This algorithm was implemented in MASIW framework, a tool for development of AADL-models.

Key words: AADL; avionics design; static analysis.

DOI: 10.15514/ISPRAS-2016-28(2)-10

For citation: Troitskiy A.M., Buzdalov D.V. A static approach to estimation of execution time of components in AADL models. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 157-172. DOI: 10.15514/ISPRAS-2016-28(2)-10

1. Introduction

Modern avionics is responsible for control of almost all aspects of aircraft operation. As a result, the complexity of such systems is really high. Thus making sure that developed system is correct is a challenging task.

Nowadays problems and their solution bring additional complexity to avionics systems. To satisfy models requirements for weight and power consumption,

integrated modular avionics (IMA [1]) approach is used. It means that several resources (e.g. universal processor modules and network) are shared between several pieces of software. The approach leads to appearing of step of the integration of the whole system, i.e. deployment of software on different hardware, network configuration and etc.

This approach solves weight and power consumption problems, but leads to potential problems of interfering of applications. It means that the whole system correctness must be checked and this problem is not solvable by checking of correctness of each part of the system.

The model-driven approach of development allows to manage with the complexity of a system being developed. In particular, models are needed to perform different kinds of analysis of the modelled system though analysis of appropriate models. Such analyses are intended to be performed on different stages of development, in particular, to eliminate errors at early steps of development.

One kind of checks that are needed to be performed is check of *timing properties* of software components.

In particular, during design and deployment stages, each particular application is bound to a processor module. Appropriate timing properties are assigned to them, for example

- *dispatch protocol*, i.e. whether an application is fired periodically, eventually (sporadically) or both;
- *period* of execution for periodic applications;
- *compute deadline*, i.e. time interval in which an application has to finish its work after it was given an ability to execute;
- *recover deadline*, i.e. time interval in which an application has to recover from recoverable errors;
- *process time*, i.e. the time between sending a processed output data after getting some input data;
- *output rate*, i.e. rate at which an application has to produce its output, when it is periodic;
- *output jitter*, i.e. maximum deviation of time for periodic output and etc.

Being assigned to some particular application, these properties can be used in schedulability analysis, data flow timing analysis, worst case execution time (WCET) analysis and etc. Some desired or expected values can appear before implementation of particular software.

During the system development, models of it are refined. In particular, for software some behaviour specifications can appear. Such behaviour specifications can be purely functional (i.e. containing only information about which outputs will be produced in particular inputs at the given state).

Also such specifications can contain how much time will be consumed in this or that situation. The addition of this information can lead to inconsistency in the model, because some assumptions about timing properties of software can already exist in the model and these assumptions can contradict with behavior specification.

To check the consistency of a model, it is important to estimate timing properties of particular behaviour specifications.

Compute deadline consistency example

Consider a periodic software component with some particular *period* set in the model. Consider also that this component has *compute deadline* property bounds set to a range p from p_1 to p_2 ms.

This property can be used in the schedule building: e.g. a time frame of p_2 ms can be reserved each period to ensure this software component has enough time to compute. This can be done on early stages of system development when no particular behavior is known yet.

Consider the case when after development this software component is refined: now its behaviour is specified with automaton with transitions containing how much time is consumed by computations assigned to them. We can estimate general time consuming of an application each period as a range h from h_1 to h_2 ms.

After getting estimations h we can compare it with bounds p from the model and there are several decisions we can take:

- when $h = p$, behavior corresponds to property and the model is consistent;
- when $h \not\subset p$, the model can be inconsistent because real execution time may miss the bounds;
- when $h \subset p$, $p \neq h$, the behaviour specification corresponds to the property; also, we can say that the property in the model can be refined to a more precise value;
- when $p \cap h = \emptyset$, the model is inconsistent.

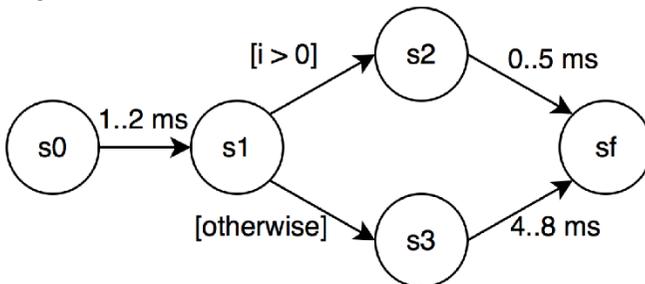


Fig. 1. Example of behavior specification

Example of consistent case

Consider an example when the model has bounds for *compute deadline* property set to be from 3 to 10 ms. Consider also that this application has behavior specification

with automaton shown on the *fig.1*. Each period this application begins in state s_0 and finishes in s_f .

In this example we can estimate execution time of the application to be between 5 and 10 ms. This value is consistent with property set in the model.

There is another case when such estimations are useful. Consider a situation when some software component in the model did not have any timing properties set. Consider then, that later it was refined and some behavior specification has appeared for it. The model still needs to be checked for schedulability and other timing-aware properties. So, we need to derive these timing properties for a component with some behaviour specification. Again, we run into an issue of estimation of timing properties having a particular behavior specification.

So, generally we can resume that there is an important issue of estimation of timing properties in responsible systems' models with behavior specifications.

2. AADL and BA

We use AADL (Architecture Analysis and Design Language, [2]) as a modelling language. It allows to describe both physical and logical parts of the modelled system, connections between components and bindings between layers of the system. AADL has a mechanism of the language extending though special language annexes and it has a number of standard annexes.

One of such extensions is called Behavior Model Annex [3] (BA). It allows to specify behavior of AADL-components using extended time-aware finite-state machine.

Behaviors are set to components of a modelled system. The basic elements used in BA behavior specifications are

- automaton states change;
- internal computations;
- accessing and assigning to internal or external variables (data components);
- interaction with the outer world using input/output ports; depending of behavior, input ports can be managed both by pulling data and by waiting for data to come;
- handling *dispatch* events, i.e. a situation when software component is allowed to perform its execution (*e.g., an operating system signals a thread to start*).

Behavior Annex automaton must contain a single initial state. When the automaton goes out from the initial state, its internal variables are being initialized. The automaton can contain several final states, in these states automaton can stop its execution.

Each state of the automaton belongs to one of the classes of *complete states* or *execution states*.

Transitions from execution states occur immediately after automaton comes to such state. In complete states automaton waits for external events (data for input ports or dispatch event). Transitions going out of complete states are fired as soon as corresponding event happens.

In BA each state transition is assigned with a list of actions which is run when automaton performs this transition.

There are actions that appear in the list of actions in BA behavior specification:

- actions with ports: reading, writing, getting of messages count in ports;
- actions with local and accessible external variables: reading and assignment;
- locking on resources: getting and releasing;
- action for modelling of time consumption $computation(t_{min}..t_{max})$;
- *stop* action for automaton interruption;
- composite actions (loops, conditionals);
- computation of arithmetical expressions.

3. Problem

We focus on AADL models with behavior specifications set using Behavior Model Annex language.

We consider a BA behavior specification of a single component in a model. Also, we consider two states s_{start} and s_{end} of the automaton are given.

We want to estimate the maximum and minimum model time the BA automaton will consume to go out from state s_{start} and to come to s_{end} .

4. Solution

Automaton can reach a given state starting from another given state in several ways depending on variables state, external events and nondeterminism. We will call an interleaving sequence of states and transitions as a *path* in automaton.

Thus we divide the original problem to considering a single path in automaton and then considering the automaton itself as a source of paths.

4.1. Estimation for a path

First, let us look at a finite path starting and ending at given states s_{start} and s_{end} , and going through states s_1, s_2, \dots, s_n , which could be equal to each other and to states s_{start} and s_{end} . We would designate it as $s_{start} \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow s_{end}$. The question is how long does it take to go along this path out from s_{start} to s_{end} .

Some of states in the path may be complete. An automaton is waiting for external events in these states while going through them. It is a hard task to estimate how much

time would it take because it is not a local property, i.e. it depends on other components in the model.

Execution states do not consume any time by definition, thus there is no such problem for them.

Also, in BA actions assigned to transitions can take some time (*e.g. computation action takes time, which is specified with its argument; input/output operations may take time too*). Time taken by composite actions (loops and conditionals) depend on very actions inside them and external conditions (state of variables and ports). Having dependency on external conditions, estimation of time consumption by conditionals it a tricky task (undecidable in the general case).

Thus, task of estimation of time, taking by execution of a finite path, can be split into two tasks: time estimation for each complete state in the path and for each list of actions assigned to a transition in the path.

4.2. Estimations for an automaton

The whole automaton containing both execution and complete states is a challenging object. Let us at first consider simpler kind of automata containing only execution states and then to consider the general case.

4.2.1 Automata with execution states only

In this case, automaton is not waiting for external events and goes through states right away. We can represent such automaton as a weighted graph. Vertexes of the graph are states of the automaton, and edges of the graph are transitions of the automaton. Weight of each edge is time estimation for the actions of corresponding transition.

We can use all known algorithms for finding minimum and maximum times (*e.g. for finding minimum time we can use Dijkstra's algorithm [4]*).

However, when the graph is cyclic these estimations can be inaccurate. *For example, we have a loop of the automaton, which is executed exactly 50 times. If this fact is not used, estimation of the time consumption of this loop may be too imprecise, up to $+\infty$ for the higher bound and to 0 for the lower bound. Considering information of the number of loop iterations, we can estimate the time to be $50t_{body}$ where t_{body} is an estimation of the time consuming by the loop body, or even more precise if t_{body} depends on the loop iteration number in a known way.*

Despite inaccuracy in some cases, time estimation for this kind of automaton is a pretty studied problem.

4.2.2 Automata with complete states too

Approaches with simple weighted graphs with weights only on edges do not model the fact that automaton can wait some time in a complete state during its execution. But we work with automata having complete states. Thus, we need to manage with it somehow while estimating automata execution time.

It seems that this problem can be reduced to the previous one, e.g. though replacing a single complete state with two connected execution states with a transition consuming the same time as automaton waits in this complete state.

But what we realized trying to implement such approach is that time of waiting in a complete state is not local and cannot be represented by some constant. This time actually depends both on the way this state was reached and on how regular external events occur. So, automata with complete states need special treatment, one variant of which will be discussed below.

4.2.3 Solution structure

So, to solve the original task we have divided the original problem to the following subtasks:

- estimation of time consumption of paths in automaton:
 - estimation of execution time for transitions;
 - estimation of time of waiting in complete states;
- estimation of time consumption by automaton itself:
 - in a particular case, when the automaton contains only execution states;
 - in the general case, when automata with both complete and execution states are considered.

The rest of the paper follows this division.

5. Estimation of time for paths

5.1. Estimation of time for transitions

Let us estimate how much time can take different Behavior Annex actions. At first, look at simple actions.

The action *computation* has a time as an argument, which is the execution time of this action.

Also, the action *get resource* can take some time, because at the moment when this action is executed, needed resource can be used by some other component. And so it will be necessary to wait for some time until the resource can be used. We will estimate this time from 0 to $+\infty$.

If action *stop* occurs at some point, then the execution of automaton became interrupted and it does not go to the next state. The action does not take time. However, since we are interested in the time between the states of the automaton, it is convenient to assume that the time of this action is $+\infty$. Indeed, if the transition from s to q with action *stop* exists, it means that automaton will not ever be in state q after this transition.

Now let us consider composite actions. Loops which contains the actions occupying some time, we will estimate with time from 0 to $+\infty$. Making this estimation to be

more accurate is possible but it is not considered in this paper. Other loops do not take any time.

We will estimate conditional constructs in the following way. Time of actions in if-block is from t_1 to t_2 , time of actions in else-block is from τ_1 to τ_2 (if there is no else-block $\tau_1 = \tau_2 = 0$). Then the estimation is the time range from $\min(t_1, \tau_1)$ to $\max(t_2, \tau_2)$.

In this way, estimations for transitions of the automaton can be performed. Now let us estimate time, that automaton is waiting in complete states.

5.2. Estimation of time for complete states

Behavior Annex allows to handle two types of external events: receiving a message to input port and a dispatch signal.

At first, look at the first type of events. Since the expectation of the receiving message can take arbitrarily much time, we will estimate this time with 0 to $+\infty$. So, this is the estimations of time of waiting in the complete states for the external event of the first type.

Estimations of time waiting for events of the second type can be performed in same way. But the estimations can be more accurate when the component is a thread. This is due to the fact, that AADL allows to set properties for the thread, which determined how often dispatch signal arrives to the thread (*such properties are Dispatch Protocol and Period*).

These properties determine the time between neighboring complete states in automaton. Consider any path in an automaton, which starts and ends in complete states, all other states are execution states, and the transition from the first complete state is the transition of the second type. Above AADL-properties can determine the execution time of this path from going out from the first complete state to going out from the second complete state. This time is determined by time range with possibly infinite bounds.

In this way, when automaton comes to complete state, the waiting time in this state is determined by the time elapsed from going out from the previous complete state and by AADL-properties.

6. Estimation of time for the whole automaton

6.1. Particular case, execution states only

6.1.1 Problem

The weighted oriented graph $G = \{V, E\}$ and two vertices s_{start}, s_{end} are given. The weights of the edges are determined by the function $w: E \rightarrow \mathbb{R}^2$.

Weight of each edge is a range of two real numbers $[r_1, r_2]; r_2 \geq r_1$, where r_1 is the lower bound, r_2 is the upper bound of the range. Weights are partially ordered in the following way:

$$[r_1, r_2] < [q_1, q_2] \Leftrightarrow r_2 < q_1.$$

Also, the addition function for weights is determined:

$$[r_1, r_2] + [q_1, q_2] = [r_1 + q_1, r_2 + q_2].$$

The task is to find the maximal and minimal weight of paths from s_{start} to s_{end} , where weight of a path is a sum of weights of path's $s_{start} \rightarrow \dots \rightarrow s_{end}$ transitions counted with multiplicity.

For example, we will consider the graph on the fig. 2 and vertices s_0 and s_6 as s_{start} and s_{end} respectively.

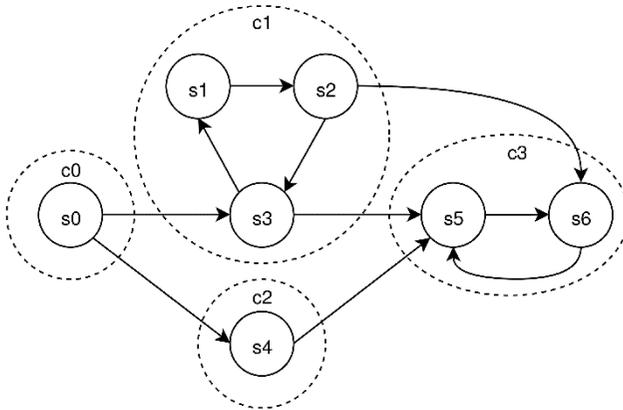


Fig. 2. Graph G and strongly connected components

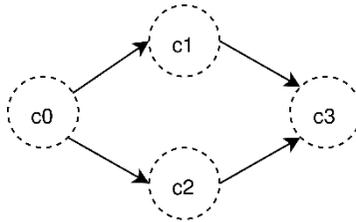


Fig. 3. Graph E

6.1.2 Algorithm

- 1) We find strongly connected components (SCC) in graph G with Tarjan's algorithm [5]. Strongly connected components of the graph G are highlighted by a dotted line on fig. 2.
- 2) We build acyclic graph E from strongly connected components of the graph G (fig. 3).
- 3) Let vertices s_{start} and s_{end} belong to strongly connected components c_{start} and c_{end} respectively. Then we find all paths in acyclic graph E (we

- call them SCC-paths) from c_{start} to c_{end} . *In the example, all paths from c_0 to c_3 are $c_0 \rightarrow c_1 \rightarrow c_3$ and $c_0 \rightarrow c_1 \rightarrow c_3$.*
- 4) For each SCC-path $c_{start} \rightarrow c_1 \rightarrow \dots \rightarrow c_{n-1} \rightarrow c_{end}$ we pick vertices from each SCC and consider the following path through them:
 $(s_{start} \Rightarrow s_0^{out}) \rightarrow (s_1^{in} \Rightarrow s_1^{out}) \rightarrow \dots \rightarrow (s_{n-1}^{in} \Rightarrow s_{n-1}^{out}) \rightarrow (s_n^{in} \Rightarrow s_{end})$,
 where $s_{start} \in c_0$, $s_{end} \in c_n$, $s_i^{in}, s_i^{out} \in c_i$, $i = 1, 2, \dots, n$, and edges $(s_j^{out} \rightarrow s_{j+1}^{in}) \in E$, $j = 1, 2, \dots, n - 1$. We will designate such paths as p_{picked} . Designation $s_i^{in} \Rightarrow s_j^{out}$ represents an automaton path from state s_i to state s_j inside a single SCC-component. Vertices s_i^{in} and s_i^{out} can be the same. *On the fig. 4 all paths are presented.* Notice that number of such paths is finite because each SCC-path is finite.
 - 5) Let us find the weight of each path p_{picked} . Weight of each transition $s_i^{out} \rightarrow s_j^{in}$ is equal to weight of edge (s_i^{out}, s_j^{in}) of graph G . To estimate weight of transitions $s_i^{in} \Rightarrow s_i^{out}$, $i = 1..n - 1$, we consider two cases.
 Case 1: c_i is acyclic (thus containing a single vertex), then weight of the transition $s_i^{in} \Rightarrow s_i^{out}$ is 0.
 Case 2: c_i is cyclic, then upper bound of weight of the transition $s_i^{in} \Rightarrow s_i^{out}$ is positive infinity, and the lower bound is calculated using Dijkstra's algorithm [4].
 - 6) For possibly infinite set of paths between s_{start} and s_{end} we have considered finite set of p_{picked} paths. We calculated weight of each p_{picked} path, got a finite set of weights. Thus, we can pick maximal and minimal ones.

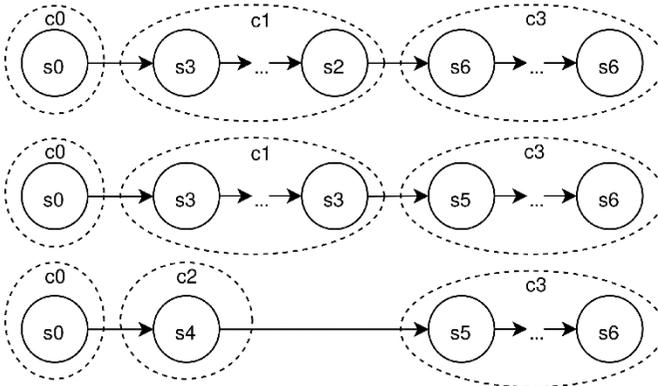


Fig. 4. Paths in graph G from s_0 to s_6

6.2. General case, both execution and complete states

6.2.1 Problem

The Behavior Annex automaton and two states of the automaton are given. The problem is to find estimation of the execution time of the automaton between leaving the state S_{start} and entering the state S_{end} .

We designate the set of states of the automaton as S . The set of execution states of the automaton is $Exec \subset S$, the set of complete states of the automaton is $Comp \subset S$.

For example, let us consider the automaton on fig. 5. Complete states are marked by white color, execute states are gray. The goal is to find time between state $e2$ and state $c2$.

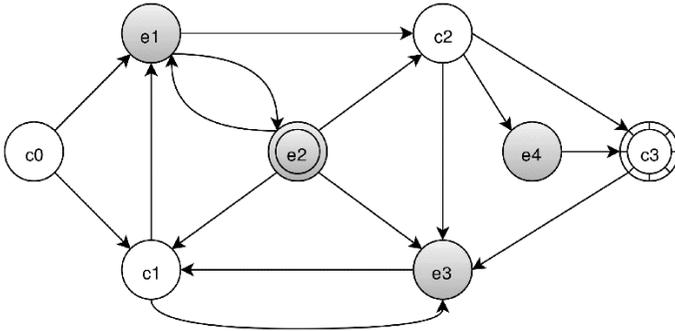


Fig. 5. Graph with complete states and execution states

6.2.2 Solution idea

Two different states types are determined in Behavior Annex. So we consider two different graphs.

We consider graph of the complete states and the graph of the execution states separately. Then if we need to find time between exit from one complete state to exit from other complete state, we use graph of complete states. In other cases we use the graph of execution states.

6.2.3 Algorithm

At first, we introduce few functions.

Function $PREV: S \rightarrow Comp$ computes all *previous complete states* for a state of the automaton, i.e. those complete states starting with which it is possible to reach the state through only execution states. More formally, $\forall s \in S \forall c \in Comp: c \in PREV(s) \Leftrightarrow \exists (c \rightsquigarrow s)$, where $c \rightsquigarrow s$ means $(c \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n \rightarrow s)$, with $n \geq 0$, $e_1, e_2 \dots e_n \in Exec$.

Function $NEXT: S \rightarrow Comp$ computes all possible *next complete states* for a state of the automaton, i.e. those complete states, which can be reached from the state through only execution states. More formally, $\forall s \in S \forall c \in Comp: c \in NEXT(s) \Leftrightarrow \exists (s \rightsquigarrow c)$. It is easy to see that $\forall c_1, c_2 \in Comp: c_1 \in PREV(c_2) \Leftrightarrow c_2 \in NEXT(c_1)$.

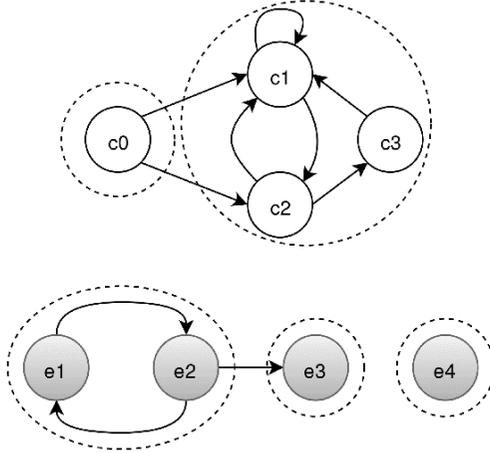


Fig. 6. Graph G_e and graph G_c

6.2.3.1 General scheme

We have two states $s_{start}, s_{end} \in S$. The aim is to find the minimum and the maximum possible time between leaving the state s_{start} and entering the state s_{end} . We will do this by estimation of time for each path $s_{start} \rightarrow \dots \rightarrow s_{end}$. The problem is that execution time of the path depends on complete states before state s_{start} , if s_{start} is execution state.

We will consider two cases: when s_{start} is complete state, and when s_{start} is execution state.

When s_{start} is complete state, each path $s_{start} \rightarrow \dots \rightarrow s_{end}$ can be divided into smaller paths: $s_{start} \rightarrow \dots \rightarrow c_{in}$ and $c_{in} \rightsquigarrow s_{end}$, where $c_{in} \in PREV(s_{end})$. For each $c_{in} \in PREV(s_{end})$ time of the path $s_{start} \rightarrow \dots \rightarrow c_{in} \rightsquigarrow s_{end}$ is $T(s_{start} \rightarrow c_{in}) + t(c_{in} \rightsquigarrow s_{end})$, where $T(s_{start} \rightarrow \dots \rightarrow c_{in})$ is time between leaving s_{start} and leaving c_{in} , and time $t(c_{in} \rightsquigarrow s_{end})$ is time between leaving c_{in} and entering s_{end} . Notice that times T and t can be different for the same path, when the last state of the path is complete state. The ways of estimation of time $T(c_i \rightsquigarrow c_j)$ were described in section 5.2.

When s_{start} is execution state, each path $s_{start} \rightarrow s_{end}$ is a part of path like $c_{out} \rightsquigarrow s_{start} \rightsquigarrow c_{med} \rightarrow \dots \rightarrow c_{in} \rightsquigarrow s_{end}$ where $c_{med} \in NEXT(s_{start})$, $c_{out} \in PREV(s_{start})$, $c_{in} \in PREV(s_{end})$. Time of the path $s_{start} \rightarrow \dots \rightarrow s_{end}$ can

be computed as $T(c_{out} \rightsquigarrow c_{med}) - t(c_{out} \rightsquigarrow s_{start}) + T(c_{med} \rightarrow \dots \rightarrow c_{in}) + t(c_{in} \rightsquigarrow s_{end})$.

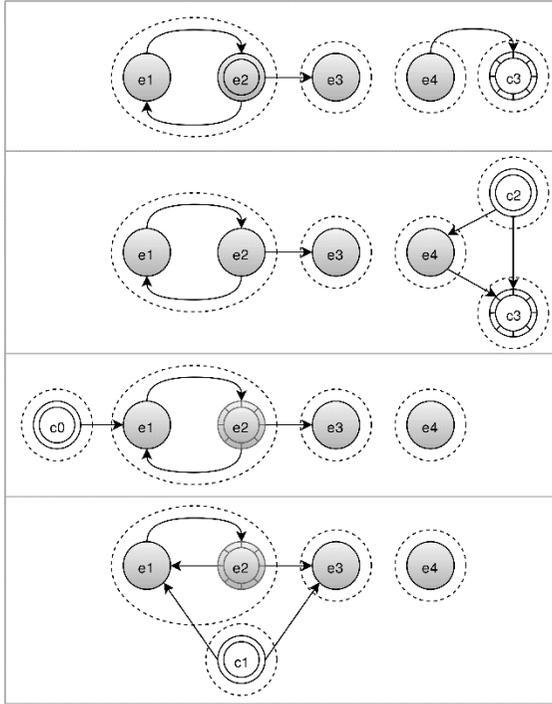


Fig. 7. Usage of graph G_e : graphs $G'_e(e_2, c_3)$, $G'_e(c_2, c_3)$, $G'_e(c_0, e_2)$, $G'_e(c_1, e_2)$.

6.2.3.2 Calculation of T

Let us focus on the function T . Value of T is described in section 5.2 for paths $c_i \rightsquigarrow c_j$, where $c_i, c_j \in Comp$. To find time T for arbitrary paths $(c_i \rightarrow \dots \rightarrow c_j)$ we build weighted oriented graph G_c . The vertices of the graph G_c are complete states of the automaton. We build edge (c_i, c_j) , if a path $c_i \rightsquigarrow c_j$ exists in the automaton. Weights of edges are determined with AADL-properties of the component as described in section 5.2, i.e. weight of an edge (c_i, c_j) equals to $T(c_i \rightsquigarrow c_j)$. Graph G_c for the considered example is presented on fig. 6. To find time $T(c_i \rightarrow \dots \rightarrow c_j)$ we execute the algorithm described in section 6.1 on graph G_c .

6.2.3.3 Calculation of t

To find time $t(s_1 \rightsquigarrow s_2)$ we build weighted oriented graph G_e . The vertices of the graph G_e are all execution states of the automaton. For each transition $e_1 \rightarrow e_2$ of the

automaton we build edge (e_1, e_2) in graph G_e . The weight of this edge is time estimation for transition's actions (see section 5.1). Graph G_e can be not connected. Graph G_e is presented on the top of fig. 6.

With graph G_e we can estimate time $t(s_1 \rightarrow s_2)$. To do this we build new graph $G'_e(s_1, s_2)$. Vertices set of graph $G'_e(s_1, s_2)$ is union of states set of G_e and $\{s_1, s_2\}$. It contains all edges from G_e . Additionally, it contains all edges, which are corresponding to outgoing transitions of automaton from state s_1 to vertices from $G'_e(s_1, s_2)$ and incoming transitions from vertices of $G'_e(s_1, s_2)$ to s_2 . To find $t(s_1 \rightarrow s_2)$ we execute the algorithm from section 6.1 on graph $G'_e(s_1, s_2)$.

On the second line of fig. 7 the graph G_e for calculating the time between exit from complete state c_2 to enter to complete state c_3 is presented.

6.2.3.4 Calculation of the result

For each path $s_{start} \rightarrow \dots \rightarrow s_{end}$ we calculate time estimation. The result of the algorithm is the smallest time range, that contains all these time ranges.

7. Related works

One close problem to the problems, considered in this paper, is WCET problem. This problem is well-known, and a lot of algorithms solving WCET exist. But these algorithms cannot be applied to our problem directly, due to considered specific object class, defined by Behavior Annex language. As Behavior Annex describes behavior based on timed automata, consider WCET algorithms working on timed automata.

The WCET problem for timed automata was considered in the paper [6]. This paper has a description of the algorithm using the difference-bound matrix data structure to represent zones (heuristic). This algorithm can be applied in the particular case, which was described in section 6.1.

The main specific construct in Behavior Annex is complete states. In the particular case we consider automata with only execution states. These automata are very similar to timed automata from the paper [6]. It means that algorithms from the paper can be applied to the particular case. We are thinking about applying it, but currently we have chosen simpler algorithm.

But to use it in the general case from 6.2, it should be adapted. We have decided that the adaptation of the algorithm would be harder, than to develop the new algorithm applied to a needed object class.

8. Conclusion

In this paper, the development of mission-critical systems is considered. In this context, we have considered the task of correct integration of the whole system. System modelling with language AADL and analysis of models are using to solve the task.

The problem is that a component of an AADL model can have behavioral properties set. At the same time the behavior of the component can be set with Behavior Model Annex. That can lead to inconsistency of the model. So, we considered a task of automated analysis of behaviors in AADL-models.

In this paper, one static approach for analysis of timing properties is proposed. An algorithm for finding of execution time estimation of behaviour of AADL-components was offered and described in the paper. This algorithm was implemented in MASIW, a framework for development and analysis of AADL models [7].

Characteristics of behaviors, acquired using proposed algorithm can be used for checking of model consistency and for model refinement, when AADL-properties are not set.

References

- [1]. B. C. Watkins, "Transitioning from federated avionics architecture to Integrated Modular Avionics", AIAA 26th Digital Avionics Systems Conference, 2007.
- [2]. Architecture Analysis & Design Language (AADL), SAE International standard AS5506B, SAE International, 2012, <http://standards.sae.org/as5506b/>.
- [3]. Architecture Analysis & Design Language (AADL), Annex Volume 2, Behavior Model Annex, SAE International, 2011, <http://standards.sae.org/as5506/2/>.
- [4]. E.W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*, 1959.
- [5]. R.E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, 1972.
- [6]. O. I. Al-Bataineh, "Verifying worst-case execution time of timed automata models with cyclic behaviour". Ph. D. dissertation, School of Computer Science & Software Engineering, 2015.
- [7]. D. Buzdalov, S. Zelenov, E. Kornychin, A. Petrenko, A. Strakh, A. Ugnenko, and A. Khoroshilov, "Tools for system design of integrated modular avioics". *Trudy ISP RAN/Proc. ISP RAS*, vol. 26, issue 1, 2014, pp. 201-230 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-6

Способ статической оценки времени работы компонентов AADL-моделей

А.М. Троицкий <troitskiy@ispras.ru>

Д.В. Буздалов <buzdalov@ispras.ru>

*Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

Аннотация. При проектировании современных систем авионики, а также других ответственных систем, неотъемлемой частью разработки является моделирование этих систем. Модели могут использоваться для проверок и валидации системы, в том числе на ранних этапах разработки. Ранняя валидация важна из-за того, что стоимость исправления ошибок растёт экспоненциально от времени внесения этой ошибки. Для

моделирования такого рода систем широко используется язык моделирования AADL, позволяющий моделировать как архитектуру разрабатываемых систем, так и некоторые поведенческие характеристики компонентов модели. В статье рассматривается задача автоматизированной проверки модели на консистентность некоторых поведенческих свойств. В частности, рассматривается проблема оценки времени работы компонентов моделей и соответствия этого времени другим свойствам в модели. Эта проблема близка к проблеме худшего времени выполнения (WCET), но имеет свою специфику в данном приложении. Рассмотрен статический подход, работающий со стандартной спецификацией поведения компонентов AADL-моделей специализированными расширенными конечными автоматами. В статье были рассмотрены особенности используемой модели поведения (специализированных конечных автоматов), в частности, за счёт работы автомата со временем и внешними событиями. Были рассмотрены проблемы оценки времени работы таких моделей поведения, связанные с нелокальностью этой характеристики в ряде случаев. Был рассмотрен важный частный случай, а также общий случай этой проблемы. В статье предлагается алгоритм, позволяющий оценить время работы таких моделей поведения в этих случаях. Данные алгоритм реализован и используется в среде разработки AADL-моделей APM СИ (MASIW).

Ключевые слова: AADL; авионика; статический анализ.

DOI: 10.15514/ISPRAS-2016-28(2)-10

Для цитирования: Троицкий А.М., Буздалов Д.В. Способ статической оценки времени работы компонентов AADL-моделей. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 157-172 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-10

Список литературы

- [1]. B. C. Watkins, "Transitioning from federated avionics architecture to Integrated Modular Avionics", AIAA 26th Digital Avionics Systems Conference, 2007.
- [2]. Architecture Analysis & Design Language (AADL), SAE International standard AS5506B, SAE International, 2012, <http://standards.sae.org/as5506b/>.
- [3]. Architecture Analysis & Design Language (AADL), Annex Volume 2, Behavior Model Annex, SAE International, 2011, <http://standards.sae.org/as5506/2/>.
- [4]. E.W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*, 1959.
- [5]. R.E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, 1972.
- [6]. O. I. Al-Bataineh, "Verifying worst-case execution time of timed automata models with cyclic behaviour". Ph. D. dissertation, School of Computer Science & Software Engineering, 2015.
- [7]. Д.В. Буздалов, С.В. Зеленов, Е.В. Корныхин, А.К. Петренко, А.В. Страх, А.А. Угненко, А.В. Хорошилов, "Инструментальные средства проектирования систем интегрированной модульной авионики", Труды ИСП РАН, том 26, выпуск 1, 2014 г., стр. 201-230. DOI: 10.15514/ISPRAS-2014-26(1)-6

Practical experience of software and system engineering approaches in requirements management for software development in aviation industry

I.V. Koverninskiy <ivkoverninsk@2100.gosniias.ru>

A.V. Kan <avkan@2100.gosniias.ru>

V.B. Volkov <vbvolkov@2100.gosniias.ru>

Yu. S. Popov <yspopov@2100.gosniias.ru>

N.K. Gorelits <nkgorelits@2100.gosniias.ru>

*State Research Institute of Aviation Systems,
125319, Russia, Moscow, Viktorenko Str, 7*

Abstract. The article describes the technical world evolution tendencies, which require proper software and system engineering approaches used for complex systems creation, for example for aircrafts creation. The substantiation of the importance and relevance of using requirements management discipline in software development is made. The main basics of software and system engineering approaches and discipline are set out. System engineering is a discipline, which integrates and harmonizes all activities around entire area of systems creation. The article contains description of information systems, which have been created in GosNIIAS and now are actively used in internal and external works: requirements management information system, problem reports management information system, technological environment for test methods preparation and test results registration. Requirements management information system contains special predefined documents and template, required by standards DO-178, DO-254, DO-330, ARP4754, State Standards GOST 51904 and GOST 34. Using of requirements management system in GosNIIAS and external enterprises is described. Problem reports management information system registers and supports the lifecycle of problem reports, which appear during the work process. Technological environment for test methods preparation and test results registration supports different activities such as test methods, test cases, test procedures preparation and testing on the integration stand, test results registration and test protocols preparation. Some perspective directions of software and system engineering approaches applying in GosNIIAS are listed.

Keywords: software engineering; system engineering; requirements management; complex on-board equipment; aircraft design.

DOI: 10.15514/ISPRAS-2016-28(2)-11

For citation: Koverninskiy I.V., Kan A.V., Volkov V.B., Popov Yu.S., Gorelits N.K. Practical experience of software and system engineering approaches in requirements management for software development in aviation industry. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 173-180. DOI: 10.15514/ISPRAS-2016-28(2)-11

1. Introduction

Nowadays there is a considerable change in industries all over the worlds. The change is related with the rapidly increasing complexity level of systems and devices, which are created and used.

Safety and reliability requirements to products of aerospace, defense and other industries become stricter as well as certification requirements to management processes of products creation. At the same time we have to use new industry standards.

Aerospace imposes some restrictions and requirements on the software development process and its result. These restrictions are caused by safety requirements to the aircrafts on which the software will be used. Requirements are set out in the industry standards, these standards must be complied very carefully for high quality results and successful certification.

2. Software and system engineering approaches realization

Using and customizing software and system engineering processes and approaches are an appropriate response to technical world complication tendencies. These processes and approaches are base of the most standards and guidelines which define methods to achieve necessary safety and reliability levels during development, design and engineering of critical technical and software systems.

Nowadays software in complex technical systems is responsible for executing of the most critical functions [1].

The most important discipline of software and system engineering for software development is requirements management. If there is no requirement management process or its bad realization then obvious or hidden defects and faults appear. It takes more and more efforts to repair these defects and faults at the later stages of development lifecycle.

Problems in requirements are leaders in projects failures reasons lists and rework costs lists (Standish Group reports).

That's why requirements are mandatory basis of design and development processes according to guidelines of standards R4754 (R4754A is now a draft, it is Russian analogue of ARP 4754), KT-178 (DO-178), KT-254 (DO-254), DO-330, GOST R 51904. Development of the software, hardware and systems begins from creation of requirements. Design is based on requirements. We also have to inspect how result corresponds with initial requirements during verification, validation, testing processes.

Some important tasks arose GosNIIAS due to the changes in the world. These tasks were about modernization of existing approaches and work processes in order to minimize potential risks for software design and development [2].

A number of current situation researches were done in GosNIIAS. Existing world approaches to the software and system engineering approaches were adapted considering the specialization of the institute. The results of analysis and adaptation as well as software and system engineering fundamental principles formed the basis of newest works of GosNIIAS.

Fundamentals of software and system engineering:

- Requirements are base of software development process,
- There should be coherent architecture of modules/subsystems and communication interfaces (points of input and output) between modules should be predefined,
- Verification process (product check for requirements compliance) should be organized for cases when accurate measurement is impossible,
- Modeling approaches and then model verification and validation are used for earlier failures and bug detection,
- Communication protocols between process participants should be defined like strict regulations.

Nowadays GosNIIAS has built the number of systems accordingly to software and system engineering approaches. The list of created systems consists of the following systems:

- Requirements management information system,
- Problem reports management information system,
- Technological testing environment,
- Practical approaches and skills in software and system engineering adapted for real tasks.

2.1 Requirements management information system

Requirements management information system (RMIS) was created for support requirements management activities in design and development of complex systems like aircraft onboard software.

RMIS processes are built based on R4754 (ARP 4754) processes.

RMIS realizes such functions and processes like:

- Cross-cutting requirement management process during the software and system development entire lifecycle,
- Single requirements change and configuration management process,
- All necessary lifecycle artifacts tracing,

- Generation and publishing of reporting documents and documents with any necessary data in accepted formats.

Documents and projects templates required by standards R4754, KT-178, KT-254, DO-330, GOST R 51904, GOST 34 are created and included in RMIS suite. These items allow to decrease labor costs for audit preparation and passage in certification authorities – processes and products must strictly comply the standards.

Some methodological materials were made to help with requirements management and configuration management using RMIS.

Using RMIS while designing and developing aircrafts allows to significantly reduce:

- Efforts for execution of works,
- Time for approval, negotiation and final products release,
- Errors from difficult work with requirements,
- Provides actual information to all the participants during entire development lifecycle.

This way RMIS gives opportunities to make reasonable and timely decisions.

RMIS was successfully implemented in some organizations. The list of successful users of RMIS in aviation industry includes companies such as GosNIIAS, SpecTechnica, Techodinamika and others.

GosNIIAS effectively uses RMIS in testing avionics processes on integration stand for Irkut MS-21 aircraft. RMIS's database contains traced data from AP-25 (like EASA CS-25, FAR-25 – Airworthiness standards for transport categories airplanes), Certification basis, Special technical conditions and some other data for Irkut MS-21 aircraft. There is active ongoing process of creation, customization and implementation of requirements management process, configuration management process, verification and validation management process in GosNIIAS.

2.2 Problem reports management information system

Specialists from GosNIIAS also made Problem reports management information system (PRMIS) during MS-21 project. PRMIS allows support of problem reports management activities on testing avionics processes on integration stand for MS-21 aircraft.

PRMIS processes are built on the base of R4754A (R4754A's part about problem reports activities). Main of PRMIS tasks are

- Collection and storage data of problem situations,
- Problem analysis,
- Resolving problem documenting,
- other functions.

2.3 Technological environment for test methods preparation and test results registration

Technological environment for test methods preparation and test results registration (TET) was made during MS-21 project as well. TET allows support of test methods preparation and testing activities on integration stand for MS-21 aircraft's avionics testing. Processes of TET are built in accordance with industry standard R4754.

TET provides the following functions:

- Preparation of test programs, test methods, test cases and test procedures for avionics, integrated flight control system testing,
- Maintenance of testing activities on integration stand,
- Creating test reports,
- Other functions.

TET provides such opportunities as:

- Test methods approval processes,
- Test methods development history logging,
- Test results control and changing of succeeding test methods accordingly to revealed remarks for test requirements, hardware, methods, etc.

Some of TET goals are:

- Reducing labor costs for test methods, test procedures and test cases creation,
- Transparent control for finished tests considering received and registered test results,
- Increasing quality of tests traced with requirements, test methods and programs and received results,
- Possibility to work with the set of integrated hardware on the integration stand,

Information integration with RMIS, PRMIS and configuration control system for further integration in entire software and system engineering process of GosNIIAS, which will allow effective reusing of prepared test organization process for certification audit.

3. Current and future tasks

Nowadays there are actively realized system engineering approaches in GosNIIAS. Some tasks about development, design and implementation such processes of system engineering as requirement management process, problem reports management process, information management process, verification and validation management

process, version and configuration management processes during software and system development lifecycle processes.

Processes listed above and traced with its software and system engineering approaches will be performed for the further researches. Real-time operation system creation and creation of Russian instrumental set for support of the software and system engineering processes were chosen as nearest researches for perform these processes. There were defined some models for chosen researches – change request lifecycle processes model and problem report lifecycle processes model.

4. Conclusion

GosNIIAS has plans to create cross-cutting process based on developed processes and realized with software which is already developed and which will be developed soon. It should be cross-cutting process of software and system engineering with necessary instrumental support in GosNIIAS.

References

- [1]. G.A. Chuyanov, V.V. Kosyanchuk, N.I Selvesyuk, [Prospects of development of complex onboard equipment on the basis of integrated modular avionics], *Izvestiya SFedU [News of SFedU]*, vol. 3, pp. 55-62, March 2013 (in Russian).
- [2]. G.A. Chuyanov, V.V. Kosyanchuk, N.I Selvesyuk and S.V. Kravchenko, [Directions of perfection on-board equipment to improve aircraft safety], *Izvestiya SFedU [News of SFedU]*, vol. 6, pp. 219-229, June 2014 (in Russian).

Практический опыт реализации подходов программной и системной инженерии для управления требованиями при разработке программного обеспечения в авиационной отрасли

И.В. Ковернинский <ivkoverninsk@2100.gosniias.ru>

А.В. Кан <avkan@2100.gosniias.ru>

В.Б. Волков <vbvolkov@2100.gosniias.ru>

Ю.С. Попов <yypopov@2100.gosniias.ru>

Н.К. Горелиц <nkgorelits@2100.gosniias.ru>

Государственный Научно-исследовательский Институт

Авиационных Систем,

125319, Russia, Moscow, Viktorenko Str, 7.

Аннотация. В статье проанализированы тенденции развития окружающего технического мира, обязывающие к использованию процессов программной и системной инженерии при создании сложных систем в целом и воздушных судов в частности. Приведено обоснование важности и актуальности использования дисциплины управления требованиями при разработке программного обеспечения.

Изложены принципы, лежащие в основе программной и системной инженерии. Системная инженерия – это научно-методологическая дисциплина, интегрирующая множество дисциплин вокруг единой области создания систем. В статье описаны созданные и активно используемые в ГосНИИАС информационные системы: информационная система управления требованиями, информационная система управления сообщениями о проблемах, технологическая среда подготовки методик и учета результатов испытаний. Информационная система управления требованиями содержит документы и шаблоны для разработки и публикации требований, требуемые руководствами и стандартами КТ-178, КТ-254, ARP-4754, DO-330, ГОСТ 51904, ГОСТ 34. Описано использование информационной системы управления требованиями в ГосНИИАС и сторонних организациях. Информационная система управления сообщениями о проблемах регистрирует и сопровождает жизненный цикл выявляемых в ходе работ проблем. Технологическая среда подготовки методик и учета результатов испытаний поддерживает деятельность по подготовке программ и методик испытаний, тестовых случаев и тестовых процедур, проведению испытаний на интеграционном стенде отработки программного обеспечения имитационной среды КБО самолета МС-21, подготовке протоколов испытаний. Описаны текущие работы ГосНИИАС в области развития и внедрения новых процессов и подходов. Приведены некоторые перспективные направления практического применения подходов программной и системной инженерии в ГосНИИАС.

Ключевые слова: программная инженерия; системная инженерия; управление требованиями; комплекс бортового оборудования; проектирование воздушного судна.

DOI: 10.15514/ISPRAS-2016-28(2)-11

Для цитирования: Ковернинский И.В., Кан А.В., Волков В.Б., Попов Ю.С., Горелиц Н.К. Практический опыт реализации подходов программной и системной инженерии для управления требованиями при разработке программного обеспечения в авиационной отрасли. *Труды ИСП РАН*, том 28, вып. 2, 2016 г., стр. 173-180 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-11

Список литературы

- [1]. Г.А. Чуянов, В.В. Косьянчук, Н.И. Сельвесюк. Перспективы развития комплексов бортового оборудования на базе интегрированной модульной авионики. *Известия ЮФУ*. № 3, 2013 г., стр. 55-62.
- [2]. Г.А. Чуянов, В.В. Косьянчук, Н.И. Сельвесюк, С.В. Кравченко. Направления совершенствования бортового оборудования для повышения безопасности полетов воздушного судна. *Известия ЮФУ*. №6, 2014 г., стр. 219-229.

Design and architecture of real-time operating system

^{1, 2} *K.M. Mallachiev <mallachiev@ispras.ru>*

^{1, 2, 3} *N.V. Pakulin <npak@ispras.ru>*

^{1, 2, 3, 4} *A.V. Khoroshilov <khoroshilov@ispras.ru>*

¹ *Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.*

² *Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

³ *Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*

⁴ *National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. Modern airliners such as Airbus A320, Boeing 787, and Russian MS-21 use so called Integrated Modular Avionics (IMA) architecture for airborne systems. This architecture is based on interconnection of devices and on-board computers by means of uniform real-time network. It allows significant reduction of cable usage, thus leading to reducing of takeoff weight of and airplane. IMA separates functions of collecting information (sensors), action (actuators), and avionics logic implemented by applied avionics software in on-board computers. International standard ARINC 653 defines constraints on the underlying real-time operation system and programming interfaces between operating system and associated applications. The standard regulates space and time partitioning of applied IMA-related tasks. Most existing operating systems with ARINC 653 support are commercial and proprietary software. In this paper, we present JetOS, an open source real-time operating system with complete support of ARINC 653 part 1 rev 3. JetOS originates from the open source project POK, created by French researchers. At that time POK was the only one open source OS with at least partial support for ARINC 653. Despite this, POK was not feasible for practical usage: POK failed to meet a number of fundamental requirements and was executable in emulator only. During JetOS development POK code was significantly redesigned. The paper discusses disadvantages of POK and shows how we solved those problems and what changes we have made in POK kernel and individual subsystems. In particular we fully rewrote real-time scheduler, network stack and memory management. Also we have added some new features to the OS. One of the most important features is system partitions. System partition is a specialized application with extended capabilities, such as access to hardware (network card, PCI controller etc.) Introduction of system partitions allowed us moving large subsystems out of the kernel and limiting the kernel to the minimal functionality: context switching, scheduling and message pass. In particular, we have moved network subsystem to system partition. This

moving reduces kernel size and potentially reduces probability on having bug in kernel and simplifies verification process.

Keywords: ARINC 653; RTOS; IMA; partitioning; real-time.

DOI: 10.15514/ISPRAS-2016-28(2)-12

For citation: Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 181-192. DOI: 10.15514/ISPRAS-2016-28(2)-12

1. Introduction

Real-time Safety-critical systems have strong requirements in terms of time and resource consumption. Most of them have several concurrently executing separate functions (applications), which communicate from time to time. The most obvious approach is running those applications on separate devices and connecting to sensors and actuators by point-to-point link, on which applications should communicate. But firstly, there will be a lot of wires in large system. And secondly, having a separate computing node for periodic application, which is idle most of the time, results in a great number of computing nodes and high cost of hardware.

Integrated modular avionics (IMA) network is a solution to those problems in avionics. Core modules are main part of IMA network. Core module runs a real-time operating system (RTOS), which supports independent execution of several avionics applications that might be supplied by different vendors. System provides partitioning, i.e., space and time separation of applications for fault tolerance (fault of one application doesn't affect others), reliability and deterministic behavior. The unit of partitioning is called *partition*. Basically partition is the same as process in commodity operating systems. ARINC 653 standardizes constraints to the underlying RTOS and associated API. [1]

Civil aircraft airborne computers are mostly PowerPC architecture. In this paper we present the project on development of an open source ARINC 653 compatible operating system, which can run on PowerPC CPU and, in the future, on other CPU architectures, such as MIPS and x86.

1.1 Overview of ARINC 653

ARINC 653 is the standard for implementing IMA architecture; it defines general purpose Application Executive (APEX) interface between avionics software and underlying real-time operating system, including interfaces to control the scheduling, communication, concurrency execution and status information of its internal processing elements.

Key concept of ARINC 653 is partitioning of applications in integrated module by space and time. [2]. A partition is a partitioning program unit representing an application. Every partition has its own memory space, so one partition cannot get access to the memory of another. Partitions are executed in user (non-privileged)

mode, so errors in partition cannot affect OS kernel (which is executed in privileged mode) and other partitions. Partition consists of one or more processes, which operate concurrently. Processes in partition have the same address space and can have a different priority. Process has an execution context (processor registers and data and stack areas), and they resemble well-known concept of threads. Fig. 1 shows example architecture.

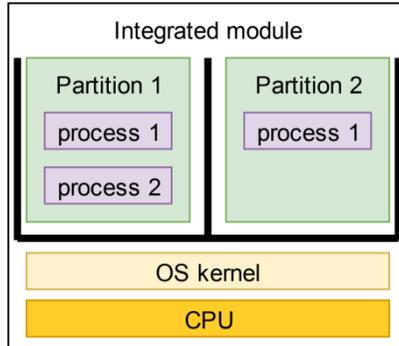


Fig. 1. Example module architecture

Partitions are scheduled using a simple round-robin algorithm. System defines a major time frame of fixed duration which is constantly repeated through integrated module execution time. Major frame is divided into several time windows. Each partition is assigned to one or more time windows, and partitions are running only during corresponding assigned time window. Assignment of time windows and major frame duration are statically configured by the system integrator, therefore scheduling is fully deterministic.

Scheduling of processes within partition is a dynamic priority based scheduling and communication and synchronization mechanism make it more sophisticated than partitions scheduling.

ARINC 653 provides interface for communication between applications (partitions), potentially running on different modules connected by onboard communication network. All inter-partition communication is conducted via messages. Message is a continuous block of data. The ARINC 653 interface doesn't support fragmented messages. Message source and destination are linked by channels; a channel links a single source to one or more destinations. Partitions have access to channels via defined access points called ports. Port has single direction; it can be either source or destination port. One port can be assigned only to one partition. Each partition can have multiple ports. It is even possible to have a channel where both source and destination ports are assigned to one partition

Partition code works with ports regardless of underlying channels. Channels are preconfigured statically.

To control the concurrent execution of processes ARINC 653 offers synchronization primitives such as semaphores, events and mutexes. Buffers and blackboards provide inter-process communication within a partition. Buffer is a messages queue, while blackboard has only one message, which is rewritten by every write operation.

2. Related works

ARINC-653 requirements results in constrains to underlying operating system. OS must support:

- space partitioning, so partitions have no access to memory areas of the other partitions and OS kernel;
- time partitioning, so not more than one partition can run at any time;
- strict and determinate inter-partition scheduler that ensures application response time.

Furthermore, in safety-critical systems the operating system must undergo certification process. As a result, size and complexity of OS become a real issue.

Popular real-time operating systems (such as RTERMS [3] and FreeRTOS[4]) don't support ARINC 653. Furthermore, RTERMS doesn't support memory protection.

Operating systems that satisfy all of these constrains are exist, but they are commercial and proprietary software. They are VxWorks[5] (by Wind River), PikeOS[6] (by Sysgo), LynxOS [7](by LynuxWorks).

There are research projects on real-time and ARINC 653 [12] enhancements of Linux. But Linux is a large system, so certification of Linux kernel seems impossible.

There are research projects that exploit the virtualization technology to support ARINC 653. But they are either proprietary like LithOS[8] (works over open hypervisor XtratuM[9]), or limited prototype VanderLeest implementation of ARINC 653 over Xen [10].

Only POK operating system [11], which is available under BSD license terms, mostly satisfies our requirements, so we decided to fork POK and continue its development.

3. POK

POK is a partitioned operating system focused on safety and security [11]. We describe it in detail here since it is the basis for the JetOS that we are working on.

POK has been designed for x86 and ported to PowerPC (PReP) and Sparc. POK has two layers: kernel and partition, where services of partition layer run at low-privileged level (user mode), and kernel services are executed at high-privileged level (kernel mode). Besides the kernel POK provides a library for partition code (libpok), which translates ARINC 653 API to POK kernel syscalls. Fig 2 shows POK architecture.

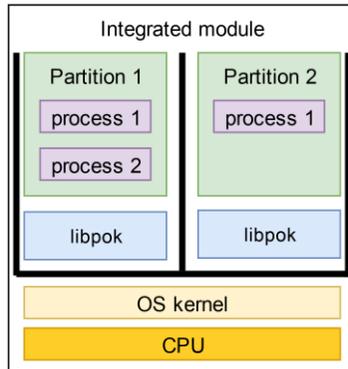


Fig. 2. POK architecture

We selected POK as the basis for our RTOS. Below in this paper we describe parts of POK that were changed or rewritten. We describe limitations of current implementation or architecture of these parts.

Partition management. POK provides partition isolation:

- in time by allocating fixed time slots for partitions in the schedule,
- in space by associating a unique memory segment to each partition.

Partition scheduling and memory management of POK partly comply the ARINC 653 specification. But PowerPC processor, on which we focus (P3041), doesn't support memory segmentation.

Processes management. POK supports ARINC 653 partition processes. All processes are represented in the kernel as array entries of a single processes array that stores process information for all partitions. POK has no logical separation in kernel representation of ARINC-653 processes of different partitions.

POK supports two intra-partition schedulers: Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF). Those partitions schedule processes within a partition when its time slot is active.

The problem with POK scheduler is that ARINC 653 requires much more from intra-partition scheduler: priority scheduling and fault management.

POK runs both inter- and intra-partition schedulers in the kernel mode.

Inter-partition communication. For every ARINC port there is a buffer of corresponding size inside the kernel. User code while sending to (or receiving from) port accesses those buffers by means of syscalls. At the beginning of every major frame POK copies data from source buffers to destinations. For large buffers there is possibility to spend significant part of partition time slot on buffer-to-buffer copying. If a process tries to send to a full port (or read from an empty one) the kernel blocks the process until buffer becomes operational. POK supported this feature but did not

obey to the ARINC-653 requirement on that the order of unblocking should be the same as the order of blocking on each priority level.

Intra-partition communication support is implemented by the user-mode library libpok, using system calls for synchronization purpose. It supports locking resources for concurrent access to shared data resources (such as buffer and blackboards) between processes in partition. When process tries to accesses a locked resource, it will be blocked (so scheduler will skip this process) until the resource is unlocked.

POK scheduler has some inherent problems with handling of locked processes. Let's consider an example. A low-priority locks a buffer for writing and before it unlocks the buffer a higher priority process wakes up. POK scheduler unconditionally switches to the second process. If the second process tries to get status information about the locked buffer it blocks and POK wakes the first process. But according to ARINC-653 standard the process that requests status information must not block.

4. JetOS

JetOS is the real time operating system with ARINC-653 support that we currently develop at ISPRAS. It originates from POK but has evolved significantly since then. Before we introduce the new features of JetOS compared to POK let us mention the facility that was removed from POK: the AADL configuration tool. Originally POK was designed and implemented as a demonstration of a number of approaches, and the developed selected rather exotic approach to configuration. The suggested way to create an embedded application by means of POK is to specify its environment and capabilities as an AADL specification. In JetOS we dropped AADL support in favor of XML-based configuration files.

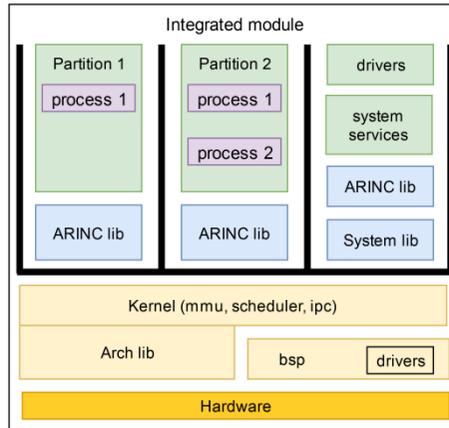


Fig. 3. JetOS architecture

Furthermore, we dropped support of the SPARC platform as there are no onboard avionics systems that are built atop of SPARC CPUs. At the moment JetOS runs on x86 and PowerPC (Book E branch).

Partition management. Unlike x86 and SPARC the new target hardware for JetOS, PowerPC platform, features direct MMU control through TLB writes. To reduce cache flushes at context switches and simplify TLB lookups PowerPC provides tagged cache where each tag is an 8 bit identifier. We use that identifier as partition identifier (pid). At context switch we just change value of the special-purpose register responsible for current pid. This is simple and secure method.

The inter-partition scheduler of POK was able to switch partitions only when the active process runs in user mode. If a process calls syscall it cannot be switched until the end of that call. Such behavior violates requirements of real-time since system calls might be prolonged. Currently we are working on kernel-mode critical section and synchronization primitives to enable context switch while a process executes a system call.

Processes management. We store process-related data in kernel separately for different partitions. Intra-partition scheduler was fully rewritten to support ARINC 653 specification. The new scheduling facility allows for multiple schedulers, and different partitions might utilize different schedulers (a.g. ARINC-653 for avionics applications and preemptive pthreads for system partitions). New intra-partition scheduler can be accessed only by functions

- start() is called when partition is starting or restarting
- on_event() is called on every event such as timer interrupt and returning control to partition.

Inter-partition communication. We use one ring buffer for every channel. Its size is the sum of source and destination ports buffers size in original POK design. It removes the need for copying from source to destinations buffers. Correct work of send and receive function achieved by two pointers, one for source port, and one for destination. Sending increases source port pointer, receiving increases destination port pointer. When pointers are met then buffer either full or empty, uncertainty is resolved by another variable associated with the channel, which stores current number of messages in the channel's buffer. Example can be seen at Fig. 4.

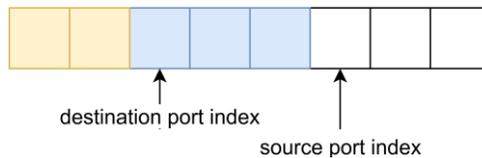


Fig. 4. Example kernel channel buffer. Yellow cells are already received messages, blue cells are sent but not yet received messages, white cells are empty

Intra-partition communication. Correct handling of concurrent data access to buffers and blackboards without violating the ARINC 653 scheduling requirements

with user mode scheduler is a hard task. Therefore the intra-partition schedulers are implemented in the kernel to simplify lock-wait-unlock and priority scheduling. In future versions we may design a solution that solves this issue while keeping a code in user space.

4.1 Configuration

The characteristic feature of real-time operating systems is deterministic behavior. The primary way to ensure reliable and dependable behavior is static pre-allocation of all resources – memory, CPU time, access to devices, etc. For instance, partition code is executed only during fixed time slots within the schedule, no sooner, but no later. Memory is pre-allocated for every partition, memory image of the partition is fixed, no pages could be added or removed during runtime.

Many parameters of our operating system are configured statically and cannot be changed dynamically. These parameters are number of partitions and their memory size, number of ports, their names, sizes and directions, channels etc.

Configuration of the system is stored in xml documents. To keep the kernel minimal we got rid of the need to include xml parser to kernel: the configuration files are processed at build time. The processor generates C code where parameters are presented as either preprocessor macros (`#define` constants) or enum constants. The generated files are included in the build process.

4.2 System partitions

Beside ordinary partitions, that interact with the kernel and the outer world through ARINC 653 APEX, the standard allows for so called *system partitions* that utilize interfaces outside the scope of APEX services, such as access to devices or network sockets. The standard doesn't specify their operations and interfaces other than constraints on time and space partitioning: system partitions are subject to scheduling. The difference between system partitions and kernel modules is that system partitions run in user space and have time and space partitioning constraints.

Our OS supports system partitions. From the kernel point of view system partitions are like ordinary partitions with some additional memory mapping and additional system calls. Communication between application partitions and system partitions is performed through ARINC-653 ports.

Currently we have only one system partition: the IO partition that is responsible for communication over the network. In the future we will implement a number of other system partitions – file system, graphics server,

IO partition has access (by corresponding entry in TLB) to special memory areas, where network card registers are mapped, so IO partition can work directly with hardware without kernel system calls.

IO partition receive and send data either from partitions in the same integrated module by ports or from other integrated modules by network card drivers. In the simple case the communication over network is based on UDP messages, and the configuration

defines mapping between ARINC 653 port and a pair of IP address and UDP port. This mapping looks like ARINC channel, so we also call it channel.

But network communication may be based on other protocols, such as AFDX. So in general, the channel maps ARINC port to some network specific data. We support parallel work with several network protocols, by assigning channel driver to channel. Channel driver is interlayer between port and device driver. In most cases channel driver is a network stack.

System can have several network cards, so we support parallel independent work of several device drivers. Currently we support three network cards drivers: virtio, ne2k family and hardware cards on the platform with P3041 processor.

Each network driver manages one or more uniform devices. During initialization each driver, which cards are connected through PCI bus, registers as PCI device in PCI driver. After initialization of all network drivers PCI driver starts enumeration of PCI bus. If it finds a physical device that matches a registered PCI device, then it signals to the corresponding network driver. Network driver dynamically for every signal registers a *network device*. Network device has a name and method to send and receive data from assigned physical device. Names to network device are assigned dynamically; name is concatenation of drivers name and sequential number of current device in driver.

The configuration assigns channel drivers to network devices by name. Example of sending two messages in parallel to two different network cards can be seen at Fig. 5.

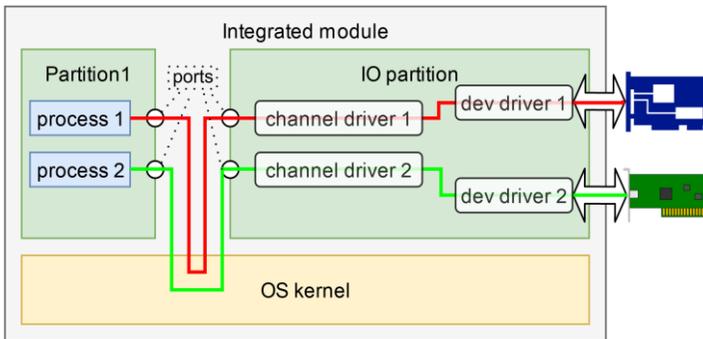


Fig.5. Two messages are being parallel sent to different network cards

Different drivers require different configuration. We have dedicated xml parsers of some specific part of xml document, this parser generates data specific for corresponding driver.

This architecture allows independent work of different drivers, which can possibly come from different developers. Furthermore, it allows adding new drivers with minimal effort and change of common parts.

5. Future work

There is research group to develop OpenGL renderer and frame buffer driver for our OS. Their work will show how well we thought out architecture of IO partition.

We finally need to measure latency without providing which we cannot tell that our operating system is a real-time system.

We are going to seek way to minimize kernel code, and move code, for which it is possible, to user-space.

Currently we use only one CPU core of the e500mc multicore processor. Newest version of ARINC 653 introduces interfaces for multicore work. We are going to support multicore CPUs as well.

Another objective is to port the OS to MIPS CPU family and another PowerPC family, namely IBM PPC 440.

6. Conclusion

In this paper, we sketched JetOS, a real-time operating system, which support ARINC 653 standard. Our system started as fork of POK OS. We describe architecture of POK, architecture of our operating system and differences between them.

References

- [1]. Avionics application software standard interface part 0 overview of ARINC 653, ARINC specification 653P0-1, August 3, 2015
- [2]. Avionics application software standard interface part 1 – required services, ARINC specification 653P1-3, November 15, 2010
- [3]. G. Bloom, J. Sherrill. 2014. Scheduling and thread management with RTEMS. SIGBED Rev. 11, 1 (February 2014), 20-25. DOI=<http://dx.doi.org/10.1145/2597457.2597459>
- [4]. C. S. Stangaciu, M. V. Micea, V. I. Cretu; Hard real-time execution environment extension for FreeRTOS Conference: IEEE International Symposium on Robotic and SEnsors Environments (ROSE 2014), At Timisoara DOI: 10.1109/ROSE.2014.6953035
- [5]. VxWorks 653 http://www.windriver.com/products/product-overviews/PO_VxWorks653_Platform_0210.pdf
- [6]. R. Kaiser, S. Wagner: Evolution of the PikeOS Microkernel, MIKES: 1st International Workshop on Microkernels for Embedded Systems. 2007
- [7]. LynxOS <http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/>
- [8]. M. Masmano, Y. Valiente, P. Balbastre, I. Ripoll, A. Crespo, J.J. Metge, 2010. LithOS: a ARINC-653 guest operating for XtratuM. In Proc. of the 12th Real-Time Linux Workshop, Nairobi (Kenya).
- [9]. M. Masmano, I. Ripoll, A. Crespo, and J.J. Metge. XtratuM: a Hypervisor for Safety Critical Embedded Systems. 11th Real-Time Linux Workshop. Dresden. Germany. http://www.xtratum.org/files/xm_rtlw09.pdf
- [10]. S. H. VanderLeest. ARINC 653 hypervisor. In Proc. Of IEEE/AIAA DASC, Oct. 2010.
- [11]. J. Delange, L. Lec, 2011. POK, an ARINC653-compliant operating system released under the BSD license. In 13th Real-Time Linux Workshop (Vol. 10). <http://julien.gunm.org/data/publications/articled111-osad111.pdf>

- [12]. S. Han and H.-W. Jin. 2012. Kernel-level ARINC 653 partitioning for Linux. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, 1632-1637. DOI=<http://dx.doi.org/10.1145/2245276.2232037>

Устройство и архитектура операционной системы реального времени

^{1, 2} К.М. Маллачиев <mallachiev@ispras.ru>

^{1, 2, 3} Н.В. Пакулин <npak@ispras.ru>

^{1,2,3,4} А.В. Хорошилов <khoroshilov@ispras.ru>

¹ Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

² Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.

³ Московский физико-технический институт (государственный университет)
141701, Россия, Московская область, г. Долгопрудный, Институтский пер., 9

⁴ Национальный исследовательский университет «Высшая школа экономики»
101000, Россия, Москва, ул. Мясницкая, д.20

Аннотация. Современные авиалайнеры, такие как Airbus A320, Boeing 787, перспективный отечественный самолёт МС-21, используют новую архитектуру построения комплекса бортового оборудования, получившую название Интегрированная модульная авионика (ИМА). В её основе лежит объединение приборов и бортовых вычислителей в единую сеть реального времени, что позволяет существенно снизить количество кабелей на борту и, тем самым, уменьшить взлётный вес лайнера. В ИМА разделяются функции сбора информации (датчики), воздействия (актуаторы) и логики оказания управляющих воздействий, которая реализуется специализированным прикладным ПО в бортовых вычислительных модулях. Международный стандарт ARINC 653 описывает требования к операционной системе реального времени, устанавливаемой на таких модулях, и программный интерфейс между прикладным авиационным ПО и операционной системой. Данный стандарт регламентирует временное и пространственное разделение прикладного ПО в соответствии с принципами ИМА. Большинство ОСПВ соответствующих стандарту ARINC 653 являются коммерческим ПО. В данной статье представляется JetOS – ОСПВ с открытым исходным кодом полностью соответствующую требованиям ARINC 653 части 1 версии 3. JetOS была основана на открытом проекте французских исследователей РОК. Некогда РОК была единственной ОСПВ с открытым исходным кодом, которая хоть скольконибудь соответствовала требованиям стандарта ARINC 653, однако была непригодна для практического использования: РОК не удовлетворяла ряду фундаментальных требований ARINC 653 и работала только в эмуляторе. При разработке JetOS код РОК был существенно переработан. В статье мы обсуждаем недостатки РОК и показываем, как нам удалось решить эти проблемы и какие изменения были внесены в архитектуру и реализацию РОК и отдельным подсистем. В частности, был полностью переписан планировщик реального времени, сетевой стек и управление памятью. Также в JetOS были добавлены новые возможности. Наиболее интересной является поддержка

системных разделов. Системный раздел – специальное прикладное ПО с расширенным набором возможностей, таких как прямой доступ к отдельным аппаратным средствам (сетевой карте, PCI контроллеру и т.п.). Наличие системных разделов позволяет вынести крупные подсистемы из ядра ОС и оставить в ядре минимальный набор задач, связанных с переключением контекстов, планировщиком и обменом сообщениями между компонентами ПО. В частности, в системный раздел вынесена подсистема, отвечающая за взаимодействие через сеть. Данное перемещение кода позволяет уменьшить размер ядра ОС, что теоретически уменьшает вероятность наличия ошибки в ядре и упрощает процесс верификации ядра.

Ключевые слова: ARINC 653; OCPB; операционная система реального времени; ИМА; интегрированная модульная авионика

DOI: 10.15514/ISPRAS-2016-28(2)-12

Для цитирования: Маллачиев К.М., Пакулин Н.В., Хорошилов А.В. Устройство и архитектура операционной системы реального времени. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 181-192 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-12

Список литературы

- [1]. Avionics application software standard interface part 0 overview of ARINC 653, ARINC specification 653P0-1, August 3, 2015
- [2]. Avionics application software standard interface part 1 – required services, ARINC specification 653P1-3, November 15, 2010
- [3]. G. Bloom, J. Sherrill. 2014. Scheduling and thread management with RTEMS. *SIGBED Rev.* 11, 1 (February 2014), 20-25. DOI=<http://dx.doi.org/10.1145/2597457.2597459>
- [4]. C. S. Stangaciu, M. V. Micea, V. I. Cretu; Hard real-time execution environment extension for FreeRTOS Conference: IEEE International Symposium on RObotic and SENSors Environments (ROSE 2014), At Timisoara DOI: 10.1109/ROSE.2014.6953035
- [5]. VxWorks 653 http://www.windriver.com/products/product-overviews/PO_VxWorks653_Platform_0210.pdf
- [6]. R. Kaiser, S. Wagner: Evolution of the PikeOS Microkernel, MIKES: 1st International Workshop on Microkernels for Embedded Systems. 2007
- [7]. LynxOS <http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/>
- [8]. M. Masmano, Y. Valiente, P. Balbastre, I. Ripoll, A. Crespo, J.J. Metge, 2010. LithOS: a ARINC-653 guest operating for XtratuM. In Proc. of the 12th Real-Time Linux Workshop, Nairobi (Kenya).
- [9]. M. Masmano, I. Ripoll, A. Crespo, and J.J. Metge. XtratuM: a Hypervisor for Safety Critical Embedded Systems. 11th Real-Time Linux Workshop. Dresden. Germany. http://www.xtratum.org/files/xm_rtlw09.pdf
- [10]. S. H. VanderLeest. ARINC 653 hypervisor. In Proc. Of IEEE/AIAA DASC, Oct. 2010.
- [11]. J. Delange, L. Lec, 2011. POK, an ARINC653-compliant operating system released under the BSD license. In 13th Real-Time Linux Workshop (Vol. 10). <http://julien.gunm.org/data/publications/articled111-osad111.pdf>
- [12]. S. Han and H.-W. Jin. 2012. Kernel-level ARINC 653 partitioning for Linux. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, 1632-1637. DOI=<http://dx.doi.org/10.1145/2245276.2232037>

Developing a Debugger for Real-Time Operating System

^{1,2} A.N. Emelenko <emelenko@ispras.ru>

^{1,3} K.A. Mallachiev <mallachiev@ispras.ru>

^{1,2,3} N.V. Pakulin <npak@ispras.ru>

¹ *Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

² *Moscow Institute of Physics and Technology (State University),*

⁹ *Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*

³ *Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. In this paper, we report on the work in progress on the debugger project for real-time operating system JetOS for civil airborne systems. It is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC 653 API specification. This operating system is being developed in the Institute for System Programming of the Russian Academy of Sciences and next step in developing this system is to create a tool to debug user-space applications on it. We also discuss the major requirements to such a debugger and show the difference between it and typical debugger, used by desktop developers. Moreover, we review a number of debuggers for various embedded systems and study their functionality. Finally, we present our solution that works both in emulator QEMU, which we use to emulate environment for our system, and on the target hardware. The presented debugger is based on GDB debugging framework but contains a number of extensions specific for debugging embedded applications. However, the implementation of the debugger is not complete yet and there is a number of features that can improve debugger usability, but it is already more functional than common GDB debugger for QEMU and, in contrast to other systems and their debuggers, where developers can use some functions to debug applications, but not all we need, our debugger meets the majority of our requirements and restrictions.

Keywords: debugger; GDB; real-time OS; remote debugger

DOI: 10.15514/ISPRAS-2016-28(2)-13

For citation: Emelenko A.N., Mallachiev K.A., Pakulin N.V. Developing a Debugger for Real-Time Operating System. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 193-204. DOI: 10.15514/ISPRAS-2016-28(2)-13

1. Introduction

Application debugger is an indispensable tool in developer's hands. But debugger in a real-time operating system is more than just plain debugger. In this paper we present an on-going project on debugger development for JetOS, a real-time operating system that is being developed in the Institute for System Programming of the Russian Academy of Sciences.

JetOS is a prototype operating system for civil airborne avionics. It is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC 653 API specification, the de-facto architecture for applied (functional) software.

The primary objectives of ARINC 653 are deterministic behavior and reliable execution of the functional software. To achieve this ARINC 653 imposes strict requirements on time and space partitioning. For instance, all memory allocations and execution schedules are pre-defined statically.

The unit of partitioning in ARINC 653 is called *partition*. Every partition has its own memory space and is executed in user mode. Partitions consist of one or more *processes*, operating concurrently, that share the same address space. Processes have data and stack areas and they resemble well-known concept of threads.

Embedded applications might be run in two different environments: in an emulator and on the target hardware. In our project we use QEMU system emulator. Although QEMU has its own debugger support, its functionality proved to be insufficient for debugging embedded applications. Therefore, we implemented a debugger not only for the target hardware, but for the emulator as well.

2. Main Targets for Debugger

Debugger for an embedded operating system has a number of specific features compared to typical debugger used by desktop developers.

Firstly, an embedded application runs under constrained conditions, such as limited on-board resources and lack of interactive facilities – no keyboard and screen. This makes it impossible to do debugging on the same device where application runs. Therefore, the debugger for embedded applications has to be remote: the developer interacts with workstation while the application runs on a target hardware.

Secondly, an embedded application typically consists of a number of interacting processes that needs to be debugged simultaneously. This means that the debugger must support dynamic and transparent switching between execution contexts during debugging session.

Thirdly, the debugged should support developers of system software, mostly device drivers and network stack. This requires switching between low-privilege code and highly privileged kernel code in the same debugging.

It is also important to mention that embedded developers widely use emulators in their work process. Typically most of development runs on top of emulators, therefore the debugger must support corresponding emulators as well.

The above mentioned features impose a number of restrictions on the design of the debugger that we considered:

- There are many different applications compiled in OS, which can have overlapping virtual address spaces.
- Typically target hardware board for embedded OS has only one port to communicate with the external world – a single serial port. Since it is used to stream console output of the running applications we need to share it between debugger traffic and applications' output.
- Multifunctional debugger is a complex program. It is very complicated to develop it from scratch, so we decided to base our debugger on an existing one.
- Support debugging both on hardware and with emulator because this support can expand developers' capabilities and improve their efficiency.
- Support capabilities of debugging for kernel and for user mode code, as well as capabilities of multiprocess mode.
- It must excel QEMU debugger, which we use to emulate environment for our system.
- Since the OS in question is real-time, it is important to minimize debugger's impact on system during debugging.

In order to meet these restrictions we selected the architecture of remote debugger with server and client parts, that communicate over a serial port using multiplexer.

We have chosen GDB (GNU debugger) for the client part of our debugger.

3. Related Works

We are not the first to consider the problem of remote debugging. For example, Pistachio microkernel uses kdebug for debugging [4]; besides, there is Fiasco debugger [1] and many different debuggers for VxWorks, for example, RTOS debugger [2].

Here we briefly consider some debuggers for embedded OSes and their primary features.

3.1 Fiasco OS

Fiasco OS is a 3rd-generation microkernel, based on L4 microkernel [1]. The kernel is simplistic, it misses most of the features available in “big” operating systems like Linux or Windows: program loading, device drivers and file system. All these features must be implemented in user-level programs on top of it (L4 Runtime Environment provides a basic set such functions).

Fiasco OS has built-in support for debugger that:

- supports threads;
- provides stack backtrace

- sets breakpoints;
- does single step;
- provides reading/writing in memory;
- provides reading hardware registers;
- support interprocess communication (IPC) monitoring.

The Fiasco Kernel Debugger (JDB) is a debugger for Fiasco. It has the following special functionality:

- It always freezes the system when it is working. It means that JDB disables all interrupts and halts clock. All processes and kernel don't work when JDB is invoked.
- JDB doesn't use any part of Fiasco kernel, because it is a stand-alone debugger with drivers for keyboard, display, etc.

In general, JDB is not a part of Fiasco μ -kernel, and Fiasco μ -kernel can run without connection with JDB or another debugger.

The debugger operates remotely over the serial line.

3.2 VxWorks

VxWorks [5] is a real-time operating system (RTOS) developed as proprietary software by Wind River of Alameda, California, US. It supports Intel (x86, including the new Intel Quark SoC and x86-64), MIPS, PowerPC, SH-4, and ARM architectures.

RTOS debugger for VxWorks implements the following set of features:

- Task Stack Coverage
- Task Related Breakpoints
- Task Context Display
- Debugging Modules (for example, Kernel module)
- Debugging Real-Time Processes
- Debugging Protection Domains
- Collecting statistics for function and tasks

RTOS debugger displays all system states, tasks, message queues, memory partitioning, modules and etc.

The key feature of the RTOS debugger is that is based on Lauterbach's TRACE32 debugger [3] that utilizes hardware interfaces like JTAG. It does not use serial port for communication with the target hardware but rather requires specific debug module.

3.3 L4Ka::Pistachio

L4Ka::Pistachio [4] is the latest L4 microkernel developed by the System Architecture Group at the University of Karlsruhe. It is the first available kernel

implementation of the L4 Version 4 kernel API, which provides support of both 32-bit and 64-bit architectures, multiprocessor and super-fast local IPC. The current release supports x86-x64 (AMD64/ EM64T, K9 / P4 and higher), x86-x32 (IA32, Pentium and higher), PowerPC 32bit (IBM 440, AMCC Ebony / Blue Gene P).

The debugger for Pistachio kernel can direct its I/O via the serial line or the keyboard/screen. It is a local debugger and does not support remote debugging mode.

This debugger is also a low-level device with very limited amount of functions.

Debugger for Pistachio can:

- Set breakpoints
- Single step
- Dump memory
- Read registers

When the processor meets special instruction (for example, *int3* instruction), it passes control to interrupt handler, which is the part of Pistachio kernel. In turn, interrupt handler checks instructions, which come next, and if they correspond to the special layout, it prints special message before passing control to interrupt handler. This feature is a simplistic implementation of a facility to trace execution.

4. Technical Description:

The primary goal of the debugger is Power PC platform, based on e500mc CPU core. The debugger is based on GDB, it uses the GDB architecture to establish link to the remote target.

The architecture includes three major components: front end, local client and remote server. The front end provides user interface, it runs on the same workstation as the client part. The latter translates the commands from the front end into GDB protocol and communicates with the remote server. The server implements the actual command embedded into protocol messages such as reading memory regions, setting breakpoints, processing debug interrupts, etc. Remote server is sometimes called “stub”.

Gdb-stub for i386 was taken as a basis for our debugger. This stub was totally redesigned for e500mc processor, which belongs to PowerPC architecture family. We left only the packet exchange and some of the packet processing mechanisms.

We use common gdb client, which was built for PowerPC with somewhat extended functional, to connect to our stub. This functional was developed using special user defines commands, so developers don't need to use special version of GDB. Instead, they can use any version, but it needs to use gdb commands file by utilizing special “source” command in GDB.

Accordingly, messaging mechanism between client and server doesn't change – the client sends a special-type packet to the server and waits for the server's answer. The server receives this message, checks control sum, which was sent in this packet, and if it matches the message contents, informs the client that the message was accepted

for processing. Then the server performs the action described in the packet and sends its own packet to the client.

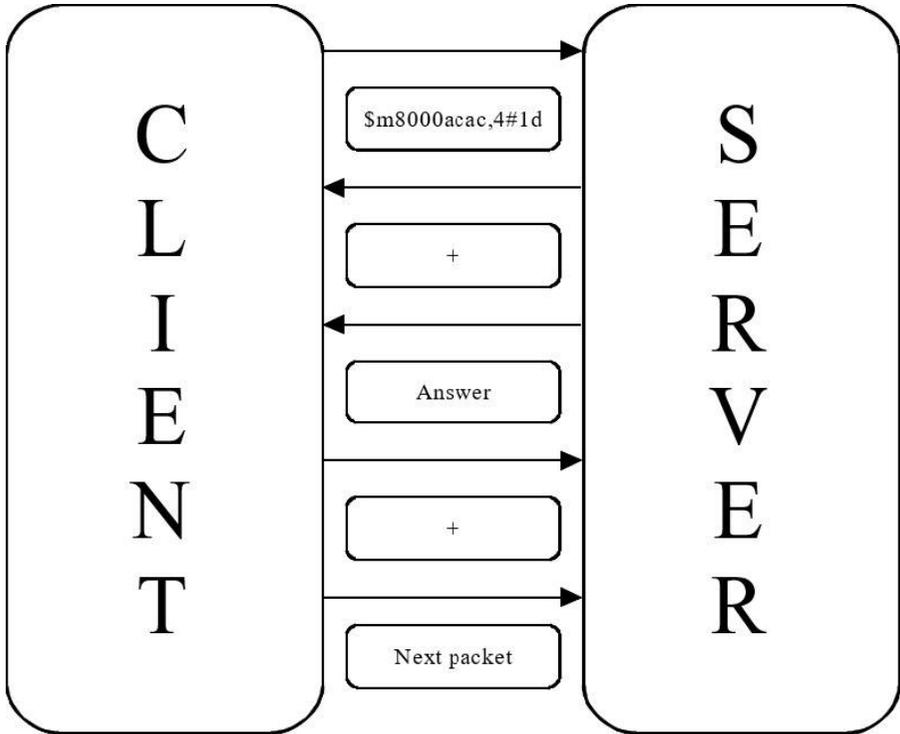


Fig. 1. GDB messaging mechanism

Let us consider an example on Fig. 1. Here client sends to server packet “\$m8000acac,4#1d”. This means that client wants to read 4 bytes of memory from virtual address 0x8000acac. In this packet “1d” is the control sum, that is, the sum of all bytes in message modulo 256. If the server fully receives this message, it sends “+”, and the client knows that the message was accepted. After that, server sends 4 bytes of memory from that address to the client in the same way, and message exchange continues. All these types of packets are described in GDB manual.

4.1 Implementation of the server side

In general, debugger's work consists of packet exchange between client and server. Client sends certain types of packets to perform the action, which the user needs. Our goal is to develop server part because we use client part from common GDB.

During the connection between server part of the debugger with client part our system stands in frozen state where no interrupts are available and the clock is halted. This opportunity allows us to work with partitions and debugger as if there is no debugger in the system.

We implemented functions in our debugger in the following way:

Breakpoints setting was implemented using special PowerPC instruction 'trap'. When the trap instruction occurs, server code in interrupt handler is called.

For Single step operation, we can use two different methods. The first one is when the system stops on the next instruction of the current partition. The second one is to stop the system stops on the next instruction wherever it is. The difference is how system calls are handled; the first method skips all kernel code and traverses application only. The second method allows entering kernel and stepping through system call implementation. Furthermore, it is sensitive to interrupts: if an interrupt occurs during the step, the debugger switched to the interrupt handler.

However, GDB structure requires interrupts to be disabled during single step. This requirement imposes restrictions on partition's work, so we gave up the second method. Because of the lack of debug registers in QEMU we need to disable interrupts and set trap instruction on the next instruction.

Watchpoints were implemented using special capabilities of hardware, such as Debug registers. Unfortunately, QEMU doesn't have such registers, so we need to use another way to set watchpoints in emulator. This method isn't implemented yet, but we are working in this direction. We also developed multiplexer to use one serial port for both GDB and another application. Multiplexer allows message exchange for debugger and for internal system service. The transformation of one serial port into two serial ports with the help of our multiplexer is not so difficult.

There are two parts of multiplexer, local and remote. Local part is a superstructure responsible for information input/output in the system. During the output it puts a special symbol before every printable symbol, determining to which of the two virtual serial ports the next symbol should be sent. Working with input symbols is very similar: two symbols are read, with the first of them specifying the application to which we want to send the second symbol. Remote part of multiplexer looks the same. This solution is not the fastest, but it provides smooth debugger's work via one serial port together with other applications. This connection between remote and local parts of multiplexer is shown on Fig. 2.

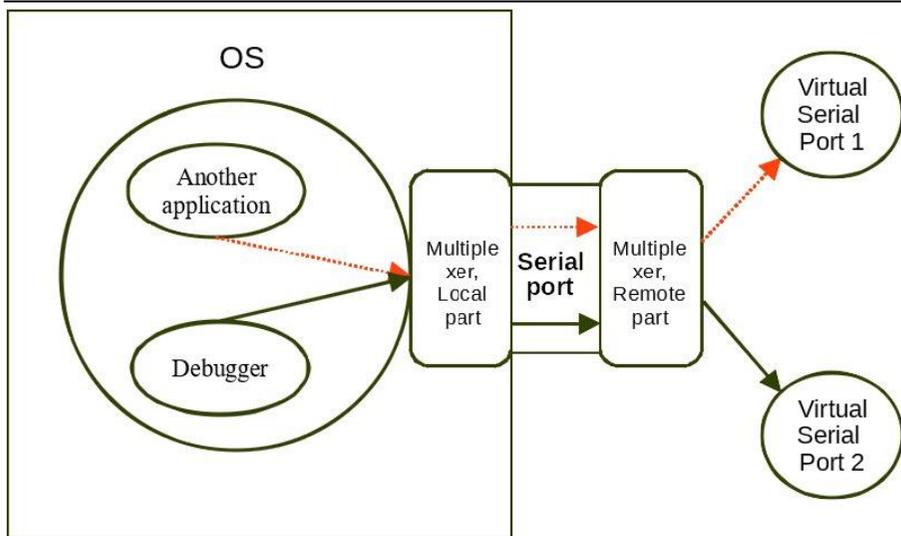


Fig. 2. Multiplexer work

5. Debugger's Capabilities

Our debugger supports all standard debugging features. Among them are:

5.1 Setting Breakpoints on Kernel and Partitions.

Setting breakpoints is the key feature of any debugger. Considering that client knows only virtual addresses, the server part of the debugger must correctly translate this address into physical address. Our debugger can do this, that's why users can debug partitions with overlapping virtual address spaces and debugger stops only on the partition that the user wants.

5.2 Single Step.

Stepping through code step by step is a convenient way of finding bugs. However, there can be a situation in real-time OS, when the next instruction in code is not the next executable instruction, for example, because of timer interrupt. That's why we disable interrupts during the single step.

5.3 Showing Information about Processes and Threads, Inspecting Memory, Instructions and Registers. Memory Reading and Writing.

Memory view must correctly translate virtual addresses into physical as with breakpoints. The capability to find out all information about threads in OS, their states, registers and memory is very important too.

Support of memory writes allows changing process state as user's discretion.

5.4 Setting Watchpoints.

Watchpoints are one of the most comfortable ways to control user's partition. They give the opportunity to follow changes in memory sectors and stop/pause while trying to read or record memory. This opportunity increases the number of ways to control partitions' states.

5.5 Stack Inspection.

Stack inspection makes tracing possible: for example, tracing the queue of called functions, which can help user to understand exactly what has happened in the system.

6. Future Work

Implementation of the debugger is not complete yet. There is a number of features that can improve debugger usability:

- Enhance debugging capabilities to the level of standard GDB functionality.
- Accelerate debugger interaction time with the system through multiplexer.
- Improve hardware support on bare metal.
- Increase user convenience in multiplexer. Enhance its functionality for working with more devices (now multiplexer supports only two devices). This solution allows us to work on bare metal with as many ports as we need, regardless of the actual amount of ports.
- Add watchpoints implementation to QEMU, which doesn't support debug registers. This is the reason why we can't use debug registers for setting watchpoints like we do on bare metal. In that case, we need to change code handling in QEMU to develop instruction for watchpoints creation.

7. Conclusion

In this paper, we have presented our project on implementation of the debugger for real-time operating system JetOS. In contrast to other systems and their debuggers, where developers can use some functions to debug applications, but not all we need, our debugger meets the majority of our requirements and restrictions. However, we will be able to update our debugger in near future and increase its functionality, but it is already more functional than common GDB debugger for QEMU.

References

- [1]. F. Mehnert, J. Glauber and J. Liedtke, "Fiasco Kernel Debugger Manual" Dresden University of Technology, Department of Computer Science, November 2008 (<https://os.inf.tu-dresden.de/fiasco/doc/jdb.pdf>)
- [2]. Lauterbach GmbH, "RTOS debugger for VxWorks", November 2015

- (http://www2.lauterbach.com/pdf/rtos_vxworks.pdf)
- [3]. Lauterbach GmbH, “RTOS-VxWorks”, 18 August 2014
(<http://www2.lauterbach.com/doc/rtosvxworks.pdf>)
- [4]. System Architecture Group University of Karlsruhe. “The L4Ka: Pistachio Microkernel”. May 1, 2003
(<http://www.l4ka.org/l4ka/pistachio-whitepaper.pdf>)
- [5]. Wind River Systems, Inc “VxWorks Product Overview”, March 2016
(<http://windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf>)
- [6]. Free Software Foundation, Inc. “Debugging with gdb: the GNU Source-Level Debugger”, The Tenth Edition
(<https://software.intel.com/sites/default/files/article/365160/gdb.pdf>)

Разработка отладчика для операционной системы реального времени

^{1, 2} А.Н. Емеленко <emelenko@ispras.ru>

^{1, 3} К.А. Маллачиев <mallachiev@ispras.ru>

^{1, 2, 3} Н.В. Пакулин <npak@ispras.ru>

¹ *Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

² *Московский физико-технический институт (государственный
университет),*

141701, Московская область, г. Долгопрудный, Институтский переулок, д.9.

³ *Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.*

Аннотация. В этой статье мы расскажем о проекте по разработке отладчика для операционной системы реального времени JetOS, созданной для гражданских авиационных систем. Она предназначена для работы в рамках архитектуры Интегрированной Модульной Авионики (ИМА) и реализует ARINC 653 спецификацию API. Эта операционная система разрабатывается в институте системного программирования РАН, и следующим шагом в ее разработке стало создание инструмента для отладки пользовательских приложений. Также в этой статье будут рассмотрены основные требования к такому отладчику и показана разница между ним и обычным отладчиком, используемым разработчиками настольных приложений. Более того, были рассмотрены другие встраиваемые операционные системы, такие как WxWorks, Fiasco OS, L4Ka:Pistachio и отладчики для них, а также был изучен их функционал. В заключение, мы представим наш отладчик, который может работать как в эмуляторе QEMU, используемом для эмуляции окружения для JetOS, так и на целевой машине. Представленный отладчик является удаленным и построен с использованием структуры GDB, но содержит ряд расширений, специфичных для отладки встроенных приложений. Однако реализация отладчика пока не завершена и существует целый ряд задач по улучшению удобства и возможностей отладчика, но на текущий момент он является уже более функциональным, чем обычный отладчик GDB для QEMU и, в

отличие от других рассмотренных систем и их отладчиков, где разработчики могут использовать некоторые функции для отладки приложений, но не все, что нам нужно, наш отладчик удовлетворяет большинству поставленных требований и ограничений, а также уже используется разработчиками приложений для JetOS.

Ключевые слова: отладчик; GDB; OCPB; удаленный отладчик; операционная система реального времени

DOI: 10.15514/ISPRAS-2016-28(2)-13

Для цитирования: Емеленко А.Н., Маллачиев К.А., Пакулин Н.В. Разработка отладчика для операционной системы реального времени. *Труды ИСП РАН*, том 28, вып. 2, 2016 г., стр. 193-204 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-13

Список литературы

- [7]. F. Mehnert, J. Glauber and J. Liedtke, “Fiasco Kernel Debugger Manual” Dresden University of Technology, Department of Computer Science, November 2008 (<https://os.inf.tu-dresden.de/fiasco/doc/jdb.pdf>)
- [8]. Lauterbach GmbH, “RTOS debugger for VxWorks”, November 2015 (http://www2.lauterbach.com/pdf/rtos_vxworks.pdf)
- [9]. Lauterbach GmbH, “RTOS-VxWorks”, 18 August 2014 (<http://www2.lauterbach.com/doc/rtosvxworks.pdf>)
- [10]. System Architecture Group University of Karlsruhe. “The L4Ka:: Pistachio Microkernel”. May 1, 2003 (<http://www.l4ka.org/l4ka/pistachio-whitepaper.pdf>)
- [11]. Wind River Systems, Inc “VxWorks Product Overview”, March 2016 (<http://windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf>)
- [12]. Free Software Foundation, Inc. “Debugging with gdb: the GNU Source-Level Debugger”, The Tenth Edition (<https://software.intel.com/sites/default/files/article/365160/gdb.pdf>)

Modelling the People Recognition Pipeline in Access Control Systems

¹ F. Gossen <frederik.gossen@lero.ie>

¹ T. Margaria <tiziana.margaria@lero.ie>

² T. Göke <thomas.goeke@systeambgmbh.com>

¹ Lero - The Irish Software Research Centre, University of Limerick,
Tierney Building, University of Limerick, V94 NYD3, Ireland.

² SysTeam GmbH,

Technologiepark, Martin-Schmeißer-Weg 14, 44227 Dortmund, Germany.

Abstract. We present three generations of prototypes for a contactless admission control system that recognizes people from visual features while they walk towards the sensor. The system is meant to require as little interaction as possible to improve the aspect of comfort for its users. Especially for people with impairments, such a system can make a major difference. For data acquisition, we use the Microsoft Kinect 2, a low-cost depth sensor, and its SDK. We extract comprehensible geometric features and apply aggregation methods over a sequence of consecutive frames to obtain a compact and characteristic representation for each individual approaching the sensor. All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic representation through a sequence of small data transformations. Every single transformation takes one or more of the previously computed representations as input and computes a new representation from them. In the example models presented in this paper, we are focusing on the generation of frontal view images of peoples' faces, which is part of the processing pipeline of our newest prototype. These frontal view images can be obtained from colour, infrared and depth data by rendering the scene from a changed viewport. This pipeline can be modelled considering the data flow between data transformations only. We show how the prototypes can be modelled using modelling frameworks and tools such as Cinco or the Cinco-Product Dime. The tools allow for modelling the data flow of the data processing pipeline in an intuitive way.

Keywords: Visual Modelling, Face Recognition, People Recognition, Computer Vision.

DOI: 10.15514/ISPRAS-2016-28(2)-14

For citation: Gossen F., Margaria T., Göke T. Modelling the People Recognition Pipeline in Access Control Systems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 205-220.
DOI: 10.15514/ISPRAS-2016-28(2)-14

1 Introduction

When using today's admission control systems some kind of interaction is required to check permission for every individual. Among the most widely used technologies are RFID chips on check cards. When attempting to pass the admission control system people have to swipe their check card through a reader to transfer a unique ID to the system. The system will then check whether or not access should be granted. Once a person is identified, it is easy to assign different levels of permission to different people. This might be useful to restrict access to certain areas in a building. Other methods for identification include PINs, passwords or keys. All these methods have one thing in common: They require the user to carry something, either physically or in mind. That means it is likely that someone who is allowed to pass the admission control system is not able to do so because he or she has forgotten his or her password or key. To overcome this issue iris recognition, fingerprint recognition and face recognition can be used [1]. All of these methods identify a person by something that cannot be forgotten such as the eye or the face.

In our work, we focus on identity recognition from colour and depth data using low-cost depth sensors such as the Microsoft Kinect 2 [2]. These sensors offer colour, infrared and depth images at high frame rates. Our goal is to recognize people with as little interaction as possible. The user should be able to walk towards the admission control system looking forwards as he or she walks. The system picks up his or her head and face and predicts the identity that is most likely to have caused the observation. Moreover, the system will have notion of certainty. In cases where the prediction is possibly wrong a fall back method for identification will be used. This can be a PIN, password or a check card but it is also possible to redirect the person to a staff member to be identified with human capabilities.

The proposed system primarily improves the aspect of comfort for everybody who uses the admission control system as they no longer have to carry check cards or keys or remember PINs or passwords. Such an admission control system can be used in many places. Starting from fitness studios, spa and swimming pools where members have to be recognized to give access, reaching to institutions where staff members have to be recognized. In these scenarios, the proposed system primarily improves the aspect of comfort. However, in some cases people are not able to use any of the alternative methods for identification. Especially in places like hospitals and retirement homes where many people suffer from impairments such a system can make a major difference. People with Parkinson's disease might be unable to swipe a check card with the tremor in way that allows the system to read the card. They will also have problems to enter a PIN or a password while a visual recognition system would not require them to interact in a particular way. Other examples include patients with Alzheimer's disease, people wearing a cast or doctors with sterilized hands.

We are currently working on the third version of a prototype for contactless admission control. Previous versions have suggested that geometric features can contribute to reliable recognition of individuals but are alone not sufficient for reliable access

control [3]. We are currently focusing on the generation of frontal view images from people's faces, which we will use to extract comprehensible and characteristic features for individuals. This will allow for recognizing them in the application of an admission control system. In what follows we will present the three versions of our prototype in Section 2. All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic feature representation. In order to show how this pipeline can be modelled, Section 3 introduces the reader to the meta modelling framework Cinco and to Cinco-Products that we use to model the pipeline. We present two alternative models of the data processing pipeline in Section 4 that are based on these Cinco-Products. Section 5 concludes this paper and points our directions of future work.

2 Prototypes

We are currently working on the third version of our prototype. The first two versions were based on the Microsoft Kinect [2] and its SDK while our new version will be based only on its low-level API that is similar to that of comparable sensors. The low-cost sensor provides capabilities to acquire colour, depth and infrared images at high frame rates. It comes with a powerful SDK that provides reliable algorithms to detect people's skeletons and faces.

In this section, we will describe the three versions of our prototype and we analyse their differences.

2.1 First Prototype

Starting with the first prototype, we decided to use the Microsoft Kinect 2 sensor. This sensor acquires colour, infrared and depth images at high frame rates. Moreover, the sensor comes with a Software Development Kit (SDK) that offers a high resolution face model and a skeleton model. The first system is based on the capabilities of the Kinect SDK. We use the high resolution face model to extract characteristic geometric features with clear interpretation. The features are extracted on frame by frame basis. The compact and comprehensible set of features is used to predict the person who is most likely to have caused the observation. In this first approach we use our own implementation of a Bayesian Classifier to perform this task on every frame separately.

As we aim to recognize human identities in a comprehensible way we need features that provide clear interpretation. As one of the first features that were used for facial recognition, geometric features fulfil this requirement [4] [5]. In contrast to early approaches we extract distances in space rather than in the image plane using the Kinect's face and its skeleton model.

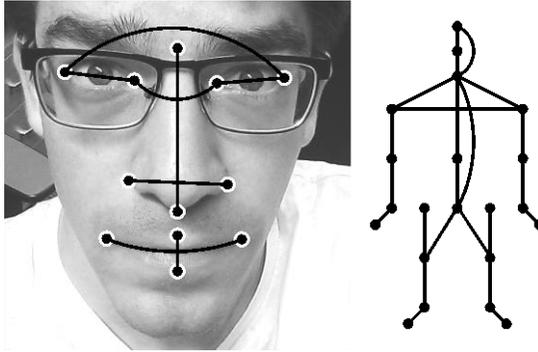


Fig. 1. Visualization of 8 distances that were extracted from the Kinect's face model (left) and 23 distances that were extracted from the Kinect's skeleton model (right). The distances are estimated in space rather than in the image plane to be invariant under the viewpoint.

The Kinect's face model provides a set of 1347 feature points many of which are interpolated. In order to obtain features with clear meaning we decided to focus on a subset of 12 feature points as shown in Figure 1. These feature points represent the eye corners, the moth corners, the lower and the upper lip and the left, right, lower and upper boundary of the nose. The face model provides these points' positions in both, the image plane and in space. As distances in the image plane are affected by the view point we use the Euclidean distances between points in space. We extract the following distances from the face model

- the inner and the outer eye distance
- the width and the height of the nose
- the width and the height of the mouth
- the width of the left and of the right eye.

Figure 1 visualizes all of the 8 facial features. Note, that some of these might vary a lot. For instance, the estimated height of the mouth will vary when people open or close their mouth. The features are therefore not invariant to facial expressions but can nevertheless characterize the face of an individual. Note, that we extract only a small subset of possible distances with clear interpretation that we expect to be characteristic for the human face. In this way we maintain comprehensibility of our representation. Moreover, too many features would lead to overfitting during the learning process as our data set was small at this stage of the development process.

The Bayesian Classifier that we used for classification assumes conditional independence of features. This assumption is obviously violated for the proposed features meaning that the Bayesian Classifier is no longer guaranteed to be optimal. However, Bayesian Classifier are often successful although their assumptions might be violated. In order to allow for a good representation of the conditional probability distribution, we introduce another assumption that the conditional probability for each features is normally distributed. Hence, the learned model for a person can be

represented by mean and variance per feature. Note, that a normal distribution is a reasonable assumption for geometric features from the Kinect face model as shown in [3]. However, this introduces yet another assumption that might be violated to some degree. The Bayesian Classifier is therefore no longer guaranteed to be optimal. However, it shows reasonable performance in many classification problems as well as in our preliminary evaluation in [3].

With recognition rates of up to 80% on a preliminary data set with only 5 people, this first prototype was far from being sufficiently accurate for reliable access control. However, it proved, that geometric features from the Kinect's models can be used to recognize people. This first system was not exploiting the redundancy of the records among consecutive frames as its prediction was on a frame by frame basis. Moreover, the feature set was extremely small.

2.2 Second Prototype

To overcome the weaknesses of our first prototype, we introduce more features and feature aggregation in the second version of our prototype and tested different classifiers to analyse the quality of the feature set. One feature that is expected to be particularly predictive for a person's identity is his or her height. The height varies a lot between different individuals and can be estimated using the Kinect's skeleton model. The model provides the position for 25 joints in both, the image plane and in space. We extract Euclidean distances in the same way as we extract them from the face model. In order to capture meaningful features from the model we consider distances from a selection of adjacent skeleton joints. In addition, we extract features between joints that are not adjacent if we expected the feature to be characteristic for a person. In particular, we wanted to represent a person's height and his or her shoulder width. Figure 1 shows all of the proposed 23 features that are considered from the skeleton model.

We introduced feature aggregation to this version of our prototype. Frames are still processed separately to extract the set of features from them leading to 31 values per frame. When a person approaches the Kinect sensor, up to 15 frames were considered during our experiments. As the Kinect's models have high computational demands, the low frame rate did not allow for more records in most situations. All of the 15 records aim to measure the same set of distances. In order to benefit from this redundancy, we aggregated the sequence into a single value per feature using one of 6 aggregation methods. As the most prominent aggregation method for real values, we used the mean in our experiments. In order to be more robust against outliers, we also analysed the median and four variants of truncated mean which can be seen as intermediate aggregation methods between mean and median.

The larger the distance to an object, the more noise can be observed in depth images. This makes approximation of the facial shape more difficult leading to lower quality of the Kinect's models. We therefore expect the measures for the proposed geometric features to be particularly noisy for records that were taken far away from the sensor.

As these measures might have a negative impact on the quality of the aggregated features, we consider only a subset of the closest N records during our experiments.

In order to select a good classifier for our system, we use Rapid Miner [6] to evaluate the quality of our feature set. We tested two classifiers in our experiments, k-Nearest Neighbour (k-NN) [5] and Linear Discrimination Analysis (LDA) and report a recognition rate of up to 88% for k-NN and up to 89% for LDA on a data set with 37 individuals.

These recognition accuracies are a significant improvement over our previous system while the aggregated features are still as comprehensible as the previously used raw features. Most importantly, the aggregation of feature values was shown to improve the recognition accuracy significantly. In order to be used in an access control system, we aim to further increase our system's performance.

2.3 Third Prototype

As based on the Kinect's face and skeleton model, the first two versions of our prototype are not easily adoptable to the use of other sensor devices. Moreover, the Kinect's face and skeleton model have high demands with regard to hardware. This might be a problem once the system is in use on site where such machines are not available or increase the costs dramatically. Hence, we wanted to become independent of the Kinect SDK's advanced capabilities while we still use the sensor and its low-level API. The subset of the provided functionality that we use in the third version of the system is available for many other low-cost sensors. We acquire colour, infrared and depth frames as well as a mapping between these data sources. This functionality is also offered in OpenNI [7] for a variety of different sensors.

Although the recognition accuracies using aggregated geometric features are a significant improvement over the first version of our prototype they are not yet sufficient for reliable access control. However, they have shown that geometric features can contribute to reliable recognition in a comprehensible manner. To explore additional features and to improve the system's accuracy, we currently focus on colour, infrared and depth data directly which were not used in the previous systems. We aim to extract comprehensible features from these images as an intermediate representation. These features can again be geometric features as the distances between certain feature points but they are not restricted in this way and more importantly, no longer based on the Kinect's models.

As a basis for feature extraction we decided to generate frontal view images of detected faces. When a person approaches the sensor, his or her head and face are detected. We also estimate the person's head pose meaning that the exact position and orientation of a person's face is known. As the depth frame provides spatial information, this allows to render the scene from a normalized position in front of a person's face. In this way we obtain depth images of detected faces that are aligned in a predefined position.

Given the mapping from depth frame to the colour frame respectively the infrared frame, it is further possible to use these as a texture. Hence, we are able to render frontal views of a detected face using either the colour frame or the infrared frame as a texture. Three different kinds of frontal views of a person's face can be computed in this way. As the system is currently under development, we want to focus on this part of the data processing pipeline in what follows. These first steps as a part of the data processing pipeline are sufficient to point out the idea of how such a system can be modelled using existing modelling frameworks.

3 Modelling Frameworks and Tools

All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic feature representation through a sequence of small data transformations. Every single transformation takes one or more of the previously computed representations as its input and computes a new representation from them. As only the final outcome is of any interest while intermediate representations are solely used for the computation of the final outcome, all of our prototypes can be modelled intuitively by focusing on the data flow only. In fact, we will show that the control flow can be derived from the data flow in our example.

In the example presented in this paper, we are focusing on the newest version of our prototype. As the final recognition is not implemented to date, we will focus on the generation of frontal view images of peoples' faces which will be part of the final data processing pipeline. These frontal view images can be obtained from colour, infrared and depth data by rendering the acquired data from a changed viewport. In order to do so the face position and its orientation have to be estimated precisely. Our newest prototype approaches this task in four steps based on a single depth frame.

We show two example models of our prototype using two different modelling tools that were generated using the modelling framework Cinco. In what follows, we will first introduce the reader to the modelling framework Cinco and to Dime, the most complex Cinco-Product to date. We will further show a small custom Cinco-Product that models the data flow only and is tailored to the needs of our prototypes' models.

3.1 Cinco

Cinco is a meta modelling framework for graphical Domain Specific Languages that is developed at TU Dortmund University since 2014 [8] [9] [10]. It is based on the popular Eclipse Open-Source IDE and allows for the generation of Cinco-Products that are themselves based on the Eclipse IDE. Graphical Domain Specific Languages in Cinco are based on the concept of directed graphs meaning that a predefined set of custom nodes and edges is defined for a particular Cinco-Product. The meta modelling framework allows to define the appearance for each kind of node and edge and allows to constrain their connectivity. In this way it is possible to allow certain

edges to connect only very particular kinds of nodes, but many other ways of constraining the graphical language are possible.

To enable rich features in Cinco-Products, the framework implements the concepts of hooks which allows to programmatically adjust the graph in case of a particular event. Such an event can be that a node was moved on the canvas or that it was removed from it. In particular, this allows to implement custom spatial arrangement of multiple nodes relative to one another but many other applications are possible.

As a meta modelling framework Cinco is used to generate modelling tools that are referred to as Cinco-Products. Due to the only assumption that a graphical Domain Specific Language is a directed graph, Cinco is very flexible and allows to generate modelling tools for a wide range of applications. Cinco itself does not associate any semantics with the graphical language but allows for the generation of an API that can be used to generate code from the graph models or to interpret them otherwise. Particularly interesting for our example models, edges can be used to represent both, control flow and data flow.

3.2 Dime

As the most complex Cinco-Product to date, Dime is the prime example of the Cinco's flexibility. As a Cinco-Product, Dime defines a set of nodes and edges, their appearance and also constraints the way they can be connected. While nodes represent situations during model's execution, edges are used to model both, control flow and data flow.

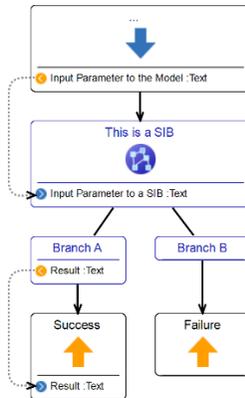


Fig. 2. Minimal example of a Dime model.

The most important nodes are the so called Service Independent Building Blocks (SIB) which represent executable code in the model. Every SIB has a list of input ports similar to function or method parameters in other programming languages. The functionality represented by the SIB relies only on the data provided by means of these input ports. The execution of a SIB can result in different cases which are

modelled using the concept of branches. Every SIB must have one or more branches as its successors, each representing one case. Depending on the outcome of the execution of the SIB, one branch is chosen that determines the SIB that is to be executed next. In this way branches are used to model the control flow of the system. In addition, the selection of a particular branch, any other outcome of a SIB will be represented as variable. In Dime the set of computed variables can be defined for every branch separately. This is often appropriate as there will be no computation result in some error cases or different results can be computed in different cases. Dime represents the outcome in terms of data by output ports that are associated with the branch nodes. Figure 2 shows a small example of a Dime model with one SIB that has two branches only one of which has an output port.

As Dime allows the user to model control flow and data flow, it has to provide at least two different kinds of edges. In fact, there are many more kinds of edges but for the sake of simplicity, we want to focus on data flow and control flow. The control flow starts at the start SIB which is represented by a blue arrow. To make the entry point unique, there can only exist one start SIB in every model. Together with the end SIBs they are the only SIBs that have no branches. During execution the start SIB will do nothing as it is solely used to represent the start of the control flow and potential input ports. The end SIBs are used to represent different cases as an outcome of the model's execution and their associated output ports. As such they serve a similar purpose as branches on the level of the entire model. In fact, this is how Dime allows users to model in a hierarchical manner, meaning that the whole model can be used as a SIB in other models. In order to define the control flow from the start SIB to one of the possible end SIBs, the user has to define the control flow. This is done by connecting branches as the outcome of SIBs to exactly one other SIB. Depending on the outcome of the execution of a SIB, this allows to define the successor separately for every case. The control flow must be defined for every possible branch to make the model valid. When the control flow reaches a SIB, all of its input ports must be available. The required data can be provided by the initial input parameters on the level of the entire model or it can be provided as the outcome of a previously executed SIB. In any case, the variable to be used as an input must be defined using data flow edges. These edges are dashed and connect exactly one output port of a branch to one input port of a SIB. Moreover, the start SIB's ports can be used as output ports and the end SIBs' ports can be used as input ports. It is the user's responsibility to define the data flow and the control flow in such a way that required input data is available when a SIB is reached. Hence, the data flow imposes constraints on the control flow and vice versa, which can be exploited in the example that we present in this paper.

As Dime is a Cinco-Product and defines a set of nodes and edges, with clear interpretation, Dime is no longer as flexible as the use of Cinco for a tailored Cinco-Product. However, by modelling both, control flow and data flow its graphical models can express similar things as many programming languages in an intuitive fashion. Dime is still flexible in the sense that SIBs can have arbitrary functionality.

3.3 Custom Cinco-Product

Although Dime is suitable for many applications as it allows to model both, data flow and control flow, there are applications that can be modelled more intuitively in other ways. In our example, we are only interested in the final outcome of the computation, respectively the final data representation. As every data transformation depends on one or more data representations, an order of all data transformations is implicitly defined by the data flow. Hence, this example allows for modelling the data flow only while the control flow can be derived automatically.

For our second example we have therefore created our own Cinco-Product that allows for modelling the data flow only. There are three kinds of nodes, namely input representations (blue), output representations (green) and intermediate representations (white) and only one kind of edges to model data flow. All of these nodes represent a form of data that will be computed during the execution of the model if necessary. Note, that intermediate representations also represent a data transformation as they are computed from one or more other representations.

4 Example Models of our Prototype

All of our prototypes were developed in a way that allows to easily model their data processing pipeline using either Dime or a custom Cinco-Product. The recognition algorithm can be clearly separated into a sequence of data transformations as will be shown by means of the following two example models. We present example models for both of the modelling tools, Dime and our own Cinco-Product.

4.1 Dime Model Example

As Dime allows for modelling control flow and data flow, the data processing pipeline can be easily modelled in Dime. Each of the data transformations as part of the data processing pipeline can be represented as a SIB. The required input representations are inputs to the SIBs and will therefore be connected to the SIBs' input ports. In our example we want to focus on the data flow and we want to show that the control flow can be automatically derived from it. For the sake of simplicity this example is therefore limited to a single branch per SIB. When the model is used to generate code in future versions of our system, more than one branch will be necessary to handle exceptions in any of the data transformation steps. For instance, there might be no person visible in an acquired frame and hence no meaningful head pose can be computed.

Ignoring these exceptions in the current version of the model, every SIB has exactly one branch which is the success branch. The success branch is necessary to provide outputs of the computation, namely a new data representation. In the example new frames are acquired from each of the three sources as a very first step. The first three SIBs fulfil this purpose and provide colour, infrared and depth frames as output ports of their success branches. While colour and infrared frames are solely used as a texture for the generation of the frontal view images, the depth frame plays an

important role. As it provides information about the facial shape it can be used to approximate the head pose. In the newest version of our prototype, this problem is approached as a sequence of four refinements as shown in the second row of the model in Figure 3. First, the head position is roughly approximated from the raw depth image as given to the first SIB's input port. The success branch provides the head position approximation which serves as an additional input for the more precise head position computation. Both, the raw depth frame and the head position approximation are connected to the head position computation SIB's input ports. In a similar fashion, the head pose is computed in two consecutive steps. Finally, a precise estimate of the head pose is available which allows for rendering of frontal views.

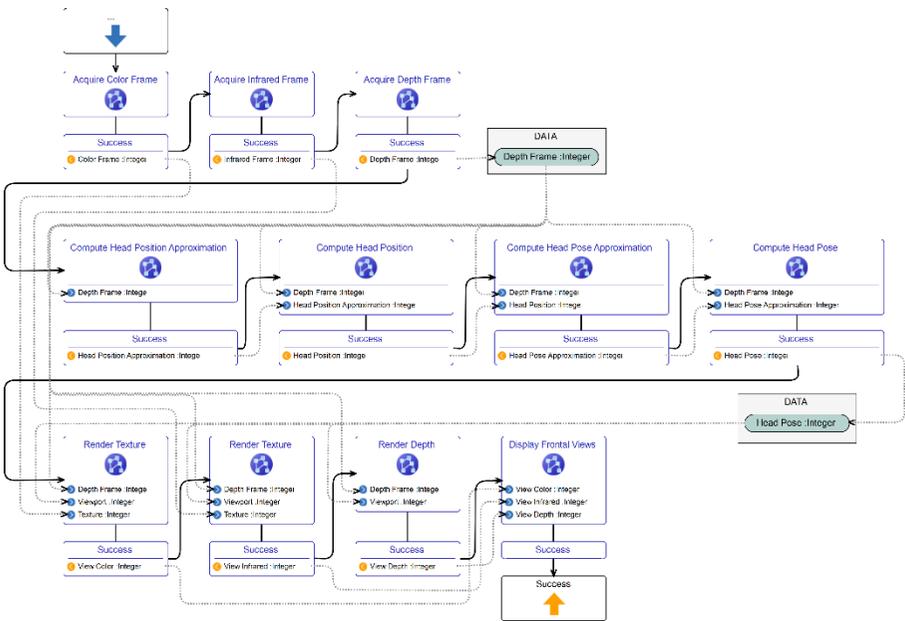


Fig.3. Dime model for the generation of frontal view images from colour, infrared and depth data.

Given the head pose, the first three SIBs in the third row of our model in Figure 3 render the frontal view images. As input parameters, all of them expect the raw depth image as acquired from the sensor which provides spatial information and also the precise head pose estimate which defines the viewport from which the scene is to be rendered. In addition, colour and infrared images are used as a texture leading to three different frontal view images of the detected face. For demonstration purposes the last SIB takes all of these images as inputs and displays them.

Note, that some data representations such as the depth image and the head pose are used more than once. In Dime this requires the use of a data context that holds

variables and allows for them to be used multiple times. In general, this is necessary as SIBs can change the value of their input parameters. However, in our example all of the input variables are only read which allows for using them multiple times in an arbitrary order.

Our goal is to define reusable components from the data processing pipeline of our final version of the admission control system and to provide them as SIBs in Dime. Not only would this allow to modify the system in a very intuitive way and would allow non-programmers to adjust the system at any time, but also would this allow for building similar systems from the existing components in a very easy way. Especially people with little to no programming skills would be enabled to create advanced systems that detect people, their head poses and many other things depending on the capabilities of the palate of provided SIBs.

4.2 Custom Cinco-Product Model Example

As only the displayed image the displayed image in this example as the final outcome of our model's execution is of any interest, the order of execution is irrelevant as long as required input representations are available for every data transformation in the data processing pipeline. In order to exploit this property, we use our custom Cinco-Product to model the data flow of the pipeline only. Every node represents data while intermediate data representations (white) implicitly represent data transformations that define how the new data representation can be obtained from others. In the example, raw sensor data is given as input representations (blue). In particular, these are colour, infrared and depth frames that were acquired using the Kinect sensor. Per execution of the pipeline one frame from each source is available and all of them can be used to obtain the final data representation. As the newest version of our prototype does not implement the final recognition of an individual yet, we limit the example to the generation of a collage of frontal view images from all three sources. The frontal views will later be used to extract facial features for the recognition of individuals. In order to render frontal view images of faces, the head pose must be known which defines the viewport from which the scene has to be rendered.

As the prototype is the same one that was used for the Dime model example, the computation of the head pose is again approached in four steps. First, the head position is approximated in the raw depth frame. The data representation *Head Position Approximation* implicitly represents the data transformation from a raw depth frame to an approximation of a person's head position. The new data represents only the approximation of the head position and no longer the depth frame. It is therefore a significantly smaller data representation than the *Depth Frame*, which was provided as an input representation. In a second step, the *Head Position* is computed more precisely from the raw *Depth Frame* and from the *Head Position Approximation*. The new data representation therefore depends on two others which have to be available before the *Head Position* can be computed. Hence, the data flow as defined in the model in Figure 4 imposes constraints on the order of data transformations.

Note, that the model in Figure 4 does not define the control flow but only the data flow. As the data flow imposes constraints on the order in which data representations must be available, a possible control flow can be deduced automatically from the topological order of the graph. More precisely every input representation must be available before a new data transformation can be applied. In our example, this means that *Head Position Approximation*, *Head Position*, *Head Pose Approximation* and *Head Pose* must be computed in exactly this order before any of the other data representations can be derived. For the generation of the separate frontal view images, the order can be arbitrary as they do not depend on one another but only on input representations and the *Head Pose*. Finally, the *Merged Image* must be computed at the very end. This is also defined as the final output representation (green) of our model. Any case in which the order of computation is irrelevant allows for parallelism. In our example, the generation of the separate frontal view images can in fact be performed in parallel once their input representations are available.

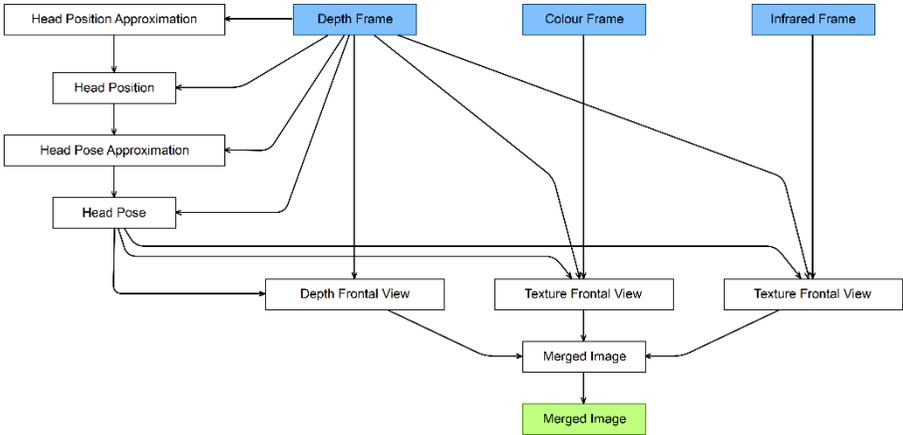


Fig. 4. Cinco-Product model for the generation of frontal view images from colour, infrared and depth data.

As an alternative to SIBs in Dime our custom Cinco-Product has strong focus on data flow. While this simplifies modelling of a data processing pipeline, there is no easy way of modelling side effects, defining the order of execution etc. This Cinco-Product is nevertheless useful for data oriented applications such as processing pipelines in computer vision systems similar to the one in our example.

5 Conclusion

In this paper, we presented three generations of prototypes for a contactless admission control system with high potential to be modelled with available modelling frameworks and tools.

We presented the three versions of our prototype and their commonalities and differences. In particular, we focused on the data processing pipeline which all prototypes implement in a similar fashion. This part of the system can be modelled intuitively with modelling tools such as Dime or our custom Cinco-Product.

In order to show our prototypes' potential to be modelled, we introduced the reader to Cinco, Dime and our custom Cinco-Product. Focusing on the first part the newest processing pipeline, we show examples of models in both tools. We discussed the possibility to derive the control flow automatically from a specification of the data flow in the data processing pipeline.

We continue to develop the most recent version of our prototype in a way that maintains its high potential to modelled visually. This will enable us to define reusable components from the data processing pipeline of our final admission control system and to provide them as SIBs in Dime. Moreover, we aim to model the system using a similar Cinco-Product to the one that we presented in this paper. Not only would this allow to modify the system in a very intuitive way and would allow non-programmers to adjust the system at a later stage, but also would this allow for building similar systems from the existing components in a very easy way. Especially people with little to no programming skills would be enabled to create advanced systems that detect people, their head poses and many other things. Once a set of powerful SIBs, respectively data transformations is developed for computer vision related applications, it can be extended continuously leading to a rich palate of SIBs. Depending on the extend of this palate, this would allow for modelling a wide range of computer vision related applications. In the long term, such a palate could also be extended to an even broader range of systems that implement any kind of a data processing pipeline.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

References

- [1]. G. S. Gagandeep Kaur and V. Kumar, "A review on biometric recognition," *International Journal of Bio-Science and Bio-Technology*, vol. 6, no. 4, pp. 69–76, 2014.
- [2]. Z. Zhang, "Microsoft kinect sensor and its effect," *IEEE MultiMedia*, vol. 19, no. 2, pp. 4–10, 2012.
- [3]. F. Gossen, "Bayesian recognition of human identities from continuous visual features for safe and secure access in healthcare environments," in *Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015 10th International Conference on, 2015.
- [4]. I. Marqués and M. Graña, *Computational Intelligence in Security for Information Systems 2010: Proceedings of the 3rd International Conference on Computational Intelligence in*

- Security for Information Systems (CISIS'10), ch. Face Processing for Security: A Short Review, pp. 89–96. 2010.
- [5]. T. Cover and P. Hart, “Nearest neighbor pattern classification,” *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.
- [6]. “Rapid miner.” <https://rapidminer.com/>. Accessed: 2016-02-02.
- [7]. “Openni 2 sdk.” <http://structure.io/openni>. Accessed: 2016-04-01.
- [8]. S. Naujokat, L.-M. Traonouez, M. Isberner, B. Steffen, and A. Legay, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, ch. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems, pp. 481–498. Springer Berlin Heidelberg, 2014.
- [9]. S. Naujokat, M. Lybecait, B. Steffen, D. Kopetzki, and T. Margaria, “Full generation of domain-specific graphical modeling tools: a meta modeling approach.” under submission, 2015.
- [10]. “Cinco scce meta tooling framework.” <http://cinco.scce.info/>. Accessed: 2016-04-01.

Моделирование конвейера распознавания людей в системах контроля доступа

¹ Ф. Гёссен <frederik.gossen@lero.ie>

¹ Т. Маргариа <tiziana.margaria@lero.ie>

² Т. Гёке <thomas.goeke@system-gmbh.com>

¹ Lero - The Irish Software Research Centre, Лимерикский университет, Tierney Building, Лимерикский университет, V94 NYD3, Ирландия.

² SysTeam GmbH,

Технопарк, Martin-Schmeißer-Weg 14, 44227 Дортмунд, Германия.

Аннотация. В работе представлено три поколения прототипов бесконтактной системы пропуска, распознающей людей по их визуальным особенностям при подходе к сенсору. Назначением системы является увеличение удобства пользователей за счет минимизации взаимодействия. Такая система может быть особенно полезна людям с нарушениями тех или иных функций. Для получения и обработки данных в системе используется Microsoft Kinect 2, недорогой сенсор глубины, и связанные с ним инструменты разработки. Распознавание приближающегося к сенсору индивида основано на построении компактного характеристического представления; для этого вычисляется множество геометрических особенностей индивида и применяются методы агрегации для последовательности кадров. Каждый из трех прототипов реализуют некоторый конвейер обработки данных; конвейер преобразует данные, полученные от сенсора, в компактное характеристическое представление путем последовательного применения простых трансформаций. Каждая отдельная трансформация получает на вход одно или несколько представлений, полученных на предшествующих стадиях, и строит по ним новое представление. Примеры моделей, представленные в этой статье, фокусируются на генерации фронтальных изображений лиц людей — это часть конвейера обработки данных последнего прототипа. Фронтальные изображения могут быть получены по данным о цвете, инфракрасном излучении и глубине путем

рендеринга сцены относительно меняющейся области просмотра. Такой конвейер может быть представлен исключительно потоками данных между трансформациями. В статье показывается, как моделировать прототипы с помощью таких сред и инструментов, как Cinco и Cinco-Product Dime. Эти средства позволяют интуитивным образом моделировать потоки данных в конвейерах.

Ключевые слова: визуальное моделирование, распознавание лиц, распознавание людей, машинное зрение.

DOI: 10.15514/ISPRAS-2016-28(2)-14

Для цитирования: Гёссен Ф., Маргариа Т., Гёке Т. Моделирование конвейера распознавания людей в системах контроля доступа. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 205-220 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-14

Список литературы

- [1]. G. S. Gagandeep Kaur and V. Kumar, "A review on biometric recognition," *International Journal of Bio-Science and Bio-Technology*, vol. 6, no. 4, pp. 69–76, 2014.
- [2]. Z. Zhang, "Microsoft kinect sensor and its effect," *IEEE MultiMedia*, vol. 19, no. 2, pp. 4–10, 2012.
- [3]. F. Gossen, "Bayesian recognition of human identities from continuous visual features for safe and secure access in healthcare environments," in *Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015 10th International Conference on, 2015.
- [4]. I. Marqués and M. Graña, *Computational Intelligence in Security for Information Systems 2010: Proceedings of the 3rd International Conference on Computational Intelligence in Security for Information Systems (CISIS'10)*, ch. Face Processing for Security: A Short Review, pp. 89–96. 2010.
- [5]. T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.
- [6]. "Rapid miner." <https://rapidminer.com/>. Accessed: 2016-02-02.
- [7]. "Openni 2 sdk." <http://structure.io/openni>. Accessed: 2016-04-01.
- [8]. S. Naujokat, L.-M. Traonouez, M. Isberner, B. Steffen, and A. Legay, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, ch. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems, pp. 481–498. Springer Berlin Heidelberg, 2014.
- [9]. S. Naujokat, M. Lybecait, B. Steffen, D. Kopetzki, and T. Margaria, "Full generation of domain-specific graphical modeling tools: a meta modeling approach." under submission, 2015.
- [10]. "Cinco scece meta tooling framework." <http://cinco.scece.info/>. Accessed: 2016-04-01.

Parallel processing and visualization for results of molecular simulation problems

D. V. Puzyrkov <dpuzyrkov@gmail.com>

V. O. Podryga <pvictoria@list.ru>

S. V. Polyakov <polyakov@imamod.ru>

*Keldysh Institute of Applied Mathematics (Russian Academy of Sciences)
Miusskaya sq., 4, Moscow, 125047, Russia*

Abstract. In this paper authors presents “mmdlab” library for the interpreted programming language Python. This library allows to carry out reading, processing and visualization of the results of numerical calculations in the tasks of molecular simulation. Considering the large volume of data obtained from such simulations, there is a need in parallel realization of algorithms for processing those volumes. Parallel processing should be performed on multicore systems, such as common scientific workstation, and on super-computer systems and clusters, where the MD simulations were held. During the development process we have study the effectiveness of the Python language for such tasks, and we have examined the tools for it’s acceleration. As well, we studied multiprocessing capabilities and tools for cluster computation using this language. Also we have investigated the problems of receiving and processing the data, located on multiple computational nodes. This was prompted by the need to process the data, produced by parallel algorithm, that was executed on multiple computational nodes, and saves its output on each of them. As a tool for scientific visualization was chosen an open-source “Mayavi2” package. The developed ”mmdlab” library was used in the analysis of the results of MD simulation of the gas and metal plate interaction. As a result, we managed to observe the effect of adsorption in details, which is important for many practical applications.

Keywords: parallel processing; visualization; molecular dynamics; Python; Mayavi2

DOI: 10.15514/ISPRAS-2016-28(2)-15

Для цитирования: Puzyrkov D.V., Podryga V.O., Polyakov S.V. Parallel processing and visualization for results of molecular simulation problems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 221-242. DOI: 10.15514/ISPRAS-2016-28(2)-15

1. Introduction

Advances in computer technology and the rapid growth of computational capabilities significantly increased the possibilities of computational experiment (CE). In particular, nowadays it is already possible to study the properties and processes in complex systems on molecular and atomic levels, for example, using molecular dynamics (MD) approach. Mathematical models, which describe such processes, may

consider huge amounts of particles: up to billions of them, and even more. In addition, each particle can be described by dozens of parameters and the volume of output data in such CE can be estimated in terabytes.

Processing of such volumes of data in serial mode can potentially take years, and optimization of computing code does not bring a significant acceleration of the computations. Therefore, currently the most widely used approach to accelerate the large-scale computing is its paralleling, which means that a great number of compute nodes would process a large amount of data each handling apart of it.

As a result of paralleling, each node receives only a small part of the data set which is easy to manipulate with.

This technique significantly reduces the time required to complete data processing, but leads to several problems concerning the data storage. Most often, after performing calculations compute nodes exchange the results of computations, and master process assembles them in RAM or in a storage device as one large array or a file. However, in the large-scale computations the size of the result array (file) can significantly exceed the resources of the master node. In this case, each compute node stores the results in isolation. The last described method of storage has several advantages. The first one is the lack of need to sequentially read all the results for further processing (for example, for visualization purpose) because each computational node only reads its part of the data. The second advantage is that each individual data file is typically not very large (compared to the full data set), and thus it takes less processing time. Such data can be reached in various ways, for example using a distributed file system, on-the-node-process reading, or using the applications allowing to send data over the network, such as the SFTP.

The scientific programs that store data in the form described above, are considered in this article. The results of the simulation based on the algorithm, described in the article [1] were used as a data set for studying parallelization capabilities of the developed "mmdlab" library.

One of the ways of CE data representation is a two- and three-dimensional visualization.

In order to assemble a complete state of the simulation results, it is required to read and process the data from each compute node, which in itself is a resource-intensive task. In most cases, the calculated data formats and storage methods differ depending on the calculation program. Therefore, such programs usually have their own visualizer, and calculate all the necessary visualization data in the process of computation, collecting them on the master node. In this case, the visualization is provided by the means of such programs (LAMMPS, and others). Another way is to save data in the well-known standardized containers (HDF5, VTK, and other), which are supported by the majority of software for scientific visualization. The problems of such methods of storage and rendering are the limited possibilities of the used visualization software in regards to visualization and post-processing, and in the case of well-known standards of data storage there occurs the problem of loading large files.

This paper presents an attempt to create a flexible tool that allows importing, processing and visualization of data from different sources, regardless of its structure: whether the data is in known formats or distributed calculation results in a custom format.

The results obtained using the computer program described in the article [1] were considered as a test case.

In view of the parallel algorithms and storage features, this data can be a one big file that describes the general state of the simulated system, as well as a distributed data, processed by every computational process separately. The results obtained from the simulation are the information about the interactions of the gas molecules with the metal atoms near the surface. This process is characteristic for many technological microsystems used in nanotechnology.

2. Problem Statement

The problem of collecting and processing the distributed data obtained as a result of some calculation program has several key features.

Firstly, it is the specifics of the problem domain. As a result of searching among the various simulation packages, there has not been found suitable means for parallel loading of distributed data relating to the considered task. This problem drove us to do this research.

Secondly, the scale of the input data can differ greatly. It can be a small one-dimensional array or a large number of files distributed across the various computational nodes and file systems. Such problems are usually solved by means of a software system that generated this data, or by development of a specialized "loader" tool, which understands input-output formats used by the calculation program.

Thirdly, there is a need to process such results for convenient representation on charts or in 3D visualization.

Due to the features described above, in this work we made an attempt to create a framework for the software complex with the following features:

- Parallel reading of data from different sources;
- User-defined data formats support;
- Custom data filters and processors support;
- Data visualization solution;

It is important to emphasize that in the case of development of such library its expandability has a significant role. It should be relatively easy to use the developed framework for processing the data stored in any format, and to integrate it with the other known solutions for visualization and data processing. As the initial stage of development we chose the problem of post-processing and visualization of the results obtained in work described in the article [1].

This task involves the consideration of all the listed features of the selected application, because of the distributed structure of the data in different computer systems with remote access to it via SSH.

3. Development tools

There are many known solutions for task-based paralleling and data visualization.

Feature of these solutions is the difficulty of their use, setup and installation.

Among the known solutions for clustering can be noted Apache Hadoop. This is a large and complex solution, which implements MapReduce model for task-based parallel processing. However, for the considered problem, it has many unnecessary features, such as a distributed file system (HDFS) and requirement of installation on computational nodes.

For general scientific visualization, there is a variety of software packages, for example, Paraview, VMD, Tecplot. Each of these software packages has its own format of data storage, and is also able to read the standardized formats. However, in the case of a custom data format or a complex data distribution all of these solutions require implementation of a special data loader.

Taking all the above into account, we decided to add into the developed library the support of the integration into such packages, and its own visualization and clustering tools. Furthermore, "mmdlab" library has a minimum set of dependency and does not require installation on the compute nodes.

In view of the need for the above-mentioned integration into well-known solutions, as well as the requirements posed by the expandability of developed framework, we decided to use an interpreted programming language Python, due to the fact that almost all of that packages use Python in their plug-in systems.

3.1 Python

Python [2] is a widely used in scientific community general-purpose high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than it would be possible in languages such as C++ or Java. The syntax of kernel of Python is very simple and short, at the same time a standard library gives the large volume of useful functions and convenient data structures. It is also a cross-platform, so you can use it (with some restrictions), both under the MS Windows and Linux operating systems.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management, full introspection, exceptions and multiprocessing.

The developers community created a lot of computer science libraries, that makes Python one of the most commonly used languages for big data analysis and scientific calculations.

Though Python already has version 3, in this study we used Python version 2.7, in view of the fact that some used libraries (for example, Mayavi2) were written in Python 2.7, and Python 3 and Python 2.7 in some cases do not have backward compatibility.

3.2 IPython

IPython [3] is an interactive shell for Python language, which adds an expanded introspection, additional command syntax, code highlighting and autocomplition. The main feature of this project is that it provides the core for Jupyter web-application, which allows to write scripts in Python, R, and BASH directly in the browser, as well as interact with the objects of visualization. In this work IPython notebook application has been selected as the web-control system.

3.3 Accelerators of computations

Despite all the advantages of the main realization of the interpreter CPython, it is necessary to remember that the Python is a high-level interpreted programming language. It cannot provide high performance itself, due to the memory management system and dynamic typification. It is very easy to use, but if performance is critical it is necessary to implement CPU-critical code in C or C++, to avoid the overhead of interpreter calls. However, there are several technologies allowing to evade the low-level programming.

Another big disadvantage of the CPython interpreter is associated with the speed and performance in multithreading. The last is caused by use of the GIL (Global Interpreter Lock) mechanism representing mutex (the elementary binary semaphore) which is not allowing different threads to process the same bytecode at the same time. Unfortunately, this lock is necessary, since the memory management system in CPython is not thread-safe. The following methods were considered to avoid this limitations.

3.4 Numpy

Numpy [4] is an open source library for Python. It implements fast multi-dimensional arrays and plenty of parallel (vectorized) algorithms for linear algebra, Fourier transform and other applications. Since Numpy is written in C, the executable code of the library is compiled into native code, and there is no need for its interpretation, gaining significant speedups of the array-processing methods. The threads that run inside Numpy do not depend on the GIL, present in the CPython, and therefore its use accelerates the execution of algorithms by parallelization. Besides Numpy has detailed documentation that facilitates the development and maintenance of the software. All these features make Numpy reasonable choice for array processing in Python.

3.5 Numba

Numba [5] is optimizing Just-In-Time (JIT) compiler, which allows to accelerate the time-critical code by compiling it into native code. Unlike Cython, Numba does not require explicit type annotations (but supports it) and does not translates the code in C language, which simplifies the use of this technology. In order to show Numba which methods are needed to be optimized, the user must use the simplest means of Python language, called a decorator.

Marked by the special decorator methods Numba optimizes and compiles to machine code using LLVM (Low Level Virtual Machine) infrastructure. With the ability to turn off the GIL, as well as the compilation to native code without using the Python C API (for the methods that operates elementary types), Numba compiler can generate more efficient and optimized bytecode. Numba also automatically vectorizes all that it can handle, utilizing the capabilities of multiprocessor systems to the maximum.

```
from numba import jit
@jit(nogil=True, nopython=True)
def numpy_numba_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider
def numpy_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider
```

Listing 1. Numba and Numpy array multiplication.

Table 1. Numba and Numpy performance comparison.

N	Numpy	Numba	Speedup
10 ⁶	0.19 ms	0.07 ms	2.77
10 ⁷	1.62 ms	0.74 ms	2.19
10 ⁸	16.06 ms	7.4 ms	2.17

Table 1 compares the speed of execution of the same Python code (multiplication arrays with multiplying and dividing by a constant, see Listing 1), in one case without Numba, in the other using this technology. Testing was performed on a system with the Intel Core I7-3630QM CPU.

It should be noted that the algorithm shown in Listing 1 is not parallel in the means of code, and the vectorization is performed by Numpy.

The Table 1 shows that Numba allows to speed up the execution nearly twice due to JIT compilation, without any special optimization, such as, most likely, would be needed while using any other tools, such as Cython.

4. Parallelization tools

Considering a GIL mechanism, presenting in CPython, the use of standard Python threads is not an effective solution for parallel processing. GIL does not allow multiple threads run simultaneously on different cores (within one interpreter process) even on a multiprocessor system. However, running multiple processes of interpreters, which can exchange data, completely solves this problem. The only distinctive in this case is that the launch of the process is a much more prolonged operation than starting threads, and usage of multi-process application on small data is not rational. There are several tools for easy management of such tasks.

4.1 Multiprocessing

Multiprocessing [2] is a standard library module that provides an interface to create and manage multiple interpreters processes. Its API is similar to the threading module of the standard library. It also adds some new features, such as the Pool class, representing the abstraction and control mechanism for a set of parallel interpreter processes. Multiprocessing also implements interprocess primitives, such as queue and mutex. It is also worth noting that each process of the interpreter works in separate memory space, therefore there is no need to worry about race conditions when writing or reading variables, unless they are declared as an object in shared memory. Communication between the processes of the interpreter within a given library is through interprocess communication channel, based on pipes, using the pickle module, allowing to "serialize" and "deserialize" the Python objects (serialization - the process of transferring any data structure into a bit sequence; deserialization - the restoration of the initial state of the data structure from a bit sequence). All the tasks of synchronization and object transferring are carried out by the Multiprocessing module. Therefore, the user does not need to solve the problem of confirming that all data used in the calculation has been updated.

4.2 ParallelPython

ParallelPython (PP) [6] is a library used to solve the problem of clustering applications. Its implementation has a client-server structure and it requires installation of the server part on the compute nodes. However, the server program of the PP is a simple one-file script, that can be transferred into the node in any possible way. Because of the simplicity of PP interface, it allows to run a computational task on a parallel cluster in few lines of code.

This library has its own load balancer, and it also monitors the status of nodes and redistributes tasks in case of non availability of one of them. With Multiprocessing module, ParallelPython allows simply and conveniently use all of the capabilities of the cluster computing.

Listing 2 shows an example of summing up the plurality of arrays in parallel mode, using ParallelPython and Multiprocessing.

```
import pp
import numpy as np
ppservers = ("10.0.0.1", "10.0.0.2", "10.0.0.3", "10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
    pool = multiprocessing.Pool(2)
    half = len(array)/2
    s = sum(pool.map(sum, [array[:half], array[half:]]))
    return s
arrays = [np.ones(5000) for i in xrange(10) ]
imports = ("multiprocessing",)
deffuncs = tuple()
jobs = [serv.submit(mpsum, (a,)), deffuncs, imports) for a in arrays]
s = sum([job() for job in jobs])
print s
```

Listing 2. Parallel Python and multiprocessing usage for multiple arrays summation.

At every computational node, two processes start by ParallelPython and each of them starts other two process by means of Multiprocessing. It is worth noting that this library, as well as Multiprocessing, uses the "pickle" module to serialize data and tcp / ip network messaging.

5. Visualization tools

As it was already mentioned, there are many third-party tools for data visualization. The "mmdlab" library presented in this work can be used as a tool for preparation of data for the visualization in such packages, however it was also decided to add its own visualization capabilities. During the research it has appeared that the listed below libraries almost do not concede in options to the well-known packages for scientific visualization.

5.1 Mayavi2

Mayavi2 [7] is a Python framework, which allows to build a general-purpose scientific visualization. It gives user a possibility to load and render the data in a separate GUI application and also has a convenient Python API for scene construction and rendering. This library is built over the well-known in scientific community VTK library.

Mayavi2 gives ample opportunities for the visualization of data, beginning from hydrodynamic calculations and finishing with atomistic data. In the case of the interactive GUI mode, tools for changing the rendering parameters, such as the size of objects, color schemes, filter settings are also available. Mayavi2 also has a possibility of the offscreen-rendering (without displaying image), that is extremely important for the server, distributed and batch operation of a large number of data.

```
import pp
import numpy as np
ppservers = ("10.0.0.1", "10.0.0.2", "10.0.0.3", "10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
    pool = multiprocessing.Pool(2)
    half = len(array)/2
    s = sum(pool.map(sum, [array[:half], array[half:]]))
    return s
arrays = [np.ones(5000) for i in xrange(10) ]
imports = ("multiprocessing",)
deffuncs = tuple()
jobs = [serv.submit(mpsum, (a,), deffuncs, imports) for a in arrays]
s = sum([job() for job in jobs])
print s
```

Listing 3. KDE calculation and visualization script using SciPy and Mayavi.

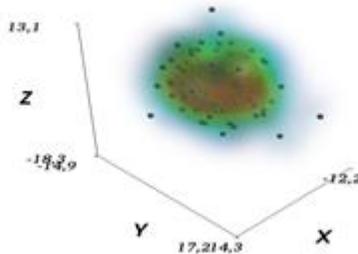


Fig. 1. Listing 3 execution result: Kernel Density Estimation as volume visualization.

Listing 3 and the Fig. 1 show an example of the density distribution calculation of points and its three-dimensional visualization using Mayavi2 and library for scientific computing SciPy.

5.2 Matplotlib

Matplotlib [8] is a Python library for building high-quality two-dimensional graphs. It is widely used in the scientific community. Usage of Matplotlib is very similar to the usage of the plot methods in MATLAB, however, they are independent projects. It is particularly convenient that the plots, which are drawn with the help of this library can be easily integrated into applications written with different libraries for GUI construction. Matplotlib can be integrated into applications written using the wxPython, PyQt and PyGTK libraries.

Matplotlib module is not included in the standard library, but it is the de facto standard for the visualization of numerical information.

6. Distributed data access

The data obtained from the algorithm, described in the article [1] has distributed structure, and is stored on the compute nodes, used for simulation.

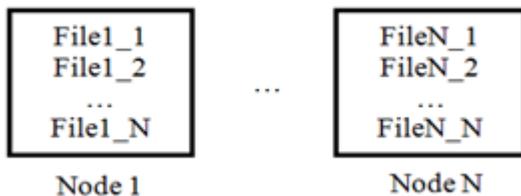


Fig. 2. Data distribution structure.

Fig. 2 shows an example of such data arrangement. The composition of all the files is a complete form of the system simulated by means of molecular dynamics. It happens that the computational nodes use the shared disk space, for example, by means of the NFS (Network File System). However, access to the data from the client-side which needs to read and process the data is open only via SSH. Paramiko library can be used to solve this problem.

6.1 Paramiko

Paramiko [9] is a library for the Python language, which provides implementation and interface for interacting with remote systems via SSHv2 protocol. This library has both client and server implementations. In addition, Paramiko provides a convenient API, which implements objects of "file" type, which are representing files on the remote filesystem. This functionality was used as a basis for the implementation of SSH collector in the represented work.

7. Implementation details

Using the tools above, there was initiated the development of the software complex, allowing to achieve the objectives, namely the parallel data reading and processing, as well as their visualization. As an initial stage, "mmdlab" package was written which implements a general purpose API for such tasks. Below are described the implementation problems we have to handle, application and solutions with the means of the developed library. There is also drawn further attention to the implementation peculiarities in some parts of the package.

7.1 Parallel data access

A module for reading and partial processing of the input data was named "datareader". In this module have been implemented the necessary objects for reading and representation of the data, such as Container, Parser and means of access to the files on the local file system and via SSH. In the terminology of "mmdlab" package,

Container is a structure that stores the read data in a user-defined format. Parser is a special object that reads binary data structure and parses them, thereby obtaining a container. The Parser class receives the raw data from the Transport object that provides an interface for the access to the local or remote file system.

Inheriting and combining objects from these classes, the user can easily make the loader, that parse a custom data format, and accesses it using any protocol, such as SSH or HTTP.

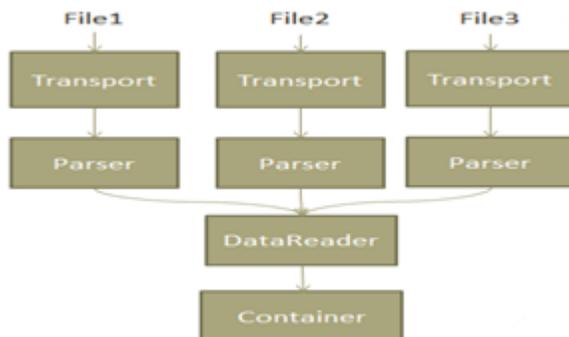


Fig. 3. MMDLAB components scheme.

On the Fig. 3 are shown the "mmdlab" components interactions.

Let's consider the reading procedure of the MD system's particular state described the article [1]. Given the distributed structure of input data, a single state of the system is a set of files of the atomistic data. For each of them it is necessary to read, parse and compile binary structure into a single container that contains the representation of the simulated system. For the performance needs it is necessary to use a parallel algorithm for the reading and processing of the data.

Master process launches N slave-processes that are able to load and parse the data.

Then it begins to give every data file address to a every free process. When the slave process has finished the reading and parsing procedure, and assembled its part of the container, the master process combines the loaded data with its master container, and then assigns a new file to the slave process. After all the slave processes are completed, and there are no more files for reading, master process provides the necessary post-processing for the container, where all of the available data is stored, and sends it to the next data processor in line. It should be noted that in some cases it is not necessary to send all the data to the master host. For those cases, the "mmdlab" supports a possibility to use the post-processing pipeline in the slave processes, so they can make necessary calculations and send back only the result, but not all the processed data set.

In order to enhance the ability of "mmdlab" package for reading the custom-format data, it is required to describe the new entity for storage and loading of such data.

As an example, consider the implementation of such entities for reading a CSV (Comma-Separated Values) format with three columns.

```
class CsvCtr(dr.containers.DummyContainer):
    def __init__(self):
        self.cols = [[], [], []]
    def append_data(self, data):
        for i, d in enumerate(data[:]):
            self.cols[i].extend(d)
class CsvParser(dr.parsers.DummyParser):
    def data(self):
        cols = [[], [], []]
        for line in self.transport.readlines():
            c = line.split(",")
            for i in range(0,3):
                cols[i].append(c[i])
        return cols
nodes = \
({ "ip": "10.0.0.1", "pwd": "123", "login": "test",
  { "ip": "10.0.0.2", "pwd": "123", "login": "test"}
remotedirs = [(sys.argv[1], node) for node in nodes]
transport = dr.transport.RemoteDirs(remotedirs)
parser = CsvParser()
rdr = dr.DistributedDataReader(file_mask="1*.csv",
                              transport=transport, parser = parser,
                              container = CsvCtr)
container = mmdlab.run([rdr, ])
```

Listing 4. CSV Container and Parser implementation using "mmdlab" package.

Listing 4 shows an example of such an extension to CSV reading from remote file systems via SSH.

In practice the user will need to describe the new class inherited from the class `DummyContainer` and to redefine the `append_data` method in it. Also it will be required to describe the class for raw data parsing.

7.2 Pipeline

In this work, to run reading and processing tasks, it is proposed pipeline-type interface (see Listing 5, the `mmdlab.run` part).

This method makes it possible to run an execution of a chain of actions in one line, each of which is carried out over the result of the previous task. Also parallel operations over the same result of the previous method are supported.

For example, the call of `mmdlab.run([generate, [f1, f2, f3], sum])` first performs the "generate" method, then in parallel mode it runs three processes: "f1", "f2", "f3" each operating on the result of the "generate", and in the end it will summarize the obtained values. Restrictions on objects in the pipeline are simple: the object has to be callable, it should take the data for processing as an argument and it should return an object.

```
from mmdlab.datareader.shortcuts import read_distr_gimm_data
import mmdlab
import sys
reader = read_distr_gimm_data(sys.argv[1],sys.argv[2])
filter_reg =
mmdlab.dataprocessor.filters.RegionFilter([0,10,0,10,0,10])
parts_descr = \
{ "Nickel" : { "id" : 0, "atom_mass" : 97.474, "atom_d" : 0.248}, \
  "Nitrogen" : { "id" : 1, "atom_mass" : 46.517,"atom_d" : 0.296} }
filter_split = mmdlab.dataprocessor.filters.SplitFilter(parts_descr)
container = mmdlab.run([reader, filter_reg, filter_split ])
met,gas = container["Nickel"], container["Nitrogen"]
mp = mmdlab.vis.Points3d(met, scalar=met.t, size=met.d,
                        colormap="black-white")
gp = mmdlab.vis.Points3d(gas, scalar=gas.t, size=gas.d,
                        colormap="cool")
mmdlab.vis.colorbar(gp, "Gas T")
mmdlab.vis.show(distance=20)
```

Listing 5. Reading, processing and visualization of the atomistic data using "mmdlab" package.

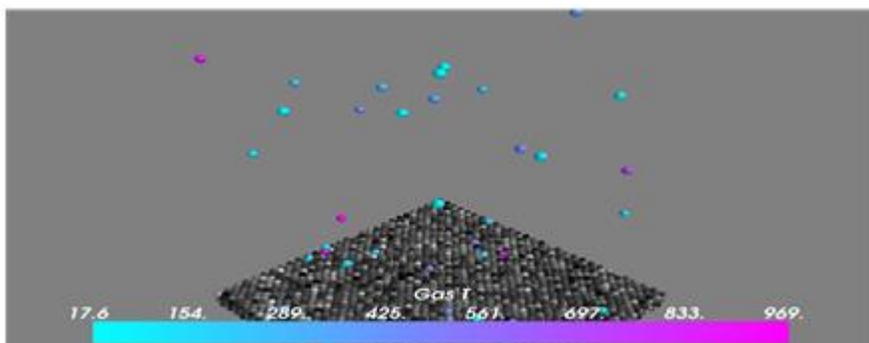


Fig. 4. The result image produced by execution of Listing 5.

At the current stage of development, when you run a multithreaded processing over the previous action the result will be copied to each of the child process.

In the future we plan to add some additional entities, allowing to manage the execution workflow, such as a special object that allows to perform an action in the master process, and to send the result's parts to the slave-processes. This may be necessary, for example, for the separation of the array into a multiple parts, and process each in a separate slave-process without sending the entire array to it.

Due to the fact that the pipeline is implemented by means of the interface module Multiprocessing, consider some of the problems encountered.

7.3 RAM leak in parallel processing

Let's consider the reading procedure of the `DistributedDataReader` class (see Listing 6).

```
class DistributedDataReader:
    ...
    def read(self):
        ...
        files = self.transport.list(self.file_mask)
        container = self.container()
        pool = Pool(processes=self.np, maxtasksperchild=self.mtpc)
        results = [pool.apply_async(rd, \
            args=(f, self.transport.filer(), self.parser)) \
            for f in files]
        for ct in results:
            container.append_data(ct.get())
        return container.finalize()
```

Listing 6. A part of DistributedDataReader class.

During the testing it was found that a resources leak appears in the multiprocessing mode. After starting the pool of processes, and performing a variety of tasks in it, memory consumption increases dramatically. It became apparent that by default the started by Multiprocessing library interpreter processes handle all the scheduled tasks without restarting.

Each task which is carried out in such processes leaves the context, which becomes bigger in the volumes of consumed memory as the more data the task returns. As a result, after long-term execution of multiple tasks at the computational node the RAM came to an end.

The proposed solution of this problem is as follows. The object of a processes pool has a special parameter of the constructor named "maxtaskperchild", allowing to set the number of tasks that a single interpreter process can handle. When the counter of finished jobs becomes more then this value, the master-process algorithm will restart the interpreter. Changing this parameter allows to vary the maximum amount of memory consumed. However, it should be noted that the smaller the value, the more often the master process will restart child processes' interpreters. It can take noticeable amount of time.

Within the considered task of processing large amounts of data, the time is not critical, and installation of rather small value is quite justified because of memory limits.

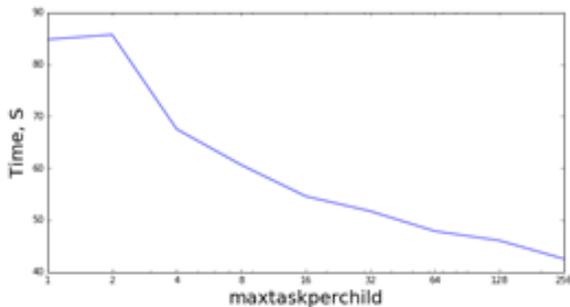


Fig. 5. Loading time of 256 files depending on a "maxtaskperchild" parameter, logarithmic scale.

Fig. 5 shows the dependence of the loading time on the "maxtaskperchild" parameter. The loader uses multiprocessing module, with the pool consisting of one process, and loads 256 data files in serial mode.

Taking into the account the Fig. 5, the optimal behavior of the processes pool is to restart the slave-workers every 16 tasks. It makes possible limiting the consumption of RAM and at the same time keeps the overhead of the interpreter restart time influence almost negligible.

7.4 Multiprocessing and Pool of Pools

Another problem encountered in the development process is the fact that the default multiprocessing library does not allow to create "nested" pools for processes.

In particular, if there appears a necessity to run in parallel the processes of reading a plurality of states of the studied system (this will start new slave-processes that should start a lot of reading processes), for example, for the particles' trajectories construction, so the Multiprocessing module will not allow to do it. The introspection which is supported by the Python language fully helps with the solution of this problem.

The "mmdlab" package developed in this work has a construction shown in Listing 7 included in it. It redefines the `_get_daemon` and `_set_daemon` methods at the "multiprocessing.Process" class and provides a new object, inherited from the Pool class. It should be used instead of the standard Pool class from Multiprocessing module.

```
import multiprocessing
import multiprocessing.pool
class NoDaemonProcess(multiprocessing.Process):
    def _get_daemon(self):
        return False
    def _set_daemon(self, value):
        pass
    daemon = property(_get_daemon, _set_daemon)
class MultiPool(multiprocessing.pool.Pool):
    Process = NoDaemonProcess
```

Listing 7. MultiPool class, allowing to run pool of processes inside child process, created by multiprocessing module.

7.5 Data processing

For processing and filtering data in developed "mmdlab" library the same mechanisms as for the data reading are used. The so-called "pipeline" architecture is used which implicates the container object passing through a chain of a great number of data processors, that can change, supplement a container or create a new one. The "run" method in the "mmdlab" package passes the container obtained from the previous task to the input of the next processing method. The implementation of these processing methods can be both serial and parallel.

In the application to the analysis specific objective of molecular dynamics simulations' results from the article [1], the objects for data post-processing have been added to the developed library. For example, a filtration of particles by various criteria, in particular for getting the particles only from specified area, for filtration by indexes and division of particles according to physical materials.

All computationally intensive procedures were optimized by using Numpy and Numba.

As a simple example, let's consider the task of visualizing of the particles' position and temperature that are divided by criteria of physical material in the predetermined area. Such problem can be solved using "mmdlab" library in the following way (see Listing 5). First, the user creates an object of the data loader, setting their location in the filesystem and a time mark.

Then they need to specify the description of particles, which the division filter will work with, and create the corresponding objects of filters (the location filter and the division filter). Lastly they need to pass these objects to the pipeline. Calculation of temperature is performed during the container's post-processing stage.

Listing 5 and Fig. 4 show the listing of such task and the execution results.

```
import sys
import mmdlab
from scipy.stats import *
from mmdlab import parallel
from mmdlab.dataareader.shortcuts import *
from mmdlab.dataprocessor.filters import *
rdr = read_distr_gimm_data(sys.argv[1],0)
def calc_kde(kde, data):
    return kde(data.T)
parts_descr = { "Nickel" : { "id" : 0}, "Nitrogen" : { "id" : 1}}
filter_split = SplitFilter(parts_descr)
filter_reg = RegionFilter([0, 100, 0, 100, 0, 100])
cont = mmdlab.run([rdr, filter_reg, filter_split])
gas = cont["Nitrogen"]
kde = gaussian_kde(np.vstack([gas.x,gas.y,gas.z]))
xi, yi, zi = np.mgrid[0:gas.x.max():30j,
                      0:gas.y.max():30j,
                      0:gas.z.max():30j]
c = np.vstack([item.ravel() for item in [xi,yi,zi]])
cores = sys.argv[2]
nodes = ("192.168.6.15", "192.168.6.20")
cluster = parallel.Cluster(nodes)
args = [(kde,a) for a in np.array_split(c.T, cores)]
cluster.map(calc_kde, args)
density = np.concatenate(results).reshape(xi.shape)
```

Listing 8. KDE Clustering example using "mmdlab" package.

7.6 Cluster processing

For testing of the cluster mode was used the combination of the master node with the Intel Xeon E5-2650 (32 cores) and 6 compute nodes (Intel Xeon 5150 2.66 GHz, 24 cores) with shared file system over NFS. It gives certain freeness in respect of access to the data: it is not required to associate the input and the node on which processing is started, as any datafile is available from any of nodes.

However, such configuration has a bottleneck: the storage input-output performance. As a result, it was decided to use the following strategy: a master node, which is a physical data storage, in the multiprocess mode loads data into memory and sends it to the cluster nodes in the form of internal representation, without the data processing. In contrast to the strategy of "reading on each node" the described way allows to use the computational capabilities of the subordinated nodes on maximum, with the minimum input-output waiting, maximizing disk input-output utilization.

In case of difficult visualization for which processing and rendering takes more time than reading one system state, such approach allows to reduce the average time of full processing almost to the data reading time, which is the potential minimum time of processing.

As an example of clustered task, consider the problem of constructing three-dimensional field of the gas density in the computational domain using Kernel Density Estimation (KDE) algorithm, implemented in SciPy library (see Listing 8).

The graphs of execution time (see Fig. 6) and the acceleration (see Fig. 7) of such calculations, depending on the number of processors for a variable number of subtasks are shown below.

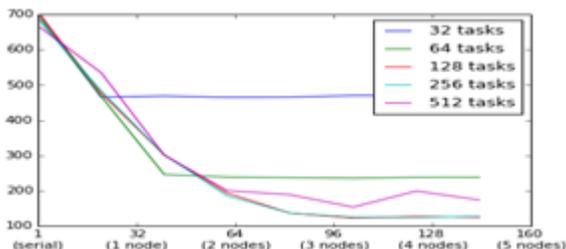


Fig. 6. Processing time for parallel KDE algorithm with various number of subtasks, depending on the number of used processors.

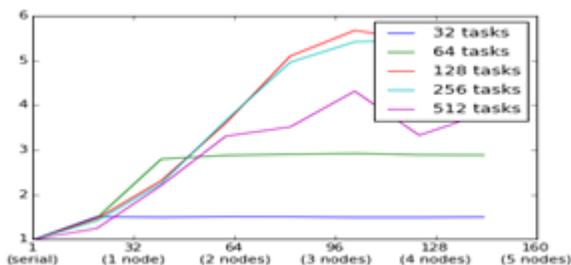


Fig. 7. Speedups for parallel KDE algorithm with various number of subtasks, depending on the number of used processors.

It should be noted that if the number of tasks is less than the number of master node processes (which is up to 32), then the increasing of the process's count in this calculation is not effective. Also, the acceleration increases with the number of nodes involved in the computation, rather than with the number of actual processes. This is due to the following two features:

- PP considers that the overhead of process start-up and data transfer is significantly less on the master-node, than on the slave-nodes. Thus, it loads the master node to the maximum, before it starts to send jobs to the slave-nodes;
- Numpy already vectorizes array operations over all available cores, and the addition of a new processor will not make a significant acceleration;

Also we need to note that the PP, which is used as a library for clustering, automatically distributes the load across nodes, depending on the tasks execution time. So it makes sense to divide the original problem into a number of subtasks more than the number of available processes, if there are some "weak" nodes in the cluster. In this case PP forms a queue and gives tasks to the nodes taking into account efficiency of each node, thereby providing a load balancing.

7.7 Visualization

For the visualization in this work Mayavi2 and Matplotlib library were used. For convenient usage of the common rendering methods, the "mmdlab.vis" module was included, which is a wrapper over the methods of these libraries, combining their capabilities to achieve the desired result. Due to the single-threaded architecture of Mayavi and Matplotlib, data visualization process is currently supported only in the single-threaded mode within a single process. However, "mmdlab" allows to run a hybrid task of reading and rendering on a set of nodes and in the multiprocessing mode, which significantly accelerates the rendering of frame-by-frame video animations.

For example, consider the task of rendering an animation, which consists of frames representing the state of the studied system in consecutive timepoints. Basic data can be distributed across the multiple nodes, thus the visualization can be run on each of the nodes, and then the result can be collected on the master-node. The following algorithm is proposed for the solution of such a problem:

- On each of the specified nodes run a sequence of reading and visualization;
- Collect all the frames that were drawn on the master node;
- Assemble an animation from collected frames;

To build an animated GIF format file "mmdlab" library uses the program "convert" from the ImageMagick [10] utils.

8. Conclusion

This paper presents the experimental version of a high-level library "mmdlab" for the Python language. Usage of such library makes it possible to perform a simple clustering and paralleling for the various types of processing tasks, such as reading, post-processing and visualization.

It can operate over the large-scale data, distributed over the computational nodes in parallel mode.

The main tasks of the development of this library are the analysis and visualization of the data obtained as the result of MD simulation of gas-metal microsystem described in the article [1]. To achieve this goals it was necessary to process about 1.3 TB of data obtained from one simulation, and there were three simulations with different materials temperatures. Usage of the "mmdlab" library allowed to closely observe the effect of nitrogen adsorption on a nickel plate (see Fig. 8) including an analysis of the individual particles' trajectories.

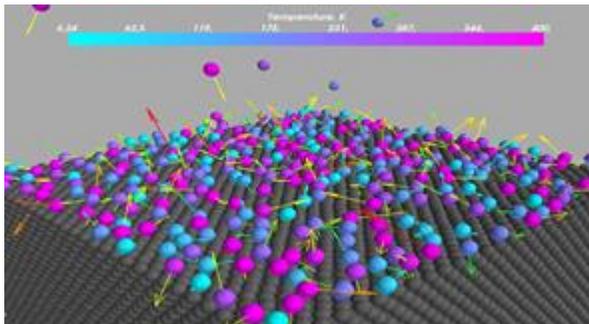


Fig. 8. Adsorption of nitrogen on nickel plate and particle trajectory visualized using the "mmdlab" package.

Special attention was paid to a possibility of extension of the library. It is possible thanks to flexibility of the used tools. As a result, usage of the developed library can be extended to reading and visualization of potentially any structures of data.

9. Acknowledgment

Work is performed with assistance of the Russian Foundation for Basic Research (grants No. 15-07-06082-a, No. 15-29-07090-ofi_m).

10. References

- [1]. V.O. Podryga, S.V. Polyakov, D.V. Puzyrkov, "Supercomputer Molecular Modeling of Thermodynamic Equilibrium in Gas–Metal Microsystems" (in Russian), in *Vychislitel'nye Metody i Programirovanie [Numerical Methods and Programming]*, vol. 16, no. 1, pp. 123-138, 2015 (in Russian).
- [2]. Python official documentation, 04, Feb. 2016, <https://www.python.org/>
- [3]. P. Fernando, E.G. Brian, "IPython: A System for Interactive Scientific Computing" (in English), in *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, 2007. (2015, Feb. 4), [Online]. Available: <http://ipython.org>
- [4]. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, *Computing in Science & Engineering*, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37
- [5]. Numba official documentation, 04, Feb. 2016, <http://www.numba.pydata.org/>
- [6]. Vanovschi V., Parallel Python Software, <http://www.parallelpython.com>
- [7]. Ramachandran, P. and Varoquaux, G., "Mayavi: 3D Visualization of Scientific Data" *IEEE Computing in Science & Engineering*, 13 (2), pp. 40-51 (2011)
- [8]. John D. Hunter. Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering*, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55
- [9]. Paramiko official documentation, 04, Feb. 2016, <http://www.paramiko.org/>
- [10]. ImageMagick official documentation, 04, Feb. 2016, <http://www.imagemagick.org/>

Параллельная обработка и визуализация для результатов моделирования методом молекулярной динамики

Д. В. Пузырьков <druzyrkov@gmail.com>

В. О. Подрыга <pvictoria@list.ru>

С. В. Поляков <polyakov@imamod.ru>

*Институт Прикладной Математики им. М. В. Келдыша Российской
Академии Наук
125047, Москва, Миусская пл., д.4*

Аннотация. В этой работе авторами представляется библиотека "mmdlab" для интерпретируемого языка программирования Python. Эта библиотека позволяет осуществлять чтение, обработку и визуализацию результатов численных расчетов задач молекулярного моделирования. Учитывая большой объем данных, получаемый в результате проведения таких симуляций, существует необходимость в параллельной реализации алгоритмов для обработки таких объемов. Параллельная обработка должна выполняться как на многоядерных системах, таких как обычный современный компьютер, так и на суперкомпьютерных системах и кластерах, где происходило численное моделирование методом молекулярной динамики. В процессе разработки данной библиотеки была изучена эффективность языка Python для таких задач и были рассмотрены инструменты, позволяющие увеличить производительность программ на этом языке. Также были изучены возможности данного языка в отношении параллельных вычислений и инструменты, позволяющие использовать для вычислений системы кластерного типа. Кроме того, были исследованы проблемы загрузки и обработки данных, расположенных на множестве вычислительных узлов. Это было вызвано необходимостью обрабатывать данные, полученные с помощью параллельного алгоритма, который выполнялся на нескольких вычислительных узлах и сохранял результаты на каждом из них. В качестве инструмента для научной визуализации был выбран пакет с открытым исходным кодом "Mayavi2". Разработанная библиотека "mmdlab" была использована для анализа результатов МД моделирования взаимодействия газа с металлической пластиной. В результате применения данной библиотеки удалось в деталях наблюдать эффект адсорбции, который важен для многих практических приложений.

Ключевые слова: параллельная обработка; визуализация; молекулярная динамика; Python; Mayavi2

DOI: 10.15514/ISPRAS-2016-28(2)-15

Для цитирования: Пузырьков Д.В., Подрыга В.О., Поляков С.В. Параллельная обработка и визуализация для результатов моделирования методом молекулярной динамики. *Труды ИСП РАН*, том 28, вып. 2, 2016 г., стр. 221-242 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-15

Список литературы

- [1]. Подрыга В.О., Поляков С.В., Пузырьков Д.В., Суперкомпьютерное молекулярное моделирование термодинамического равновесия в микросистемах газ-металл. Вычислительные методы и программирование, том 16, no. 1, стр. 123-138, 2015.
- [2]. Python official documentation, 04, Feb. 2016, <https://www.python.org/>
- [3]. P. Fernando, E.G. Brian, "IPython: A System for Interactive Scientific Computing" (in English), in *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, 2007. (2015, Feb. 4), [Online]. Available: <http://ipython.org>
- [4]. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, *Computing in Science & Engineering*, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37
- [5]. Numba official documentation, 04, Feb. 2016, <http://www.numba.pydata.org/>
- [6]. Vanovschi V., Parallel Python Software, <http://www.parallelpython.com>
- [7]. Ramachandran, P. and Varoquaux, G., `Mayavi: 3D Visualization of Scientific Data` *IEEE Computing in Science & Engineering*, 13 (2), pp. 40-51 (2011)
- [8]. John D. Hunter. Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering*, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55
- [9]. Paramiko official documentation, 04, Feb. 2016, <http://www.paramiko.org/>
- [10]. ImageMagick official documentation, 04, Feb. 2016, <http://www.imagemagick.org/>

Memristor-based Hardware Neural Networks Modelling Review and Framework Concept

¹ *D.D. Kozhevnikov <ddkozhevnikov@edu.hse.ru >*

² *N.V. Krasilich <nadezhda.krasilich@mail.ru>*

¹ *National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.*

² *National Research University Higher School of Economics,
38, Studencheskaya st., Perm, 614070, Russia.*

Abstract. This paper is a report of a study in progress that considers development of a framework and environment for modelling hardware memristor-based neural networks. An extensive review of the domain has been performed and partly reported in this work. Fundamental papers on memristors and memristor related technologies have been given attention. Various physical implementations of memristors have mentioned together with several mathematical models of the metal-dioxide memristor group. One of the latter has been given a closer look in the paper by briefly describing model's mechanisms and some of the important observations. The paper also considers a recently proposed architecture of memristor-based neural networks and suggests enhancing it by replacing the utilized memristor model with a more accurate one. Based on this review, a number of development requirements was derived and formally specified. Ontological and functional models of the domain at hand have been proposed to foster understanding of the corresponding field from different points of view. Ontological model is supposed to shed light onto the object-oriented structure of memristor-based neural network, whereas the functional model exposes the underlying behavior of network's components which is described in terms of mathematical equations. Finally, the paper shortly speculates about the development platform for the framework and its prospects.

Keywords: memristor; memristor model; hardware neural network model; memristor-based neural networks.

DOI: 10.15514/ISPRAS-2016-28(2)-16

For citation: Kozhevnikov D.D., Krasilich N.V. Memristor-based Hardware Neural Networks Modelling Review and Framework Concept. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 243-258. DOI: 10.15514/ISPRAS-2016-28(2)-16

1. Introduction

Until 1970-s the world has been aware of only three passive elements of electrical circuitry: resistors, capacitors and inductors. The three stated elements coupled with

natural relationships provide five connections for four basic notions of electrical circuit theory (voltage, charge, current and flux). Mathematics, however, claims that four things can be mutually interconnected in six different ways. Indeed, the relation between charge and flux was not present. It wasn't until 1971 that the discordance has been formulated and solved. A new element – memristor - has been proposed by Leon Chua in his paper in IEEE Transactions on Circuit Theory completing the mathematical symmetry of circuit theory. It took nearly 40 years for memristor to transform from a purely theoretic concept into feasible implementation. In 2008 a group of scientists from Hewlett-Packard Labs lead by Stan Williams has finally built working memristors [1].

One of the most promising domains of memristor application, seem to be artificial neural networks [2]. These often come in either software or hardware implementations, sometimes in a combination of both. While digital neural networks simulate the data processing mechanism of biological neural networks, hardware ones strive to emulate it. It is worth mentioning that since most of computer architectures conform to the von Neumann architecture, neural network simulation becomes a challenging task because of the paradigm mismatch. Instead of simulating the ways of nature, hardware neural networks try to directly replicate them, creating non-von-Neumann architectures. In comparison with digitally simulated networks, hardware ones can achieve better speed, less power consumption and chip space.

On the other hand, hardware networks often prove to be far less accurate than their software counterparts, due to the nonuniformity of analog components [3]. Another disadvantage of modern hardware neural networks, which they actually share with the software ones, is the volatile storage of synaptic weights. There are ways to achieve the nonvolatile weight storage within hardware networks, but usually such weights are either static (cannot be changed once manufactured), quickly digress (require frequent updating) or are rather hard to program [4]. The emergence of memristor, however, seems to have opened new possibilities in addressing the stated problems. Memristors seem to be a perfect match for synapses, making hardware implementations of neural networks more reliable and greatly increasing productivity of neural computations [5].

Nevertheless, memristors are still scarcely available and lack industrial-grade production. Being such a new technology, they are often hard and expensive to acquire for experimentation, but a large variety of memristor models has already been produced, making it possible to model memristor-based devices.

Thus, considering the domain of artificial intelligence, a need in profound and correct model of artificial memristor-based feedforward neural network arises. Such model would be of great help in assessing the qualities of modeled system: computation performance, time and energy expenses, material costs, etc. Consequently, the goal of the research is to develop a framework for modelling artificial memristor-based neural networks.

2. Theoretical Memristor

The concept of memristor has been recognized since 1971, when Leon Chua has proposed for the first time in a well-organized and mathematically described way [6]. The 1971 Chua’s paper in IEEE Transactions on Circuit Theory is considered to be the pioneer work in the corresponding field of research. Although, the concept of memristor-like devices has been suggested earlier in 1960 by Bernard Widrow, Leon Chua was the first one not only to provide a feasible foundation for memristor’s existence, but also to estimate and mathematically describe its’ supposed behavior and properties.

Memristor fulfills the mathematical symmetry of relationships between major circuit notions. The relationship created by a memristor, according to Chua, is expressed as follows:

$$v(t) = M(q(t))i(t),$$

where $M(q(t))$ is the memristance defined as

$$M(q) \equiv \frac{d\varphi(q)}{dq}.$$

The definition of memristance may be represented in a more convenient form by substituting flux and charge with their integral definitions:

$$M(q(t)) = \frac{d\varphi/dt}{dq/dt} = \frac{d\left[\int_{-\infty}^t v(\tau)d\tau\right]/dt}{d\left[\int_{-\infty}^t i(\tau)d\tau\right]/dt} = \frac{v(t)}{i(t)}.$$

The similarity of memristor to the remainder of classical circuit elements can be better reflected by expressing their definitions via differential equations as it is done in fable 1.

Table 1. Differential equations of basic circuit elements

Device	Electronic Symbol	Unit	Differential equation
Resistor		R, ohm	$R = \frac{dv}{di}$
Capacitor		C, farad	$C = \frac{dq}{dv}$
Inductor		$L, \frac{Wb}{A}$ or henry	$L = \frac{d\varphi}{di}$
Memristor		$M, \frac{Wb}{c}$ or ohm	$M = \frac{d\varphi}{dq}$

The first important property of memristors, which commonly is referred to as memristance and stands for the ability to change its resistance gradually via a controlled mechanism (e.g. memory of device's history of charge).

The second significant attribute of memristors, figured out by Chua, is the non-volatility property, which stands for the absence of internal power supply. In other words, Chua proposed that memristor is able to store the value of own resistance without the need to be connected to a power source.

In 1976, Leon Chua and his fellow colleague Sung Kang proceeded exploring the mathematical and physical properties of the memristor [7].

They had come to an understanding, that since memristor is a dynamic device, one equation is not enough to describe it, henceforth memristor's behavior is represented by following equations for current-controlled memristor

$$\begin{aligned}x &= f(x, i, t) \\v &= R(x, i, t)i\end{aligned}$$

and for voltage-controlled one

$$\begin{aligned}x &= f(x, v, t) \\v &= R(x, v, t)i\end{aligned}$$

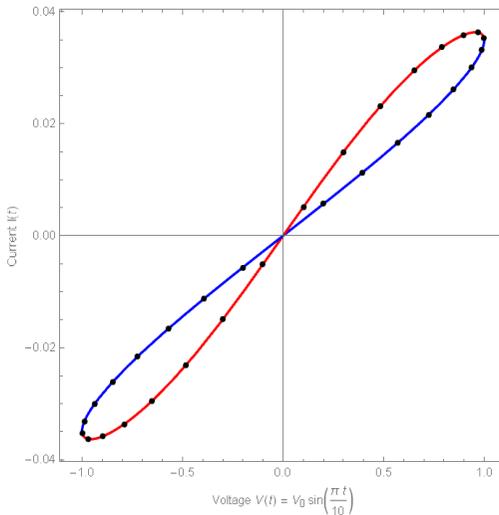


Fig. 1. Pinched hysteresis loop in the i-v curve

where v and i denote the input voltage and current respectively and x stands for the internal state of the device. In their paper, Chua and Kang also provided a more generalized concept of memristive systems with no specific reference to particular physical variables.

One noteworthy peculiarity derived from these equations is that regardless of the state x (which implements the memory effect), the output voltage is equal to zero whenever input voltage or current are equal to zero as well. This zero-crossing property, Chua and Kang write, manifests itself vividly in the form of a Lissajous figure, which always passes through the origin. Thus, they extended the definition of memristor that is now to encompass any system able to demonstrate a Lissajous figure (later called pinched hysteresis loop by Chua) in the i - v curve, which is presented on fig. 1.

3. Memristor Models

However, the true interest has been sparked by the notable work of Richard Stanley Williams' group of researchers at Hewlett-Packard laboratories. Despite this fact, the idea of memristors not being a purely theoretical concept has captivated minds of many researchers around the world, resulting in more than 120 publications about memristors and memristive systems by 2011. [8].

After the concept of memristor was brought back to the public's sight, several implementations of memristors and memristive systems have been proposed. Different implementations of memristor rely on various physical and chemical reactions that give rise to both memristance and nonvolatility, properties essentially constituting the definition of memristor. There have been reported polymeric [9,10], spintronic [11], ferroelectric [12] and layered [13] implementations of memristor, but titanium dioxide memristors remain the most well studied group. During this research four models were closely considered, namely linear ion drift model[1], nonlinear ion drift model[14], Simmons tunnel barrier model[15], and threshold adaptive memristor model (TEAM)[16]. Unfortunately, due to the paper size considerations only the last one of them will be reported. This model, however, was decided to be further utilized throughout the work.

TEAM model, proposed by Kvatinsky et al., incorporates advantages of ion drift models' explicitness and Simmons tunnel barrier accuracy, yet manages to preserve relatively high computational performance and generalizability. TEAM model is based on the same physical behavior as Simmons tunnel barrier model. But it manages to convey it with simpler mathematical functions. The model introduces several assumptions for the sake of analytical simplicity: state variable does not change below a certain threshold and exponential dependence is replaced with a polynomial one. Detailed mathematical foundation of the model may be found in the corresponding paper.

A major advantage of such a relation is the explicitness of current and voltage relationship as opposed to the Simmons tunnel barrier model. Nevertheless, Kvatinsky et al. were able to perform a fitting procedure forcing TEAM model to match the latter with reasonable and sufficient accuracy. In their paper, authors of

TEAM model also report the results of comparison between the fitted TEAM and Simmons tunnel barrier model. The feasible preciseness of TEAM model was proved by the average discrepancy between models' state variable difference of only 0.2%. The maximum difference of this value constituted 12.77%, however the run time of the model was nearly halved (47.5%) Kvatinsky et al. had been also able to fit the model with different types of physical memristor models, namely STT-MRAM and Spintronic memristors.

4. Memristor Bridge Neural Network

This paper considers the neural network architecture proposed by Adhikari et al. in 2012 [4]. The architecture is based on the memristor-bridge synapse [17] and aims to solve the issue of nonvolatile synaptic weight storage and implement a newly proposed hardware learning method.

4.1 Memristor Bridge Synapse

Memristor bridge synapse architecture was first proposed in [17], it is a Wheatstone-bridge-like circuit that consists of four identical memristors of opposite polarities. When positive or negative strong pulse $v_{in}(t)$ is applied at the input, the memristance of each memristor is increased or decreased depending upon its polarity.

Kim et al. write, that if input pulse voltage is equal to v_{in} , voltages at memristors can be calculated according to "voltage-divider formula". Then given memristances M_1 , M_2 , M_3 , and M_4 stand for the corresponding memristors at time t , the output voltage is reported to be equal to the voltage difference between terminals A and B:

$$v_{out} = v_A - v_B = \left(\frac{M_2}{M_1 + M_2} - \frac{M_4}{M_3 + M_4} \right) v_{in}.$$

4.2 Memristor Bridge Neuron

In artificial neural networks neurons are required to sum a set of input postsynaptic signal and, according to the activation function, propagate (or not propagate) the signal further on to the next layer of the network. The neuron is then required to sum the input postsynaptic signals. Kim et al. point out, that the signal summing operation is easier to be performed in current mode: postsynaptic signals should be connected to a single node, so that the following neuron would receive the sum of currents via Kirchhoff's current law. In order to achieve current summation, the memristor bridge synapse has to be modified because it provides voltage output. Kim et al. suggest combining the memristor bridge with differential amplifier. The latter converts post-bridge negative and positive voltage into corresponding currents. Hence, for a set of synapses there exist two nodes: one for positive postsynaptic current and one for negative postsynaptic current. These nodes sum the output currents of each individual synapse in the set. Neuron itself is then comprised of the summation nodes, but also of the active load circuit that implements the activation function as in fig. 2. The sum

of all postsynaptic currents is converted back to voltage (presynaptic signal for next layer of neural network) by the active load circuit according to the activation function. In their paper, Adhikari et al. also provide rigorous mathematical explanation of the suggested architecture behavior.

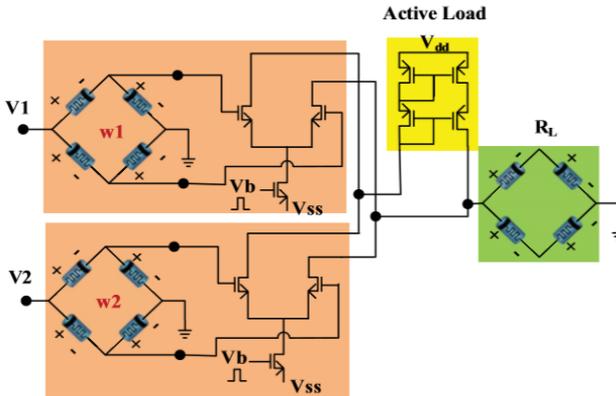


Fig. 2. Memristor Neural Network Circuit Fragment [4]

4.3 Neural Network Training

A composition of an arbitrary number of neurons connected via memristor-bridge synapses therefore constitutes the artificial network. Adhikari et al. intend to use Chip-in-the-Loop technique for training the network of proposed architecture. They, however, suggest modifying this technique slightly in order to take into account peculiar properties of memristor-based circuits. This technique is a viable choice since it provides a way to deal with memristor bridge non-idealities without explicitly modelling these nonidealities. According to this technique, the circuit performs the forward computation of the network, whereas back-propagation and weight update is done on the software side.

The hardware circuit network is reproduced by a software clone, which is used to process the training data. After the computer network has processed all the training data, synaptic weights of each individual synapse circuit are programmed by direct application of strong voltage pulses in order to match with the weights from computer network's weight matrix. Hence, the whole of the hardware network is treated as it consists of a set of simple single-layer networks. Each one of those single layer networks is trained separately, according to the weight matrix. Because of the nature of memristor bridge synapses, the need in additional circuitry is eliminated.

5. Framework Concept

As one can see, plenty of research has been carried out in the field of memristors and memristor-based neural networks. Multiple approaches to both creating and modelling memristors have been mentioned in previous sections.

It is needed to create a reliable framework for simulating memristor-based neural networks. So far, rather abundant overview of the domain has been presented. Despite the vast variety of works mentioned, the domain at hand lacks general integrity and is not formalized enough to start composing the framework at least in its basic form. Hence, the domain must be formalized to a certain extent. In order to derive this degree of formalization, the requirements for the stated framework are to be determined. This will enable framework to be designed properly and will ensure it complies with the needs and wants of its users. Requirements are decided to include four major points: accuracy, performance, flexibility, and explicitness. Accuracy stands for reliability of framework and if its output data can be trusted. Performance reflects how quick does the simulation proceed. Flexibility corresponds to how easy it is to swap components and models within framework. Finally, explicitness is determined by the overall convenience of the framework and how well does it represent results of the simulation. Insights into these requirements can be better revealed according to the SMART criteria (a project management technique for elaborating objectives), which is done in Table 2.

The requirements described above help determine what is to be expected from the framework, what kind of formalization for the domain is required, and set guidelines for further process of design and development. The domain may be formalized by representing it as a graphical scheme, henceforward called ontological model. The reason for such naming is that this model encompasses relevant entities of the domain under discourse, as well as reflects their major properties and interrelations, which in turn roughly corresponds to the definition of ontology. This model will limit the complexity of the field of memristor-based neural networks and expose the intrinsic connections between the notions at hand.

First, let us derive a set of entities to be found within this model. At the very core of every network there are neurons and synapses. These three notions (neural network, synapse and neuron) constitute the heart of designed model as well.

Multilayer network usually distinguishes between input layer neurons, output layer neurons and hidden layer neurons, which may slightly differ. Input neurons should be able to receive input signals, which may not necessarily coincide with how the signals are conveyed within the network. Similarly, output neurons must provide output signals. Consequently, input and output program modules should be introduced, in order to convert electrical output signals into human-comprehensible format and vice versa for the input signals.

Table 2. Framework requirements according to SMART

Criterion	Accuracy	Performance	Flexibility	Explicitness
Specificity	Results of simulation within framework must coincide with corresponding experimental data.	Simulation processing must be performed in a reasonable time.	Frameworks components must be easy to change and replace, due to the domain's novelty.	Simulation results should be clear and easy to observe.
Measurability	Given the same input data the framework must produce the same output data as in either experimental data or in verified models. Thus, the discrepancy between these results may be used to measure accuracy of the framework.	Time taken to perform the simulation and calculate the results reflects how well does the framework perform in terms of performance.	Framework's flexibility can be measured in regard with how many approaches to memristor modelling and network training and architecture does it implement.	Explicitness is the most subjective of all requirements and should be estimated by direct responses of framework's users.
Achievability	Accuracy is achieved through testing the framework and tuning it match with known data.	Performance is achieved through optimization of frameworks algorithms and architecture.	If designed correctly the architecture (structure) of the framework should provide sufficient flexibility.	Various parameters of framework's components must be accessible and visualization methods (graphs, visual models, etc.) should be provided.
Relevance	Accuracy is arguably the most important requirement, without sufficient accuracy, the purpose of the framework is defeated.	Performance is quite relevant since long runtime may hinder the research progress when using framework.	Because the domain is so new, it is extremely important to make the framework able to adapt to possible changes.	Visual representation of simulation results is very important for the end user.
Timeliness	Accuracy may be achieved after tuning the initial version of framework.	Performance should be taken into account during the development, but can be also improved by later optimization.	Flexibility must be ensured from the very beginning of the development.	Visualization may be introduced after the basis of the framework is complete.

Both neurons and synapses of hardware neural networks are implemented through
 Both neurons and synapses of hardware neural networks are implemented through

circuits. Circuit design may vary from one implementation to another, therefore, the general concept of neurons and synapses should be decoupled from its' particular hardware implementation to ensure flexibility. This will enable the framework to safely switch between specific circuit implementations of neurons and synapses, but will also ensure framework's operability. The framework must as well be able to switch between different realizations of memristor, namely, memristor models. Hence, the latter should be considered a separate entity, which is contently used as a component in synapse circuitry. For the time being only the metal dioxide class of memristors is considered to limit already reasonable complexity of the framework.

Finally, the network must should be able to employ different learning techniques. Despite the fact that this work considers only chip-in-the-loop method, the framework should be designed being able to implement various ways of network training. Here it is necessary to take into account not only the learning algorithm, but also how this algorithm is applied to hardware circuit components of the network.

The ontological model is depicted on fig. 3. Solid border circles correspond to the entities of the domain; dashed border circles stand for the properties (attributes) of certain entities; filled arrows represent association relation between entities; empty arrows reflect inheritance (or, possibly, interface implementation); finally, dashed lines reflect attribution connections.

It must be noticed, that the ontological model is likely to be changed in the following works and presented version is not final. Some of the anticipated issues include particular implementations of learning techniques, for instance, chip-in-the-loop does not require auxiliary circuitry, whereas spike timing-dependent plasticity usually does. Another bottleneck to be expected relates to the circuit implementations of neurons and synapses. The latter may consist of multiple circuits that should be represented as separate entities in order to preserve flexibility of the system, yet should conform to the same interface for the sake of integrity.

In this way we shed light onto the structural peculiarities of the future framework. This model is to help composing the classes to be implemented as well as their interrelations. Let us now consider the other side of the developed system, namely, its functional requirements. In this paper, the latter refer to a certain number of capabilities expected by users from the framework.

Framework under development strives to model memristor-based neural network suggested by Adhikari et al., which is described in the previous section. It is also expected to make possible modeling with better level of preciseness by enabling swappable memristor models. For instance, employing TEAM memristor model may significantly raise the relevance of proposed hardware neural network model through fostering the accuracy of memristor's physical model.



Fig. 3. Domain's Ontological Model

The functional scope of the framework may be represented as a set of intertwined mathematical equations that describe various parts of the network model. Each entity of the framework can be characterized with equations that have adjustable parameters, which are usually derived by the authors of corresponding models from experimental data analysis. These equations are extracted from relevant models and are bound in such way, that one equation's output usually corresponds to input of the other equation. This set of equations is depicted on fig. 4.

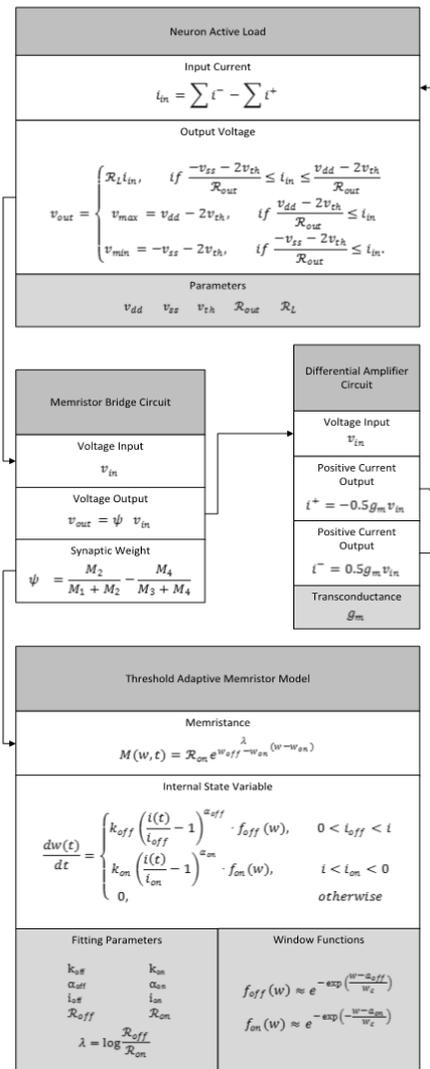


Fig. 4. Functional Structure

Each separate square on the scheme reflects an entity of the framework, while arrows denote the input-output connections between equations. One may notice that relations of equations form a cycle, where one iteration of this cycle corresponds to one layer of hardware memristor network. This figure depicts what set of functions is expected to be provided by the future framework.

6. Conclusion and Prospects

In this paper, a range of memristor models has been reviewed together with some of the fundamental papers on memristor-related technologies. Based on this review, a concept of framework for modeling memristor-based hardware neural networks has been proposed. This framework represents an implementation of neural network architecture considered in the paper, but implies ability to swap memristor models in order to increase the overall flexibility and, possibly, relevance of models generated with the help of proposed framework. The ability to switch between model is also expected to help comparing suggested implementations. In the process of framework structure discovery a set of criteria has been formulated to assess the future software product, domain of memristor-based neural networks has been formalized to a certain extent, and, finally, the framework has been given a functional structure strictly defining its' capabilities.

Specific platform for framework implementation is yet to be chosen. As of current state of affairs, Unity engine is expected to be the most favorable candidate. Its architecture perfectly fits the nature of soft simulation (which the framework ultimately represents), providing some software patterns that greatly alleviate the development. Considered engine is also able to realize extensive visualization of models as well as equip them with user-friendly interface to further enhance model explicitness and facilitate employment of the future framework for academic purposes. Finally, implementing a circuit simulation framework in Unity also pursues an exploration goal, since such attempts have not been previously well studied.

References

- [1]. D. Strukov, G. Snider, D. Stewart and R. Williams, "The missing memristor found", *Nature*, vol. 453, no. 7191, pp. 80-83, 2008.
- [2]. J. Mullins, "Memristor minds: The future of artificial intelligence", *NewScientist Magazine*, no. 2715, 2016.
- [3]. S. Draghici, "Neural Networks in Analog Hardware - Design and Implementation Issues", *International Journal of Neural Systems*, vol. 10, no. 1, pp. 19-42, 2000.
- [4]. S. Adhikari, Changju Yang, Hyongsuk Kim and L. Chua, "Memristor Bridge Synapse-Based Neural Network and Its Learning", *IEEE Trans. Neural Netw. Learning Syst.*, vol. 23, no. 9, pp. 1426-1435, 2012.
- [5]. T. Simonite, "A Better Way to Build Brain-Inspired Chips", *Cacm.acm.org*, 2015. [Online]. Available: <http://cacm.acm.org/news/186782-a-better-way-to-build-brain-inspired-chips/fulltext>. [Accessed: 30- Mar- 2016].
- [6]. L. Chua, "Memristor-The missing circuit element", *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507-519, 1971.
- [7]. L. Chua and S. Kang, "Memristive devices and systems", *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209-223, 1976.
- [8]. A. Thomas, "Memristor-based neural networks", *Journal of Physics D: Applied Physics*, vol. 46, no. 9, p. 093001, 2013.

- [9]. V. Erokhin and M. Fontana, "Electrochemically controlled polymeric device: a memristor (and more) found two years ago", Arxiv.org, 2008. [Online]. Available: <http://arxiv.org/abs/0807.0333>. [Accessed: 30- Mar- 2016].
- [10]. F. Alibart, S. Pleutin, D. Guerin, C. Novembre, S. Lenfant, K. Lmimouni, C. Gamrat and D. Vuillaume, "An Organic Nanoparticle Transistor Behaving as a Biological Spiking Synapse", *Adv. Funct. Mater.*, vol. 20, no. 2, pp. 330-337, 2010.
- [11]. X. Wang, Y. Chen, H. Xi, H. Li and D. Dimitrov, "Spintronic Memristor Through Spin-Torque-Induced Magnetization Motion", *IEEE Electron Device Lett.*, vol. 30, no. 3, pp. 294-297, 2009.
- [12]. A. Chanthbouala, V. Garcia, R. Cherifi, K. Bouzouhouane, S. Fusil, X. Moya, S. Xavier, H. Yamada, C. Deranlot, N. Mathur, M. Bibes, A. Barthelemy and J. Grollier, "A ferroelectric memristor", *Nature Materials*, vol. 11, no. 10, pp. 860-864, 2012.
- [13]. A. Bessonov, M. Kirikova, D. Petukhov, M. Allen, T. Ryhänen and M. Bailey, "Layered memristive and memcapacitive switches for printable electronics", *Nature Materials*, vol. 14, no. 2, pp. 199-204, 2014.
- [14]. E. Lehtonen and M. Laiho, "CNN using memristors for neighborhood connections", 2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010), 2010.
- [15]. M. Pickett, D. Strukov, J. Borghetti, J. Yang, G. Snider, D. Stewart and R. Williams, "Switching dynamics in titanium dioxide memristive devices", *J. Appl. Phys.*, vol. 106, no. 7, p. 074508, 2009.
- [16]. S. Kvatinsky, E. Friedman, A. Kolodny and U. Weiser, "TEAM: ThrEshold Adaptive Memristor Model", *IEEE Trans. Circuits Syst. I*, vol. 60, no. 1, pp. 211-221, 2013.
- [17]. H. Kim, M. Sah, C. Yang, T. Roska and L. Chua, "Memristor Bridge Synapses", *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2061-2070, 2012.

Обзор предметной области и концепция фреймворка для разработки моделей мемристоров и мемристорных нейронных сетей

¹ Д.Д. Кожевников <ddkozhevnikov@edu.hse.ru>

² Н.В. Красилич <nadezhda.krasilich@mail.ru>

¹ НИУ Высшая Школа Экономики,
101000, Россия, г. Москва, ул. Мясницкая, д. 20.

² НИУ Высшая Школа Экономики,
614070, Россия, г. Пермь, ул. Студенческая, д. 38.

Аннотация. В данной работе представлены предварительные результаты текущего исследования по разработке среды моделирования аппаратных мемристорных нейронных сетей. Проведен анализ релевантных трудов, описаны фундаментальные работы по мемристорам и мемристорным технологиям, рассмотрены различные физические реализации мемристоров, а также несколько математических моделей мемристоров из металло-диоксидной группы. Одна из таких моделей более подробно представлена в работе, описаны ее основные механизмы и наиболее интересные свойства. В работе также рассматривается недавно предложенная архитектура мемристорной нейронной сети, описывается методика обучения подобной аппаратной

нейронной сети, реализация ее компонент: нейронов и синапсов на основе мемристорных мостов. В данной работе также выдвинуто предложение по улучшению этой архитектуры путем использования более точной модели мемристора в рамках сети. Основываясь на проведенном анализе предметной области, составлены и формально описаны требования к разработке среды моделирования мемристорных нейронных сетей. Кроме того, для лучшего понимания рассматриваемой предметной области составлены онтологическая и функциональная модели. Первая модель необходима для формализации объектной структуры предметной области, в то время как вторая модель используется для явного представления математических формул, описывающих физическое поведение соответствующих объектов. В совокупности обе модели позволяют составить полное, формализованное и многостороннее описание предметной области мемристорных нейронных сетей и перейти к процессу проектирования и разработки программного продукта. В конце работы кратко представлены дальнейшие перспективы разработки среды моделирования мемристорных нейронных сетей.

Ключевые слова: мемристор; модель мемристора; аппаратная нейронная сеть; мемристорная нейронная сеть.

DOI: 10.15514/ISPRAS-2016-28(2)-17

For citation: Кожевников Д.Д., Красилич Н.В. Обзор предметной области и концепция фреймворка для разработки моделей мемристоров и мемристорных нейронных сетей. *Труды ИСП РАН*, том 28, вып. 2, 2016, стр. 243-258 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-16

References

- [1]. D. Strukov, G. Snider, D. Stewart and R. Williams, "The missing memristor found", *Nature*, vol. 453, no. 7191, pp. 80-83, 2008.
- [2]. J. Mullins, "Memristor minds: The future of artificial intelligence", *NewScientist Magazine*, no. 2715, 2016.
- [3]. S. Draghici, "Neural Networks in Analog Hardware - Design and Implementation Issues", *International Journal of Neural Systems*, vol. 10, no. 1, pp. 19-42, 2000.
- [4]. S. Adhikari, Changju Yang, Hyongsuk Kim and L. Chua, "Memristor Bridge Synapse-Based Neural Network and Its Learning", *IEEE Trans. Neural Netw. Learning Syst.*, vol. 23, no. 9, pp. 1426-1435, 2012.
- [5]. T. Simonite, "A Better Way to Build Brain-Inspired Chips", *Cacm.acm.org*, 2015. [Online]. Available: <http://cacm.acm.org/news/186782-a-better-way-to-build-brain-inspired-chips/fulltext>. [Accessed: 30- Mar- 2016].
- [6]. L. Chua, "Memristor-The missing circuit element", *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507-519, 1971.
- [7]. L. Chua and S. Kang, "Memristive devices and systems", *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209-223, 1976.
- [8]. A. Thomas, "Memristor-based neural networks", *Journal of Physics D: Applied Physics*, vol. 46, no. 9, p. 093001, 2013.
- [9]. V. Erokhin and M. Fontana, "Electrochemically controlled polymeric device: a memristor (and more) found two years ago", *Arxiv.org*, 2008. [Online]. Available: <http://arxiv.org/abs/0807.0333>. [Accessed: 30- Mar- 2016].
- [10]. F. Alibart, S. Pleutin, D. Guerin, C. Novembre, S. Lenfant, K. Lmimouni, C. Gamrat and D. Vuillaume, "An Organic Nanoparticle Transistor Behaving as a Biological Spiking Synapse", *Adv. Funct. Mater.*, vol. 20, no. 2, pp. 330-337, 2010.

- [11]. X. Wang, Y. Chen, H. Xi, H. Li and D. Dimitrov, "Spintronic Memristor Through Spin-Torque-Induced Magnetization Motion", *IEEE Electron Device Lett.*, vol. 30, no. 3, pp. 294-297, 2009.
- [12]. A. Chanthbouala, V. Garcia, R. Cherifi, K. Bouzehouane, S. Fusil, X. Moya, S. Xavier, H. Yamada, C. Deranlot, N. Mathur, M. Bibes, A. Barthelemy and J. Grollier, "A ferroelectric memristor", *Nature Materials*, vol. 11, no. 10, pp. 860-864, 2012.
- [13]. A. Bessonov, M. Kirikova, D. Petukhov, M. Allen, T. Ryhänen and M. Bailey, "Layered memristive and memcapacitive switches for printable electronics", *Nature Materials*, vol. 14, no. 2, pp. 199-204, 2014.
- [14]. E. Lehtonen and M. Laiho, "CNN using memristors for neighborhood connections", 2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010), 2010.
- [15]. M. Pickett, D. Strukov, J. Borghetti, J. Yang, G. Snider, D. Stewart and R. Williams, "Switching dynamics in titanium dioxide memristive devices", *J. Appl. Phys.*, vol. 106, no. 7, p. 074508, 2009.
- [16]. S. Kvatinsky, E. Friedman, A. Kolodny and U. Weiser, "TEAM: ThrEshold Adaptive Memristor Model", *IEEE Trans. Circuits Syst. I*, vol. 60, no. 1, pp. 211-221, 2013.
- [17]. H. Kim, M. Sah, C. Yang, T. Roska and L. Chua, "Memristor Bridge Synapses", *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2061-2070, 2012.

Композиционная модель и способ построения функционально-ориентированных информационных ресурсов информационно-управляющих систем*

И.И. Чукляев <smolrsu@mail.ru>

*Военная академия войсковой противовоздушной обороны
Вооруженных Сил Российской Федерации
имени Маршала Советского Союза А.М.Василевского,
214027, Россия, г. Смоленск, ул. Котовского, 2.*

Аннотация. В статье представлена композиционная модель функционально-ориентированных информационных ресурсов информационно-управляющих систем, а также способ построения этой модели. Предлагаемая модель и способ построения функционально-ориентированных информационных ресурсов обеспечивают расширенные возможности по созданию перспективных средств защиты информационно-управляющих систем, ориентированных на комплексную защиту выполнения задач с учетом уровней управления сложных организационно-технических систем. Композиционная модель соответствует модели данных в не первой нормальной форме, образована функциональным комплексом данных и представляется в виде многоосновной алгебраической структуры, который отображает структуру, взаимосвязи, а также специфику операций манипулирования и обработки над сложно структурированными данными на различных уровнях управления информационно-управляющих систем. Способ построения функционально-ориентированных информационных ресурсов информационно-управляющих систем основан на дедуктивном методе построения аксиоматических теорий. Представлено формализованное описание функционально-ориентированных информационных ресурсов информационно-управляющих систем для решения задач различных уровней управления; функционального комплекса данных многоосновной алгебраической структуры композиционной модели; функциональных зависимостей и отношения, образующих систему аксиом композиционной модели. Показаны графические элементы, на которых отображено распределение компонентов подсистем информационно-управляющей системы на смежных уровнях управления; структурно-

* Исследование выполнено при поддержке РФФИ в рамках научного проекта № 13-07-97518 и гранта Президента Российской Федерации № МК-3603.2014.10.

функциональная диаграмма взаимодействия компонентов информационно-управляющих систем, распределенных по уровням управления; структура композиционной модели функционально-ориентированных информационных ресурсов информационно-управляющей системы, представленная таблицей сложно структурированных данных в не первой нормальной форме; структурно-логическая схема способа построения композиционной модели функционально-ориентированных информационных ресурсов информационно-управляющих систем.

Ключевые слова: информационно-управляющая система; функционально-ориентированные информационные ресурсы.

DOI: 10.15514/ISPRAS-2016-28(2)-17

Для цитирования: Чуляев И.И. Композиционная модель и способ построения функционально-ориентированных информационных ресурсов информационно-управляющих систем. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 259-270. DOI: 10.15514/ISPRAS-2016-28(2)-17

1. Введение

Интеграция информационно-телекоммуникационных технологий в сложных организационно-технических системах (ОТС) специального назначения актуализирует вопросы обеспечения их защищенности от несанкционированных внешних и/или внутренних воздействий дестабилизирующего характера (НСВ), заключающихся в разрушении, повреждении компонентов, модификации (искажении) данных ([1]), ведущих к нарушению выполнения задач управления.

В настоящее время предложены разнообразные методы и средства обеспечения защищенности ОТС и циркулирующих данных в условиях НСВ. Однако, как правило, они «локализованы» относительно отдельных совокупностей данных и процессов и не ориентированы на комплексную защиту выполнения задач с учетом уровней управления ОТС [2, 3].

Предлагается композиционная модель функционально-ориентированных информационных ресурсов (ФОИР) информационно-управляющих систем (ИУС), которая отображает структуру, взаимосвязи, а также специфику операций манипулирования и обработки функционального комплекса данных на различных уровнях управления информационно-управляющих систем, соответствующего модели данных в не первой нормальной форме (*non-first normal form – NFNF*). Предлагаемая модель и способ построения композиционной модели ФОИР ИУС обеспечивают расширенные возможности по созданию перспективных средств защиты ИУС, ориентированных на комплексную защиту выполнения задач с учетом уровней иерархии ОТС.

2. Функционально-ориентированные информационные ресурсы информационно-управляющих систем

Информационно-управляющая система включает функциональную, информационную, организационную и техническую подсистемы. Компоненты этих подсистем распределены по уровням управления ИУС. В табл. 1 показан пример распределения компонентов этих подсистем для смежных уровней управления ИУС[4].

Табл. 1. Пример распределения компонентов подсистем информационно-управляющей системы на смежных уровнях управления.

Table 1. An example of distribution of subsystem information system's components on the adjacent levels of control.

Уровни управления ИУС	Подсистемы ИУС			
	Функциональная	Информационная	Организационная	Техническая
ИУС	Задачи управления <Функция>	Информационные потоки данных <IR>	Должностные лица органов управления <ДЛОУ>	Техническая основа <Механизм>
Уровень A_i	Задачи управления, реализуемые ДЛОУ <Функция ¹ _{nm} >	Информационные потоки <IR>, <Вход Функция ¹ _{nm} >, <Управление Функция ¹ _{nm} >, <Вызов Функция ¹ _{nm} >, <Выход Функция ¹ _{nm} >	Должностные лица органов управления <ДЛОУ _{bc} >	Средства автоматизации <СрАвт>
Уровень A_j	Задачи управления, реализуемые устройствами (аппаратурой) <Функция ³ _{nmlk} >	Показатели структуры <N _{IR} > информационных потоков данных, <Вход Функция ³ _{nml} >, <Управление Функция ³ _{nml} >, <Вызов Функция ³ _{nml} >, <Выход Функция ³ _{nml} >	Действия ДЛОУ <ДЛОУ _{bcd} >	Устройства (аппаратура) <Устройства>

На рис. 1 показана структурно-функциональная диаграмма взаимодействия компонентов подсистем ИУС, распределенных по уровням управления.

Функционально-ориентированные информационные ресурсы ИУС содержат многоаспектную информацию для выполнения всей совокупности задач управления ИУС и представляются в виде [5]:

$$\begin{aligned} \Phi OIP_{\text{ИИС}} = & \{ \Phi OIP_{A_i} \} \triangleright \triangleleft \{ IR \} \triangleright \triangleleft \{ \text{Функция} \} \triangleright \triangleleft \{ \text{Право} \} \triangleright \triangleleft \{ \text{ДЛОУ} \} \triangleright \triangleleft \\ & \triangleright \triangleleft \{ \text{Механизм} \} \triangleright \triangleleft \{ \text{Взаимосвязи} \} \triangleright \triangleleft \{ \text{Свойства} \} \triangleright \triangleleft \{ \text{Временные} \\ & \text{параметры} \}; \end{aligned}$$

Функционально-ориентированные информационные ресурсы для решения задач i -го уровня управления:

$$\begin{aligned} \Phi OIP_{A_i} = & \{ \Phi OIP_{A_j} \} \triangleright \triangleleft \{ \text{Вход}_{\text{Функция}}^i \} \triangleright \triangleleft \{ \text{Управление}_{\text{Функция}}^i \} \triangleright \triangleleft \{ \text{Вызов}_{\text{Функция}}^i \} \triangleright \triangleleft \{ \text{Выход}_{\text{Функция}}^i \} \triangleright \triangleleft \\ & \triangleright \triangleleft \{ \text{Функция}_{nm}^i \} \triangleright \triangleleft \{ \text{Право} \} \triangleright \triangleleft \{ \text{ДЛОУ}_{bc} \} \triangleright \triangleleft \{ \text{СрАвт} \} \triangleright \triangleleft \{ \text{Взаимосвязи} \} \\ & \triangleright \triangleleft \\ & \triangleright \triangleleft \{ \text{Свойства} \} \triangleright \triangleleft \{ \text{Временные параметры} \}; \end{aligned}$$

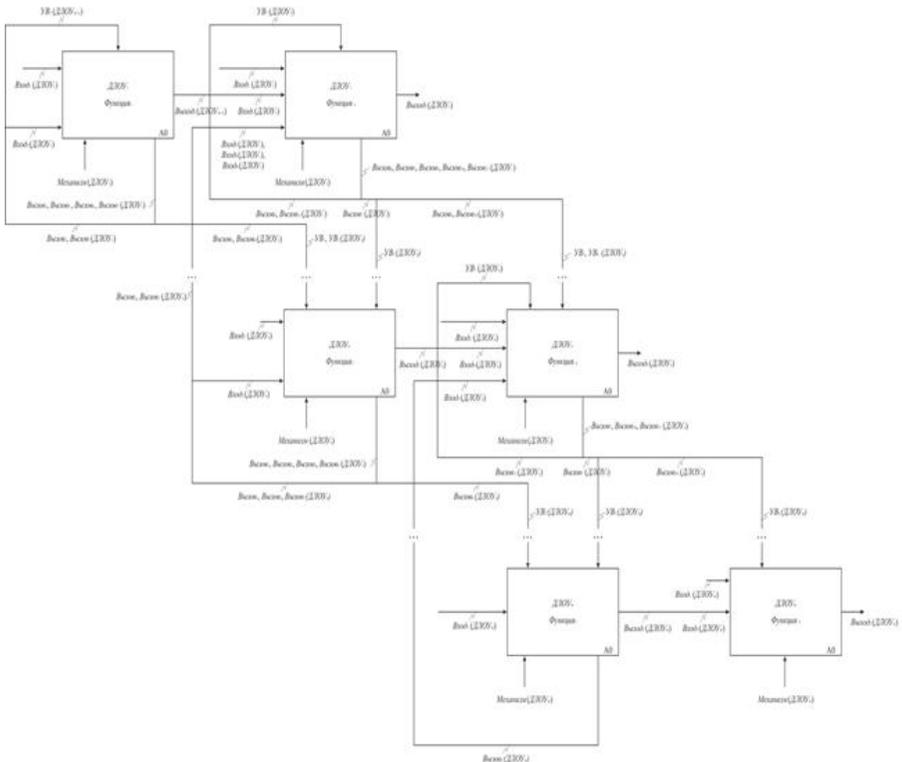


Рис. 1. Структурно-функциональная диаграмма информационно-управляющей системы.

Fig. 1. Structural and functional diagram of information and control system

Функционально-ориентированные информационные ресурсы для решения задач j -го уровня управления:

$$\begin{aligned} \Phi OIP_{A_j} = \{N_{IR}\} \triangleright \triangleleft \{ \text{Функция}_{nmkl}^3 \} \triangleright \triangleleft \{ \text{Право} \} \triangleright \triangleleft \{ \text{ДЛОУ}_{bcd} \} \triangleright \triangleleft \\ \{ \text{Устройства} \} \triangleright \triangleleft \\ \triangleright \triangleleft \{ \text{Взаимосвязи} \} \triangleright \triangleleft \{ \text{Свойства} \} \triangleright \triangleleft \{ \text{Временные параметры} \}, \end{aligned}$$

- где
- $\{ \Phi OIP_{IUC} \},$ – функционально-ориентированные ИУС
 - $\{ \Phi OIP_{A_i} \},$ – информационные ресурсы ИУС
 - $\{ \Phi OIP_{A_j} \}$ – соответствующих уровней управления;
 - $\{ \text{Взаимосвязи} \}$ – взаимосвязи функционально-ориентированных информационных ресурсов ИУС соответствующих уровней управления;
 - $\{ \text{Свойства} \}$ – свойства защищенности, предъявляемые к функционально-ориентированным информационным ресурсам ИУС соответствующих уровней управления;
 - $\{ \text{Временные параметры} \}$ – временные параметры, характеризующие изменения показателей функционально-ориентированных информационных ресурсов ИУС соответствующих уровней управления;
 - $\triangleright \triangleleft$ – операция агрегирования, характеризующая объединение и укрупнение показателей функционально-ориентированных информационных ресурсов ИУС соответствующих уровней управления.

3. Композиционная модель функционально-ориентированных информационных ресурсов информационно-управляющих систем

Композиционная модель ФОИР ИУС включает в себя функциональный комплекс данных и представляется в виде многоосновной алгебраической структуры с учетом уровней управления ИУС:

$$\langle D_1, D_2, \dots, D_n; R; \Sigma \rangle,$$

- где D_1, D_2, \dots, D_n – заданные множества (остовы) данных, соответствующих уровням управления

R	–	функционально-ориентированных информационных ресурсов ИУС; конечный набор функций функционально-ориентированных информационных ресурсов ИУС, определенных на D_1, D_2, \dots, D_n (характеристика структуры);
Σ	–	ограничительные условия, накладываемые на множества D_1, D_2, \dots, D_n и функции данных из R .

Для формирования остовов D_1, D_2, \dots, D_n , характеристик структуры R и ограничительных условий Σ требуется:

выполнить анализ показателей ФОИР ИУС и определить имена, цепи, схемы, ранг;

построить граф N -дерева и таблицы в не первой нормальной форме ФОИР ИУС; выявить F -зависимости и FD -отношение, сегментировать «файлами» функциональные схемы произвольных порядков композиционной модели ФОИР ИУС.

Структура ФОИР ИУС является остовами D_1, D_2, \dots, D_n композиционной модели ФОИР ИУС, а их значения определяют структуру типовых характеристик R . Формализованное описание остовов D_1, D_2, \dots, D_n композиционной модели ФОИР ИУС представлено функциональными схемами FSh произвольных порядков.

Сегментирование «файлами» ($B_{ИУС}^1, \dots, B_{A_i}^i, \dots, B_{A_j}^j$) функциональные схемы FSh произвольных порядков в структуре композиционной модели ФОИР ИУС позволяет идентифицировать ФОИР различных уровней управления количественно (в виде экземпляров).

Выявленные F -зависимости ФОИР ИУС уровней управления, образующие FD -отношение ФОИР ИУС, обобщены в систему аксиом Σ :

$$\Sigma = \left\{ \begin{array}{l} FD_1 : \text{Функция} \xrightarrow{B_{ИУС}^1} IR_{\text{Функция}}, \text{Функция} \xrightarrow{B_{ИУС}^1} ДЛОУ, \text{Функция} \xrightarrow{B_{ИУС}^1} \text{Механизм}, \\ FD_2 : \text{Функция} \xrightarrow{B_{ИУС}^1} \text{Механизм}; \\ FD_3 : (\text{Функция} \cup \text{Механизм}) \xrightarrow{B_{ИУС}^1} (IR_{\text{Функция}} \cup \text{Механизм}) \xrightarrow{B_{ИУС}^1} (ДЛОУ \cup \text{Механизм}). \end{array} \right.$$

Функциональный комплекс данных композиционной модели ФОИР ИУС, структура которой показана на рис. 2, представляется в виде:

$$\Phi КД_{\text{ФОИР ИУС}} = (\text{ФОИР ИУС})$$

```
{(Наименование экз.: число, ФОИР  $A_i$ 
  {(Наименование экз.: число, ФОИР  $A_j$ : значение
  })
});
Σ
).
```

```
ФОИР ИУС =
(ФОИР ИУС
  {(Функция
    {(Наименование экз.: число,  $FSh_{\text{функция}}$ : значение
    )},
  IR
    {(Наименование экз.: число,  $FSh_{IR}$ : значение
    )},
  ДЛОУ
    {(Наименование экз.: число,  $FSh_{\text{ДЛОУ}}$ : значение
    )},
  Механизм
    {(Наименование экз.: число,  $FSh_{\text{Механизм}}$ : значение
    )}
  })
).
```

4. Способ построения композиционной модели функционально-ориентированных информационных ресурсов информационно-управляющих систем

Предлагаемый способ (рис. 3) основан на дедуктивном методе построения аксиоматических теорий ([6]) и позволяет построить композиционную модель ФОИР ИУС, отображающую структуру, взаимосвязи, а также специфику операций манипулирования и обработки над сложно структурированными данными ([7-9]) на различных уровнях управления ИУС, соответствующими модели данных в не первой нормальной форме.

5. Заключение

В качестве основы для средств защиты ИУС выступают ФОИР ИУС, структуру которых составляет функциональный комплекс данных многоосновной

алгебраической структуры, организованные для выполнения задач ИУС с учетом уровней управления.

Комплексная защита выполнения задач ИУС с учетом уровней управления является перспективным направлением создания современных средств защиты ОТС.

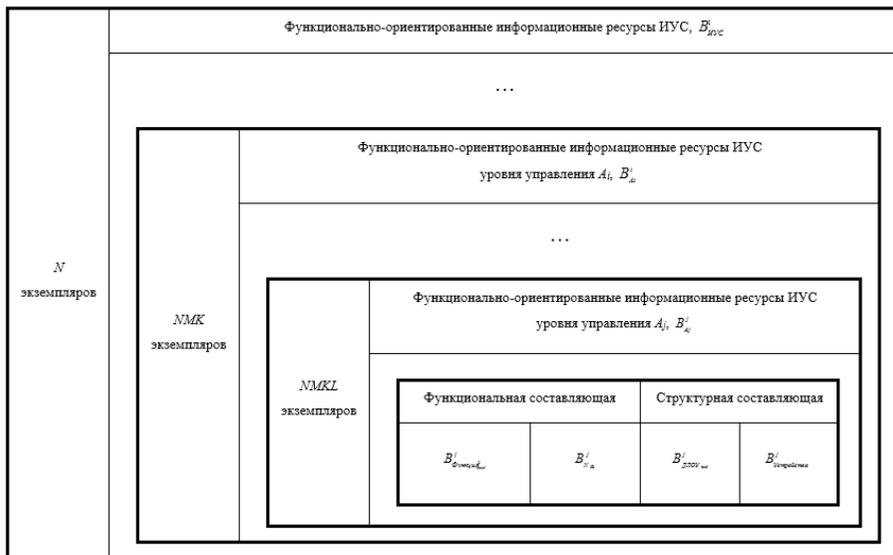


Рис. 2. Структура композиционной модели функционально-ориентированных информационных ресурсов информационно-управляющей системы, представленная таблицей сложно структурированных данных в не первой нормальной форме.

Fig. 2. The structure of the composite model for function-oriented information resources of information and control system presented by table of complexly structured data is not the first normal form.

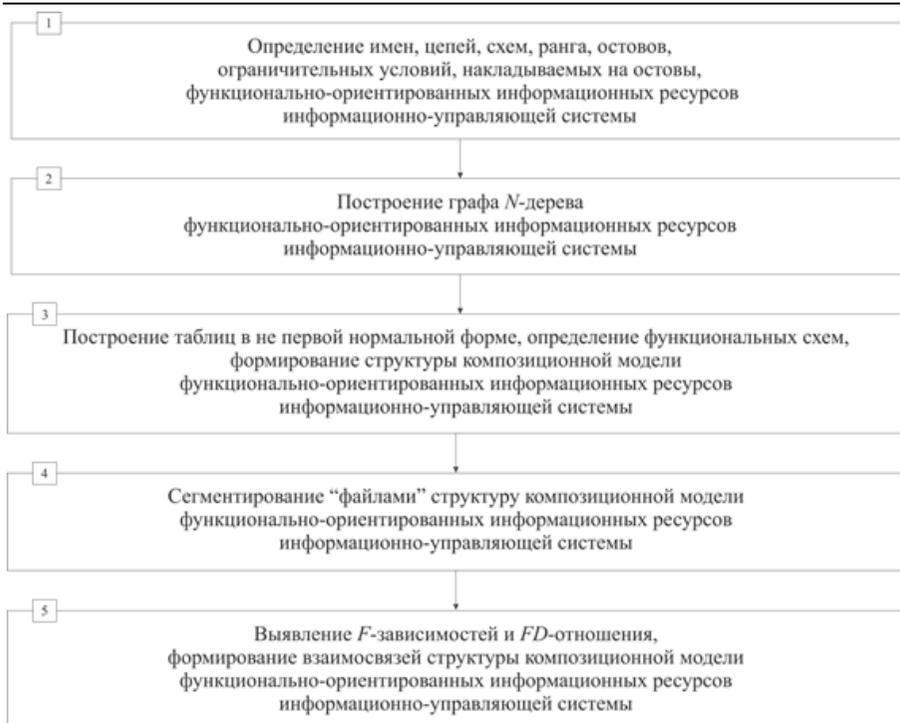


Рис. 3. Структурно-логическая схема способа построения композиционной модели функционально-ориентированных информационных ресурсов информационно-управляющих систем.

Fig. 3. Structural and logical scheme for the method to build a composite model of functional-oriented information resources of information and control systems.

Предлагаемая композиционная модель ФОИР ИУС отображает структуру, взаимосвязи, а также специфику операций манипулирования и обработки над сложно структурированными данными на различных уровнях управления информационно-управляющих систем, соответствующими модели данных в не первой нормальной форме.

Использование данной модели позволит расширить возможности по созданию и внедрению перспективных средств защиты ИУС, ориентированных на комплексную защиту выполнения задач сложных ОТС.

Список литературы

- [1]. Руководящий документ Гостехкомиссии России от 30.03.1992 г. «Защита от несанкционированного доступа к информации. Термины и определения». 5 с.

- [2]. А.В. Морозов, В.В. Борисов, И.И. Чукляев. Вычислительные системы: теоретическое обобщение, развитие, практические результаты. Монография. Смоленск: ВА ВПВО ВС РФ. Издательство «Смоленская городская типография». 2013. 448 с.
- [3]. И.И. Чукляев. Теоретическое обобщение предметной области «информационная безопасность». Тенденции развития методов и средств. М.: ОАО «Концерн «Системпром». Статья. В кн.: Научно-технический сборник ОАО «Концерн «Системпром». Вып. № 1(6)-2015, 2015. С. 471-486.
- [4]. И.И. Чукляев. Информационно-управляющая система в условиях многоуровневого функционально-ориентированного информационного конфликта и подавления. Смоленск: ВА ВПВО ВС РФ. Статья. В кн.: Научно-технический сборник «Вестник войсковой ПВО». Вып. № 13, 2015. С. 183-189.
- [5]. И.И. Чукляев. Метод и модели управления рисками защищенности в информационно-управляющих системах. М.: ОАО «Концерн «Моринформсистема-Агат». Статья. В кн.: Научно-технический сборник ОАО «Вычислительные системы реального времени и цифровые устройства». Вып. № 9, 2015. С. 14-33.
- [6]. Е.П. Емельченков. Базы данных. Современных подход. Смоленск: ВА ВПВО ВС РФ. 2010. 59 с.
- [7]. Е.П. Емельченков, Ю.С. Маленин. О функциональном подходе в теории баз данных. Смоленск: СГПУ. Деп. в ВИНТИ № 6046-84. 1984. 29 с.
- [8]. Ye. Yemelchenkov, M. Tsalenko Functional dependencies in hierarchical Structures of Data. // Lect. notes in Compute Science. Berlin, 1991. V. 495. P. 258-275.
- [9]. Е.П. Емельченков, Н.А. Левин О моделировании сложных предметных областей. // Проблемы и методы информатики. II Научная сессия ИПИ РАН. Тез. докл. / под ред. И. Соколова. М., ИПИ РАН. 2005. С. 89-91.

Composition model and method of creation of functionally-oriented information resources

*I. Chucklyaev <smolrsu@mail.ru>
Military academy of army anti-aircraft defense
Armed forces of the Russian Federation,
2 Kotovskiy Str., Smolensk, 214027, Russian Federation*

Abstract. In the paper the composition model of functionally oriented informational resources is provided as well as the method for creating this model. The composition model and method of functionally oriented informational resources provide enhanced features on creation of informational security systems. The composition model corresponds to a data model in not the first normal form, is formed by the functional complex of data and is presented in the form of polybasic algebraic structure which displays structure, correlations, and also specifics of operations of handling and processing over difficult structured data. The method of creation of function-oriented information resources is based on a deductive method of creation of axiomatic theories. The formalized description of function-oriented information resources is provided; the functional complex of data of polybasic algebraic structure of composition

model; the functional dependences and the relation forming system of axioms of composition model. Graphic elements on which it is displayed components of subsystems are shown; structurally functional chart of component interaction of levels of control; the structure of composition model of function-oriented information resources provided by the table of difficult structured data in not the first normal form; structural logic circuit of a method of creation of composition model of function-oriented information resources.

Keywords: management informational system; composition model; functionally oriented informational resources.

DOI: 10.15514/ISPRAS-2016-28(2)-17

For citation: Chucklyaev I. Composition model and method of creation of functionally-oriented information resources. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 259-270 (in Russian). DOI: 10.15514/ISPRAS-2016-28(2)-17

References

- [1]. Rukovodyashiy document Gostehkomissii Russian Federation [The leading document of Gostekhcommissia of Russia]. «Zashita ot nesancionorovannogo dostupa k informacii. Termini I opredeleniya» [Protection against illegal access to information. Terms and determination], 1992. 5 p. (in Russian).
- [2]. A.V. Morozov, V.V. Borisov, I.I. Chucklyaev. [Computing systems: theoretical generalization, development, practical results]. *Monografiya*. Smolensk, Air defense of Military academy, 2013. 448 p. (in Russian).
- [3]. I.I. Chucklyaev. [Theoretical generalization of data domain "information security". Tendencies of development of methods and means]. Moscow, Konzern «Systemprom» Publ., 2015, volume 1(6)-2015, pp. 471-486 (in Russian).
- [4]. I.I. Chucklyaev. [Management information system in the conditions of the multi-level function-oriented information conflict and suppression]. Smolensk, Air defense of Military academy, *Vestnik PVO* [Messenger army air defense], 2015, volume 13, pp. 183-189 (in Russian).
- [5]. I.I. Chucklyaev. [Method and models of risk management of security in management information systems]. Moscow, Konzern «Morinformsistema-Agat» Publ., 2015, volume 9, pp. 14-33 (in Russian).
- [6]. E.P. Emelchenkov. [Databases. The modern approach]. *Monografiya*. Smolensk, Air defense of Military academy, 2010. 59 p. (in Russian).
- [7]. E.P. Emelchenkov, Yu.S. Malein. [About the functional approach in the database theory]. Smolensk, SGPU, VINITI [VINITI], 1984, volume 6046-84, 29 p. (in Russian).
- [8]. Ye. Yemelchenkov, M. Tsalenko. Functional dependencies in hierarchical Structures of Data. *Lect. notes in Compute Science*. Berlin, 1991. V. 495. P. 258-275.
- [9]. E.P. Emelchenkov, N.A. Levin. [About simulation of difficult data domains]. *Problemi I metodi informatiki. II Nauchnaya sessia IPI RAN. Tez. dokl.* Moscow, IPI RAN Publ., 2005, pp. 89-91 (in Russian).