UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DIOGO RAPHAEL CRAVO

Module Integration using Graph Grammars (MIGRATE)

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Leila Ribeiro

Porto Alegre
August 2021

CIP — CATALOGING-IN-PUBLICATION

Cravo, Diogo Raphael

Module Integration using Graph Grammars (MIGRATE) / Diogo Raphael Cravo. – Porto Alegre: PPGC da UFRGS, 2021.

112 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Leila Ribeiro.

1. Graph grammar. 2. Software integration. 3. Verification tool. I. Ribeiro, Leila. II. Título.

"Take risks. Ask big questions.
Don't be afraid to make mistakes;
if you don't make mistakes,
you're not reaching far enough"

— DAVID PACKARD

AGRADECIMENTOS

Agradeço à Professora Leila por todo o apoio, pelas muitas e longas reuniões de orientação em que discutimos esta dissertação. Sem sua supervisão este trabalho não seria possível.

Agradeço também ao Professor Rodrigo, que cedeu seu tempo para explicar conceitos sobre gramáticas de grafos e discutir dificuldades no procedimento de verificação. Também aos colegas do grupo de pesquisa, ao Arthur pela ajuda com o Verigraph.

Agradeço aos colegas de empresa pelas discussões e também pela compreensão em todas reuniões de que não pude participar.

Agradeço à Hellena por todo apoio neste último ano. Seus conselhos me ajudaram a persistir!

Por fim, agradeço especialmente à minha família e a minha companheira Nick por todo amor, apoio e compreensão.

ABSTRACT

Software, be it desktop, mobile or web, is becoming more and more connected. Software development is also becoming more connected with ecosystems comprised of networks of millions of packages. Engineering software today involves writing code that weaves together libraries, services and applications. Such processes are under constant changes due to both internal requests (e.g. new features) or external demands (e.g. dependency updates). Avoiding integration bugs in this scenario can be quite a challenge regardless of common strategies such as testing and versioning. We studied graph grammars to find a set of grammars (verification grammars) that represent how software modules integrate and leveraged existing graph grammar analysis, specifically critical pair analysis, to point out possible integration problems in such grammars automatically. Furthermore, we created a formalism (module nets) to represent how software modules share information and leveraged graph grammars, by the fact that they can be proven to have functional behavior (confluence), to translate instances of module nets to verification grammars, enabling developers to create and modify module nets and then have warnings concerning integration problems automatically generated. We summarized this process in a framework we call module integration using graph grammars (MIGRATE), which we illustrate in this work through a case study with a fictitious search engine for research articles. Our approach demonstrates how to leverage critical pair analysis of graph grammars to automatically uncover a few integration bugs. It also serves as a pathway for future research exercising other graph grammar analyses to full extent.

Keywords: Graph grammar. software integration. verification tool.

Integração de módulos utilizando gramáticas de grafos (MIGRA)

RESUMO

Software, seja desktop, mobile ou web, está se tornando mais e mais conectado. Desenvolvimento de software também está se tornando mais conectado com ecossistemas feitos de redes de milhões de pacotes. Construir software hoje corresponde a escrever código que integra bibliotecas, serviços e aplicações. Essas redes estão sob mudanças constantes devido a necessidades internas (e.g. novas funcionalidades) ou demandas externas (e.g. atualização de dependências). Evitar defeitos de integração neste cenário pode ser um grande desafio, apesar de estratégias como teste e versionamento. Nós estudamos gramáticas de grafos para encontrar um conjunto de gramáticas (gramáticas de verificação) que representam como módulos de software se integram e aproveitamos análises de gramáticas de grafos existentes, especificamente análise de pares críticos, para apontar automaticamente possíveis problemas de integração nessas gramáticas. Além disso, nós criamos um formalismo (redes de módulos) que representa de que forma módulos compartilham informação e aproveitamos gramáticas de grafos, pelo fato de que pode-se provar seu comportamento funcional (confluência), para traduzir instâncias de redes de módulos para gramáticas de verificação, possibilitando que desenvolvedores criem e modifiquem redes de módulos e então gerem automaticamente avisos que dizem respeito a problemas de integração. Nós resumimos este processo em um framework que chamamos de integração de módulos utilizando gramáticas de grafos (MIGRATE), o qual ilustramos neste trabalho através de um estudo de caso com um motor de busca por artigos de pesquisa. Nossa abordagem demonstra como aproveitar análise de pares críticos de gramáticas de grafos para descobrir automaticamente alguns defeitos de integração. Ela também serve como caminho para pesquisas futuras exercitando todo o potencial de análises de gramáticas de grafos.

Palavras-chave: Gramática de Grafos, integração de software, ferramenta de verificação.

LIST OF ABBREVIATIONS AND ACRONYMS

MIGRATE	Module Integration using Graph Grammars
TAGG	Typed Attributed Graph Grammar
AGG	The Attributed Graph Grammar System
NPM	Node Package Manager
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Software is becoming more and more connected. Desktop applications, which used to save data to disk, now synchronize with the cloud and allow simultaneous editing by different users. Smartphone applications connect to backend services that store data and carry out requests. Not to mention the web itself, a huge network of sites, which are connected, by design, to each other and to services bearing all the business logic.

Software development is also becoming more connected. Each programming language brings with itself an ecosystem of package managers full of thousands of libraries, sometimes even millions (DECAN; MENS; GROSJEAN, 2019). Such libraries usually make use of one another, giving rise to vast networks of dependencies. Engineering software today means to write code that weaves together libraries, services and applications.

There are many reasons to leverage software as a dependency rather than develop software from scratch. The first is to avoid reinventing the wheel: if someone has already been through the development and testing, there is no reason to make that effort again. Also, although developers know their software entirely, the development of some libraries may require domain expertise, which is too different from the project goals, such as research algorithms, or even database drivers and cloud development kits. Finally, constraints such as time and budget shrink how much work a team can perform, forcing teams to concentrate on their business logic and leverage everything they can.

The inherent limitation on how much work a team can perform also forces organizations to scale in order to achieve big goals. In such context each team is responsible for a set of assets, and reuse is strongly encouraged within the organization to save up resources. An evidence of this approach to developing software is the growing interest in microsservices by the software engineering community (FOWLER, 2014; SOLDANI; TAMBURRI; HEUVEL, 2018).

Software can be leveraged in many ways. Control over integration is greatest when the dependency owner and the consumer are the same, because not only dependency code is known, but also updates can be controlled to keep integration. As dependencies attract more consumers, these consumers will have less control over updates, but they may still have access to code, as it happens with open source software. Integration is harder when consumers neither own nor have access to dependency code, in which case consumers have to rely fully on documentation and dependency updates may break the integration to consumer software.

Consumers can choose when they update libraries, but they are rarely given the chance to choose when to update dependency services. The least control consumers have over integration is when their dependency is a service they do not develop themselves, because this service can be updated independently of consumer software, possibly introducing integration faults.

Integration faults can arise for multiple reasons (AUÉ et al., 2018; MOSTAFA; RODRIGUEZ; WANG, 2017). One example is when a consumer uses a API that is removed from a dependency in an update. Another common example is changes to method signatures in dependency updates. More subtle integration faults are related to changes in how dependencies handle data. Even if an update keeps the APIs of a dependency, it may change its input/output relation, which can cause all sorts of trouble when it is not integrated properly.

There are many approaches to preventing integration faults. Semantic versioning (PRESTON-WERNER, 2020) aims to signaling when a change to a dependency is considered "breaking", warning developers that updating a dependency to that version may require changes to integration code. Automated tests written by developers can also uncover integration faults (FOWLER, 2018). Additionally, some languages have tooling that prevents compilation and bundling of code when it does not integrate well to its dependencies (ORACLE, 2020).

Despite all of the approaches in place to prevent integration faults, some of these faults are still subtle and hard to catch. As an example, let us consider the NodeJS library axios, a very useful and popular HTTP client. Release 0.19.2 of axios had a configuration, called *maxContentLength*, which served both as maximum response length as well as maximum request length[1], the latter was relayed to another library called follow-requests. The confusion of responsibilities assigned to the same parameter led axios contributors to create a new parameter, *maxBodyLength*[2], which was relayed to follow-requests, while *maxContentLength* retained its original purpose of being the maximum response length. This change was release in version 0.20.0 of axios.

An unsuspecting developer when updating axios from version 0.19.2 (or older) to version 0.20.0 (or newer) may overlook this change, which can lead to trouble, because the newer version will constrain maximum upload sizes to its default *maxBodyLength*, which the previous version would not do.

To brake integration faults before they make it into production, many researchers

[1]https://github.com/axios/axios/issues/2696
[2]https://github.com/axios/axios/pull/2781

are proposing new approaches. This work is dedicated to proposing a method and tool to automatically uncover some integration faults.

1.1 A verification tool for software integration

We propose a verification method and tool that can automatically uncover faults arising from the integration of modules. Our method uses a novel formalism, which we call *module net*, to model the data flow between and within modules. Module nets are obtained by code inspection, translating programming language to module net diagrams. Verification happens by first translating a module net to a graph grammar, which is an automatic procedure, and then performing a series of analyses over the verification grammar. The outcome of our verification procedure is a set of warnings to users highlighting which integrations deserve their attention.

As previously mentioned, software is connected and built not only on top of libraries, but also on top of services, and software integration happens at both levels. For this reason, we consider modules as any piece of software that handles data, which can be anything: application, library or service.

Although we recognize control flow plays an important role in the behavior of software, such as conditions and loops, our method does not address control flow. Adding control flow would incur an even bigger problem to solve, whereas our goal in this work is to show a prototype. Control flow can still be addressed in future work. Additionally, because the verification is carried out at the level of module nets, we ignore language specifics, which can be addressed when performing the extraction of module nets from programming languages, and it is also left for future work.

The verification method we propose can be used in many ways. When given a module net, it is able to output warnings regarding that specific module net. A developer can then make changes to the module net and perform the verification again to assess whether integration faults are resolved. Alternatively, that same developer can choose to apply changes directly to code and then have this code automatically translated into module net to perform the verification. Library and service developers can use module nets to check that the changes they make to their code do not impact consumers.

Moreover the verification can be used to determine the correct version that should be assigned to a module after it is updated, by checking the integration of the updated module to a fixture module that uses all of its functionalities. Additionally, such checks could be used to rank libraries according to stability, in terms of how many warnings are present. Also, because the verification is concerned with how data flows between modules, it could be used to point out resources which are shared by many services and factor responsibilities, achieving lower coupling. Versioning, ranking and coupling functionalities are not covered in this work.

1.2 Main contribution

Our goal with this research is to eventually reach a point where we can highlight integration faults automatically and independently of technology. This work provides the first steps in this direction. These are the main contributions of this work:

- A formalism to model how data flows between software modules, which we call module net;
- A method to verify and highlight integration faults arising from the data flow of module nets;
- A prototype tool that implements the method above and its evaluation in a case study.

1.3 Outline

Chapter 2 addresses the problem of integrating software modules in different ecosystems.

Chapter 3 is a brief introduction to typed attributes graph grammars, which are used extensively in this work.

Chapter 4 describes the module integration framework that we want to build, MIGRATE, with its extraction, translation and verification procedures and all artifacts used in the process.

Chapter 5 defines module nets, which are a formalism to describe software integration.

Chapter 6 presents a procedure to translate module nets into graph grammars using graph grammar derivation, and proof of confluence and discussion of well-definedness of such translation.

Chapter 7 shows how to use critical pairs generation to find integration issues and how to report such issues to developers in the form of warnings.

Chapter 8 condenses the work done until here by showing a case study with three versions of a fictitious search engine for research articles.

Chapter 9 contains related work and tools to solve this problem for libraries as well as services.

Chapter 10 provides concluding remarks.

In the appendix we have included the entire graph transformation system used to translate module nets into verification grammars, which we presented in Chapter 6.

2 PROBLEM

Software is built on top of other software. Users integrate to dependencies, which can be classes, libraries or services. Dependencies are developed either by users themselves, by others that belong to the same organization or by third-parties. When a dependency is updated, it may break its integration to some or even all of its users. The problem of integration is part of software engineering and although many tactics have been devised to tackle it, there is just no way to compute whether two arbitrary functions are the same, as observed in (RAEMAEKERS; DEURSEN; VISSER, 2014).

2.1 Compatibility matters

There is a multitude of software ecosystems out there with all kinds of package managers such as Debian packages for Linux, Chocolatey packages for Windows, plugins for Eclipse, etc. For programming languages we have[1]:

- JavaScript/NodeJS and NPM[2] with over one million packages
- R and CRAN[3] with 17 thousand packages
- Java and Maven with over six million indexed artifacts
- Ruby and RubyGems with more than 164 thousand gems
- Python and PyPI/pip with over 288 thousand projects
- .NET and NuGET with more than 242 thousand packages
- PHP and Packagist with almost 300 thousand packages
- Perl and CPAN[4] with almost 200 thousand Perl modules
- Rust and Crago with more than 54 thousand crates
- and many others such as Go, Haskell, Smalltalk, etc. See Libraries.io[5] for 37 package managers and almost four million packages.

Package ecosystems have been extensively studied. One study shows that most packages in NPM, CRAN and RubyGems directly depend on other packages, and there are even more transitive dependencies (DECAN; MENS; CLAES, 2017). Additionally

[1]Data extracted on 2021 from official websites of each package manager.
[2]Node Package Manager
[3]The Comprehensive R Archive Network
[4]The Comprehensive Perl Archive Network
[5]https://libraries.io/

the amount of packages that have the most dependencies seems to grow on NPM, and these packages have a high impact on the ecosystem being dependencies of up to 30% of the entire ecosystem (DECAN; MENS; CLAES, 2017). Decan *et al.* later extended this study to Cargo, CPAN, NuGET and Packagist on top of the ecosystems previously mentioned, finding that less than 17% of packages are dependencies of more than 80% packages (DECAN; MENS; GROSJEAN, 2019). Breaking changes to such fundamental packages can have catastrophic effects.

There not seems to be recent analogous studies of the size and structure of service ecosystems. Services are agnostic to programming languages and can expose interfaces of all kinds, including simple remote procedure call (RPC) up to more elaborate representational state transfer (REST) architectures. Recent trends in service interfaces are gRPC and graphQL. Services exchange messages through some protocol, usually HTTP. A common portal for public access services is ProgrammableWeb[6], which lists almost 24 thousand services. Postman API Network[7] also contains over 1700 APIs and 21 thousand Postman collections.

2.2 Compatibility often breaks

Compatibility often breaks. A study with more than 22 thousand Java libraries released prior to 2011 has shown that roughly 35% of minor releases and 23% of patch releases contained breaking changes (RAEMAEKERS; DEURSEN; VISSER, 2014). This is in contrast to semantic versioning principles, which restrict patch and minor updates to non-breaking changes. Perhaps what is more surprising is that breaking changes were found not to influence the adoption time of new updates (RAEMAEKERS; DEURSEN; VISSER, 2014). A more recent study with 317 popular Java libraries has shown that roughly 28% of changes were considered breaking, and the larger a library gets, the more breaking changes it introduces (XAVIER et al., 2017). The authors also studied 260 thousand clients of those libraries, estimating that in the worst case scenario, on median, roughly 2.5%, but up to 100% in rare cases, were affected by breaking changes (XAVIER et al., 2017). Developers are compelled to update their libraries in an effort to escape vulnerabilities, but the very fixes may also contain breaking changes. Wang *et al.*, when studying the update risk of Java libraries have found that an impressive 35% of studied

[6]https://www.programmableweb.com/

[7]https://www.postman.com/explore

libraries, more than 4200 libraries, had more than 300 deleted APIs between a vulnerable version and its corresponding fix (WANG et al., 2020).

Fear of breaking changes also scares developers away from updates. This is true in the mobile development environment too, where just 13% of sampled apps update dependencies constantly and 63% never update them (SALZA et al., 2018). The longer developers take to update, the more they lag behind. This is known as "technical lag"[8] and has been studied in the NPM ecosystem showing that out of 120 thousand packages, almost 1,5 million releases and 8 million dependencies roughly 25% of dependencies and 40% of releases present technical lag, with an average of 7 to 9 months (DECAN; MENS; CONSTANTINOU, 2018). Decan *et al.* highlight that patch and minor dependency updates are the most affected by technical lag[9], corroborating the findings of Raemarkers *et al.*, which is that developers are and should be afraid of updates, even though these studies were carried out in different ecosystems. Wang *et al.* approach this phenomenon with various metrics, such as "usage outdatedness", "update intensity" and "update delay". They find that very few projects keep all their libraries up-to-date and more than 50% of projects take longer than 60 days to update dependencies (WANG et al., 2020).

Integration issues are not restricted to libraries, services also suffer a great deal of compatibility problems. Aué *et al.* have studied millions of faults logged by a large scale web service in the payments business and came up with eleven categories for those faults (AUÉ et al., 2018). While some errors can be attributed to end users, such as providing a maxed out credit card, others are due to the programming that integrates clients, service and third-parties. Even microservices, a relatively new architectural pattern known for achieving loosely coupled modules, are affected by compatibility problems, and API versioning and contracts are mentioned in 13 of 51 grey literature papers (SOLDANI; TAMBURRI; HEUVEL, 2018).

2.3 Why compatibility breaks

As mentioned in the introduction, there are different levels of complexity when handling module integration. This complexity can be classified in three dimensions: Users, Influence and Distribution. In the following, we discuss these three dimensions,

[8]A formal definition can be found in referenced articles.

[9]Although Decan *et al.* find that patch and minor are the most affected by technical lag, they judge that such updates should be automatic, contrary to Raemarkers *et al.* findings that patch and minor updates are often not backwards compatible, when they should be.

considering easier to more difficult integrations in each one.

i) Users

- Single user, such as a module developed for a single project
- Few users, such as modules shared within an organization across projects
- Many users, such as open source modules that integrate to many modules

With a single or even few users, it is often possible for a module to verify its integration to users prior to releasing changes, but the more users a module has, the tighter these users integrate to dependencies, increasing the impact of breaking changes. On the other end, breaking change aversion encumbers dependency developers more and more to the point of stagnation. This is what happened to the NPM request module, which has gone into maintenance mode[10] due to the tremendous amount of users, making it impossible to publish breaking changes without causing huge trouble.

ii) Influence

- Users own code or can propose changes to code
- Users can view code
- Users only have access to documentation or live/compiled software

Integration is a difficult task, which requires insights on how each of the integrated components work. The ability to see dependency code comes in handy many times and can make a difference to avoid bugs. Even better is when users can propose/make changes to code, as it enables development of better interfaces.

iii) Distribution

- Released as single entity, such as a class
- Released coupled with consumer, such as a library
- Released independently for a subset of users, such as backend services
- Released independently, such as a generally available service

We include distribution because it affects the time a user has to adapt to changes. If the dependency is part of a project, either a class or a library, its update can be coupled with a project release, thus meeting project schedule. Of course, libraries have their own release schedule and when security updates are released, users are required to update as soon as possible. Services are the hardest to integrate because they often are not even aware of their users' schedules, and a service release can immediately break users.

[10] https://github.com/request/request/issues/3142

2.3.1 Weighing decisions

When faced with the choice of whether or not to break compatibility, developers have to weigh the costs of decaying code and the benefits of opting to break. Keeping compatibility can incur maintenance costs in the form of more code, interfaces, releases and branches to maintain as well as holding-off new features, whereas opting to break compatibility can incur costs of providing support for broken clients and communicating changes to users (BOGART et al., 2016; BRITO et al., 2018b). Keeping compatibility has the benefit of preventing cascading breaks, but breaking compatibility can have many benefits such as addressing technical debt in terms of style, deprecation, refactorings, etc., performance improvements and bug fixes and even add new features (BOGART et al., 2016; BRITO et al., 2018b).

2.3.2 Ecosystem pressure

Whether breaking compatibility is acceptable or not has a lot to do with the ecosystem the software inhabits. In a survey with 28 developers from three different ecosystems, researchers have found that Eclipse developers value backwards compatibility, whereas NPM developers make use of versioning to deliver breaking code due to the pressure of a fast moving environment and finally CRAN developers have easy installations at the cost of fast reaction to updates (BOGART et al., 2016). To support these values, Eclipse platform plans yearly releases and only then are developers compelled to make changes, NPM allows dependency versions to be pinned and thus dependency updates are decoupled from package releases, and R/CRAN requires users to update their dependencies shortly after new releases and does not allow version pinning as in NPM (BOGART et al., 2016).

These ecosystem values also have to do with the functionality provided by the tooling. Whereas NPM is capable of installing various releases of a same package in the same project, thus enabling package maintainers to delay updates, CRAN require that only the latest release of a package is installed, forcing timely updates (DECAN; MENS; CLAES, 2017).

2.4 How to avoid breaking compatibility

Integration is a problem pertaining to two actors: dependency users and dependency developers. While users will apply tactics, run tools, implement tests, manage dependency versions and avoid deprecated interfaces, developers will follow best practices, apply versioning guidelines and deprecation and run regression tests. All of this to ensure the healthy integration of dependencies.

2.4.1 Project tactics

A common strategy adopted by users to avoid breaking compatibility is to keep dependencies to a minimum and select dependencies they trust (BOGART et al., 2016). Despite this strategy, trivial packages are very popular for example in the NPM ecosystem. Trivial packages are packages with low cyclomatic complexity and few LOC and should be considered with care due to lack of tests and high dependency count (ABDALKAREEM et al., 2017). They later extended this study and included the PyPI ecosystem with similar results (ABDALKAREEM et al., 2020).

For services, common advice is to relax the assumptions on data received, ignoring any extra information sent and avoiding any kinds of format enforcing, unless it is absolutely necessary. By ignoring extra information and avoiding format checks, services ensure longer compatibility with clients that use old versions of the API. This pattern is known as the Tolerant Reader (FOWLER, 2011; LüBKE et al., 2019).

2.4.2 Programming language support

From the library developers perspective, language specifications can provide means to avoid breaking compatibility. Java specification includes the notion of "Binary Compatibility", which defines what kinds of changes are compatible, such as adding modifiers, inserting new types, and many others (ORACLE, 2020).

2.4.3 Versioning

Versioning schemes signal to users the degree of changes and enable users to specify rules for automatic updates of their dependencies. Perhaps the most popular scheme in practice is Semantic Versioning (PRESTON-WERNER, 2020), which uses the form "major.minor.patch"[11], where major releases increase the major version (and reset minor and patch versions) and contain breaking changes, while minor and patch releases keep major versions and do not contain breaking changes. Another kind of versioning scheme is the Eclipse versioning[12], also known as OSGi semantic versioning, which is very much like semantic versioning, except for the inclusion of a fourth version number to address builds.

Semantic versioning assigns a special meaning to versions of major 0. Such versions are considered in "initial development" and are permissive to API changes, being considered unstable (PRESTON-WERNER, 2020). Despite this meaning, many popular NPM modules are released with major version 0 and are nonetheless widely adopted by the community. Axios is one such example, a HTTP client for JavaScript/NPM with almost 15 million weekly downloads, that still has a major version 0 and there is no forecast as to when 1.0.0 will be released[13].

The widespread use of major 0 can be a concern precisely because it hinders the release of security fixes for older versions. Additionally, unstable APIs even prevent the automatic update of such fixes when affected modules are transitive dependencies, because minor version increases with major 0 contain breaking changes, which delays updates even more and exposes the ecosystem to vulnerabilities. As we write this text, this is the current state of axios[14].

Versioning is also used in services, where it can be found in addresses, HTTP headers or bodies (LüBKE et al., 2019). Applying versioning in services gives clients a buffer to transition to new APIs, which otherwise would have to be done coupled with service deploy. This of course comes at the expense of service developers who then need to maintain multiple versions of an API running. Operational costs constrain the amount of versions a service can expose and usually only major versions are exposed, with at most two versions running in parallel (LüBKE et al., 2019).

[11]Semantic versioning is in fact a little bit more complex making room for prerelease versions and initial development phase (version 0).

[12]https://wiki.eclipse.org/Version_Numbering

[13]https://github.com/axios/axios/issues/1333

[14]https://github.com/axios/axios/issues/3407

In spite of the available schemes, but perhaps by mistake, many projects release breaking changes without signaling so (RAEMAEKERS; DEURSEN; VISSER, 2014).

2.4.4 Tests

Tests are yet another approach to ensure compatibility for both users and developers. There are all kinds of tests in the literature, but specifically integration tests address the problem of compatibility. In his blog, Fowler says that integration tests are meant to show whether software modules work as expected when brought together (FOWLER, 2018) and splits integration tests into two categories: "narrow integration tests" and "broad integration tests" (FOWLER, 2018). While the former requires some kind of mock to replace the actual module integrated and thus isolate tested code to just a single module, the latter tests all modules working together. The blog advocates in favor of "narrow integration tests", which according to Fowler are more effective because it spares the trouble of building an entire test environment.

The issue with unit tests is that they mock integration points, effectively shadowing precisely what we want to test here. Integration tests can help us find such bugs, but they are often very expensive to write because they require such test environments as we have mentioned.

2.4.5 Deprecation

Deprecation is a valid strategy to signal to users that a feature is not to be used anymore and that it may be removed in the future, however deprecation is seldom used properly (ZHOU; WALKER, 2016). Deprecation is also practiced in services and coupled with "aggressive obsolescence", essentially enforcing deprecation with deadlines (LüBKE et al., 2019).

2.5 Our contribution

Software modules essentially share data. Users send data to dependencies, which perform computations, carry out side effects and provide back some data. Even side effects are the exchange of data with third-party modules. When a dependency is updated and it is not compatible anymore to a user, the dependency either still shares the same kind of data and computations or not. In case it does, then fixing compatibility is just a matter of rewiring the programming to the user so that the right type or the right method or the right name, etc., is used.

When the dependency changes the way it handles data, then this change can go unnoticed and cause trouble when it gets to production. Even worse, sometimes the change is so fundamental there is no amount of rewiring that will bring back the compatibility it had before to a user: it just does not perform the same functions that user needs anymore. This dissertation is concerned with finding those kinds of compatibility issues, when compatibility breaks due to deeper problems than syntactic changes.

Our work can be divided into three phases: (i) transformation of code into module nets (model extraction), (ii) translation of module nets into graph grammars with graph grammars as the translation engine and (iii) verification of module nets as graph grammars.

3 TYPED ATTRIBUTED GRAPH GRAMMARS

The formalism of Graph Grammars (or Graph Transformations) is based on defining states of a system as graphs and state changes as rules that transform these graphs. In this chapter we review informally the main concepts of the area needed for our work. Graph grammars have been studied for almost six decades and there are many approaches backed by various researchers all around the world (EHRIG, 1979; CORRADINI et al., 1997; EHRIG et al., 1999). We use the algebraic approach to graph grammars, which bases all definitions on category theory. More concretely, we use the DPO-approach (EHRIG et al., 1997; EHRIG, 2006) and use typed attributed graph grammars.

Graph-based formal description techniques are a friendly means of explaining complex situations in a compact and understandable way. Graph grammars are a generalization of Chomsky grammars from strings to graphs suitable for the specification of distributed, asynchronous and concurrent systems. The basic notions of this formalism are: states are represented by graphs and possible state changes are modeled by rules, where the left- and right-hand sides are graphs. Graph rules are used to capture the dynamical aspects of the systems. That is, from the initial state of the system (the initial graph), the application of rules successively changes the system state.

The definitions presented in this chapter are informational, they are simplified and presented mostly in terms of examples, as these definitions are classical definitions of typed attributed graph grammars and can be found in the literature. All of the definitions in this section come from (EHRIG, 2006). Throughout the chapter, each time we define a concept, we will mark it in boldface.

3.1 Nodes and Arrows

A **graph** is a tuple (V_G, E_G, s_G, t_G) with sets of graph vertices V_G and edges E_G, and functions source $s_G : E_G \to V_G$ and target $t_G : E_G \to V_G$. Graphs can be augmented with data sets, forming tuples $(V_G, V_D, E_G, E_{VD}, E_{ED}, (s_i, t_i)_{i \in \{G,VD,ED\}})$ which we call **e-graphs**[1]. The set V_D contains the values that may be used as attributes of vertices and edges (this set is potentially infinite, containing, for example all natural numbers, strings, etc.). Sets E_{VD} and E_{ED} denote connections that assign values (of V_D) to vertices (of V_G) and edges (of E_G), respectively. Figure 3.1 shows an example e-graph E_1 and Figure

[1] "e" is for extended

3.4a shows this same graph with visual notation.

An algebra is a mathematical structure containing sets (called carrier sets) and functions over these sets (called operations). Algebras can be specified using algebraic specifications, that are composed of a set of sorts (set names), operations (signatures of functions) and equations (to specify the behaviour of the functions). An **attributed graph** is a pair (G, A) of an e-graph and an algebra, where V_D is the disjoint union of all carrier sets of the algebra. The advantage of using algebras to obtain values of V_D is that we can specify which values belong to this set. Moreover, the use of term algebras allows to use variable and terms as attributes, which is particularly useful to describe general rules.

Figure 3.1: Example e-graph E_1 in mathematical notation where $\uplus$ is the disjoint union of sets, Nat is the set of natural numbers and Bool is the set of boolean values

$$\begin{gathered}
E_1 = (V_G, V_D, E_G, E_{VD}, E_{ED}, (s_i, t_i)_{i\in\{G,VD,ED\}}) \\
V_G = \{T_1, T_2\} \\
V_D = Nat \uplus Bool \\
E_G = \{t_1, t_2\} \\
E_{VD} = \{nt_1, nt_2\} \\
E_{ED} = \{\} \\
s_G = \{t_1 \mapsto T_1, t_2 \mapsto T_2\}, t_G = \{t_1 \mapsto T_2, t_2 \mapsto T_2\} \\
s_{VD} = \{nt_1 \mapsto T_1, nt_2 \mapsto T_2\}, t_{VD} = \{nt_1 \mapsto 1, nt_2 \mapsto 2\} \\
s_{ED} = \{\}, t_{ED} = \{\}
\end{gathered}$$

Figure 3.2: Example algebra A_1 corresponding to a signature Σ_1

$$\begin{gathered}
\Sigma_1 = (S, OP) \\
S = \{S_1, S_2\} \\
OP = \{op_1, op_2\} \\
op_1 :\rightarrow S_1 \\
op_2 : S_1 \rightarrow S_2 \\
A_1 = (\{Nat, Bool\}, \{0, iszero\})
\end{gathered}$$

We can use all graph definitions provided so far to build categories, where graphs, e-graphs or attributed graphs are the objects and the morphisms are graph morphisms. A **graph morphism** $m : G_1 \rightarrow G_2$ is a tuple (m_V, m_E) that maps nodes with $m_V : V_{G_1} \rightarrow V_{G_2}$ and edges with $m_E : E_{G_1} \rightarrow E_{G_2}$ from a graph to another preserving edge source and target along the morphism. For e-graphs, we have morphisms $m_i, i \in \{V_G, V_D, E_G, E_{VD}, E_{ED}\}$ as depicted in Figure 3.3. For attributed graphs, morphisms require morphisms between the algebras of the two attributed graphs as well.

Figure 3.3: An e-graph morphism (EHRIG, 2006)

$$\begin{tikzcd}
E^1_{ED} \arrow[rr, "target^1_{ED}"] \arrow[dr, "source^1_{ED}"'] \arrow[drrrr, dashed, "m_{E_{ED}}"] & & V^1_D \arrow[drrrr, dashed, "m_{V_D}"] & & E^1_{VD} \arrow[ll, "target^1_{VD}"'] \arrow[dl, "source^1_{VD}"] \arrow[drrrr, dashed, "m_{E_{VD}}"] & & & & \\
& E^1_G \arrow[rr, bend left, "source^1_G"] \arrow[rr, bend right, "target^1_G"'] \arrow[drrrr, dashed, "m_{E_G}"] & & V^1_G \arrow[drrrr, dashed, "m_{V_G}"] & E^2_{ED} \arrow[rr, "target^2_{ED}"] \arrow[dr, "source^2_{ED}"'] & & V^2_D & & E^2_{VD} \arrow[ll, "target^2_{VD}"'] \arrow[dl, "source^2_{VD}"] \\
& & & & & E^2_G \arrow[rr, bend left, "source^2_G"] \arrow[rr, bend right, "target^2_G"'] & & V^2_G &
\end{tikzcd}$$

Figure 3.4: Example typed attributed graph TAG_1, where the morphism from AG_1 to T_1 is indicated by using in AG_1 the same names as in T_1 with indices (values of Nat are an exception)

$$\begin{tikzcd}
& 1 \\
T_1 \arrow[ur, dotted, "nt_1"] \arrow[dd, "t_1"'] & \\
& 2 \\
T_2 \arrow[ur, dotted, "nt_2"] \arrow[loop right, "t_2"] &
\end{tikzcd}$$

(a) Attributed graph AG_1.

$$\begin{tikzcd}
& Nat \\
T \arrow[ur, dotted, "nt"] \arrow[loop right, "t"] &
\end{tikzcd}$$

(b) Type graph T_1

A **typed attributed graph** is a triple (G, m, T) where G and T are attributed graphs and $m : G \to T$ is a graph morphism, T is called type graph (because it defines all types of vertices and edges of a graph grammar). Figure 3.4 shows the typed attributed graph $TAG_1 = (AG_1, m_1, T_1)$. From here on, whenever we refer to graphs, we mean typed attributed graphs.

3.2 Grammars

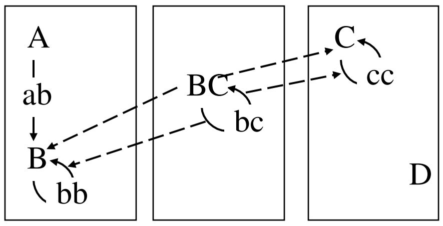
A **graph transformation rule** (or **production**) p is a pair of graph morphisms l and r and graphs L, G and R as depicted in Figure 3.5, where G is usually called the **gluing graph**. Graph L denotes the items that must be present for the rule to be applied, graph R the ones that will be present after rule application (including preserved and created items), and G represents the preserved items. Morphisms l and r are used to connect the preserved items from L to R, using the gluing graph G. Figure 3.6 shows an example graph transformation rule, we omit sets V_D, E_{VD} and E_{ED} as well as typing for conciseness. A graph transformation rule can be applied to a graph if there is a **match**, that is, a morphism from the left-hand side of the rule L to the graph, such application is usually called a **graph transformation**. Figure 3.7 depicts graph transformation $L_i \overset{p,m}{\Rightarrow} R_i$, which is exemplified by Figure 3.8. A graph transformation rule may additionally have **negative application conditions**, which are situations (graphs) that prevent the application of the rule. A procedure to apply a transformation rule follows (see Figure 3.7 for the names of graphs):

1. Choose graph L_i and production p
2. Choose a morphism m that matches graph L to L_i
3. The rule is not applicable if it has a negative application condition n with $n : L \to N$ and there is a morphism $m_N : N \to L_i$. In this case, application is aborted and the next steps are not executed.
4. Given match m, construct a gluing graph G_i (also called context graph) such that PO_1 is a pushout. This pushout construction basically deletes from L_i all items that are to be removed by the rule. If this construction is not possible (due to conflicts between preserving and deleting items or trying to delete vertices without deleting corresponding edges), rule application is aborted.
5. Finally, pushout PO_2 is construted, giving rise to the resulting graph R_i. This pushout adds to G_i all items created by the rule.

Figure 3.5: Definition of graph transformation rule.

$$L \xleftarrow{l} G \xrightarrow{r} R$$

Figure 3.6: Example of a graph transformation rule P_1.

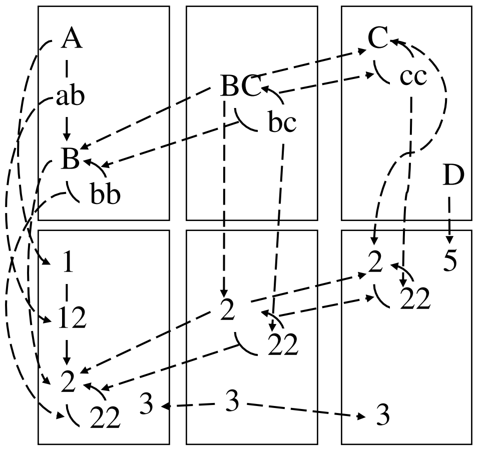


A **graph transformation system** is a set of graph transformation rules. A **graph grammar** is a graph transformation system with an initial graph.

Figure 3.7: Application of a graph transformation rule.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "PO_1"] & G \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "PO_2"] & R \arrow[d] \\
L_i & G_i \arrow[l] \arrow[r] & R_i
\end{tikzcd}$$

Figure 3.8: Example of a graph transformation rule application, which preserves, deletes and creates nodes and edges.



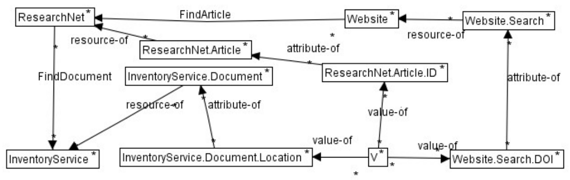
3.3 Properties

In this dissertation we are interested in critical pairs, which are used in the verification process, and also in the property of confluence, which is fundamental for correct model transformations. A confluent graph transformation system presents functional behavior, in the sense that given an initial graph it will always produce the same final graph (EHRIG, 2006). In order to define critical pairs and confluence, there are a few properties of grammars that we will have to review.

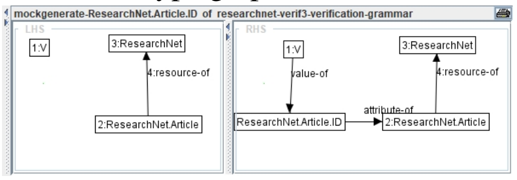
An ordered pair of transformations is **parallel dependent** if the first transformation disables the second. Similarly, such pair is **sequentially dependent** if the first transformation enables the second. More formally, given transformations $t_K^{p_1} = K \overset{p_1,m_1}{\Rightarrow} R_1$ and $t_K^{p_2} = K \overset{p_2,m_2}{\Rightarrow} R_2$, we say $(t_K^{p_1}, t_K^{p_2})$ is parallel dependent if $t_K^{p_1}$ disables $t_K^{p_2}$. Given transformations $t_K^{p_1} = L_1 \overset{p_1,m_1}{\Rightarrow} K$ and $t_K^{p_2} = K \overset{p_2,m_2}{\Rightarrow} R_2$, we say $(t_K^{p_1}, t_K^{p_2})$ is sequentially dependent if $t_K^{p_1}$ enables $t_K^{p_2}$. In both cases, K is known as **context graph**.

A pair $(t_K^{p_1}, t_K^{p_2})$ of transformations is a **critical pair of conflict** if it is parallel dependent, and $(t_K^{p_1}, t_K^{p_2})$ is a **critical pair of dependency** if it is sequentially dependent.

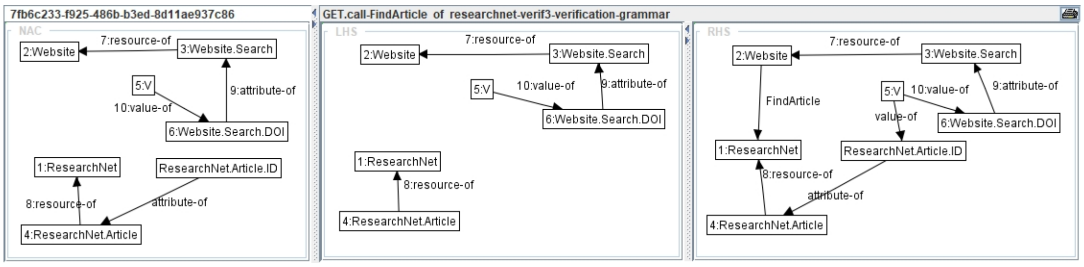
Figure 3.9: Examples of graph grammar rules and type graph in AGG



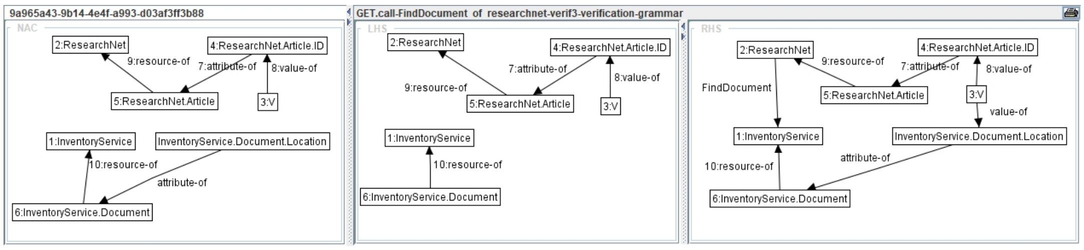
(a) Type Graph



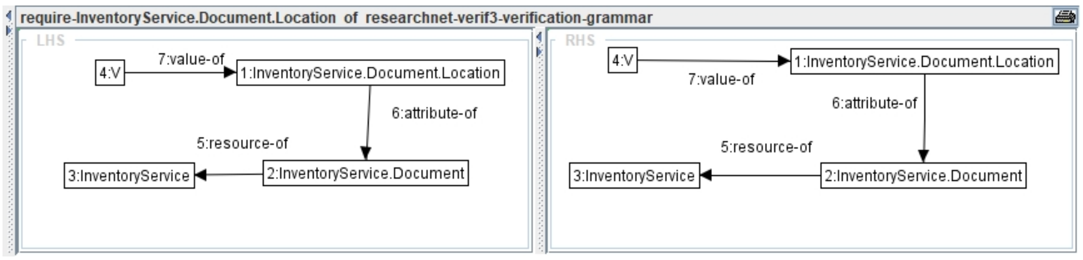
(b) Rule *mockgenerate-ResearchNet.Article.ID*



(c) Rule *GET.call-FindArticle*



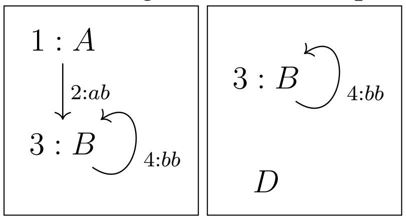
(d) Rule *GET.call-FindDocument*


require-InventoryService.Document.Location of researchnet-verif3-verification-grammar
LHS
4:V
7:value-of
1:InventoryService.Document.Location
6:attribute-of
5:resource-of
3:InventoryService
2:InventoryService.Document
RHS
4:V
1:InventoryService.Document.Location
7:value-of
6:attribute-of
5:resource-of
3:InventoryService
2:InventoryService.Document

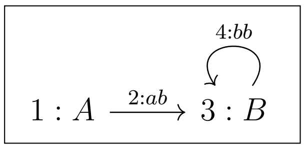
(e) Rule *require-InventoryService.Document.Location*

In order to compute critical pairs arising from any pair of productions (p_1, p_2), it is necessary to compute every single context graph K for which we have $(t_K^{p_1}, t_K^{p_2})$. Because there can be infinite such graphs, usually the set of context graphs is reduced by allowing only graphs which are not subgraphs of other graphs in such set. To compute this set, provided $p_1 = L_1 \leftarrow G_1 \rightarrow R_1$ and $p_2 = L_2 \leftarrow G_2 \rightarrow R_2$, we can compute the finite set of overlaps between L_1 and L_2 for critical pairs of conflict, or the finite set of overlaps between R_1 and L_2 for critical pairs of dependency, provided L_1, L_2 and R_1 are finite graphs. The elements of such set are known as **overlapping graphs**. In summary, we say $(t_O^{p_1}, t_O^{p_2})$ is a critical pair between transformations of rules p_1 and p_2 with overlapping graph O. As just explained, there can be many such pairs for two rules p_1 and p_2 with

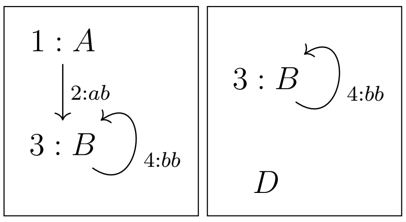
Figure 3.10: Example critical pairs $(t^{P_1}_{O_1}, t^{P_1}_{O_1})$ and $(t^{P_1}_{O_2}, t^{P_1}_{O_2})$



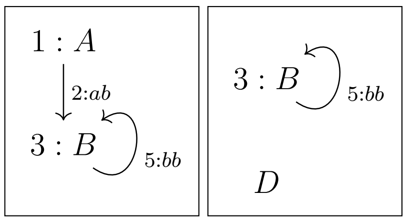
(a) Rule P1 which is both left and right component of $(t^{P_1}_{O_1}, t^{P_1}_{O_1})$



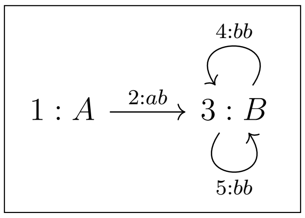
(b) Overlapping graph O_1 of $(t^{P_1}_{O_1}, t^{P_1}_{O_1})$



(c) Rule P1 left component of $(t^{P_1}_{O_2}, t^{P_1}_{O_2})$



(d) Rule P1 right component of $(t^{P_1}_{O_2}, t^{P_1}_{O_2})$



(e) Overlapping graph O_2 of $(t^{P_1}_{O_2}, t^{P_1}_{O_2})$

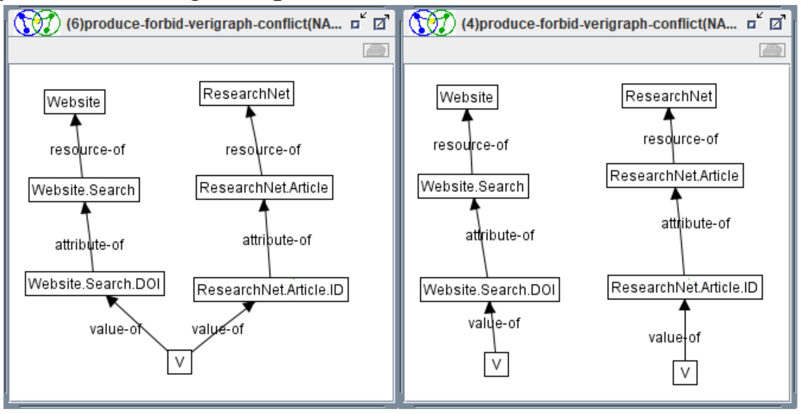
different O. Figure 3.10 shows two critical pairs of conflict between rule P_1 and itself. In this example, we can see that rule P_1 on the left hand side deletes node A and edge ab, which are needed to trigger rule P_1 on the right hand side of both critical pairs. This is captured by the overlapping graphs O_1 and O_2 and morphisms from the graphs of rules to O_1 and O_2.

There are different kinds of critical pairs, depending on how its productions relate to each other. For critical pairs of conflict we have *delete-use*, *produce-forbid*, *change-use* and *change-forbid*. For critical pairs of dependency we have *produce-use*, *delete-forbid*, *change-use* and *change-forbid*. All of these kinds of critical pairs can be found in more detail in the AGG Manual[2].

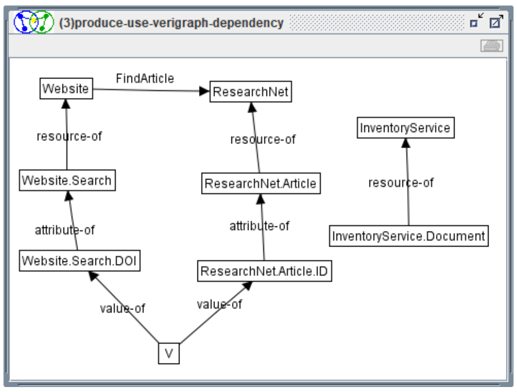
Finally, if an initial graph is large enough, it may be that both rules of a critical pair of conflict are applicable to different subgraphs. Similarly, it may be that after some transformations, a context graph never arises which would induce a pair of transformations that comprise a specific critical pair. Also, even if such context graph is found, it may be a third transformation is applied which again disables one or both transformations of that specific critical pair. In these regards, we say that critical pairs are *potential*.

[2]https://www.user.tu-berlin.de/o.runge/agg/AGG-ShortManual/node36.html

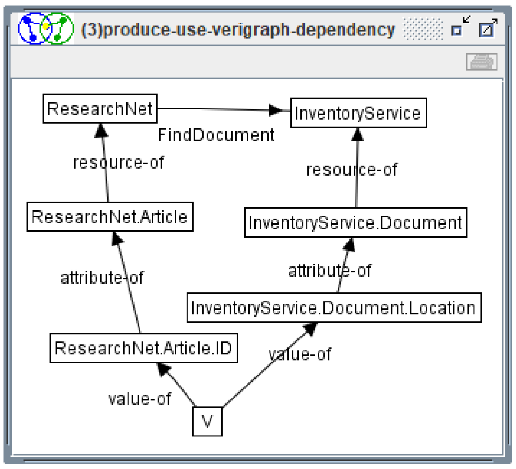
Figure 3.11: Examples of overlapping graphs as seen in AGG, we omit morphisms from rule graphs to overlapping graphs as there is only one possible morphism for each case (it maps as many nodes and edges as possible)



(a) Overlapping graphs of two different produce-forbid critical pairs of conflict between rules *mockgenerate-ResearchNet.Article.ID* and *GET.call-FindArticle*



(b) Overlapping graph of produce-use critical pair of dependency between rules *GET.call-FindArticle* and *GET.call-FindDocument*



(c) Overlapping graph of produce-use critical pair of dependency between rules *GET.call-FindDocument* and *require-InventoryService.Document.Location*

If we can show that for a critical pair of conflict $(t_O^{p_1}, t_O^{p_2})$ there are transformations (or a series of transformations) $t_X^{p_i} = R_1 \overset{p_i,m_i}{\Rightarrow} X$ and $t_X^{p_j} = R_2 \overset{p_j,m_j}{\Rightarrow} X$, then we say this critical pair is **confluent** (EHRIG, 2006). In other words, if we can get around the critical pair by applying other transformations such that we arrive at a common graph X, then the critical pair is confluent. Furthermore, we say this critical pair is **strictly confluent** if this series of transformations preserves a subgraph of O (EHRIG, 2006). A graph transformation system is **locally confluent** if all its critical pairs of conflict are strictly confluent.

To define termination criteria for graph transformation systems, we may use the concept of **production layers**. Intuitively, we classify the productions of a grammar in layers, such that elements of some type are only created by productions of the same layer, and may be deleted only by productions of subsequent layers. This ensures that if an element of a type is created by a transformation, some other transformation will delete it using rules of the same or subsequent layers only, when creation is no longer possible. To guarantee **termination** we additionally have to prove that each layer terminates before the grammar moves to the next layer of productions. For deletion layers, i.e., layers which contain only deleting rules, **layer termination** is shown by arguing that there is only a finite number of items that can be deleted in this layer and that these items are the ones that have been created by previous layers, but which are not created in the current layer (or else this deletion layer would not terminate). Layer termination of creation layers, i.e., layers that have rules that create items of some type, is shown by arguing that at some point creation will be halted by negative application conditions. The procedure assigning productions to deletion and creation layers, and thus showing whether a graph transformation system terminates, can be automated (EHRIG, 2006).

A graph transformation system is **confluent** if it is locally confluent and terminates. Confluence is relevant when we expect a system to exhibit a deterministc behavior, i. e. to produce unique final graph (up to isomorphism) for a given initial graph. In this work, we will use graph grammars in two different ways: (i) verification grammar: to express the semantics of a module net, and (ii) translation grammar: to associate a semantics to a module net. A verification grammar is a grammar that describes the integration behavior of the underlying module net, whereas a translation grammar basically defines a model transformation, generating the verification grammar that corresponds to a module net. Verification grammars may be non-deterministic (since they express behavior of possibly non-deterministic systems), but the translation grammar must be confluent to associate a unique meaning (verification grammar) to each module net.

4 MIGRATE FRAMEWORK OVERVIEW

Module Integration using Graph Grammars (MIGRATE) is a framework that aims to help developers in the process of integrating software modules. MIGRATE takes as input software artifacts, such as source code, and automatically produces a set of warnings informing developers what needs their attention.

This chapter presents the general framework for MIGRATE. It is a **framework**, because it is not usable out of the box. In order to use this framework, each of its abstract procedures has to be instantiated with a concrete procedure. In fact, the next few chapters of this dissertation are dedicated to explaining in detail a few of such concrete procedures. As previously mentioned, this framework can be instantiated to verify many kinds of **modules**, be it a class, library or service. If framework procedures are instantiated, we call the resulting procedure comprised of concrete procedures a **verification tool**.

The goal of our framework is to provide developers with useful information (warnings) concerning the integration of modules that compose their software. To produce such warnings, we start with software artifacts, from which we extract a single module net (see next section for its definition). We use a confluent graph grammar, which we call the translation grammar, to translate this module net into another graph grammar, which we call verification grammar. We generate critical pairs of rules of verification grammars and analyse these pairs to produce warnings, which we then report to users. Figure 4.1 shows an overview of the approach (figures are just meant to give an overview, each step will be explained better in next sections).

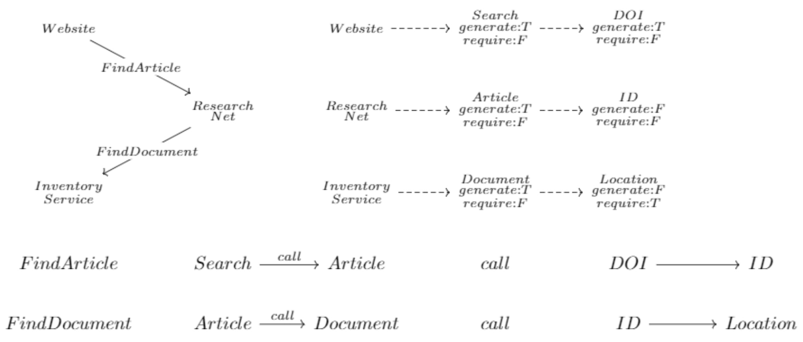
Many integration bugs are related to how information is passed from a module to another. For example, if a service asks for more data than it uses, then we can suggest that excess attributes should be deprecated. On the other hand, if a client fails to provide information required by a service, then we can tell developers we have likely found a bug. In order to uncover such bugs, we have to analyse how information is used by each module and what are the actual dependencies that emerge from data.

Our verification procedure consists of interpreting the critical pairs of rules generated for a verification grammar. This verification grammar does not mirror the exact behavior of the software that originated it, but rather it reflects how information flows in this software. In that regard, we can say the verification grammar describes a kind of software integration semantics and it is not suitable for simulation.

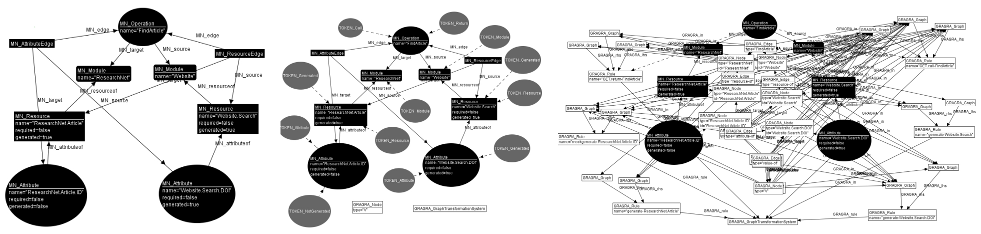
Figure 4.1: Overview of the proposed approach

```
ReadSearch() {          FindArticle(doi) {   FindDocument(id) {
  DOI := read();          id := find(doi);     location := "res/" + id;
  FindArticle(DOI);       FindDocument(id);    retrieve(location);
}                       }                    }
```

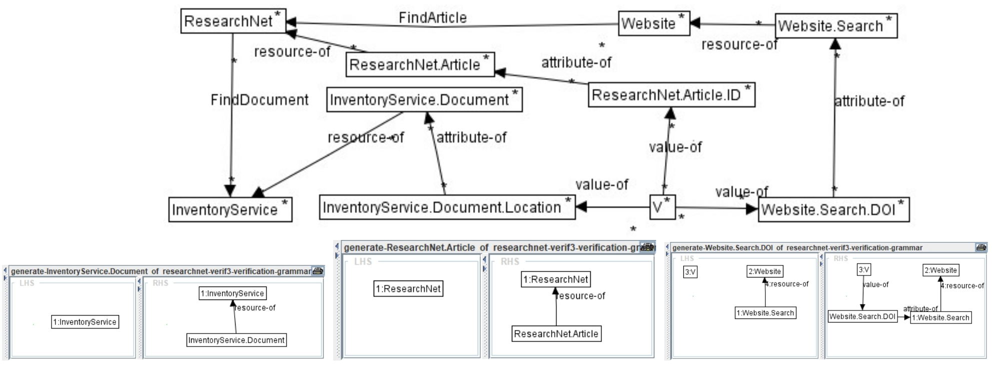
(a) Software artifacts are taken as input



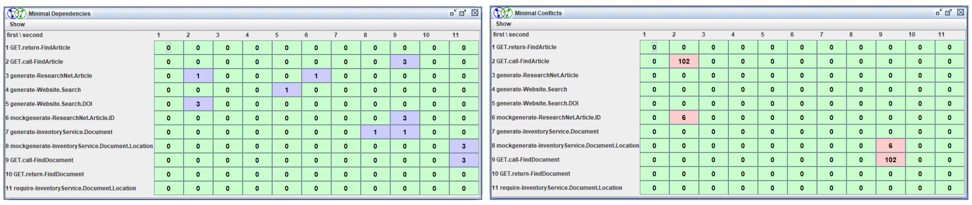
(b) Module net is extracted from software artifacts



(c) Module net is translated into verification grammar using AGG



(d) Verification grammar resulting from translation (some rules have been omitted)



Minimal Dependencies

first \ second	1	2	3	4	5	6	7	8	9	10	11
1 GET.return-FindArticle	0	0	0	0	0	0	0	0	0	0	0
2 GET.call-FindArticle	0	0	0	0	0	0	0	0	3	0	0
3 generate-ResearchNet.Article	0	1	0	0	0	1	0	0	0	0	0
4 generate-Website.Search	0	0	0	0	1	0	0	0	0	0	0
5 generate-Website.Search.DOI	0	3	0	0	0	0	0	0	0	0	0
6 mockgenerate-ResearchNet.Article.ID	0	0	0	0	0	0	0	0	3	0	0
7 generate-InventoryService.Document	0	0	0	0	0	0	0	1	1	0	0
8 mockgenerate-InventoryService.Document.Location	0	0	0	0	0	0	0	0	0	0	3
9 GET.call-FindDocument	0	0	0	0	0	0	0	0	0	0	3
10 GET.return-FindDocument	0	0	0	0	0	0	0	0	0	0	0
11 require-InventoryService.Document.Location	0	0	0	0	0	0	0	0	0	0	0

Minimal Conflicts

first \ second	1	2	3	4	5	6	7	8	9	10	11
1 GET.return-FindArticle	0	0	0	0	0	0	0	0	0	0	0
2 GET.call-FindArticle	0	102	0	0	0	0	0	0	0	0	0
3 generate-ResearchNet.Article	0	0	0	0	0	0	0	0	0	0	0
4 generate-Website.Search	0	0	0	0	0	0	0	0	0	0	0
5 generate-Website.Search.DOI	0	0	0	0	0	0	0	0	0	0	0
6 mockgenerate-ResearchNet.Article.ID	0	6	0	0	0	0	0	0	0	0	0
7 generate-InventoryService.Document	0	0	0	0	0	0	0	0	0	0	0
8 mockgenerate-InventoryService.Document.Location	0	0	0	0	0	0	0	0	6	0	0
9 GET.call-FindDocument	0	0	0	0	0	0	0	0	102	0	0
10 GET.return-FindDocument	0	0	0	0	0	0	0	0	0	0	0
11 require-InventoryService.Document.Location	0	0	0	0	0	0	0	0	0	0	0

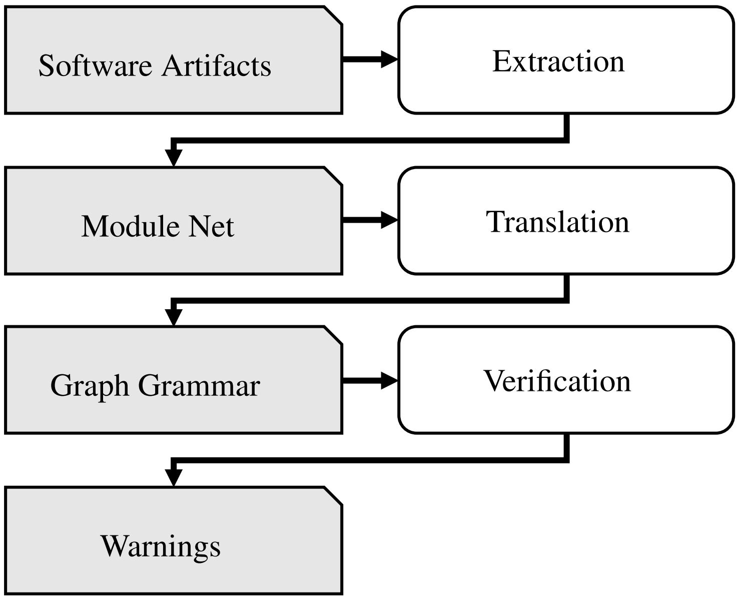
(e) Critical pairs generated for verification grammar using Verigraph

```
Website.Search.DOI can become outdated (generated and FindArticle)
The return of operation FindArticle is not used
The return of operation FindDocument is not used
```

(f) Warnings are generated based on critical pairs

Figure 4.2 depicts the module integration verifier framework. Dark squares with snipped corner represent artifacts, while squares with rounded corners and no fill are procedures that have to be instantiated. We will explain each of those squares in detail shortly.

Figure 4.2: Module Integration using Graph Grammars framework.



The framework is given as input a set of **software artifacts**. These artifacts can be anything that is machine readable and provides insight into how information flows in a system. For services, these can be OpenAPI documents. For libraries and classes, the actual code and interfaces. Models (such as UML) can also be used as software artifacts for all kinds of modules. Further kinds of artifacts can be used such as dynamic data of real payload exchanges and logs for services, and execution traces and automated tests for libraries and classes. In Chapter 8 we illustrate the framework using pseudocode as software artifacts.

The **extraction** procedure takes software artifacts and produces a module net. Extraction strategy can vary depending on what we choose as software artifacts. If a verification tool implements extraction procedures for different kinds of artifacts, these extraction procedures can be applied over all artifacts and the resulting module nets can be joined together to obtain a single output module net that provides a system wide view. Because different kinds of artifacts provide different insights into a system, this strategy allows building richer module nets. At the time of writing we do not have an implemented extraction procedure yet.

Chapter 5 introduces the idea of a **module net**. This structure is used to present how modules of a system exchange data. It is language agnostic and thus enables a verification procedure that is independent of language. Furthermore, module nets are visual models that can be edited by developers who wish to design and verify such design, skipping the extraction procedure partially or entirely.

Module nets are solely a syntactic structure and their semantics is defined via a **translation** procedure that translates a module net into a single **graph grammar**, which, in turn, can be verified later. Chapter 6 is dedicated to showing one translation procedure.

Finally, the **verification** procedure builds upon existing graph grammar verification techniques to produce different kinds of **warnings** concerning the system under analysis. Chapter 7 explains ways to use the results of the verification in defining concrete warnings.

Translation and verification procedures presented in this dissertation are implemented to provide an illustration of the proposed framework. However, the resulting verifier is a prototype, as it is not ready to find real faults, but only very simple cases in controlled environments (see case study in Chapter 8).

5 MODULE NETS

A module net describes how information flows in a system. Module nets can be drawn as diagrams for better visualization. Because of this visualization, module nets are also suitable for manual editing, enabling software designers to specify systems using this kind of notation and later verify that their design is correct using a module integration verifier tool. The semantics of module nets is defined via a translation to graph grammars, which is described in Chapter 6.

5.1 Definition

First we have to define **graphs**, which will be used later to define operations and module networks. This is a classical definition of directed graphs.

Definition 1 (Directed Graph)**.** *A **directed graph**, is a tuple $G = (N, E, s, t)$ where N and E are sets of nodes and edges, respectively, and $s, t : E \to N$ are total functions assigning a source/target node to each edge. A **subgraph** of a graph G is a graph which contains subsets of the sets of nodes and edges of G, while preserving source and target functions.*

Quadripartite graphs are graphs partioned in four subgraphs, but whose set of nodes is partitioned in two. In this work, we will partition the set of nodes N in two sets, denoted N_l and N_r, representing the nodes in the left-hand side and right-hand side of a graph, respectively. This induces a partition of the set of edges E in sets E_{ll} (representing edges between nodes of N_l), E_{lr} (representing edges from N_l to N_r), E_{rl} (representing edges from N_r to N_l) and E_{rr} (representing edges between nodes of N_r). Considering these different kinds of edge partitions, we can build four different subgraphs of a graph.

Definition 2 (Quadripartite Directed Graph)**.** *A **quadripartite graph** is a graph $Q = (N, E, s, t)$ such that*

- $N = N_l \cup N_r$ *and* $N_l \cap N_r = \emptyset$
- $E = \bigcup_{i \in \{ll,lr,rl,rr\}} E_i$ *and* $(E_i \cap E_j)_{i,j \in \{ll,lr,rl,rr\}, i \neq j} = \emptyset$ *are pairwise disjoint*
- $s = \bigcup_{i \in \{ll,lr,rl,rr\}} s_i$ *with* $(s_{ij} : E_{ij} \to N_i)_{i,j \in \{l,r\}}$
- $t = \bigcup_{i \in \{ll,lr,rl,rr\}} t_i$ *with* $(t_{ij} : E_{ij} \to N_j)_{i,j \in \{l,r\}}$

The graphs $(Q_{ij} = (N_i \cup N_j, E_{ij}, s_{ij}, t_{ij}))_{i,j \in \{l,r\}}$ are subgraphs of Q.

Example 1 (Quadripartite graph Q_1)**.** *For a graph $Q_1 = (N, E, s, t)$ with*

- $N = \{La, Lb, Ra, Rb\}$
- $E = \{la, lara, ralb, rarb\}$
- $s = \{la \mapsto La, lara \mapsto La, ralb \mapsto Ra, rarb \mapsto Ra\}$
- $t = \{la \mapsto La, lara \mapsto Ra, ralb \mapsto Lb, rarb \mapsto Rb\})$

we can build the following partitions:

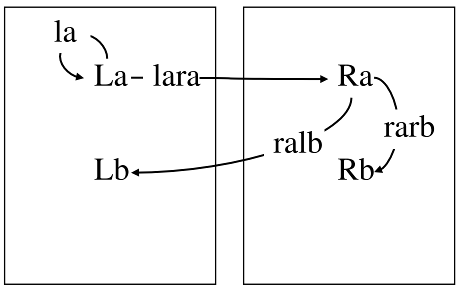
- $N_l = \{La, Lb\}$*,* $N_r = \{Ra, Rb\}$
- $E_{ll} = \{la\}$*,* $E_{lr} = \{lara\}$*,* $E_{rl} = \{ralb\}$*,* $E_{rr} = \{rarb\}$
- $s_{ll} = \{la \mapsto La\}$*,* $s_{lr} = \{lara \mapsto La\}$*,* $s_{rl} = \{ralb \mapsto Ra\}$*,* $s_{rr} = \{rarb \mapsto Ra\}$
- $t_{ll} = \{la \mapsto La\}$*,* $t_{lr} = \{lara \mapsto Ra\}$*,* $t_{rl} = \{ralb \mapsto Lb\}$*,* $t_{rr} = \{rarb \mapsto Rb\}$

Finally, the following graphs are subgraphs of Q_1:

- $Q_{ll} = (\{La, Lb\}, \{la\}, \{la \mapsto La\}, \{la \mapsto La\})$
- $Q_{lr} = (\{La, Lb, Ra, Rb\}, \{lara\}, \{lara \mapsto La\}, \{lara \mapsto Ra\})$
- $Q_{rl} = (\{La, Lb, Ra, Rb\}, \{ralb\}, \{ralb \mapsto Ra\}, \{ralb \mapsto Lb\})$
- $Q_{rr} = (\{Ra, Rb\}, \{rarb\}, \{rarb \mapsto Ra\}, \{rarb \mapsto Rb\})$

Figure 5.1 shows a visual representation of Q_1, where nodes belonging to N_l and N_r are depicted in the left and right-hand side rectangles respectively.

Figure 5.1: Example quadripartite directed graph Q_1



The first module network definition is that of a **resource**, which is a unit of information, any kind of data a system may share between its modules, either structured data or not. Resources can be database entities, API models, HTTP tickets, instances of classes, files, any information at all. We can either see resources as names independently of their values or as instances of data types.

To further specify what kind of data a resource contains, we provide **attributes**, which are pieces of information that comprise a resource. Attributes can represent primitive data types, such as integers or characters, but also complex data types such as entire objects. It is left for verification tool developers to decide[1] how to map the data of a system to resources and attributes.

Resources can either have all their attributes listed, or none at all. Listing all attributes provides the added value of verifying the information flow of such attributes, at the cost of having to list them. Omitting such attributes yields a poorer verification, but still a valid one.

Definition 3 (Resource)**.** *A **resource** is a pair composed of a resource name and a (finite) set of **attributes** (i.e., a set of names). Given a resource $r = (name, attr)$ we denote its name by $name^r$ and its attribute set by $attr^r$.*

Example 2 (Resources R_1^A, R_1^B)**.** *Using the definition of a resource, we define resources $R_1^A = (R_1^A, \{a_1, a_2\})$ and $R_1^B = (R_1^B, \{b_1, b_2\})$, and we have $attr^{R_1^A} = \{a_1, a_2\}$ and $attr^{R_1^B} = \{b_1, b_2\}$. As abuse of notation, we will frequently use the same symbol for a resource and its name.*

Modules are the units of a system. Just like resources represent any kind of data, modules represent any kind of subsystem: a service, a library, a class, anything. Modules contain resources, which are the types of information a module of this kind may share with its peers. Modules contain functions **req** (for required) and **ger** (for generated) defined over its resources and their attributes. Required resources/attributes are necessary to perform some kind of unspecified but essential operation. These can be, for example, side effects such as data that is written to the screen, or data that is shared with a third-party module to which we have no access. Generated resources/attributes are generated by some unspecified operation within a module, such as data that is input by a person or data received from a third-party module to which we do not have access. Required and generated resources/attributes can also be used to omit modules obtaining smaller module networks, if we wish so.

Definition 4 (Module)**.** *A **module** is a tuple $\mathcal{M} = (name, R_{\mathcal{M}}, req^{\mathcal{M}}, ger^{\mathcal{M}})$ where*

- *$name$ is its name*
- *$R_{\mathcal{M}}$ is a finite set of resources with unique names,*

[1] See Chapter 4 for more details on the module integration verifier framework and tool development.

- *$req^{\mathcal{M}}, ger^{\mathcal{M}} : R_{\mathcal{M}} \uplus A_{\mathcal{M}} \to \{T, F\}$ are total functions, assigning to each attribute/resource a boolean value indicating whether they are required/generated in this module, where $A_{\mathcal{M}} = \biguplus_{r \in R_{\mathcal{M}}} attr^r$.*

We denote by $Resources^{\mathcal{M}}$ the set of resource names of a module $\mathcal{M}$.

Example 3 (Modules M_A, M_B)**.** *Using the definition of a module and resources from Example 2, we define modules $M_A = (M_A, \{R_1^A\}, \{R_1^A \mapsto F, a_1 \mapsto T, a_2 \mapsto F\}, \{R_1^A \mapsto T, a_1 \mapsto T, a_2 \mapsto T\})$ and $M_B = (M_B, \{R_1^B\}, \{R_1^B \mapsto F, b_1 \mapsto F, b_2 \mapsto F\}, \{R_1^B \mapsto T, b_1 \mapsto T, b_2 \mapsto T\})$. Figure 5.2 shows these modules in visual notation.*

Figure 5.2: Modules M_A and M_B from Example 3 in visual notation.

$$\begin{tikzcd}
M_A \arrow[r, dashed] & \substack{R_1^A \\ generate{:}T \\ require{:}F} \arrow[r, dashed] \arrow[rr, dashed, bend left] & \substack{a_1 \\ generate{:}T \\ require{:}T} & \substack{a_2 \\ generate{:}T \\ require{:}F} \\
M_B \arrow[r, dashed] & \substack{R_1^B \\ generate{:}T \\ require{:}F} \arrow[r, dashed] \arrow[rr, dashed, bend left] & \substack{b_1 \\ generate{:}T \\ require{:}F} & \substack{b_2 \\ generate{:}T \\ require{:}F}
\end{tikzcd}$$

Operations are the bindings from module to module[2]. Whereas modules are containers of information, generating and requiring information, operations define how information flows from a module to another. Even though it is not stated directly in the definition, operations range over two modules, a source or **caller** and a target or **callee**, just like an edge of a graph.

Operations are quadripartite graphs augmented with an attribute relation. The nodes of an operation graph are resources of its caller and callee. The operation graph shows how the information flows from a resource of a module to a resource of another module. We will often refer to the subset of the edges from resources of the caller as the **request** or **call**, and to the other subset with resources of the callee as source, as the **response** or **return**. Note these two subsets of edges (call and return) comprise the whole set of edges of an operation graph.

Each edge of an operation graph (from a resource to another) is augmented with a relation[3] from the attributes of the first resource to the attributes of the second. Edges of operation graphs represent the transfer of a value from attribute to attribute.

[2]Note the actual modules are not part of the definition of an operation, see module network definition for that part.

[3]Even though we define it as a relation, we will draw attribute relations the same way we draw graphs for better visualization.

Definition 5 (Operation)**.** *Given a set of resources $\mathcal{R}$, an* ***operation*** *op defines how the operation acts on resources/attributes, where $op = (\mathcal{R}, E^{op}, s^{op}, t^{op}, rel^{op})$ is a quadripartite graph and $rel^{op} : E^{op} \to REL$ is a total function that maps each edge $e \in E^{op}$ to a relation $REL \subseteq attr^{s^{op}(e)} \times attr^{t^{op}(e)}$. We write $\mathcal{R} = \mathcal{R}_\rho \cup \mathcal{R}_\epsilon$ the node partitioning of op and $E^{op} = E^{op}_{\rho\epsilon} \cup E^{op}_{\epsilon\rho}$ its edge partitioning. Note that $E^{op}_{\rho\rho} = E^{op}_{\epsilon\epsilon} = \emptyset$.*

Example 4 (Operation $E^1_{A,B}$)**.** *Using the definition of operation and modules from Example 3, we define operation $E^1_{A,B} = (\{R^A_1, R^B_1\}, \{E_{call}, E_{return}\}, s, t, rel)$ and*

- $s = \{E_{call} \mapsto R^A_1, E_{return} \mapsto R^B_1\}$, $t = \{E_{call} \mapsto R^B_1, E_{return} \mapsto R^A_1\}$
- $rel = \{E_{call} \mapsto \{(a_2, b_2)\}, E_{return} \mapsto \{(b_1, a_1), (b_2, a_2)\}\}$
- $\mathcal{R}^{E^1_{A,B}}_\rho = \{R^A_1\}$, $\mathcal{R}^{E^1_{A,B}}_\epsilon = \{R^B_1\}$, $E^{E^1_{A,B}}_{\rho\epsilon} = \{E_{call}\}$, $E^{E^1_{A,B}}_{\epsilon\rho} = \{E_{return}\}$

Figure 5.3: Operation from Example 4 in visual notation.

$$\begin{tikzcd}
E^1_{A,B} & R^A_1 \arrow[r, bend left, "E_{call}"] & R^B_1 \arrow[l, bend left, "E_{return}"] & E_{call} & a_2 \arrow[r] & b_2 \\
 & & & E_{return} & b_1 \arrow[r] & a_1 \\
 & & & & b_2 \arrow[r] & a_2
\end{tikzcd}$$

Theorem 1 (Resource-attribute compatibility of operations)**.** *Operations are compatible with the attributes of their resources as stated below.*

$$\forall op = (\mathcal{R}, E^{op}, s^{op}, t^{op}, rel^{op}), e \in E^{op}.$$

$$(s^{op}(e) = r1 \to dom(rel^{op}(e)) \subseteq attr^{r1})$$

$$and$$

$$(t^{op}(e) = r2 \to rng(rel^{op}(e)) \subseteq attr^{r2})$$

Proof. Follows from the definition of operations, we write the proof for s and omit proof for t as it is similar:

$$\begin{aligned}
op =&(\mathcal{R}, E^{op}, s^{op}, t^{op}, rel^{op}), e \in E^{op}. \\
&\Rightarrow rel^{op}(e) \subseteq attr^{s^{op}(e)} \times attr^{t^{op}(e)} && \text{(by Definition 5)} \\
&\Rightarrow dom(rel^{op}(e)) \subseteq dom(attr^{s^{op}(e)} \times attr^{t^{op}(e)}) \\
&\Rightarrow dom(rel^{op}(e)) \subseteq attr^{s^{op}(e)} \\
&(s^{op}(e) = r1) \rightarrow (dom(rel^{op}(e)) \subseteq attr^{r1})
\end{aligned}$$

■

A **module network**, or short module net, is a graph whose nodes are modules and edges are operations. Additionally, modules of a module network do not share resources, i.e., resources are unique, and the set of all resources in a module network is the union of the resources in its modules. Essentially, operations of a module network carry information between modules.

To be well defined, a module network has to satisfy two properties. Equation 5.1 requires that no two modules of a module network share resources, that is, the only way for modules to share data is through an operation. Equation 5.2 ensures that operations have a caller ($s^{\mathcal{M}}(op)$) and a callee module ($t^{\mathcal{M}}(op)$) and that the resources of an operation (R^{op}) are subsets of the resources of caller ($R_{s^{\mathcal{M}}(op)}$) and callee modules ($R_{t^{\mathcal{M}}(op)}$) in the module net.

Definition 6 (Module network)**.** *A **module network** is a tuple $MN = (\mathcal{M}, Op, s^{\mathcal{M}}, t^{\mathcal{M}})$ where*

- *$\mathcal{M}$ is a finite set of modules;*
- *Op is a finite set of operations over the resources $\mathcal{R} = \biguplus_{m\in\mathcal{M}} R_m$;*
- *resources are unique:*

$$\forall m1, m2 \in \mathcal{M}. Resources^{m1} \cap Resources^{m2} = \emptyset \tag{5.1}$$

- *MN is a graph such that each operation edge is compatible with the modules of* *the module network:*

$$\forall op \in Op.\mathcal{R}^{op} = \mathcal{R}^{op}_{\rho} \cup \mathcal{R}^{op}_{\epsilon} \rightarrow \mathcal{R}^{op}_{\rho} \subseteq R_{s^{\mathcal{M}}(op)} \; and \; \mathcal{R}^{op}_{\epsilon} \subseteq R_{t^{\mathcal{M}}(op)} \tag{5.2}$$

Example 5 (Module Net MN_1)**.** *We can use the previously defined modules M_A and M_B and operation $E^1_{A,B}$ to define the module net $MN_1 = (\{M_A, M_B\}, \{E^1_{A,B}\}, \{E^1_{A,B} \mapsto M_A\}, \{E^1_{A,B} \mapsto M_B\})$. See Figure 5.4 for its visual notation. To make sure MN_1 is well defined, we need to check a few of its properties. First we check that its resources are unique according to Equation 5.1:*

$$\begin{aligned} &M_A, M_B \in \mathcal{M}_{MN_1}.Resources_{M_A} \cap Resources_{M_B} = \emptyset \\ &\qquad\qquad\qquad \Rightarrow M_A, M_B \in \mathcal{M}_{MN_1}.\{R^A_1\} \cap \{R^b_1\} = \emptyset \end{aligned}$$

now we check MN_1 is compatible with its modules according to Equation 5.2:

$$\begin{aligned} &E^1_{A,B} \in Op_{MN_1}. \\ &\quad \mathcal{R}^{E^1_{A,B}} = \mathcal{R}^{E^1_{A,B}}_{\rho} \cup \mathcal{R}^{E^1_{A,B}}_{\epsilon} \rightarrow \mathcal{R}^{E^1_{A,B}}_{\rho} \subseteq R_{s^{\mathcal{M}}(E^1_{A,B})} \; and \; \mathcal{R}^{E^1_{A,B}}_{\epsilon} \subseteq R_{t^{\mathcal{M}}(E^1_{A,B})} \\ &\Rightarrow E^1_{A,B} \in Op_{MN_1}.\mathcal{R}^{E^1_{A,B}} = \{R^A_1\} \cup \{R^B_1\} \rightarrow \{R^A_1\} \subseteq R_{M_A} \; and \; \{R^B_1\} \subseteq R_{M_B} \\ &\Rightarrow E^1_{A,B} \in Op_{MN_1}.\mathcal{R}^{E^1_{A,B}} = \{R^A_1\} \cup \{R^B_1\} \rightarrow \{R^A_1\} \subseteq \{R^A_1\} \; and \; \{R^B_1\} \subseteq \{R^B_1\} \end{aligned}$$

Figure 5.4: Module net from Example 5 in visual notation. Usually a module net will be drawn next to its modules and operations, but we omit these here because they are already drawn in Figures 5.2 and 5.3 respectively.

$$\begin{tikzcd} M_A \arrow[r, "E^1_{A,B}"] & M_B \end{tikzcd}$$

5.2 Limitations and final remarks

With the current definition of module nets, it is difficult to express every kind of behaviors a system may have. For example, because operations are simple quadripartite graphs, it is impossible for two different operations to be mapped to different graph grammar structures during a translation, because there is nothing in the module net operation definition that would tell one operation kind from the other. For that reason, in Chapter 6 we define all operations with a simple retrieval behavior, ignoring other system behaviors such as delete or create.

A solution to this problem is to define module net operations as *typed* quadripartite graphs instead. With this definition, it would be possible to give all kinds of semantics to module net operations, thus covering much better the different behaviors a system may exhibit. It is left for future work to define module net operations as typed quadripartite graphs and implement a verifier tool with this definition.

6 TRANSLATION

This chapter presents an implementation of a translation procedure from the language of module nets to graph grammars providing a semantics for module nets. We split this chapter into four sections. First we set a terminology to talk about this translation and give an overview of the procedure. Section 6.2 shows a small end-to-end example: from module net to verification grammar. Section 6.3 presents properties of the translation procedure, such as well-definedness and confluence. Finally, the last section discusses a few improvements, some of which have been implemented.

6.1 Translation

The **translation** takes a model in the source language (module net) and produces a model in the target language (graph grammar). We refer to the source model as just **module net** and to the target model as **verification grammar**. This translation was defined by a graph grammar called **translation grammar**. The translation grammar has an **initial graph**, which is a module net encoded as a graph, and after the rule application process is carried out until termination we obtain a **final graph**, which is an encoded verification grammar. To summarize, the translation procedure is comprised of three steps:

- **Encoding**: takes a module net and encodes it, producing an initial graph
- **Derivation**: applies translation grammar rules until termination, producing a final graph
- **Extraction**: extracts a verification grammar from a final graph

Figure 6.2 shows the type graph used in the translation. This graph has three kinds of nodes, that can be distinguished by the prefix of their names (and by color):

- nodes whose names start with "MN" (black) are used to describe module net components
- nodes whose names start with "TOKEN" (gray) denote auxiliary items used in the translation process
- nodes whose names start with "GRAGRA" (white) describe components of the resulting verification grammar

Before beginning a translation, the initial graph is expected to have only nodes of

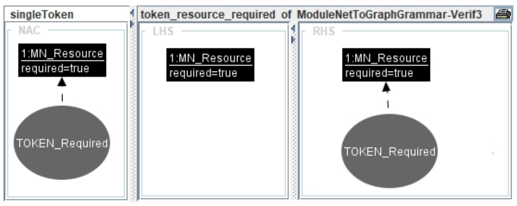
"MN" types. During translation, nodes of "TOKEN" type will be created and deleted. At the end of the procedure, the final graph will contain only nodes of type "GRAGRA".

The translation grammar has 35 rules and for that reason we omit these rules here. To see the rules, please refer to the corresponding appendix. To keep text concise, we assign codes to rule names in Table 6.1 and refer to these codes later when necessary. Each rule belongs to a specific layer. The translation process starts by applying only rules from layer 0, rules from subsequent layers may only be applied once it is not possible to apply any rule from the current layer. This classification of rules in layers is what guides the translation termination (as will be discussed later). We present a summary of what each of these 35 rules does in Table 6.2.

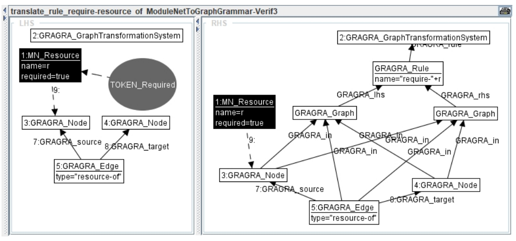
Table 6.1: Translation grammar rule codes.

Codes	Layer	Rule Name
TK1 to TK11	0	token_*
TR12	0	translate_graphTransformationSystem
TR13	0	translate_value
TR14	1	translate_module
TR15	1	translate_resource
TR16	1	translate_attribute
TR17	1	translate_rule_require-resource
TR18	1	translate_rule_generate-resource
TR19	1	translate_rule_mockgenerate-resource
TR20	1	translate_rule_require-attribute
TR21	1	translate_rule_generate-attribute
TR22	1	translate_rule_mockgenerate-attribute
TR23	1	translate_rule_call_modules
TR24	1	translate_rule_call_resources
TR25	1	translate_rule_call_attributes
TR26	1	translate_rule_return_modules
TR27	1	translate_rule_return_resources
TR28	1	translate_rule_return_attributes
CL29 to CL33	2	clean_*
AD34 to AD35	3	adjust_*

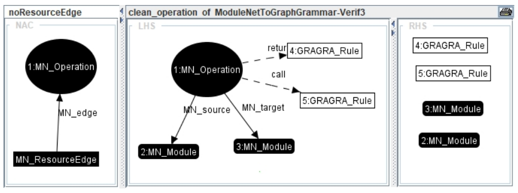
Figure 6.1: Examples of translation rules



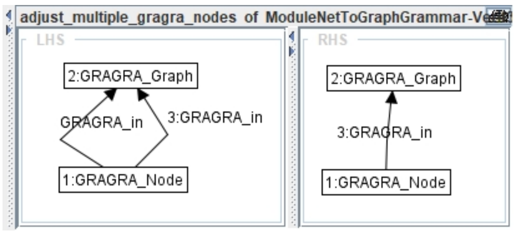
(a) Translation rule TK1



(b) Translation rule TR17



(c) Translation rule CL29


adjust_multiple_gragra_nodes of ModuleNetToGraphGrammar-Verif3
LHS
RHS
2:GRAGRA_Graph
GRAGRA_in
3:GRAGRA_in
1:GRAGRA_Node

(d) Translation rule AD34

Figure 6.2: Translation grammar type graph

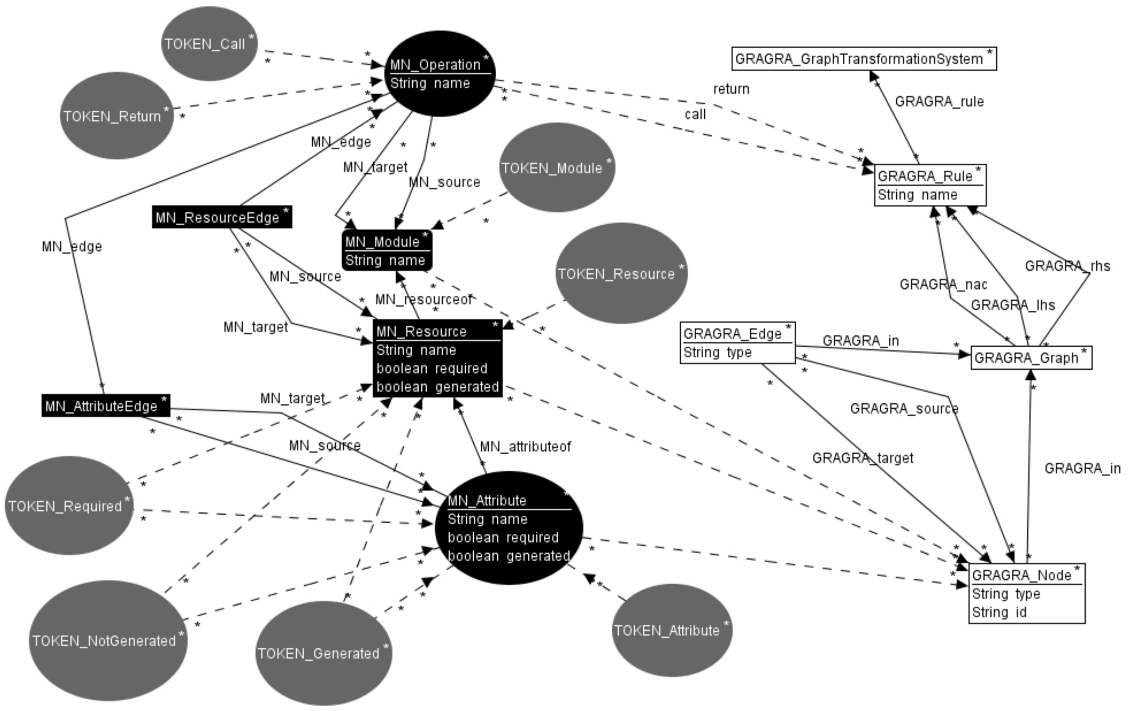
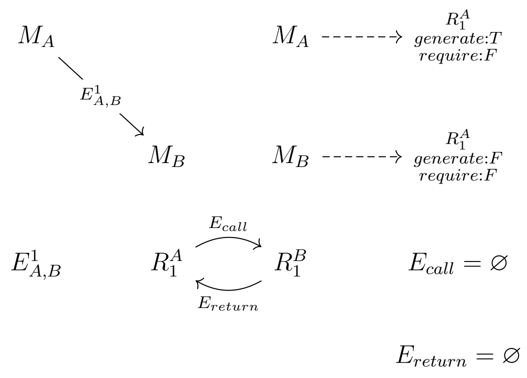


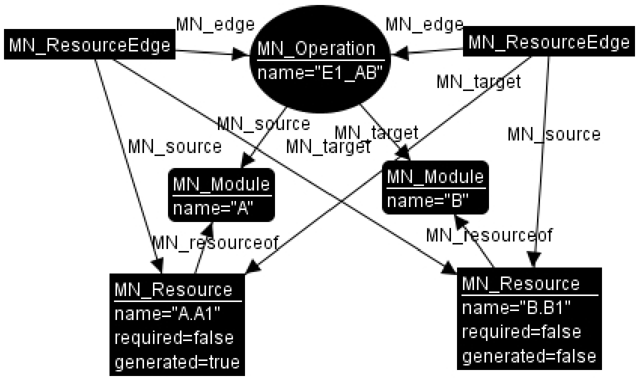
Table 6.2: Summary of translation grammar rules. For the actual rules, see appendix.

Codes	Purpose
TK1 to TK11	Insert "TOKEN" nodes to be consumed by TR* rules.
TR12 to TR13	Create "GRAGRA" nodes for GTS and value.
TR14 to TR16	Create "GRAGRA" nodes and edges for modules, resources and attributes, consuming "TOKEN" nodes.
TR17 to TR19	Create "GRAGRA" rules for required resources (TR17) and for generated (TR18) or non-generated (TR19) resources, consuming "TOKEN" nodes.
TR20 to TR22	Create "GRAGRA" rules for required attributes (TR20) and for generated (TR21) or non-generated (TR22) attributes, consuming "TOKEN" nodes.
TR23 to TR25	Create "GRAGRA" rules for operations calls (from source to target), consuming "TOKEN" nodes (TR23), resource edges (TR24) and attribute edges (TR25).
TR26 to TR28	Create "GRAGRA" rules for operations returns (from target to source), consuming "TOKEN" nodes (TR26), resource edges (TR27) and attribute edges (TR28).
CL29 to CL33	Remove "MN" nodes and edges that have been translated already, operations (CL29), "resourceof" edges (CL30), modules (CL31), "attributeof" edges (CL32) and resources (CL33).
AD34 to AD35	Remove duplicate "GRAGRA_in" edges for nodes (AD34) and edges (AD35).

Figure 6.3: Module net MN_1 and corresponding graph encoding.



(a) Module net MN_1.



(b) Encoded module net MN_1

6.2 Operation

Figure 6.3a shows the module net that will be translated in this section. As mentioned, the first step to be performed is encoding. In this phase, we encode module net components into graph components, obtaining the graph illustrated in Figure 6.3b.

We begin the translation with the initial graph and applying rules of layer 0 until no more rule can be applied, obtaining the graph from Figure 6.4. Note that layer 0 is responsible only for adding TOKEN and two GRAGRA nodes (GTS and value). Figure 6.5 is the result of the application of all layer 1 rules. We can see all tokens have been consumed and the result is a graph consisting of MN and GRAGRA nodes only, with some mappings from MN nodes to GRAGRA nodes. We omit the graph resulting from the application of rules of layer 2 because it looks like the one in Figure 6.5, except that all MN types have been removed. In this example, rules of layer 3 are not applicable (because at this point the host graph does not have duplicate GRAGRA_in edges for nodes or edges), so the result of layer 2 is the final graph. As a result from the extraction procedure from the final graph, we obtain the verification grammar depicted in Figure 6.6.

Figure 6.4: Graph after application of layer 0 - translation of MN_1

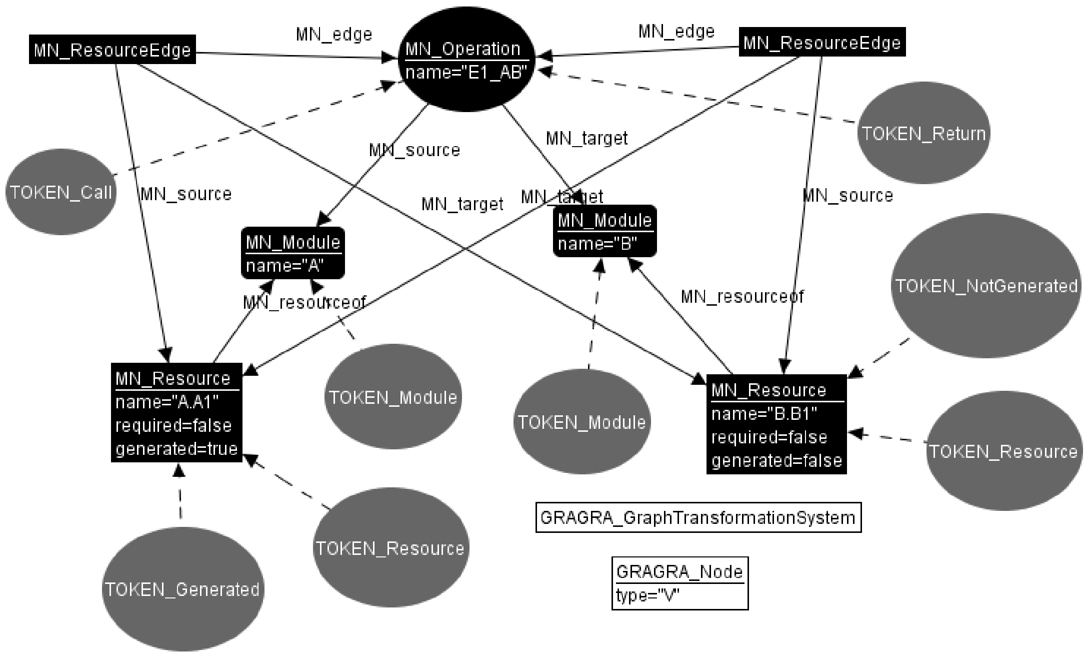


Figure 6.5: Graph after application of layer 1 - translation of MN_1

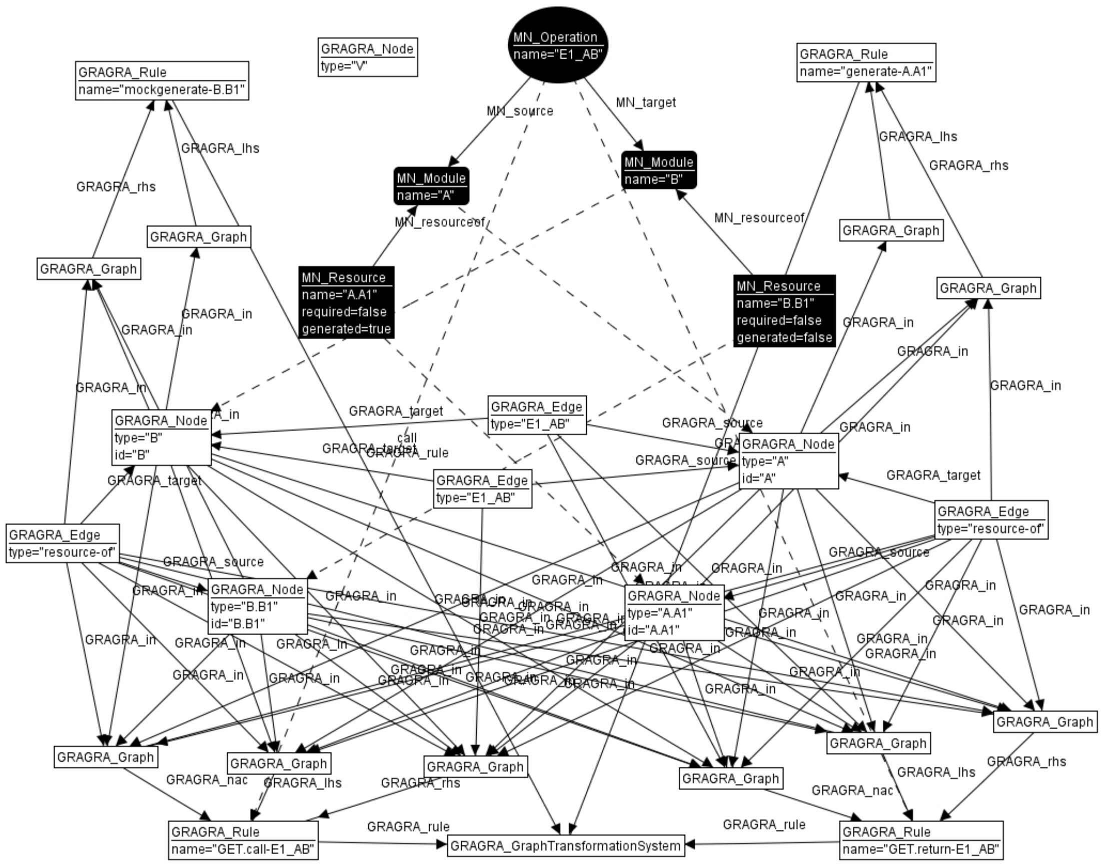
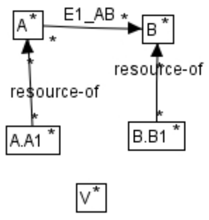
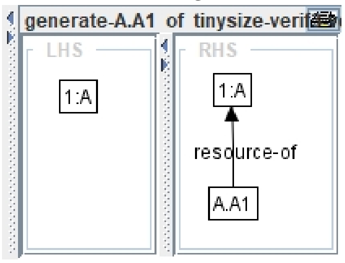


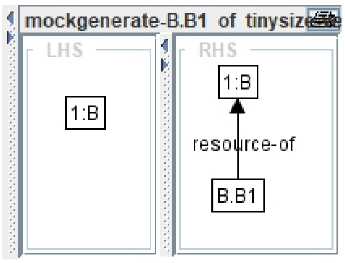
Figure 6.6: Verification grammar for module net MN_1



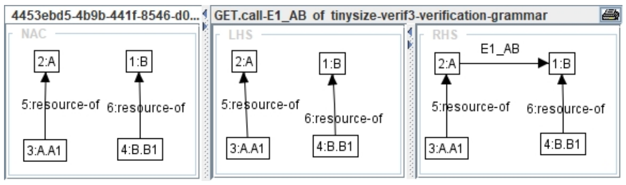
(a) Type graph



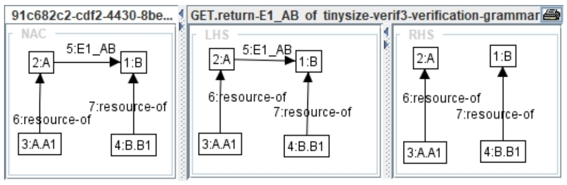
(b) *generate-A.A1*



(c) *mockgenerate-B.B1*



(d) *GET.call-E1_AB*



(e) *GET.return-E1_AB*

6.3 Properties

In this section we analyze the translation that was defined in previous sections. As stated before, this translation gives a semantics to module nets in terms of graph grammars. Thus, we have to guarantee that the translation procedure generates a valid graph grammar (well-definedness) and always terminates with an unique resulting grammar (confluence). Confluence can be formally proven, since this is a property of the translation grammar.

6.3.1 Well-definedness

We begin by laying out the following requirements for a final graph to encode a well-defined graph grammar:

R1 every edge must have source and target nodes.

R2 for every edge with source and target nodes, and included in a graph, its source and target nodes must be included in the same graph.

R3 every rule must have left handside and right handside graphs.

Note that requirements R1 and R2 are slightly different, whereas R2 requires source and target to be in the same graph, it says nothing about edges missing source or target. R1 covers this case, requiring that all edges have a source and a target.

For each of the requirements above, we argue that they are preserved by the translation grammar rules:

Preservation of R1 we analyse rules looking for the creation of "GRAGRA_edge" nodes:

- rules TR15, TR16, TR20 through TR23, TR25, TR26 and TR28 create "GRAGRA_edge" nodes. All of these rules add source and target nodes.
- no other rule creates "GRAGRA_edge" nodes.
- no rule deletes sources or targets of "GRAGRA_edge" nodes.

Preservation of R2 let us analyse all rules in the translation grammar, looking for rules that create "GRAGRA_in" edges from "GRAGRA_edge" nodes to "GRAGRA_graph" nodes (note that no rule removes "GRAGRA_in" edges):

- TK1 through TK11, TR12 through TR14 and CL29 through CL33 do not contain "GRAGRA_edge" nodes.

- TR15 and TR16 create "GRAGRA_edge" nodes, but they neither create nor remove "GRAGRA_in" edges.
- TR17 through TR23 and TR26 create "GRAGRA_in" edges, in all of those rules, for every "GRAGRA_edge" node that is included in a graph, its source and target "GRAGRA_node" nodes are included in the same graph, thus preserving R1.
- TR24 places just the sources of "GRAGRA_edge" nodes (of type "resource-of") in different graphs, but the targets of these "GRAGRA_edge" nodes are "GRAGRA_node" modules added by TR23.
- TR27 is analogous to TR24, where TR26 adds the missing targets (modules).
- TR25 places just the sources of "GRAGRA_edge" nodes (of type "attribute-of") in different graphs, but the targets of these "GRAGRA_edge" nodes are "GRAGRA_node" resources added by TR24.
- TR28 is analogous to TR25, where TR27 adds the missing targets (resources).
- AD34 and AD35 remove redundant "GRAGRA_in" edges.

Preservation of R3 rules that create "GRAGRA_rule" nodes are TR17 through TR23 and TR26. All such rules add left- and right-handside graphs to rules created. No rule removes a graph from a "GRAGRA_rule".

Based on the arguments above, we believe that the translation creates well-defined graph grammars. In addition to being a graph grammar, we also require that the translation produces a grammar that can be used for the verification procedure. We lay down following additional requirements for a well-defined verification grammar:

R4 every node has a single role, which is one of the following: a module, a resource, an attribute or a value.

R5 *required* resources and attributes create rules of type *require*, which induce critical pairs of dependencies with rules that create such resources or attributes.

R6 *generated* resources and attributes create rules of type *generate*, which induce critical pairs of dependencies with rules that need such resources or attributes and conflicts with rules that change their values.

R7 *non-generated* resources and attributes create rules of type *mockgenerate*, just as rules created by S2, the only difference being the rule name.

R8 operations create two rules each, *call* and *return*, with resource and attribute edges translated as mappings from left- to right-handside graphs of rules.

For each of the additional requirements above, we argue that they are preserved by the translation grammar rules:

Preservation of R4 let us analyse each role separately, keeping in mind that no rule adds source or target to existing edges, in other words, edges are always created with their sources and targets, as observed before when showing the preservation of R1:

- Modules are the targets of "resource-of" edges, which are created by TR15. This rule ensures the target of "resource-of" is a module due to its mapping from a "MN_Module".
- Resources are the sources of "resource-of" edges, and TR15 ensures the source of "resource-of" is a resource due to its mapping from a "MN_Resource".
- Resources are also the targets of "attribute-of" edges, which are created by TR16. This rule ensures the target of "attribute-of" is a resource due to its mapping from a "MN_Resource".
- Attributes are the sources of "attribute-of" edges, and TR16 ensures the source of "attribute-of" is an attribute due to its mapping from a "MN_Attribute".
- Attributes are the targets of "value-of" edges, which are created by TR20, TR21, TR22, TR25 and TR28. All of these rules ensure the target of "value-of" is an attribute due to its mapping from a "MN_Attribute".
- Values are the sources of "value-of" edges, and TR20, TR21, TR22, TR25 and TR28 ensure the source of "value-of" is a value because they either create their sources "V" or have the value "V" as source (which is created by TR13)

Preservation of R5 rule TR17 creates rules for required resources, and TR20 for required attributes

Preservation of R6 rule TR18 creates rules for generated resources, and TR21 for generated attributes

Preservation of R7 rule TR19 creates rules for non-generated resources, and TR22 for non-generated attributes

Preservation of R8 rule TR23 creates rules for operations calls, and TR26 for operation returns

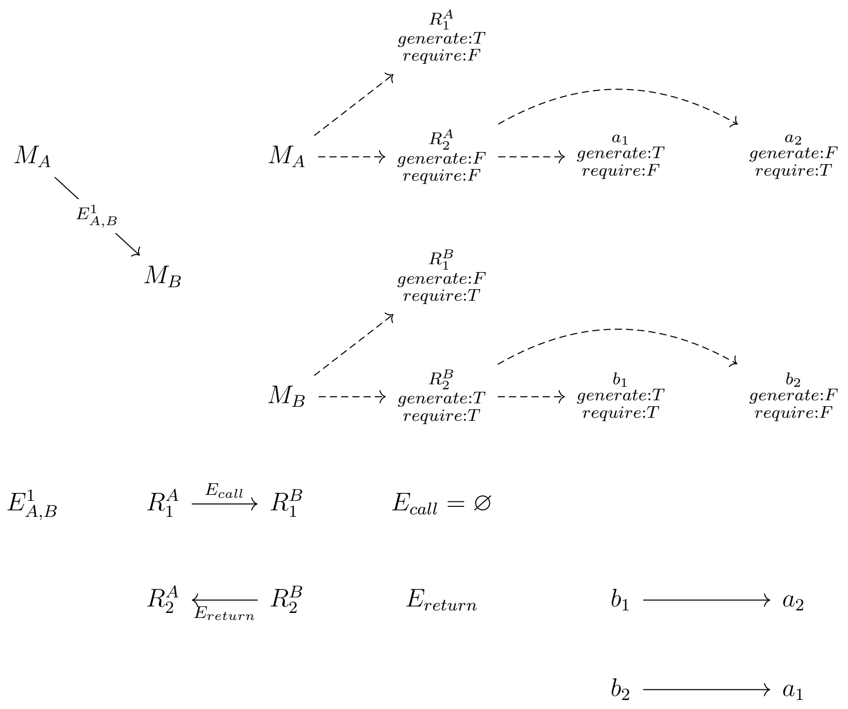
With all of the above, we argue that the translation produces well-defined verification grammars. In the following let us put this into practice with an actual translation. First we build the following atomic constraints (the actual constraints are available in the appendix) that can be mapped back to requirements:

- **A1** (*GRAGRA_edge_has_source_target*): checks that every edge has source and target nodes. When this is fulfilled, it shows R1.
- **A2** (*GRAGRA_edge_source_target_in_graph*): checks that for every edge in a graph, its source and target are in the same graph. When this is fulfilled, it shows R2.
- **A3** (*GRAGRA_rule_has_lhs_rhs*): checks that every rule has left handside and right handside graphs. When this is fulfilled, it shows R3.
- **A4** (*MN_module_is_not_just_module*): checks that a "GRAGRA" node that is a module is also a resource, attribute or value. When this is not fulfilled, it shows R4.
- **A5** (*MN_resource_is_not_just_resource*): checks that a "GRAGRA" node that is a resource is also a module, attribute or value. When this is not fulfilled, it shows R4.
- **A6** (*MN_attribute_is_not_just_attribute*): checks that a "GRAGRA" node that is an attribute is also a module, resource or value. When this is not fulfilled, it shows R4.
- **A7** (*MN_value_is_not_just_value*): checks that a "GRAGRA" node that is a value is also a module, resource or attribute. When this is not fulfilled, it shows R4.
- **A8** (*MN_self_resource*): check that a "GRAGRA" node is a resource of itself. When this is not fulfilled, it shows R4.
- **A9** (*MN_self_attribute*): check that a "GRAGRA" node is an attribute of itself. When this is not fulfilled, it shows R4.
- **A10** (*MN_self_value*): check that a "GRAGRA" node is a value of itself. When this is not fulfilled, it shows R4.

For each of the atomic constraints, we have built test graphs such that a test graph fulfills an atomic constraint that is not to be fulfilled or, the opposite, it does not fulfill an atomic that is to be fulfilled. These test graphs are only meant to test each of the atomic constraints.

Now we build a module net MN_2 as depicted in Figure 6.7. We use the translation procedure previously described to obtain the verification grammar depicted in Figures 6.8 and 6.9. The final graph obtained in this translation is omitted because it is huge with 75 nodes and 248 edges. MN_2 serves as a good example, because it contains all combinations of values for generate and require (T and T, T and F, F and T, F and F) for resources and attributes, and it also contains an operation edge without attribute edges

Figure 6.7: Module net MN_2.

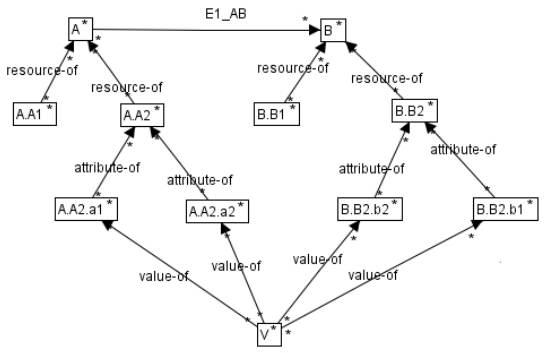

M_A
$E^1_{A,B}$
M_B
R^A_1 generate:T require:F
M_A
R^A_2 generate:F require:F
a_1 generate:T require:F
a_2 generate:F require:T
R^B_1 generate:F require:T
M_B
R^B_2 generate:T require:T
b_1 generate:T require:T
b_2 generate:F require:F
$E^1_{A,B}$
$R^A_1 \xrightarrow{E_{call}} R^B_1$
$E_{call} = \varnothing$
$R^A_2 \xleftarrow[E_{return}]{} R^B_2$
E_{return}
$b_1 \longrightarrow a_2$
$b_2 \longrightarrow a_1$

(E_{call}) and an operation edge with multiple attribute edges (E_{return}).

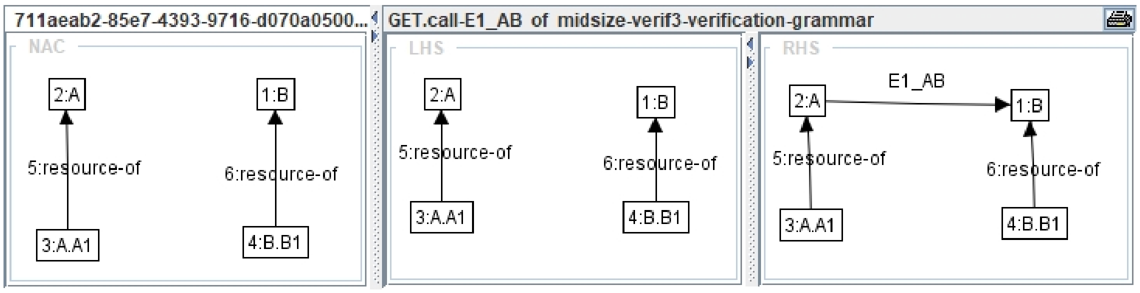
With the atomic constraints we build the constraint: **(A1) AND (A2) AND (A3) AND (NOT A4) AND (NOT A5) AND (NOT A6) AND (NOT A7) AND (NOT A8) AND (NOT A9) AND (NOT A10)**. Using AGG (RUNGE; ERMEL; TAENTZER, 2012), it is possible to see that the final graph for the translation of MN_2 fulfills this constraint, thus the verification grammar produced is well-defined in respect to requirements R1 through R4.

Since requirements R5 through R8 are not covered by the atomic constrains, we now analyse the rules produced for MN_2 to show these requirements are fulfilled. First we show R5 by noticing the required resources R^B_1 and R^B_2, and required attributes a_2 and b_1 on Figure 6.7 and the corresponding rules *require-B.B1*, *require-B.B2*, *require-A.A2.a2* and *require-B.B2.b1* from Figure 6.9. We check R6 by comparing generated resources and attributes R^A_1, R^B_2, a_1, b_1 with rules *generate-A.A1*, *generate-B.B2*, *generate-A.A2.a1* and *generate-B.B2.b1*. R7 is confirmed by non-generated resources and attributes R^A_2, R^B_1, a_2 and b_2 with rules *mockgenerate-A.A2*, *mockgenerate-B.B1*, *mockgenerate-A.A2.a2* and *mockgenerate-B.B2.b2*. Finally, we show R8 with $E^1_{A,B}$ and rules $GET.callE1\_AB$ and $GET.returnE1\_AB$. Thus the translation of MN_2 is well-defined.

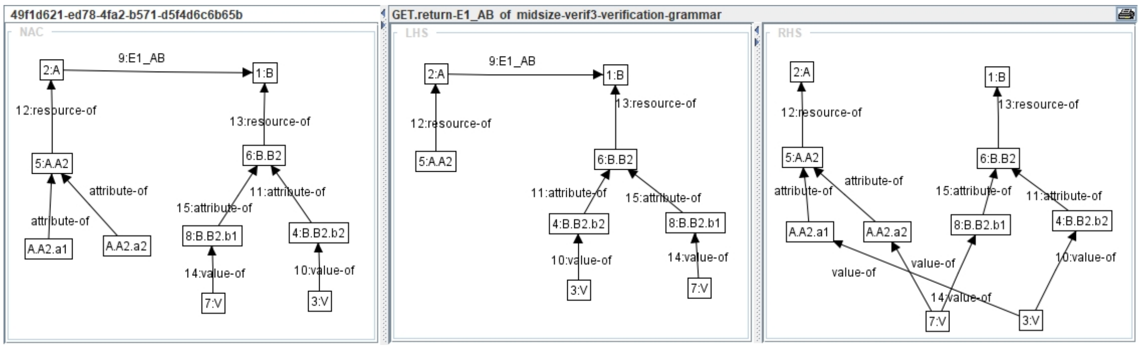
Figure 6.8: Verification grammar for module net MN_2



(a) Type graph

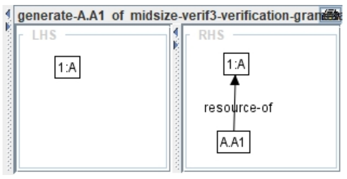


(b) *GET.call-E1_AB*

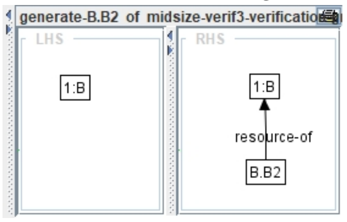


(c) *GET.return-E1_AB*

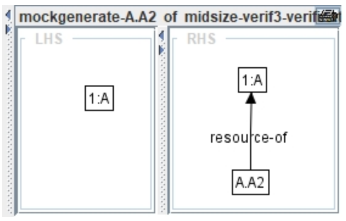
Figure 6.9: Remaining rules of verification grammar for module net MN_2



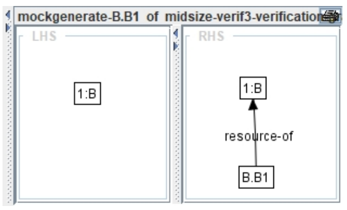
(a) *generate-A.A1*



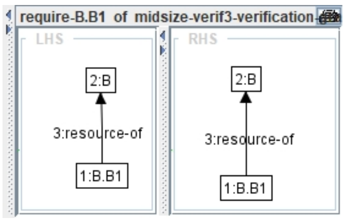
(b) *generate-B.B2*



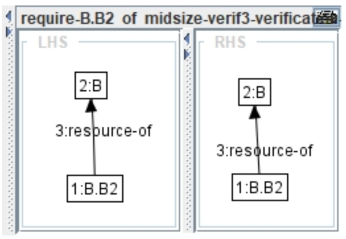
(c) *mockgenerate-A.A2*



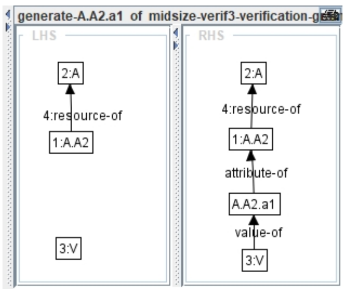
(d) *mockgenerate-B.B1*



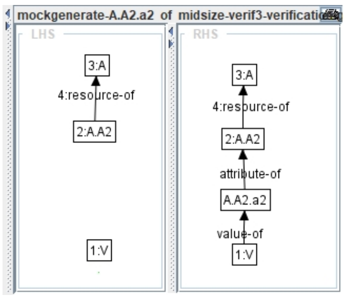
(e) *require-B.B1*



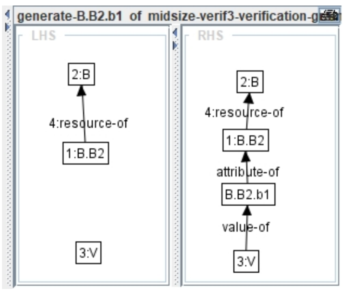
(f) *require-B.B2*



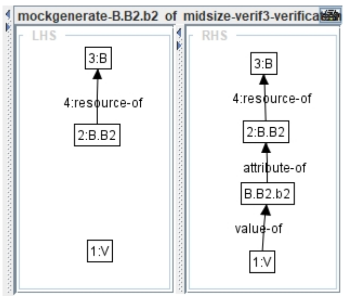
(g) *generate-A.A2.a1*



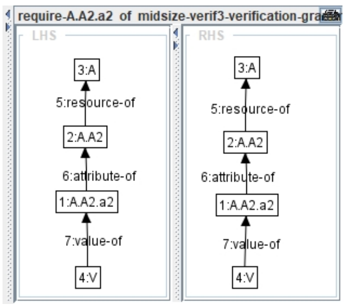
(h) *mockgenerate-A.A2.a2*



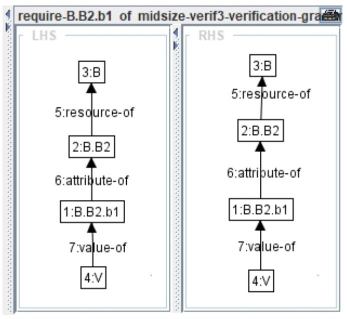
(i) *generate-B.B2.b1*



(j) *mockgenerate-B.B2.b2*



(k) *require-A.A2.a2*

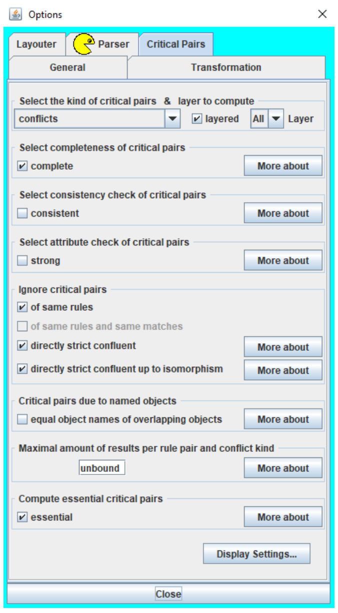


(l) *require-B.B2.b1*

6.3.2 Confluence

As explained previously in Chapter 3, confluence is a combination of two properties, namely termination and local confluence. Luckily, and differently than we did for well-definedness, we can show both of these properties automatically using AGG. We start by showing local confluence, configuring the AGG analysis as shown in Figure 6.10 (a). We ignore critical pairs of same rules, directly strictly confluent critical pairs and mark the "essential" option to compute just essential critical pairs. We choose these options because they do not influence the local confluence result, and because they greatly improve analysis time. With such configurations, no critical pairs of conflicts are generated for the translation grammar, and therefore the translation grammar is locally confluent.

Figure 6.10: CPA options and termination of translation grammar.



Options

Layouter | Parser | Critical Pairs

General | Transformation

Select the kind of critical pairs & layer to compute
conflicts ☑ layered All Layer

Select completeness of critical pairs
☑ complete

Select consistency check of critical pairs
☐ consistent

Select attribute check of critical pairs
☐ strong

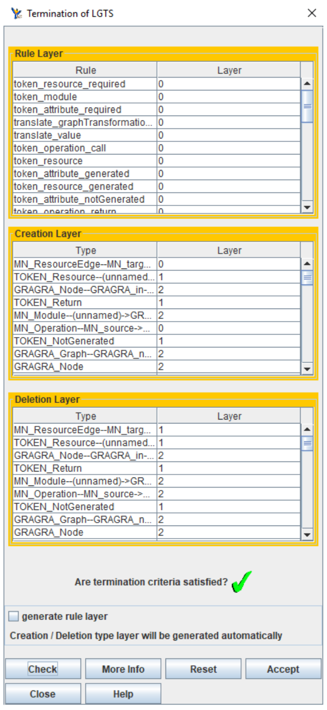
Ignore critical pairs
☑ of same rules
☐ of same rules and same matches
☑ directly strict confluent
☑ directly strict confluent up to isomorphism

Critical pairs due to named objects
☐ equal object names of overlapping objects

Maximal amount of results per rule pair and conflict kind
unbound

Compute essential critical pairs
☑ essential

(a) Critical pairs analysis configuration



Termination of LGTS

Rule Layer

Rule	Layer
token_resource_required	0
token_module	0
token_attribute_required	0
translate_graphTransformatio...	0
translate_value	0
token_operation_call	0
token_resource	0
token_attribute_generated	0
token_resource_generated	0
token_attribute_notGenerated	0
token_operation_return	0

Creation Layer

Type	Layer
MN_ResourceEdge--MN_targ...	0
TOKEN_Resource--(unnamed...	1
GRAGRA_Node--GRAGRA_in-...	2
TOKEN_Return	1
MN_Module--(unnamed)->GR...	2
MN_Operation--MN_source->...	0
TOKEN_NotGenerated	1
GRAGRA_Graph--GRAGRA_n...	2
GRAGRA_Node	2

Deletion Layer

Type	Layer
MN_ResourceEdge--MN_targ...	1
TOKEN_Resource--(unnamed...	1
GRAGRA_Node--GRAGRA_in-...	2
TOKEN_Return	1
MN_Module--(unnamed)->GR...	2
MN_Operation--MN_source->...	2
TOKEN_NotGenerated	1
GRAGRA_Graph--GRAGRA_n...	2
GRAGRA_Node	2

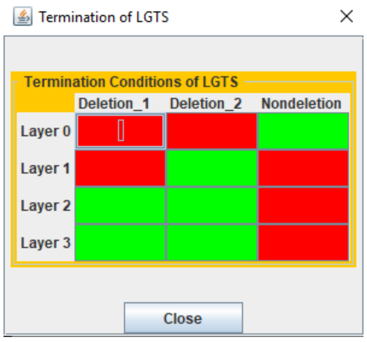
Are termination criteria satisfied? ✓

☐ generate rule layer
Creation / Deletion type layer will be generated automatically

(b) Proof of termination

To ensure that the translation grammar terminates, we first assign layers to rules as previously explained in Table 6.1. With this configuration, we run the termination analysis of AGG, which finds that the grammar terminates according to Figure 6.10 (b). In the following, we present and explain the results obtained by AGG when analysing termination. Figure 6.12 shows the layer kinds AGG finds for each rule layer, where a green box means the layer represented by that row fulfills criteria for the layer kind

Figure 6.12: Termination layers.



Termination Conditions of LGTS

	Deletion_1	Deletion_2	Nondeletion
Layer 0	🟥	🟥	🟩
Layer 1	🟥	🟩	🟥
Layer 2	🟩	🟩	🟥
Layer 3	🟩	🟩	🟥

represented by the column (so for example layer 0 fulfills criteria to be a non-deletion layer) and a red box means that layer does not fulfill criteria for the layer kind represented by the column (so again layer 0 fulfills neither criteria set 1 nor criteria set 2 to be a deletion layer).

Deletion layers of the first criteria set (Deletion_1 in Figure 6.12) decrease the amount of nodes or edges in graphs, and are eventually halted for lack of elements to delete. By looking at the rules in the appendix, note that rules CL29 through CL33 of layer 3, and AD34 and AD35 of layer 4 decrease the amount of nodes or edges in graphs.

Deletion layers of the second criteria set (Deletion_2 in Figure 6.12) delete elements previously created, and like the first criteria set are eventually halted for lack of elements to delete. By looking at the rules in the appendix again, note that all rules of layer 1, TR14 through TR28, either delete "TOKEN" nodes created by layer 0, or delete "MN" nodes which are not created anywhere. Also note that rules of layer 1 increase the amount of nodes or edges in graphs, which is why this layer does not fulfill the first criteria set for deletion layers.

Finally, non-deletion layers are guaranteed to terminate due to Negative Application Conditions (NACs). By looking at the rules in the appendix, note that all rules of layer 0 have NACs, which are TK1 through TK11, TR12 and TR13 rules, so it makes sense that layer 0 is a non-deletion layer.

Tables 6.3, 6.4, 6.5 and 6.6 show the creation and deletion layers assigned by AGG to each of the types in the translation grammar. We note that for each type, either its creation layer precedes its deletion layer or they are the same. As an example, "MN_Resource" has a creation layer of 0 and a deletion layer of 2, but the creation and deletion layers of "TOKEN_Return" are the same layer 1. In other words, no type has a creation layer greater than its deletion layer, which would imply the translation does not terminate.

Table 6.3: Node type creation layers as created by AGG.

Layer	Types
0	MN_Operation, MN_Resource, MN_ResourceEdge, MN_AttributeEdge, MN_Module, MN_Attribute,
1	TOKEN_Return, TOKEN_NotGenerated, TOKEN_Attribute, TOKEN_Call, GRAGRA_GraphTransformationSystem, TOKEN_Module, TOKEN_Generated, TOKEN_Required, TOKEN_Resource,
2	GRAGRA_Node, GRAGRA_Graph, GRAGRA_Edge, GRAGRA_Rule,

Table 6.4: Node type deletion layers as created by AGG.

Layer	Types
1	TOKEN_Return, TOKEN_NotGenerated, MN_ResourceEdge, TOKEN_Attribute, TOKEN_Call, GRAGRA_GraphTransformationSystem, MN_AttributeEdge, TOKEN_Module, TOKEN_Generated, TOKEN_Required, TOKEN_Resource
2	GRAGRA_Node, MN_Operation, MN_Resource, GRAGRA_Graph, GRAGRA_Edge, MN_Module, GRAGRA_Rule, MN_Attribute

[1]GTS abbrv. for GraphTransformationSystem

Table 6.5: Edge type creation layers as created by AGG.

Layer	Source type	Edge type	Target Type
0	MN_ResourceEdge	MN_target	MN_Resource,
	MN_Operation	MN_source	MN_Module,
	MN_AttributeEdge	MN_edge	MN_Operation,
	MN_resourceEdge	MN_edge	MN_Operation,
	MN_AttributeEdge	MN_target	MN_Attribute,
	MN_Attribute	MN_attributeof	MN_Resource,
	MN_Resource	MN_resourceof	MN_Module,
	MN_AttributeEdge	MN_source	MN_Attribute,
	MN_Operation	MN_target	MN_Module,
	MN_ResourceEdge	MN_source	MN_Resource
1	TOKEN_Resource	(unnamed)	MN_Resource,
	TOKEN_Attribute	(unnamed)	MN_Attribute
	TOKEN_Call	(unnamed)	MN_Operation,
	TOKEN_NotGenerated	(unnamed)	MN_Resource,
	TOKEN_Required	(unnamed)	MN_Attribute,
	TOKEN_Return	(unnamed)	MN_Operation,
	TOKEN_Required	(unnamed)	MN_Resource,
	TOKEN_Module	(unnamed)	MN_Module,
	TOKEN_NotGenerated	(unnamed)	MN_Attribute,
	TOKEN_Generated	(unnamed)	MN_Attribute,
	TOKEN_Generated	(unnamed)	MN_Resource
2	GRAGRA_Node	GRAGRA_in	GRAGRA_Graph,
	MN_Module	(unnamed)	GRAGRA_Node,
	GRAGRA_Graph	GRAGRA_nac	GRAGRA_Rule,
	GRAGRA_Edge	GRAGRA_target	GRAGRA_Node,
	MN_Operation	call	GRAGRA_Rule,
	MN_Operation	return	GRAGRA_Rule,
	GRAGRA_Graph	GRAGRA_lhs	GRAGRA_Rule,
	MN_Resource	(unnamed)	GRAGRA_Node,
	GRAGRA_Graph	GRAGRA_rhs	GRAGRA_Rule,
	MN_Attribute	(unnamed)	GRAGRA_Node,
	GRAGRA_Edge	GRAGRA_in	GRAGRA_Graph,
	GRAGRA_Rule	GRAGRA_rule	GRAGRA_GTS[1],
	GRAGRA_Edge	GRAGRA_source	GRAGRA_Node

Table 6.6: Edge type deletion layers as created by AGG.

Layer	Source type	Edge type	Target Type
1	MN_ResourceEdge	MN_target	MN_Resource,
	TOKEN_Resource	(unnamed)	MN_Resource,
	MN_AttributeEdge	MN_edge	MN_Operation,
	MN_resourceEdge	MN_edge	MN_Operation,
	TOKEN_Attribute	(unnamed)	MN_Attribute,
	MN_AttributeEdge	MN_target	MN_Attribute,
	TOKEN_Call	(unnamed)	MN_Operation,
	TOKEN_NotGenerated	(unnamed)	MN_Resource,
	TOKEN_Required	(unnamed)	MN_Attribute,
	TOKEN_Return	(unnamed)	MN_Operation,
	MN_AttributeEdge	MN_source	MN_Attribute,
	MN_ResourceEdge	MN_source	MN_Resource,
	TOKEN_Required	(unnamed)	MN_Resource,
	TOKEN_Module	(unnamed)	MN_Module,
	TOKEN_NotGenerated	(unnamed)	MN_Attribute,
	TOKEN_Generated	(unnamed)	MN_Attribute,
	TOKEN_Generated	(unnamed)	MN_Resource
2	GRAGRA_Node	GRAGRA_in	GRAGRA_Graph,
	MN_Module	(unnamed)	GRAGRA_Node,
	MN_Operation	MN_source	MN_Module,
	GRAGRA_Graph	GRAGRA_nac	GRAGRA_Rule,
	MN_Attribute	MN_attributeof	MN_Resource,
	GRAGRA_Edge	GRAGRA_target	GRAGRA_Node,
	MN_Operation	call	GRAGRA_Rule,
	MN_Resource	MN_resourceof	MN_Module,
	MN_Operation	return	GRAGRA_Rule,
	MN_Operation	MN_target	MN_Module,
	GRAGRA_Graph	GRAGRA_lhs	GRAGRA_Rule,
	MN_Resource	(unnamed)	GRAGRA_Node,
	GRAGRA_Graph	GRAGRA_rhs	GRAGRA_Rule,
	MN_Attribute	(unnamed)	GRAGRA_Node,
	GRAGRA_Edge	GRAGRA_in	GRAGRA_Graph,
	GRAGRA_Rule	GRAGRA_rule	GRAGRA_GTS,
	GRAGRA_Edge	GRAGRA_source	GRAGRA_Node

6.4 Limitations and final remarks

We had to face many challenges when implementing the translation. The first challenge was the node and edge explosion. Considering just the two examples we presented in this chapter:

- MN_1 had just two modules, two resources, no attributes and a single operation with two resource edges, but its final graph had 24 nodes and 69 edges.
- MN_2 had again two modules, four resources, four attributes and a single operation with two resource egdes and two attribute edges, but its final graph had 75 nodes and 248 edges.

It is easy to see that the translation scales badly. Based on translation rules (see appendix) we can draw the following conclusions:

- Each rule created has at least three nodes (GRAGRA_Rule and GRAGRA_Graphs) and three edges (GRAGRA_rule, GRAGRA_lhs and GRAGRA_rhs).
- Require, generate and mockgenerate rules have none / one[2] additional node (GRAGRA_Edge) and 6 / 14 more edges (GRAGRA_in) worst case for resources / attributes respectively. For each resource and attribute, one or two rules are created, depending on whether it is required or not.
- Each operation implies the creation of two rules with an additional node (GRAGRA_Edge) and 7 additional edges (GRAGRA_in).
- Each resource edge implies the creation of 10 edges (GRAGRA_in).
- Each attribute edge implies the creation of three nodes (GRAGRA_Node and GRAGRA_Edges) and 21 edges (mostly GRAGRA_in).
- Each module implies the creation of one node (GRAGRA_Node).
- Each resource and each attribute imply the creation of two nodes (GRAGRA_Node and GRAGRA_edge) and two edges (GRAGRA_source and GRAGRA_target).

Based on the above, given a module net with **M** modules, **R** resources, **A** attributes, **O** operations, **RE** resource edges and **AE** attribute edges we can estimate the maximum number of nodes in the final graph $\mathbf{MAX}_{nodes} = \mathbf{M} + 8 * \mathbf{R} + 10 * \mathbf{A} + 8 * \mathbf{OP} + 3 * \mathbf{AE}$ and maximum number of edges $\mathbf{MAX}_{edges} = 20 * \mathbf{R} + 36 * \mathbf{A} + 20 * \mathbf{OP} + 10 * \mathbf{RE} + 21 * \mathbf{AE}$ Note this is not a precise measure, but an upper bound, because some edges are removed

[2]for require and generate/mockgenerate respectively, generate and mockgenerate differ only by trigger condition (value of "mock" attribute) and generated rule name

by AD* rules and some rules are not created if resources / attributes are not required. If we compute this for the examples we have:

- $\mathbf{MAX}_{nodes}(MN_1) = 26 > 24$ and $\mathbf{MAX}_{edges}(MN_1) = 80 > 69$
- $\mathbf{MAX}_{nodes}(MN_2) = 88 > 75$ and $\mathbf{MAX}_{edges}(MN_2) = 308 > 248$

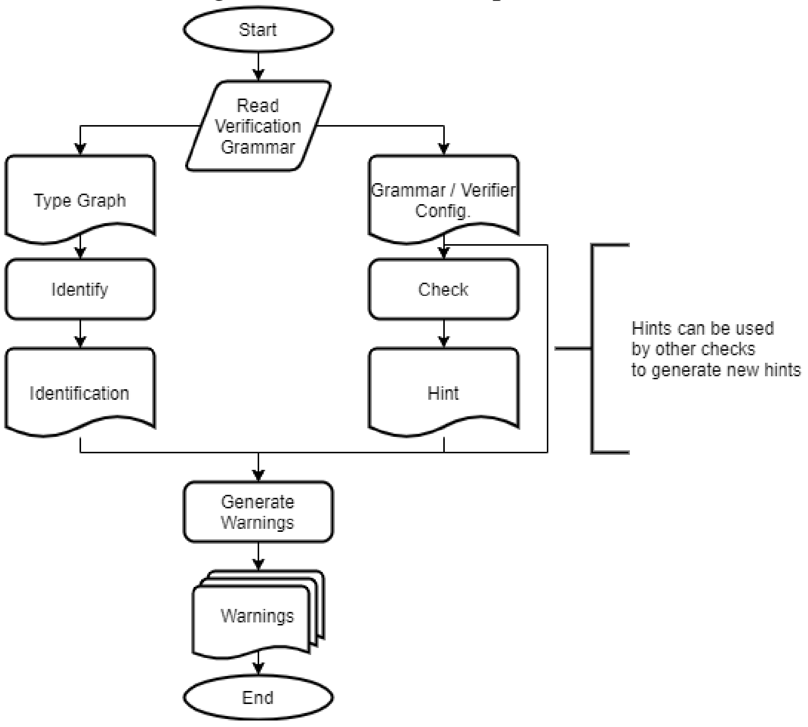
This rapid growth in the number of nodes and even more in the number of edges led us to consider performance improvements. One improvement we did during development was to get rid of type graphs. During initial phases, the translation would create a GRAGRA_Graph node to represent type graphs, just as it does for rule left- and right-handside and NAC graphs. This would contribute to the enourmous growth in edges, as every node would require a GRAGRA_Edge from itself to the type graph. We were able to improve this by introducing the "type" attribute and then get rid of the type graph GRAGRA_Graph node. Another improvement we did during the initial development phase was to reuse GRAGRA_Node nodes, instead of creating new nodes. Each time one had to be added to a GRAGRA_Graph, we just created a GRAGRA_in arrow from an existing node to that GRAGRA_Graph.

Despite these changes, the amount of nodes and edges is still too high. Even worse is that each additional node and edge makes the translation take longer, because it increases the amount of matches a rule has to a graph. For module nets just slightly bigger than MN_2, the translation slows down to the point of stagnation. To avoid this scenario, we use a divide and conquer approach: before translating big module nets, we split a module net into a series of smaller module nets, each with just one operation. The module nets are then translated and later they are joined together during the extraction procedure. From an outside view, the result of the translation is still the same, it takes as input a module net and produces a verification grammar.

7 A VERIFICATION METHOD FOR SOFTWARE INTEGRATION

The verification step takes place after translation. This step takes as input a verification grammar and produces a set of warnings. In more details, it performs a set of checks against the verification grammar and a verifier configuration, producing a set of hints. It also creates an identification that allows the mapping of verification structures back to module nets. Combining hints and identification, the verification procedure makes a set of warnings, which are displayed to users. Figure 7.1 illustrates this procedure.

Figure 7.1: Verification procedure


Start
Read Verification Grammar
Type Graph
Grammar / Verifier Config.
Identify
Check
Identification
Hint
Hints can be used by other checks to generate new hints
Generate Warnings
Warnings
End

Provided a verification grammar and verifier configuration, the verification procedure is automatic. The verification procedure presented here is finished, in the sense that it is ready to use, but it is open, that is, this verification procedure can be easily extended to support other checks and warnings.

7.1 Identification

The identification maps the structure of a verification grammar back to the module net that originated it. This procedure is essential to produce meaningful warnings to users.

Although it is similar to translation, identification is not the same as translation. When translating, we start with a model and end with another model in a different language, whereas identification produces a dictionary of structures in the source model to structures of the target model.

Another difference between identifications and translations is that an identification is partial, that is, it does not produce a full module net. Instead, an identification produces just enough information to make warnings meaningful to an user. In short, identifications do not allow the recovery of the original module net that was translated into a verification grammar.

7.1.1 Graph identification

Graph identification is the only identification procedure currently implemented. This procedure takes as input the type graph of a verification grammar and walks the graph producing mappings from node and edge IDs to module net names.

The source of a mapping is a node or edge of the verification grammar type graph and the target of the mapping is a module net name. Some structures of the source do not have counterparts in the target. The creation of mappings is straightforward:

- Values are nodes which are the source of an edge named "value-of". Values do not have a module net counterpart and thus are ignored.
- Attributes are nodes which are the source of an edge named "attribute-of". The names of such nodes are the names of attributes in a module net.
- Resources are nodes which are the source of an edge named "resource-of". The names of such nodes are the names of resources in a module net. Additionally, all "attribute-of" edges with this resource as target assign attributes to this resource.
- Modules are all other nodes. The names of such nodes are the names of modules in a module net. Additionally, all "resource-of" edges with this module as target assign resources to this module.

Graph identification can be implemented by a simple loop over graph edges, which identifies edge targets as modules, resources, attributes or values (based on edge names as explained above), and edge sources as resources or attributes. Graph identification has a complexity of $O(E_{TG})$ over a type graph with edge set E_{TG}.

7.2 Hints

Hints are produced by the verification algorithm when it analyses a graph grammar. A hint is a piece of information that can lead to another hint or to a warning. By itself, a hint is not useful to the user. Here is a brief description of dependencies between hints:

Table 7.1: Dependencies between hints and warnings

Hint	Used in Hint	Used in Warning
Critical pairs	all hints except rule decoration	-
Rule decoration	all hints except critical pairs	Unreachable operation
Information flow	Required path	-
Optional path	-	Optional operation, module, resource and attribute
Required path	-	Strictly optional attribute
Reachable rule	-	Unreachable operation
Critical pair explanation	-	Dangling resource and outdated attribute

7.2.1 Critical pairs hint

This hint takes a verification grammar and applies a critical pair analysis, yielding pairs of dependencies and conflicts, as well as a set of overlays for each critical pair. This hint is used by several other hints described shortly.

7.2.2 Rule decoration hint

Rule decoration takes a verification grammar, extracts its set of rules, and decorates each rule with a set of properties based on rule pattern. The exact decoration for each pattern is determined by the verifier configuration. These are the decoration properties:

- Rule pattern (string): determined based on verifier configuration and rule name, this is used when computing critical pair explanation hints.
- Mock (boolean): indicates whether this rule represents a mock or a real property of the module net when computing reachable rules hints.

- Maps to operation (boolean): this decorator is used to filter out auxiliary rules when computing unreachable operation warnings.
- Required by default (boolean): indicates whether a rule is required when computing optional and required path hints.
- Modules, resources, attributes (arrays): these arrays contain the names of module net structures in this rule and are used throughout the verification.

Rule decoration hint can be implemented with a loop over the rules of the verification grammar. The values of each decorator are determined by the rule pattern, which in turn is determined by a regular expression applied on rule name. Finding modules, resources and attributes in rules is $O(N_R)$ with N_R being the number of nodes in all graphs of a rule, provided we already have a graph identification, which saves the time of going through edges.

Rule decoration has a complexity of $O(R * N_R)$ over a verification grammar rule set with R rules with names that have a fixed amount of characters.

7.2.3 Information flow hint

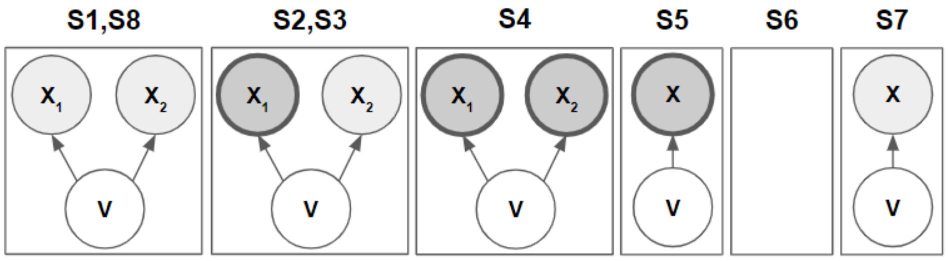
Information flow takes the produce-use dependency critical pairs of a verification grammar and attaches pairs of attributes to these dependencies, such that the information flows from the first to the second component of the pair. For example, supposing the we have an information flow pair X_1 and X_2, this means that:

- Source rule of the dependency pair sets the value of X_2 to the value of X_1 (or a combination of this and values of other attributes).
- Target rule of the dependency requires the value of X_2.

Information flow pairs are then useful later when computing required paths, where we will use information flow pairs to check which attributes carry important information to required attributes.

Figure 7.2 shows possible subgraphs of critical pair overlays. Nodes X_1 and X_2 represent attributes, while node V is a value node. Attributes marked in dark grey are critical objects, that is, they belong both to source as well as to target rule, and attributes marked in light grey belong either to source or to target rule. Codes S1 to S8 are references to Table 7.2, which shows how to process each of the subgraphs from Figure 7.2. We denote $X_1 \to X_2$ if there is information flow from X_1 to X_2.

Figure 7.2: Possible subgraphs of critical pair overlays and correspondence to Table 7.2



Note that critical pairs are potential dependencies, which means that the information flows shown here are potential, but they may not happen, depending on how each of the rules matches the host graph. Nevertheless, we can still use this information because we are looking for attributes which are never used to carry information, these are the attributes that are not part of any information flow pairs.

Table 7.2: Information flow in subgraphs of critical pair overlays

Code	Source rule	Target rule	Information flow
S1	X_1	X_2	There is no information flow.
S2	X_1	X_1, X_2	It is possible that $X_1 \to X_2$
S3	X_1, X_2	X_1	It is possible that $X_2 \to X_1$
S4	X_1, X_2	X_1, X_2	It is possible that both $X_1 \to X_2$ and $X_2 \to X_1$
S5	X_1	X_1	There is no information flow.
S6	$\varnothing$	$\varnothing$	There is no information flow.
S7	X_1	$\varnothing$	There is no information flow, the same swapping source/target.
S8	X_1, X_2	$\varnothing$	There is no information flow, the same swapping source/target.

Information flow can be implemented with a loop of overlays. For each overlay, we find all its subgraphs from Figure 7.2, by first looping over the nodes of the overlay to find value nodes and then looping over nodes connected to value nodes. So far this is the same complexity for traversing V graphs, which is $O(V * (N_V + E_V))$ with V overlays of N_V nodes and E_V edges.

Once we have the list of all attributes connected to a specific value, we then take all pairs of attributes in that list. In general this is $O(l^2)$ for a list of size l. This adds to our complexity so far making $O(V * (E_V + N_V^2))$ (note we have removed the N_V addend).

Now for each component of a pair, we determine whether or not it is in source and target rules, which is $O(N_{Rs} + N_{Rt})$, but we simplify to $O(N_R)$ with N_R the greatest of the two, and we use this information to compute the information flow according to table above. This makes $O(V * (E_V + N_V^2 * N_R))$.

In addition to the complexity of finding critical pairs, finding information flows has a complexity of $O(V * (E_V + N_V^2 * N_R))$ with V overlays of N_V nodes and E_V edges, between rules of N_R nodes.

7.2.4 Optional path hint

Optional path takes the produce-use dependency critical pairs as well as decorated rules and produces lists of nodes and rules which are not necessary. This list of optional nodes and rules can later be identified back to module net structures with the help of an identification. Such nodes and rules are later presented as warnings to the user, so that they can decide whether or not it makes sense to keep these structures in their modules.

Optional paths could be for instance operations which have been used in the past but now are deprecated. As soon as a deprecated operation is not used anymore, it can be removed.

Optional path is implemented with a traversal of the produce-use critical pairs graph. Rules are considered required when they have the required by default decorator. Rules which are the source of produce-use critical pairs with required rules as target, are considered required as well. Nodes which are not part of any required rules are optional. Cycles are handled as follows:

- An isolated cycle without required rules is optional.
- An isolated cycle with required rules is not optional.
- A cycle with rules that are not part of a path taking to a required rule is optional.
- A cycle with a rule that is part of a path that takes to a required rule is not optional.

Optional path is a simple graph traversal of $N_{CP} + E_{CP} = O(R^2)$ for critical pair graphs with N_{CP} nodes and E_{CP} edges.

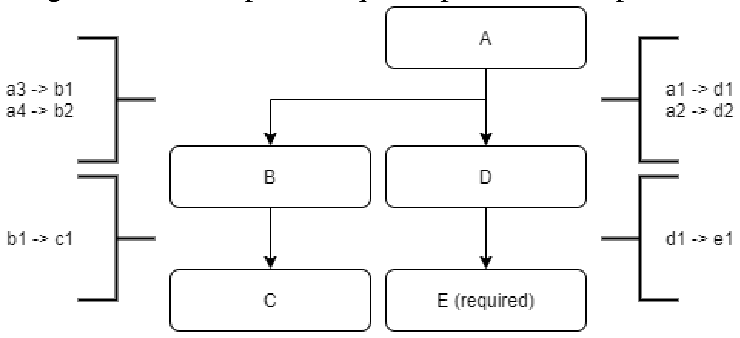
7.2.5 Required path hint

Required path takes information flows as well as decorated rules and produces a list of attributes considered to be required. The goal of this hint is to find attributes which carry information to required attributes. Note that the set of required attributes is not necessarily the complement of optional attributes over the universe of attributes.

The attributes of required rules which do not carry information to required attributes are named strictly optional attributes.

Required path is implemented with a loop over all decorated rules of a verification grammar, filtering those rules which are required by default. Then, we perform a traversal of the produce-use critical pairs graph starting from each of the required rules and going backwards. That is, in the first traversal iteration we will look for critical pairs with rules required by default as a target. The second iteration will look for rules whose target is the result of the first iteration and so on. If we take Figure 7.3 as an example, we would start by looking at rule E, which is required by default, then go backwards to rule D and stop this iteration. The next iteration will start at rule D and work backwards to rule A and stop, as there is nowhere else to go.

Figure 7.3: Example of required path hint computation



During each iteration of the traversal, for each critical pair that is found, we will look for the information flow pairs of that critical pair and start building a path that takes to a required rule. Only attributes in the path that takes to a required attribute are considered required. Going back to Figure 7.3 example, in the first iteration we will note down attribute d1, in the second iteration we note down attribute a1. Attributes belonging to information flow pairs which are not part of any such paths are not considered required. This is the case of a2, a3, a4 and d2 in the path to E, as well all attributes of B and C.

To summarize the example presented in Figure 7.3, we have a produce-use critical pair graph with five rules. The only rule decorated with required by default is rule E. The brackets indicate pairs of information flows between attributes belonging to each rule. We can see that rules B and C, as well as attributes b1, b2 and c1 are optional, because they are not in any required path. Rules A and D are required, as they lead to a required rule (E). Attributes a1 and d1 are also required, because they belong to an information flow path leading to attribute e1, which is required. Attributes a2 and d2 are neither required nor optional, they are strictly optional attributes.

In the worst case scenario, we will traverse the entire critical pairs graph[1], which is $N_{CP} + E_{CP} = O(R^2)$, with N_{CP} and E_{CP} nodes and edges in the critical pairs graph and R rules. In addition to that, we will search information flow list, which makes $O(R^2 + P)$ with P information flow pairs.

7.2.6 Reachable rule hint

Reachable rule takes the produce-use and produce-forbid critical pairs of a verification grammar, as well as its set of decorated rules, and produces a set of rules whose left handside requirements can be met, that is, a set of reachable rules. This hint operates on a few assumptions:

- For each attribute and resource in the module net, verification grammar will have a rule that produces it. We will name these rules as generators for convenience.
- If a resource/attribute is generated in the module net, its generator will be decorated "mock: false", otherwise it will be decorated "mock: true".
- Rules are built in such a way that they create produce-use dependencies for each attribute/resource they require, with generator rules as source.
- Rules are built in such a way that they create produce-forbid conflicts for each attribute/resource they produce, with generator rules as source.

Reachable rule hint can be implemented with a loop over all rules, running the whole algorithm for each single rule. When trying to determine if a rule is reachable, it will traverse the critical pairs graph and look for produce-use dependencies with generators as source and the analysed rule as a target. If there is no such dependency, then the rule is assumed to be reachable. If there is such a dependency, then either:

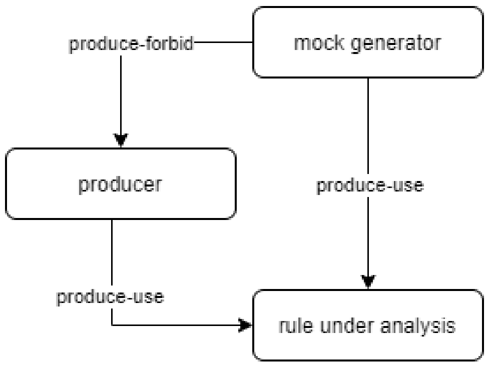
- The generator is not a mock. This is the trivial case, the rule requires a resource or attribute which is generated, everything is fine.
- The generator is a mock. In this case, we need to find some other rule that produces the resource or attribute this rule needs. If we cannot find a producer rule, then the rule under analysis is not reachable.

If the rule we are analysing is the target of a produce-use dependency from a mock generator, we will have to find some rule that produces the resource/attribute this rule needs.

[1]Note that here we mean there is at most one edge between nodes of critical pairs graph, we use overlay count to distinguish between critical pairs of same rules

This is achieved by looking for produce-forbid conflicts with that same mock generator as source. Figure 7.4 illustrates this situation. In general, if we can find that produce-forbid conflict, its target will be such a rule that a produce-use dependency from this rule to the rule currently under analysis exists. The reasoning is simple: that rule produces an attribute/resource the rule currently under analysis needs.

Figure 7.4: Example of attribute or resource not generated by the module net, but still generated by some operation of the module net



Reachable rule hint has a complexity of $2 * (N_{CP} + E_{CP}) = O(R^2)$ with R rules in the verification grammar and N_{CP} and E_{CP} nodes and edges in the critical pairs graph.

7.2.7 Critical pair explanation hint

A critical pair explanation takes the critical pairs of a verification grammar and the verifier configuration and produces a list of hints. Some of the possible verifier configurations for this hint are:

- Critical pairs of delete-use conflicts between rules with pattern operation can be explained by dangling resources. The dangling resource is the critical object in the delete-use conflict.
- Critical pairs of produce-use dependencies between rules with pattern operation or between a generator and an operation can be explained by outdated attributes. The outdated attribute is the critical object in the produce-use dependency.

Additionally, critical pair explanations can also be extended to neighbor rules in the critical pairs graph. For example, if a resource is considered dangling due to a delete-use conflict between rules R1 and R2, and rule R3 reads that same resource, then a new dangling resource hint is created also for R3, even though there is no delete-use conflict between R1 and R3.

Critical pair explanation can be implemented by a traversal of the critical pairs graph. Matching critical pairs to a configuration is simple, provided we already have the rule pattern (from rule decoration hint) and critical objects from the critical pair analysis. Checking whether the critical object is a resource or attribute is also a simple operation using the identification we have already computed. The complexity of critical pair explanation is a simple graph traversal of $N_{CP} + E_{CP} = O(R^2)$ with N_{CP} and E_{CP} nodes and edges in the critical pairs graph and R rules in the verification grammar.

7.3 Warnings

Warnings are the result of the verification and contain valuable information to users. As such, when creating warnings, we must combine hints with identifications to produce meaningful messages. There are different kinds of warnings and the following subsections explain each one of them.

7.3.1 Optional attribute, resource, module or operation warning

This warning is given when the verification is able to demonstrate that there is no scenario where this structure enables a required feature of the module net. To name a few of such scenarios:

- A trivial example of optional attribute is an attribute which is never read and also not required by the module net.
- A more elaborate optional attribute example is one such that its value is passed on to another attribute, but that attribute is a trivial optional attribute.
- An example of optional operation is one that does not contribute to reaching a required structure.
- Figure 7.3 shows optional operations B and C, optional modules B and C, optional resources B.B1 and C.C1 and optional attributes B.B1.b1, B.B1.b2 and C.C1.c1.

An optional structure is potentially a structure that was important historically, but now it is deprecated and can be removed without affecting the required information flow of the network. In that sense, this warning helps users identify which structures they may want to remove from their module interfaces.

Optional structures warning is derived from optional path hints. Optional path hints related to rules give rise to optional operation warnings, and optional path hints related to nodes may create optional attribute, resource or module warnings, depending on the identification of the node.

7.3.2 Strictly optional attribute warning

Strictly optional attribute warnings point out attributes which are in the path of required attributes, but do not contribute directly to the information flow. Just as optional attributes, strictly optional attributes are also structures that can be deprecated and eventually removed. An example of strictly optional attributes is depicted on Figure 7.3, where attributes A.A1.a2 and D.D1.d2 are strictly optional. Strictly optional attribute warnings are derived from required path hints.

7.3.3 Unreachable operation warning

Unreachable operation warnings are shown when it is impossible for an operation to be executed due to missing resources or attributes. This is not just the case that in a given situation the operation becomes impossible, but rather that it is never going to get executed. Examples of unreachable operations are:

- An operation that requires resource X.X1, but that resource is not generated.
- An operation that requires attribute X.X1.x1, which is not generated, but still it is created by another operation, which in turn is unreachable due to some other reason.

Unreachable operation warnings provide useful information for developers to detect issues in the information flow, which can be due to typos in resource or attribute names or even other kinds of failures in the specification. This kind of warning is derived from reachable rule hints.

7.3.4 Dangling resource warning

Dangling resource warnings are shown when a resource copied from module X to module Y is deleted from module X, leaving the Y copy of that resource dangling.

A dangling resource is not necessarily an issue, it is left for users to read and check whether each dangling resource can lead to issues in their specification. Dangling resource warnings are derived from critical pair explanation hints.

7.3.5 Outdated attribute warning

Outdated attribute warnings highlight attributes copied from module X to module Y and then changed in module X, leaving the Y copy of that attribute with an outdated value. An outdated attribute is not necessarily an issue, users must read and check whether each outdated attribute can lead to issues in their specification. Outdated attribute warnings are derived from critical pair explanation hints.

7.4 Complexity

Table 7.3 summarizes the complexity of the verification procedure. In particular, the time complexity of all warnings is the same: they follow straight from hints. Here are the meanings of each variable used:

- N_X, E_X: node and edge set of X
- R: set of rules
- P: set of information flow pairs
- V: set of overlays of a critical pair
- H: set of hints
- TG: type graph

Table 7.3: Time complexity of each step in the verification procedure.

Step	Type	Time complexity
Graph Identification	Identification	$O(\|E_{TG}\|)$
Rule Decoration	Hint	$O(\|R\| * \|N_R\|)$
Information Flow	Hint	$O(\|V\| * (\|E_V\| + \|N_V\|^2 * \|N_R\|))$
Optional Path	Hint	$O(\|R\|^2)$
Required Path	Hint	$O(\|R\|^2 + \|P\|)$
Reachable Rule	Hint	$O(\|R\|^2)$
Critical Pair Explanation	Hint	$O(\|R\|^2)$
Warning Generation	Warning	$O(1)$

Most of our hints are simple graph traversals, which are processed in linear time.

This complexity is good, considering that this is not a time critical application, and users may tolerate to wait a little longer. The biggest concern is with critical pairs computation, which we believe to be non-polynomial time, and on top of that is required for all of our hints and warnings.

7.5 Limitations and final remarks

Warnings available today are very limited and some are very easy to spot, such as an optional attribute or resource, which is just an attribute or resource that is not seen in any operations neither is it required. As mentioned in Chapter 5, we could expand the supported warnings by augmenting module nets with types, providing the information we need to better analyse module nets.

There is also a concern with the dangling resource warnings, which are just not possible with the current translation procedure, as dangling resource requires deletion of resources, which the current translation procedure does not support. Also outdated attribute warnings will always show when an attribute is generated or set by an operation and then read by an operation. We believe in most of the cases this will not be useful information, and it would be much more useful if we could couple this with some control flow information to infer whether or not the kinds of concurrency issues we want to avoid really can happen.

Another limitation is complexity. All of our hints and warnings are based on critical pairs computation, and this analysis can take a lot of time and memory, when we analyse large grammars, in terms of quantity of rules and graph sizes of those rules. There are two ways we can approach this: we could first try to reduce rule size by improving our translation procedure, or we could adapt verification to base it on faster critical pair analysis configurations, such as essential critical pairs computation.

8 CASE STUDY

In this chapter, we review the concepts presented so far in a case study that serves as an end-to-end example, showing that our framework can be implemented. We do not approach the subject of scalability in this chapter.

In this case study, let us consider the example of a search engine to find research articles and let us call this engine Research Net. Our engine is responsible for searching for scientific articles, but it does not contain the actual research articles, which are stored in an Inventory Service. When visiting the Research Net website, researchers can look for scientific articles by providing their Digital Object Identifier (DOI) and clicking a "submit" button. This button sends the provided DOI to Research Net, which then computes the ID of the article using the DOI provided and relays the request to its Inventory Service to find its Location. An example of a source code for Research Net is depicted in Figure 8.1.

Figure 8.2 shows module net V_1, which stands for the version one of our example Research Net system. This V_1 contains three modules that cascade information. Attribute *DOI* is generated in resource *Search* of module *Website*, and cascaded to attribute *ID* in resource *Article* of module *ResearchNet*, and then again to attribute *Location* in resource *Document* of module *InventoryService*. With this module net we mean that the DOI provided by the user is sent from the Website to Research Net, which then uses the DOI to compute an ID. The ID is then relayed to the inventory service, which possesses the actual article.

In V_1 we have marked *DOI* with $generate : T$, because this input is provided by the user, who is an entity external to our module net. Differently than *DOI*, ID is

Figure 8.1: Pseudocode for Research Net V_1, functions $read$, $find$ and $retrieve$ are part of external modules

```
function readSearch() {
  DOI := read();
  FindArticle(DOI);
}
```

(a) Website module

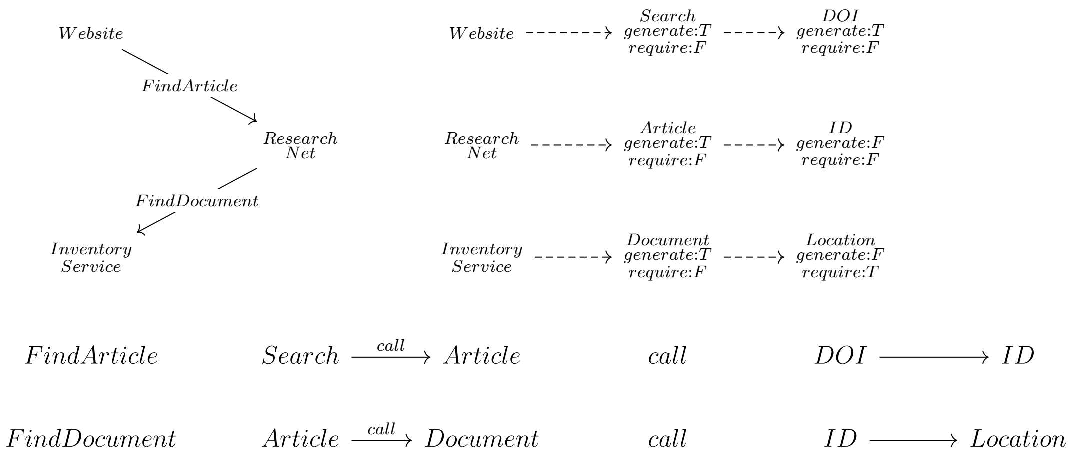
```
function FindArticle(doi) {
  id := find(doi);
  FindDocument(id);
}
```

(b) Research Net module

```
function FindDocument(id) {
  location := "res/" + id;
  retrieve(location);
}
```

(c) Inventory Service module

Figure 8.2: Example module net V_1



$generate : F$, because it requires *DOI* to be computed, and the same goes for *Location*, which requires *ID* to be computed. We mark all resources $generate : T$, as we assume they are always available.

In this module net, we have not modeled the return path, where the article is returned back to the user, and for that reason we mark the *Location* attribute with $required : T$, meaning that this attribute is passed to some other operation external to our module net.

8.1 Outdated attribute warnings in research net V_1

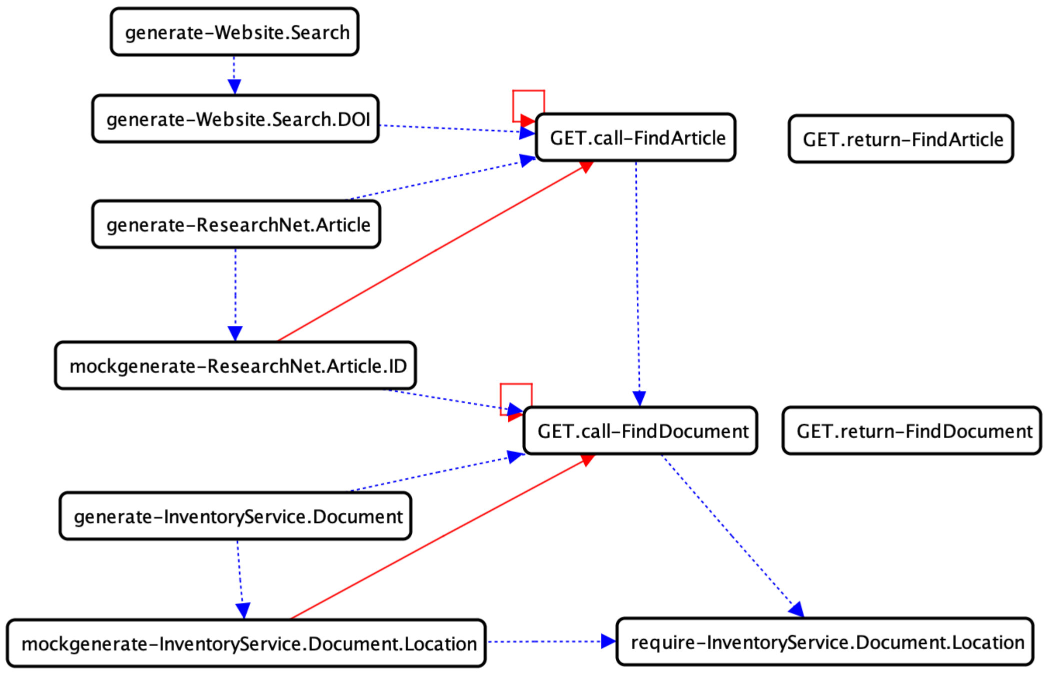
When translated with our MIGRATE prototype, module net V_1 turns into a graph grammar with eleven rules, all of which are in the critical pairs analysis (CPA) graph in Figure 8.3. Looking at the graph we notice the produce-use dependency from *generate-Website.Search.DOI* to *GET.call-FindArticle*, and because we know *DOI* is generated by the first rule (based on the name of that rule), we can conclude[1] *DOI* is the product of first rule that is used by the second rule. In addition to that, because this is a produce-use dependency between a rule of type generator and a rule of type operation, we have found a critical pair explanation hint and *DOI* is an outdated attribute.

If we run the verification procedure it will find just three warnings:

- Attribute *Website.Search.DOI* is an *outdated-attribute* because of rules *generate-Website.Search.DOI* and *GET.call-FindArticle*, meaning that *generate-Website.Search.DOI*

[1]This reasoning is not valid for graph grammars in general, but it works for verification grammars due to how we have built and named rules of such grammars.

Figure 8.3: Critical pair analysis graph for research net V_1


generate-Website.Search
generate-Website.Search.DOI
GET.call-FindArticle
GET.return-FindArticle
generate-ResearchNet.Article
mockgenerate-ResearchNet.Article.ID
GET.call-FindDocument
GET.return-FindDocument
generate-InventoryService.Document
mockgenerate-InventoryService.Document.Location
require-InventoryService.Document.Location

can change the value of *Website.Search.DOI* after it has been read by *GET.call-FindArticle*

- Rules *GET.return-FindArticle* and *GET.return-FindDocument* are *optional-rules*, because they do not contribute to reaching any required attributes or resources

The interpretation we give to the warnings above is that a user may decide to look for DOI *00.0000/fizz* and submit this search, but shortly after that, user changes their mind and sends a new search for *00.0000/buzz*, all that while ResearchNet is still trying to find *00.0000/fizz*.

8.2 Unreachable operation warnings in research net V_2

If we were to change V_1 and create a new module net for version two, V_2, where everything is the same, except that attribute *Website.Search.DOI* is not generated, then we would get a CPA graph very similar to that of Figure 8.3, except that in place of rule *generate-Website.Search.DOI*, we now have *mockgenerate-Website.Search.DOI*. With such, we note that we now have a situation with a produce-use dependency from a mock rule M (*mockgenerate-Website.Search.DOI*) to some other rule R (*GET.call-FindArticle*) and we must find a rule P such that $M \rightarrow_{PF} P \rightarrow_{PU} R$. We note that there is no such P and thus rule *GET.call-FindArticle* is unreachable. This causes a cascade effect on rules that depend on *GET.call-FindArticle* using the same reasoning we just did. This time we have a few different warnings:

- Operation $FindArticle$ is a *unreachable-operation*, because it needs attribute *Website.Search.DOI* to execute, but that attribute is not obtainable anywhere in V_2
- Operation $FindDocument$ is a *unreachable-operation*, because it needs attribute *ResearchNet.Article.ID* to execute, but that attribute is not obtainable anywhere, attribute *ResearchNet.Article.ID* used to be obtained by $FindArticle$, but that operation is not reachable in V_2
- Rule *require-InventoryService.Document.Location* is a *unreachable-operation*, because this attribute is not obtainable anywhere
- Just as before, rules *GET.return-FindArticle* and *GET.return-FindDocument* are *optional-rules*, because they do not contribute to reaching any required attributes or resources

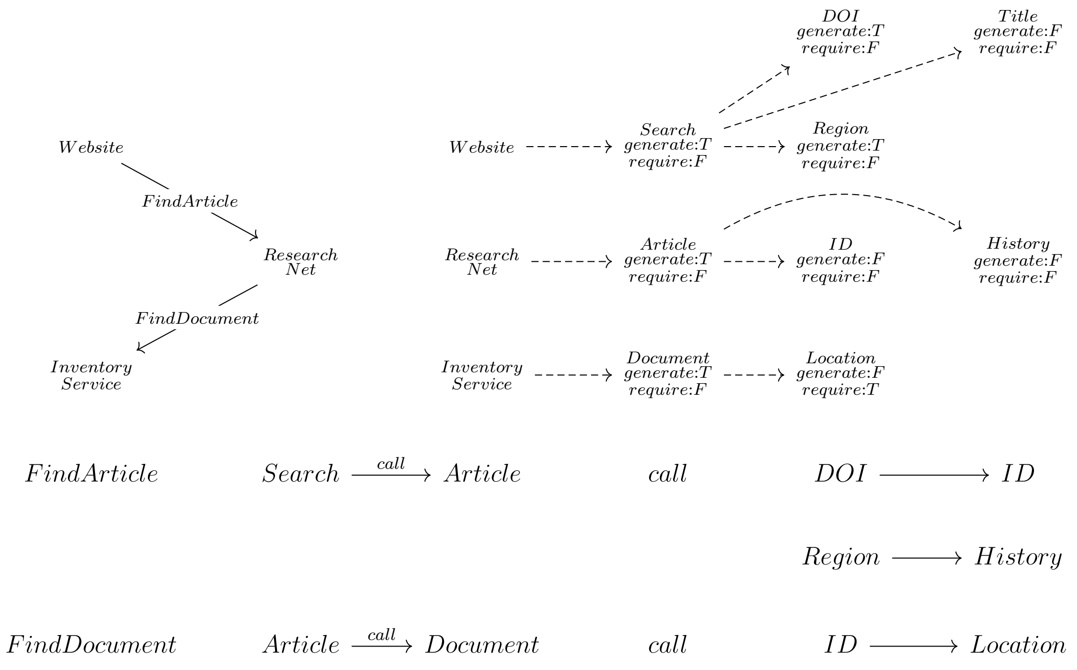
This means that Research net cannot work without users, because the entire search engine is triggered by user demand. If we were to remove user input, as we did in V_2, then MIGRATE prototype warns us that we lack the user input necessary to perform our operations. In addition to that, we are not able to send articles to users anymore (*require-InventoryService.Document.Location* is a *unreachable-operation*).

8.3 Optional attribute warnings in research net V_3

Figure 8.4 shows module net V_3, which is a modified version of V_1 where we have added two new attributes to module *Website*, *Website.Search.Title* (meaning the title of article that users want to search) and *Website.Search.Region* (meaning the geographical place where users are), and one new attribute to module *ResearchNet*, *ResearchNet.Article.History* (meaning a history of regions of users that have accessed this article).

If we look at the CPA graph for V_3 in Figure 8.5, we notice that rule *mockgenerate-Website.Search.Title* is not source of any dependencies or conflicts, meaning this attribute is completely isolated, therefore it must be optional, since it does not contribute to anything. Now for the other attributes, we notice in particular that *Region* and *History* are in a path to reaching rule *require-InventoryService.Document.Location*, so they could be necessary. For that reason, we analyse the information flow hint for this CPA graph:

Figure 8.4: Example module net V_3



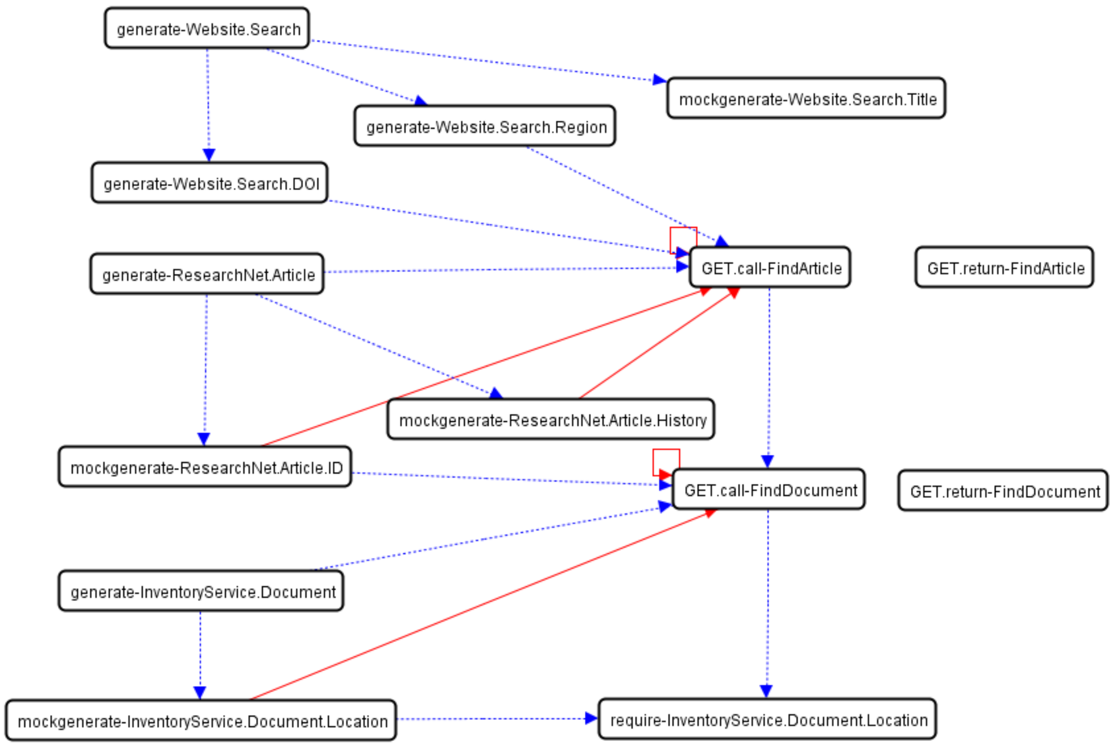
- Information flows from *DOI* to *ID* due to produce-use dependency from *GET.call-FindArticle* to *GET.call-FindDocument*
- Information flows from *ID* to *Location* due to produce-use dependency from *GET.call-FindDocument* to *require-InventoryService.Document.Location*

Looking at information flow hints, we realize neither *Region* nor *History* is necessary, thus these attributes are *strictly-optional-attributes*.

This time we have ten warnings:

- Attribute *Website.Search.Title* is both *optional-attribute* and *strict-optional-attribute*, because it is neither required nor used anywhere
- Attribute *Website.Search.Region* is a *strict-optional-attribute*, because although it is used in a operation that contributes to a required attribute (*InventoryService.Document. Location*), attribute *Region* is not necessary to reach that attribute neither is it required
- Attribute *ResearchNet.Article.History* is a *strict-optional-attribute*, for the same reasons as *Website.Search.Region*
- Like in V_1, attribute *Website.Search.DOI* is a *outdated-attribute* because of rules *generate-Website.Search.DOI* and *GET.call-FindArticle*, meaning that *generate-Web site.Search.DOI* can change the value of *Website.Search.DOI* after it has been read by *GET.call-FindArticle*

Figure 8.5: Critical pair analysis graph for research net V_3


generate-Website.Search
mockgenerate-Website.Search.Title
generate-Website.Search.Region
generate-Website.Search.DOI
generate-ResearchNet.Article
GET.call-FindArticle
GET.return-FindArticle
mockgenerate-ResearchNet.Article.History
mockgenerate-ResearchNet.Article.ID
GET.call-FindDocument
GET.return-FindDocument
generate-InventoryService.Document
mockgenerate-InventoryService.Document.Location
require-InventoryService.Document.Location

- Rules *mockgenerate-Website.Search.Title* and *mockgenerate-ResearchNet.Article. History* are *optional-rules*, because they do not contribute to reaching any required attributes or resources
- Just as before, rules *GET.return-FindArticle* and *GET.return-FindDocument* are *optional-rules*, because they do not contribute to reaching any required attributes or resources

By looking at warnings above, we realize that although we have declared attribute *Title*, we have not used it anywhere, so it really is not necessary. In addition to that, we have added attributes *Region* and *History*, and we have used *Region* to compute the value of *History*, but we have not used *History* in any way (it is not required), thus although we compute this information, it is also not necessary.

In V_3, we saw optional attributes. The difference between them and non-optional attributes is that optional attributes were neither required nor did they contribute directly to reach some required attribute. To prevent optional attribute warnings, we could have simply marked these attributes as required, but what makes an attribute required? In this example, we choose to set $required : T$ for those attributes that are handed over to some function, which does not have a counterpart in the module net.

9 RELATED WORK

Unlike other approaches, this work tackles the problem of integration from a perspective of modules, independently of whether a module is a library, service or something else. Because related work aims strictly to either library integration or service integration, we separate these two in different sections here. We also have a third section for the applications of graph grammars, which are used extensively in this work. Throughout this section we distinguish static from dynamic approaches, where by static we mean approaches that do not require code execution and by dynamic we mean approaches that do.

9.1 Integration of Libraries

Dynamic checks with tests: don't-break is a JavaScript module that uses automated tests of dependencies to determine whether an update to a library breaks its dependencies (BAHMUTOV, 2018). In (MUJAHID et al., 2020) the authors applied this same approach to ten faulty versions of different JavaScript modules and successfully identified the breaking changes in six of ten. Automated tests can be used either by dependency developers to find out if their changes to a module break dependencies or by consumers of a dependency who want to update a dependency, while making sure that it does not break their software. Developers often commit code that breaks tests and thus it can be difficult to find dependency tests on which to rely (MUJAHID et al., 2020).

NOREGRETS and its successor NOREGRETS+ (MøLLER; TORP, 2019) use consumer tests to create models of data flow together with type information between a library and its consumers. The models can then be used to check that each call to an updated library function still returns the same type and that the updated library still uses the same subset of arguments it used before the update. NOREGRETS+ can output false positives, for example, when a library introduces a new configuration argument and reads this new argument, thus changing the subset of arguments it used before the update (MøLLER; TORP, 2019).

A drawback of test-based approaches is the requirement of having tests. Tests can have low code coverage and reach only a few functions of dependencies, which imposes a requirement for a huge number of consumers and consumer tests in order to have meaningful results. In other words, the approach of using tests is better advised for libraries which are widely used and whose clients have many tests and use the whole interface of their dependencies. Even when code coverage is high, unit tests tend to mock dependencies, which makes these tests less useful for the kinds of integration issues we want to find. Another drawback of approaches based purely on tests is that, although tests may indicate that something is broken, they usually do not point out exactly the cause.

Static checks with diffs: Veracode is a verification tool that statically checks software to determine which dependencies can be updated to fix vulnerabilities. In (FOO et al., 2018) the authors present the Veracode algorithm to determine if the update of a dependency is safe to apply. The algorithm first computes the diff between current and target dependency version, consisting of inserted, deleted or changed methods. A few other types of changes are used to combine version updates and produce a diff between arbitrary versions, not just single updates. Next, Veracode computes the call graph of consumer code and matches it to the computed diff to determine whether the changes in the target version break consumer code. A major drawback of this approach is that it needs two versions to compute a diff, which gets worse when we consider transitive dependencies (FOO et al., 2018). Additionally, due to hashes used to determine whether a function was changed, simple syntactic changes can give rise to false positives (FOO et al., 2018).

APIDIFF (BRITO et al., 2018a) is another static check tool that looks for syntactic changes to library interfaces. Specifically, APIDIFF lays out three elements that can suffer changes, namely types, methods and fields, and for each element APIDIFF can detect a series of breaking and non-breaking changes, such as adding final modifier to a type or removing a method or field. In opposition to Veracode, APIDIFF does not address changes to the implementation of libraries (BRITO et al., 2018a). Additionally, APIDIFF can present false positives if the breaking changes are found in internal or external elements that are not used by clients (BRITO et al., 2018a).

Static checks with symbolic execution: (MORA et al., 2018) presents a tool called CLEVER with a new concept of "client-specific equivalence", which disregards changes to signatures and concentrates on behavior equivalence. This approach takes consumer code and two library versions to determine whether the updated version keeps the behavior the previous version had, when consumed by the client provided. CLEVER works by applying symbolic execution to both pairs of (client, previous lib version) and (client, new lib version) and determining whether code activated by client has been changed in the new lib version considering input constraints provided by the symbolic execution. If code has been changed, CLEVER tries to find counterexamples where the behavior of the new lib version is different from the previous lib version. CLEVER requires library interfaces to remain unchanged between versions compared (MORA et al., 2018). Moreover, due to the symbolic execution engines underneath, CLEVER only supports variables of integer types (MORA et al., 2018). There are many symbolic execution tools available for use. The authors of CLEVER claim that their approach is better than others to solve integration issues because CLEVER only considers paths of integration between consumer and library, whereas other tools take all paths into consideration, solving a much larger problem than needed (MORA et al., 2018).

Table 9.1 summarizes the approaches listed so far. Approaches based on diffs determine whether or not a change was made to the implementation. Whereas diff checks cannot handle the behavior of functions, testing approaches are able to verify that the behavior is the same, but only for those cases covered by tests. In contrast, symbolic execution can cover every possible path by computing constraints on inputs imposed by client code. In comparison, our approach does not require automated tests and we suggest extracting module nets from source code directly. Unlike other approaches, we do not reason about the behavior of functions such as control flow, but rather we concentrate on data flow to find issues such as whether or not a module finds the data it needs in order to call an operation of another module.

Table 9.1: Approaches found in related works to evaluate library integration.

Execution	Analysis	Languages	Name	Reference
Dynamic	Testing	JavaScript	-	(MUJAHID et al., 2020)
Dynamic	Models	JavaScript	NOREGRETS+	(MøLLER; TORP, 2019)
Static	Diffs	Java, Python, Ruby	Veracode	(FOO et al., 2018)
Static	Diffs	Java	APIDIFF	(BRITO et al., 2018a)
Static	Symbolic Execution	Python	CLEVER	(MORA et al., 2018)
Static	Graph Grammar / CPA	None so far	MIGRATE	this work

9.2 Integration of Services

Perhaps the most similar work to ours is (HAUSMANN; HECKEL; LOHMANN, 2004), which couples class diagrams (representations of ontologies) with graph grammars[1] that specify the behavior of services. With such specifications:

[1]Graph grammars in that approach are linked to operations that have names, arguments and return values. Arguments of such operations can be referenced in graph grammar rules. One key difference of the graph grammar formalism used in the cited work is that they have two kinds of negative application conditions: pre-conditions, which apply before rule application and post-conditions, which apply after rule application.

1. clients can leverage ontologies to generate type graphs, then specify requirements as graph grammar rules and match these requirements to service specifications to check if a service does what clients want (HAUSMANN; HECKEL; LOHMANN, 2004)
2. service registries can take service specifications in the form of graph grammars and generate and execute tests, for single operations as well as sequences of operations generated with critical pair analysis (CPA), ensuring service correctness prior to making that service available (HECKEL; MARIANI, 2005)
3. visual contracts, essentially graph grammars, which they define with a UML meta-model, are used as input to a tool that generates annotations containing java modeling language (JML) contracts to be evaluated during runtime, essentially monitoring that the application / service conforms to its specification (ENGELS et al., 2006)

All of the above was consolidated in (LOHMANN; MARIANI; HECKEL, 2007). In (KHAN; HECKEL, 2011) this approach was extended reducing the set of tests that need to run to find regression issues when evolving services. The key is to lay out traces, sequences of operations, and keep track of changes in critical pairs graph (KHAN; HECKEL, 2011). The approach was also further extended to generate test coverage criteria based on critical pair analysis (CPA) (KHAN; RUNGE; HECKEL, 2012).

The work mentioned above has many similarities to our work, such as the goal of checking the integration of services and achieving that goal using graph grammars, as well as pairing operations with graph grammar rules and even using AGG. Because the work above is so similar to ours, we list the following points of divergence:

- In their solution, models come first, which means that instead of spending time writing integration code, developers will specify intended behavior through graph grammar rules that can be translated back to ontologies. Our work puts models in background. First, developers write integration code and then we generate models automatically from code. Additionally, their work requires knowledge of graph grammars, while we provide tools to automate the verification process.
- Their solution uses a single graph grammar rule as representation for an operation, whereas we use two rules.
- They concentrate on services, while we created the notion of module nets, enabling verification of not only services but also libraries and anything else that can be considered a module.

- They use data flow for example in (HECKEL; MARIANI, 2005) to generate coverage criteria, interpreting def-use pairs based on graph grammar rules. We consider def-use pairs to extract models from source code.
- Whereas their goals are to match and test specifications, we generate a set of warnings.
- Perhaps most important is that even though the work above abstracts service implementation and handles it at the interface level, it still considers the behavior of service operations, including expressions to compute attribute values and control flow conditions. We only consider data flow, abstracting away everything else.

Dynamic checks: "Differential Regression Testing" for REST services is presented in (GODEFROID; LEHMANN; POLISHCHUK, 2020), where they compare the outputs of two tests with the same inputs looking for regressions. The authors suggest interacting with services through automatically generated clients (SDKs[2]) and thus they test the client/service integration. Furthermore, they describe two kinds of tests: (i) one that keeps the client version and varies the service version and (ii) one where client version varies. Whilst (i) looks for regressions on the service side, (ii) looks for regressions in the generated clients and specifications (GODEFROID; LEHMANN; POLISHCHUK, 2020). The authors highlight that in comparison to static diff approaches, testing ensures that faults are actually present and not just issues in the documentation (GODEFROID; LEHMANN; POLISHCHUK, 2020). An issue with this approach is that it is not a fully automatic solution, but instead, provides experts with diff files that need to be manually inspected. In general, an issue with testing approaches is shadowing, where a bug in a request prevents testing other requests that depend on the first one.

Static preventive approaches: Besides graph grammars, there are many other approaches to service integration problems. Relaxing signatures is suggested in (BOROVSKIY et al., 2009), where authors suggest that service designers keep their interfaces as generic as possible, thus preventing integration faults.

Static checks: In (GUINEA; SPOLETINI, 2011) the authors suggest a new language they call ISC (Interaction Sequence Charts) and provide an algorithm to evaluate the degree of compatibility of candidate services to replace another service. Like in our approach, they do not include expressions to calculate attribute values, but differently than we do, they do consider control flow and operation order.

[2]software development kits, usually automatically generated from service specifications such as OpenAPI / Swagger.

A diff approach has been proposed in (BECKER et al., 2008), where authors split service interfaces in types (operations and attributes) and models (classes and associations). They provide an algorithm for computing whether a new service version is backwards compatible or incompatible[3] based on a set of allowed diffs such as adding new methods or classes. Any diff that is not in that set causes the changes to be regarded incompatible (BECKER et al., 2008).

Table 9.2: Approaches found in related works to evaluate service integration.

Execution	Analysis	Languages	Reference
Dynamic	Test generation and execution	REST / HTTP	(GODEFROID; LEHMANN; POLISHCHUK, 2020)
Dynamic	Test generation and execution	Visual contracts	(HECKEL; MARIANI, 2005)
Dynamic	JML generation and monitoring	Visual contracts	(ENGELS et al., 2006)
Static	Specification matching	Visual contracts	(HAUSMANN; HECKEL; LOHMANN, 2004)
Static	Specification matching	ISC	(GUINEA; SPOLETINI, 2011)
Static	Diffs	any	(BECKER et al., 2008)
Static	Sign. relaxing (preventive)	any	(BOROVSKIY et al., 2009)
Static	Graph Grammar / CPA	Module nets	this work

9.3 Graph grammar applications

Our work can be divided into three phases: (i) transformation of code into module nets (model extraction); (ii) translation of module nets into graph grammars with graph grammars as the translation engine; and (iii) verification of module nets as graph grammars.

The model extraction (i) is concerned with the extraction of models from source code, where the output models may be graph grammars (in our case the output is a module net). Extraction of graph grammars from a Java code was first seen in (CORRADINI et al., 2004), although they used hypergraphs and only covered a fragment of the language, excluding features such as loops and arrays. They also highlight the existence of "control garbage", code paths that are never activated (CORRADINI et al., 2004). One of our goals in this work is to uncover some kinds of "control garbage".

In (DUARTE; RIBEIRO, 2017) the authors have translated Java code into graph grammars using code annotations to generate traces, and then using a tool previously built to generate context information out of traces. With this approach, they are able to extract dynamic information from traces, ensuring that paths analysed can actually occur, but possibly missing behaviors that do not occur in tests. The main drawback is that they included control flow in the translation, which increases largely the amount of rules. The

[3]The algorithm has also a third output value, undetermined, in case the version under analysis is lower than the previous version, which contradicts the backwards compatibility analysis goal.

authors suggest algorithms to merge rules (DUARTE; RIBEIRO, 2017). Our work does not include control flow.

The model transformation (ii) is concerned with the transformation of models from one language to another using graph grammars as the transformation engine. A desirable property for model transformation is functional behavior. Graph grammars can exhibit functional behavior as long as they are terminating and confluent, which is illustrated in (HECKEL; KüSTER; TAENTZER, 2002) with the transformation from statecharts to Communicating Sequential Processes (CSP). Graph grammars have also been used to transform class diagrams encoded in XMI format (XML metadata interchange) into entity-relationship diagrams in WebML format (Web Modeling Language) (TAENTZER; CARUGHI, 2006). An advantage of this approach is that it produces reversible transformations. The authors highlight as main disadvantage the fact that matching a rule to a graph is a NP-complete problem (TAENTZER; CARUGHI, 2006).

Triple graph grammars (TGG) are the more standard approach to model transformation, because they allow forward and backward transformations. Although TGGs exist since 1994, it was only in (EHRIG et al., 2007) that they were defined in terms of category theory, enabling their use in AGG. This same article is illustrated with an example transformation from class diagrams to entity-relation diagrams.

Graph grammars have been used to transform models of REST[4] services, from conversation based into interaction based models (HAUPT; LEYMANN; PAUTASSO, 2015). With the automatic transformation, the authors intend to allow easy specification on simple conversation models and code synthesis from more detailed interaction based models (HAUPT; LEYMANN; PAUTASSO, 2015). Because they want to enable code synthesis, they concentrate on REST implementation features, such as HTTP protocol and HATEOAS (hypermedia as the engine of application state), whereas our approach abstracts these features away into a module net which keeps just the information flow.

Table 9.3: Related works with graph grammars concerning model extraction (i) and model transformation (ii)

Purpose	Approach	Source	Target	Reference
(i)	Source code parsing	Java (fragment)	AGG	(CORRADINI et al., 2004)
(i)	Traces from tests	Java	AGG	(DUARTE; RIBEIRO, 2017)
(i)	Source code parsing	JavaScript	Module net	this work
(ii)	Derivation (AGG)	Statecharts	CSP	(HECKEL; KüSTER; TAENTZER, 2002)
(ii)	Derivation (AGG)	XMI	WebML	(TAENTZER; CARUGHI, 2006)
(ii)	Derivation (AGG, TGG)	class diagrams	ER diagrams	(EHRIG et al., 2007)
(ii)	Derivation (AGG)	REST / HTTP	REST / HTTP	(HAUPT; LEYMANN; PAUTASSO, 2015)
(ii)	Derivation (AGG)	Module net	Graph Grammar	this work

[4]representational state transfer (FIELDING, 2000)

The verification (iii) is concerned with the verification of systems specified using graph grammars, in particular such grammars model how a system behaves, with its rules representing state transitions in the system. We do not propose new verification methods for graph grammars, but instead we leverage existing methods to create our own verification algorithms.

AGG 2.0 is a full graph grammar engine and graphical user interface one of the few supporting critical pair analysis (CPA) (RUNGE; ERMEL; TAENTZER, 2012) along with Verigraph (COSTA et al., 2016). Many other tools exist which handle graph grammars differently, such as GROOVE.

Table 9.4: Related works with graph grammars concerning verification.

Approach	Name	Reference
Critical Pair Generation	AGG 2.0	(RUNGE; ERMEL; TAENTZER, 2012)
Critical Pair Generation	VeriGraph	(COSTA et al., 2016)
Critical Pair Interpretation	MIGRATE	this work

10 CONCLUSION

In this work we set out to address the problem of module integration using graph grammars. We have created a verification framework that allows us to generate warnings telling developers which integrations need their attention. We have provided a prototype that implements the translation and verification procedures of MIGRATE framework and we demonstrated these procedures in a case study.

MIGRATE framework imposes certain requirements on graph grammars it analyses, such as having specific rule names and rule patterns, as well as nodes and edges that can be interpreted back to operations, modules, resources, attributes and values. We needed a way to ensure graph grammars that we analyse meet such requirements, while at the same time providing developers with the ability to make changes to models directly. For those reasons, we have created module nets, which are a formalism to express how modules integrate to each other. However, we have not defined a semantics of module nets, instead we have created a translation procedure that assigns a (verification) graph grammar as semantics of a module net. This enabled us to leverage existing critical pairs analysis theory, and also will enable us to leverage all different analysis techniques that have been developed for graph grammars in future work. We have demonstrated how to translate module nets into verification grammars using graph grammars and proven that our translation procedure terminates, as well as argued that it creates well-defined verification grammars. We have leveraged critical pairs analysis of verification grammars to build a verification procedure that points out different kinds of warnings related to the structures in module nets.

We have many limitations to address in the future:

Extraction lacks an algorithm to extract module nets from source code

Module nets operations are interpreted as simple read operations, we cannot specify creations or deletions, and operation graphs do not allow edges between resources of the same module, which also constrains the warnings we support

Translation is difficult for small module nets, and unfeasible for bigger module nets, because the host graph grows with each application of a translation rule, which makes it harder and harder to find matches

Verification is built on top of critical pairs analysis, which is very costly

Warnings concentrate on simple flow of information and, in general, are not very useful

There are two major areas that will need our attention when extending the prototype and building a verifier tool using the concepts of this work: support of larger module nets in terms of memory and time, and meaning of warnings.

Larger module nets are specially difficult to handle during the derivation procedure applied in translation and the critical pairs computation. Considering our translation grammar is confluent, we could improve derivation by avoiding match randomization and always applying the first match we find. Critical pairs could be improved if we restricted it to the generation of essential (LAMBERS; EHRIG; OREJAS, 2008) or even initial (AZZI; CORRADINI; RIBEIRO, 2019) critical pairs, thus greatly reducing the amount of pairs generated. Another improvement would be to stop the search once we have found meaningful critical pairs, avoiding generation of redundant pairs that would lead to the same warnings in the end.

In order to improve the warnings we support, we will have to improve each of the procedures we have presented, extending module nets to support more operation types, and changing MIGRATE procedures accordingly. Also we have concentrated in finding issues due to information flow, such as a attribute which is not necessary because it does not carry information to required attributes. However, we completely disregard control flow issues, which can lead to all sorts of bugs when changed, even if the flow of information is unchanged. Expanding the types of warnings we support is left for future work.

REFERENCES

ABDALKAREEM, R. et al. Why do developers use trivial packages? an empirical case study on npm. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017**. Paderborn, Germany: ACM Press, 2017. p. 385–395. ISBN 978-1-4503-5105-8. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3106237.3106267>.

ABDALKAREEM, R. et al. On the impact of using trivial packages: an empirical case study on npm and PyPI. **Empirical Software Engineering**, v. 25, n. 2, p. 1168–1204, mar. 2020. ISSN 1382-3256, 1573-7616. Available from Internet: <http://link.springer.com/10.1007/s10664-019-09792-9>.

AUÉ, J. et al. An exploratory study on faults in web API integration in a large-scale payment company. In: **Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18**. Gothenburg, Sweden: ACM Press, 2018. p. 13–22. ISBN 978-1-4503-5659-6. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3183519.3183537>.

AZZI, G. G.; CORRADINI, A.; RIBEIRO, L. On the essence and initiality of conflicts in M-adhesive transformation systems. **Journal of Logical and Algebraic Methods in Programming**, v. 109, p. 100482, dec. 2019. ISSN 2352-2208. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S2352220818301639>.

BAHMUTOV, G. **dont-break**. 2018. Available from Internet: <https://www.npmjs.com/package/dont-break>.

BECKER, K. et al. Automatically Determining Compatibility of Evolving Services. In: **2008 IEEE International Conference on Web Services**. Beijing, China: IEEE, 2008. p. 161–168. Available from Internet: <http://ieeexplore.ieee.org/document/4670172/>.

BOGART, C. et al. How to break an API: cost negotiation and community values in three software ecosystems. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016**. Seattle, WA, USA: ACM Press, 2016. p. 109–120. ISBN 978-1-4503-4218-6. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950325>.

BOROVSKIY, V. et al. Ensuring service backwards compatibility with Generic Web Services. In: **2009 ICSE Workshop on Principles of Engineering Service Oriented Systems**. Vancouver, BC, Canada: IEEE, 2009. p. 95–98. ISBN 978-1-4244-3716-0. Available from Internet: <http://ieeexplore.ieee.org/document/5068827/>.

BRITO, A. et al. APIDiff: Detecting API breaking changes. In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Campobasso: IEEE, 2018. p. 507–511. ISBN 978-1-5386-4969-5. Available from Internet: <http://ieeexplore.ieee.org/document/8330249/>.

BRITO, A. et al. Why and How Java Developers Break APIs. **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**, p. 255–265, mar. 2018. ArXiv: 1801.05198. Available from Internet: <http://arxiv.org/abs/1801.05198>.

CORRADINI, A. et al. Translating Java Code to Graph Transformation Systems. In: EHRIG, H. et al. (Ed.). **Graph Transformations**. Berlin, Heidelberg: Springer, 2004. (Lecture Notes in Computer Science), p. 383–398. ISBN 978-3-540-30203-2.

CORRADINI, A. et al. ALGEBRAIC APPROACHES TO GRAPH TRANSFORMATION – PART I: BASIC CONCEPTS AND DOUBLE PUSHOUT APPROACH. In: **Handbook of Graph Grammars and Computing by Graph Transformation**. WORLD SCIENTIFIC, 1997. p. 163–245. ISBN 978-981-02-2884-2 978-981-238-472-0. Available from Internet: <http://www.worldscientific.com/doi/abs/10.1142/9789812384720_0003>.

COSTA, A. et al. Verigraph: A System for Specification and Analysis of Graph Grammars. In: RIBEIRO, L.; LECOMTE, T. (Ed.). **Formal Methods: Foundations and Applications**. Cham: Springer International Publishing, 2016. (Lecture Notes in Computer Science), p. 78–94. ISBN 978-3-319-49815-7.

DECAN, A.; MENS, T.; CLAES, M. An empirical comparison of dependency issues in OSS packaging ecosystems. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Klagenfurt, Austria: IEEE, 2017. p. 2–12. ISBN 978-1-5090-5501-2. Available from Internet: <http://ieeexplore.ieee.org/document/7884604/>.

DECAN, A.; MENS, T.; CONSTANTINOU, E. On the Evolution of Technical Lag in the npm Package Dependency Network. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Madrid: IEEE, 2018. p. 404–414. ISBN 978-1-5386-7870-1. Available from Internet: <https://ieeexplore.ieee.org/document/8530047/>.

DECAN, A.; MENS, T.; GROSJEAN, P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. **Empirical Software Engineering**, v. 24, n. 1, p. 381–416, feb. 2019. ISSN 1382-3256, 1573-7616. Available from Internet: <http://link.springer.com/10.1007/s10664-017-9589-y>.

DUARTE, L. M.; RIBEIRO, L. Graph Grammar Extraction from Source Code. In: CAVALHEIRO, S.; FIADEIRO, J. (Ed.). **Formal Methods: Foundations and Applications**. Cham: Springer International Publishing, 2017. v. 10623, p. 52–69. ISBN 978-3-319-70847-8 978-3-319-70848-5. Series Title: Lecture Notes in Computer Science. Available from Internet: <http://link.springer.com/10.1007/978-3-319-70848-5_5>.

EHRIG, H. Introduction to the algebraic theory of graph grammars (a survey). In: CLAUS, V.; EHRIG, H.; ROZENBERG, G. (Ed.). **Graph-Grammars and Their Application to Computer Science and Biology**. Berlin, Heidelberg: Springer, 1979. (Lecture Notes in Computer Science), p. 1–69. ISBN 978-3-540-35091-0.

EHRIG, H. (Ed.). **Fundamentals of algebraic graph transformation**. Berlin ; New York: Springer, 2006. (Monographs in theoretical computer science). OCLC: ocm69242087. ISBN 978-3-540-31187-4.

EHRIG, H. et al. Information Preserving Bidirectional Model Transformations. In: DWYER, M. B.; LOPES, A. (Ed.). **Fundamental Approaches to Software**

Engineering. Berlin, Heidelberg: Springer, 2007. (Lecture Notes in Computer Science), p. 72–86. ISBN 978-3-540-71289-3.

EHRIG, H. et al. (Ed.). **Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools**. USA: World Scientific Publishing Co., Inc., 1999. ISBN 978-981-02-4020-2.

EHRIG, H. et al. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In: ____. **Handbook of Graph Grammars and Computing by Graph Transformation**. [s.n.], 1997. p. 247–312. Available from Internet: <https://www.worldscientific.com/doi/abs/10.1142/9789812384720_0004>.

ENGELS, G. et al. Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract. In: CORRADINI, A. et al. (Ed.). **Graph Transformations**. Berlin, Heidelberg: Springer, 2006. (Lecture Notes in Computer Science), p. 336–350. ISBN 978-3-540-38872-2.

FIELDING, T. **Architectural Styles and the Design of Network-based Software Architectures**. Thesis (PhD) — University of California, 2000.

FOO, D. et al. Efficient static checking of library updates. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018**. Lake Buena Vista, FL, USA: ACM Press, 2018. p. 791–796. ISBN 978-1-4503-5573-5. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3236024.3275535>.

FOWLER, M. **TolerantReader**. 2011. Available from Internet: <https://martinfowler.com/bliki/TolerantReader.html>.

FOWLER, M. **Microservices**. 2014. Available from Internet: <https://martinfowler.com/articles/microservices.html>.

FOWLER, M. **IntegrationTest**. 2018. Available from Internet: <https://martinfowler.com/bliki/IntegrationTest.html>.

GODEFROID, P.; LEHMANN, D.; POLISHCHUK, M. Differential regression testing for REST APIs. In: **Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. Virtual Event USA: ACM, 2020. p. 312–323. ISBN 978-1-4503-8008-9. Available from Internet: <https://dl.acm.org/doi/10.1145/3395363.3397374>.

GUINEA, S.; SPOLETINI, P. Evaluating the compatibility of conversational service interactions. In: **Proceeding of the 3rd international workshop on Principles of engineering service-oriented systems - PESOS '11**. Waikiki, Honolulu, HI, USA: ACM Press, 2011. p. 29–35. ISBN 978-1-4503-0591-4. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1985394.1985399>.

HAUPT, F.; LEYMANN, F.; PAUTASSO, C. A Conversation Based Approach for Modeling REST APIs. In: **2015 12th Working IEEE/IFIP Conference on Software Architecture**. Montreal, QC, Canada: IEEE, 2015. p. 165–174. ISBN 978-1-4799-1922-2. Available from Internet: <http://ieeexplore.ieee.org/document/7158518/>.

HAUSMANN, J. H.; HECKEL, R.; LOHMANN, M. Model-based Discovery of Web Services. In: **Proceedings of the IEEE International Conference on Web Services**. USA: IEEE Computer Society, 2004. (ICWS '04), p. 324. ISBN 978-0-7695-2167-1.

HECKEL, R.; KüSTER, J. M.; TAENTZER, G. Confluence of Typed Attributed Graph Transformation Systems. In: CORRADINI, A. et al. (Ed.). **Graph Transformation**. Berlin, Heidelberg: Springer, 2002. (Lecture Notes in Computer Science), p. 161–176. ISBN 978-3-540-45832-6.

HECKEL, R.; MARIANI, L. Automatic Conformance Testing of Web Services. In: CERIOLI, M. (Ed.). **Fundamental Approaches to Software Engineering**. Berlin, Heidelberg: Springer, 2005. (Lecture Notes in Computer Science), p. 34–48. ISBN 978-3-540-31984-9.

KHAN, T. A.; HECKEL, R. On Model-Based Regression Testing of Web-Services Using Dependency Analysis of Visual Contracts. In: GIANNAKOPOULOU, D.; OREJAS, F. (Ed.). **Fundamental Approaches to Software Engineering**. Berlin, Heidelberg: Springer, 2011. (Lecture Notes in Computer Science), p. 341–355. ISBN 978-3-642-19811-3.

KHAN, T. A.; RUNGE, O.; HECKEL, R. Testing against Visual Contracts: Model-Based Coverage. In: EHRIG, H. et al. (Ed.). **Graph Transformations**. Berlin, Heidelberg: Springer, 2012. (Lecture Notes in Computer Science), p. 279–293. ISBN 978-3-642-33654-6.

LAMBERS, L.; EHRIG, H.; OREJAS, F. Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs. **Electronic Notes in Theoretical Computer Science**, v. 211, p. 17–26, abr. 2008. ISSN 1571-0661. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S1571066108002417>.

LOHMANN, M.; MARIANI, L.; HECKEL, R. A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services. In: BARESI, L.; NITTO, E. D. (Ed.). **Test and Analysis of Web Services**. Berlin, Heidelberg: Springer, 2007. p. 173–204. ISBN 978-3-540-72912-9. Available from Internet: <https://doi.org/10.1007/978-3-540-72912-9_7>.

LüBKE, D. et al. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In: **Proceedings of the 24th European Conference on Pattern Languages of Programs - EuroPLop '19**. Irsee, Germany: ACM Press, 2019. p. 1–24. ISBN 978-1-4503-6206-1. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3361149.3361164>.

MORA, F. et al. Client-specific equivalence checking. In: **Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018**. Montpellier, France: ACM Press, 2018. p. 441–451. ISBN 978-1-4503-5937-5. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3238147.3238178>.

MOSTAFA, S.; RODRIGUEZ, R.; WANG, X. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In: **Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017**. Santa Barbara, CA, USA: ACM Press, 2017. p. 215–225. ISBN 978-1-4503-5076-1. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3092703.3092721>.

MUJAHID, S. et al. Using Others' Tests to Identify Breaking Updates. In: **Proceedings of the 17th International Conference on Mining Software Repositories**. Seoul Republic of Korea: ACM, 2020. p. 466–476. ISBN 978-1-4503-7517-7. Available from Internet: <https://dl.acm.org/doi/10.1145/3379597.3387476>.

MøLLER, A.; TORP, M. T. Model-based testing of breaking changes in Node.js libraries. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019**. Tallinn, Estonia: ACM Press, 2019. p. 409–419. ISBN 978-1-4503-5572-8. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3338906.3338940>.

ORACLE. **Chapter 13. Binary Compatibility**. 2020. Available from Internet: <https://docs.oracle.com/javase/specs/jls/se15/html/jls-13.html>.

PRESTON-WERNER, T. **Semantic Versioning 2.0.0**. 2020. Available from Internet: <https://semver.org/>.

RAEMAEKERS, S.; DEURSEN, A. van; VISSER, J. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In: **2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation**. Victoria, BC, Canada: IEEE, 2014. p. 215–224. ISBN 978-1-4799-6148-1. Available from Internet: <http://ieeexplore.ieee.org/document/6975655/>.

RUNGE, O.; ERMEL, C.; TAENTZER, G. AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations. In: SCHüRR, A.; VARRó, D.; VARRó, G. (Ed.). **Applications of Graph Transformations with Industrial Relevance**. Berlin, Heidelberg: Springer, 2012. (Lecture Notes in Computer Science), p. 81–88. ISBN 978-3-642-34176-2.

SALZA, P. et al. Do developers update third-party libraries in mobile apps? In: **Proceedings of the 26th Conference on Program Comprehension - ICPC '18**. Gothenburg, Sweden: ACM Press, 2018. p. 255–265. ISBN 978-1-4503-5714-2. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3196321.3196341>.

SOLDANI, J.; TAMBURRI, D. A.; HEUVEL, W.-J. V. D. The pains and gains of microservices: A Systematic grey literature review. **Journal of Systems and Software**, v. 146, p. 215–232, dec. 2018. ISSN 01641212. Available from Internet: <https://linkinghub.elsevier.com/retrieve/pii/S0164121218302139>.

TAENTZER, G.; CARUGHI, G. T. A Graph-Based Approach to Transform XML Documents. In: BARESI, L.; HECKEL, R. (Ed.). **Fundamental Approaches to Software Engineering**. Berlin, Heidelberg: Springer, 2006. (Lecture Notes in Computer Science), p. 48–62. ISBN 978-3-540-33094-3.

WANG, Y. et al. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In: **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Adelaide, Australia: IEEE, 2020. p. 35–45. ISBN 978-1-72815-619-4. Available from Internet: <https://ieeexplore.ieee.org/document/9240619/>.

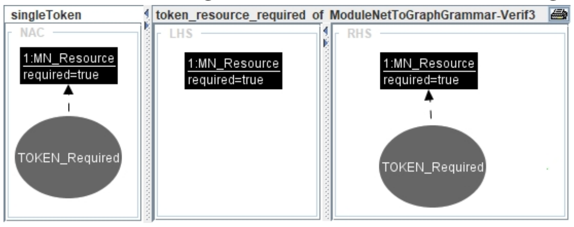
XAVIER, L. et al. Historical and impact analysis of API breaking changes: A large-scale study. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Klagenfurt, Austria: IEEE, 2017. p. 138–147. ISBN 978-1-5090-5501-2. Available from Internet: <http://ieeexplore.ieee.org/document/7884616/>.

ZHOU, J.; WALKER, R. J. API deprecation: a retrospective analysis and detection method for code examples on the web. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016**. Seattle, WA, USA: ACM Press, 2016. p. 266–277. ISBN 978-1-4503-4218-6. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950298>.

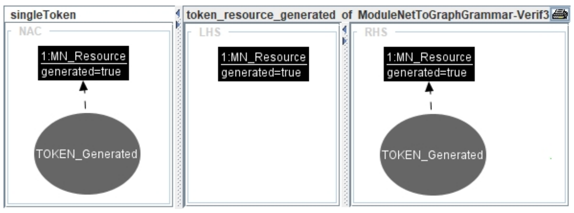
APPENDIX A — TRANSLATION GRAPH TRANSFORMATION SYSTEM

This appendix presents the entire translation Graph Transformation System (GTS) which has been explained in chapter 6. We omit here the type graph, which has already been presented in that chapter. Figures A.1, A.2, A.3 and A.4 present all rules in the GTS and Figures A.5 and A.6 present the atomic constraints used to show correctness of translations. We label each rule and atomic with codes. These codes are listed in chapter 6.

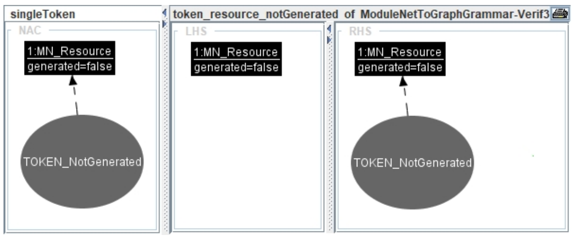
Figure A.1: Rules TK1 through TK10 of translation grammar.



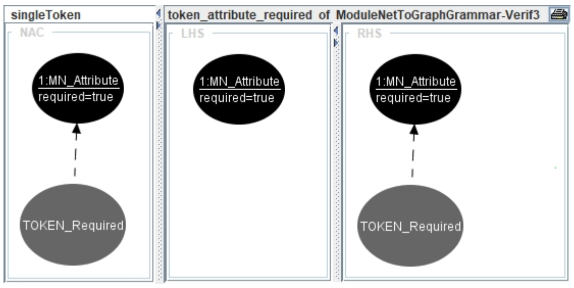
(a) TK1



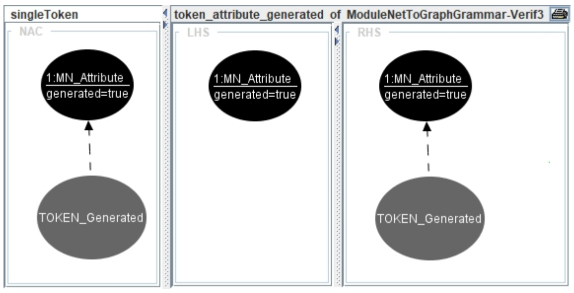
(b) TK2



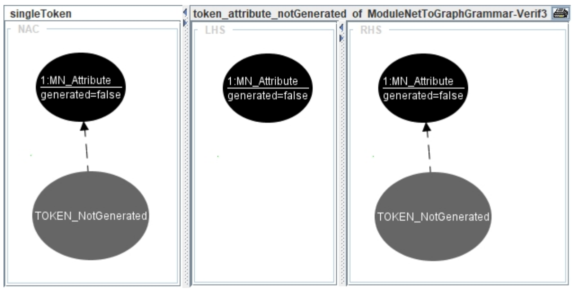
(c) TK3



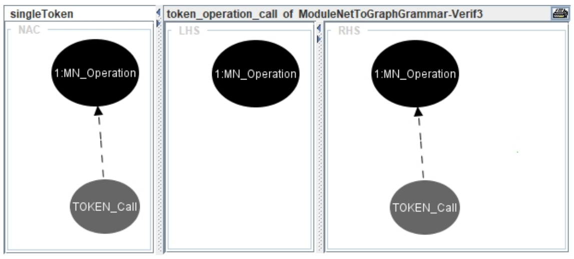
(d) TK4



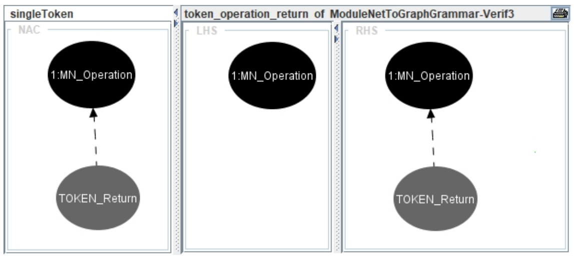
(e) TK5



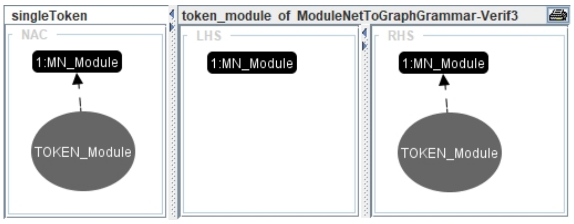
(f) TK6



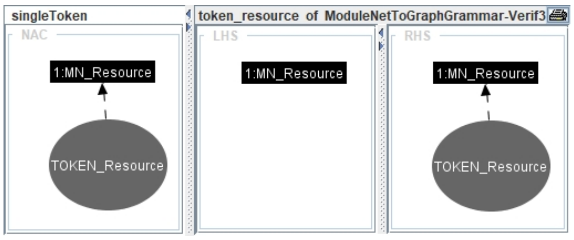
(g) TK7



(h) TK8

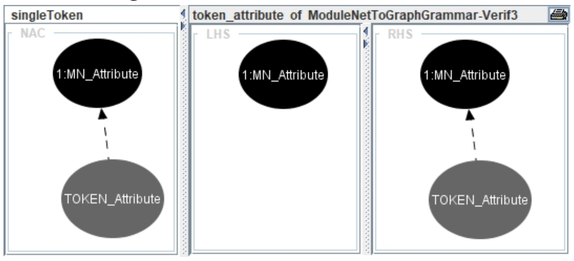


(i) TK9

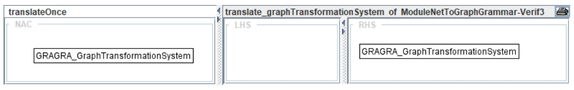


(j) TK10

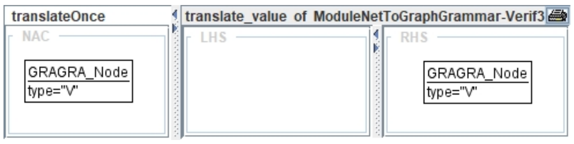
Figure A.2: Rules TK11 and TR12 through TR20 of translation grammar.



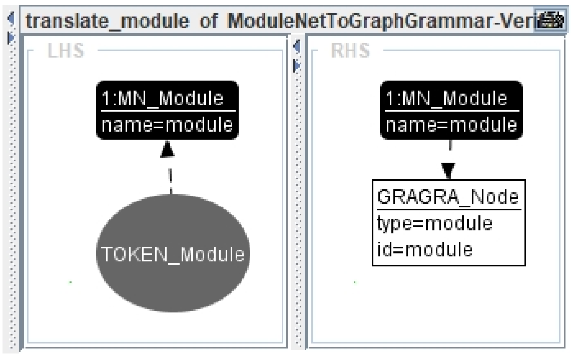
(a) TK11


translateOnce
translate_graphTransformationSystem of ModuleNetToGraphGrammar-Verif3
NAC
LHS
RHS
GRAGRA_GraphTransformationSystem

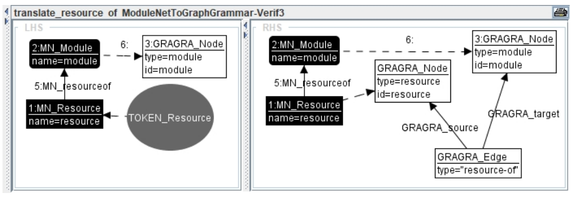
(b) TR12



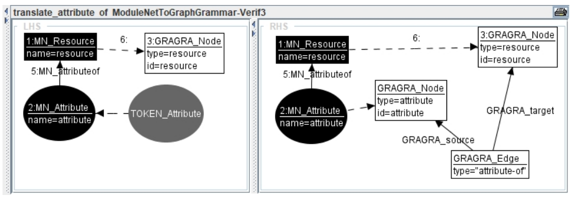
(c) TR13



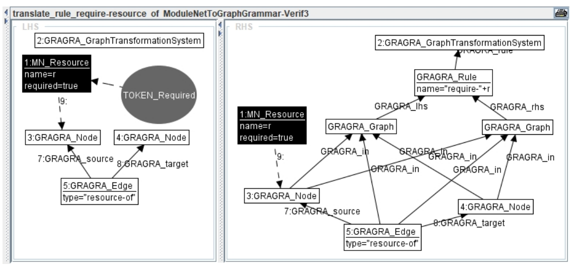
(d) TR14



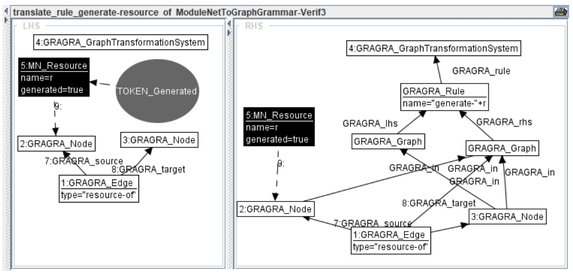
(e) TR15



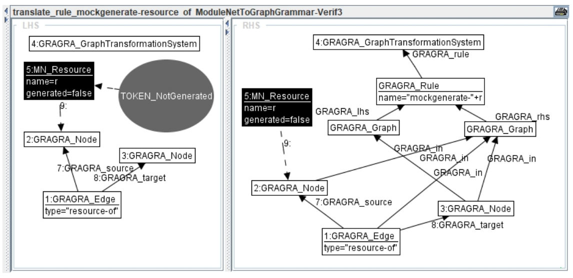
(f) TR16



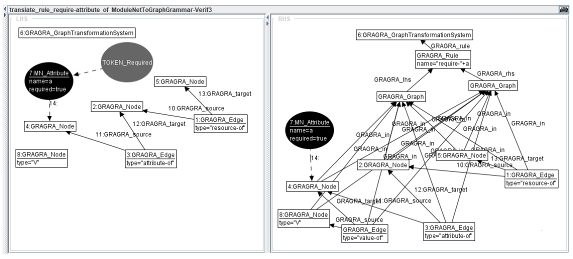
(g) TR17



(h) TR18

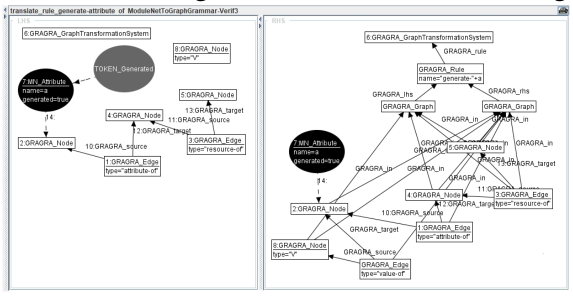


(i) TR19

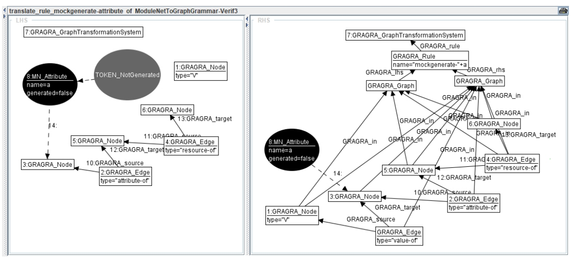


(j) TR20

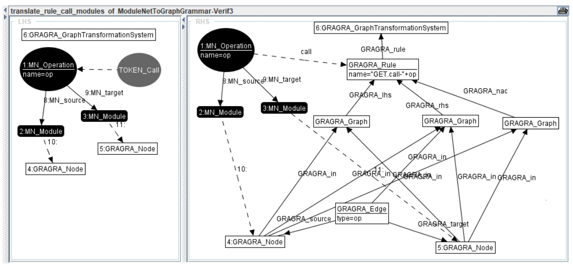
Figure A.3: Rules TR21 through TR27 of translation grammar.


translate_rule_generate-attribute of ModuleNetToGraphGrammar-Verif3
LHS
RHS

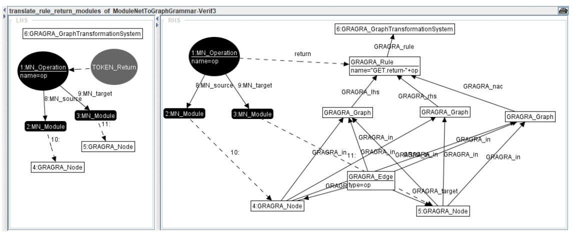
(a) TR21


translate_rule_mockgenerate-attribute of ModuleNetToGraphGrammar-Verif3
LHS
RHS

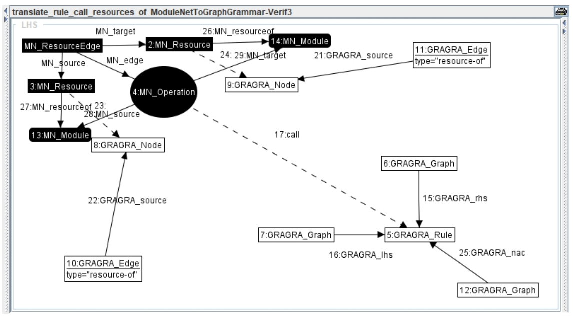
(b) TR22


translate_rule_call_modules of ModuleNetToGraphGrammar-Verif3
LHS
RHS

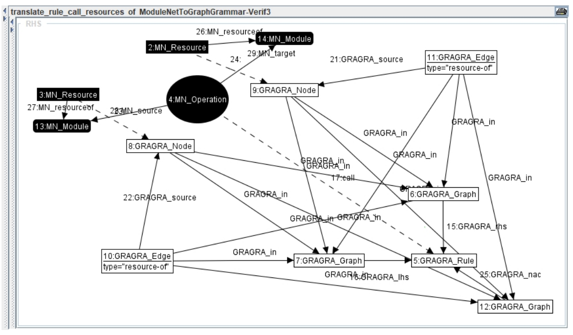
(c) TR23


translate_rule_return_modules of ModuleNetToGraphGrammar-Verif3
LHS
RHS

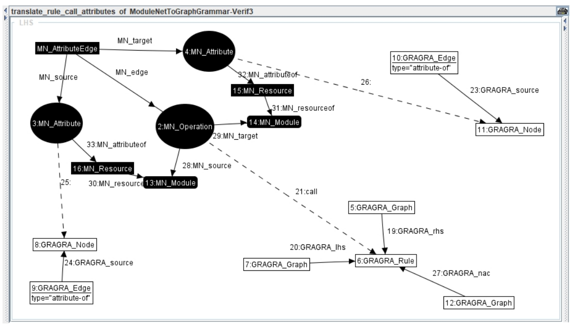
(d) TR26


translate_rule_call_resources of ModuleNetToGraphGrammar-Verif3
LHS

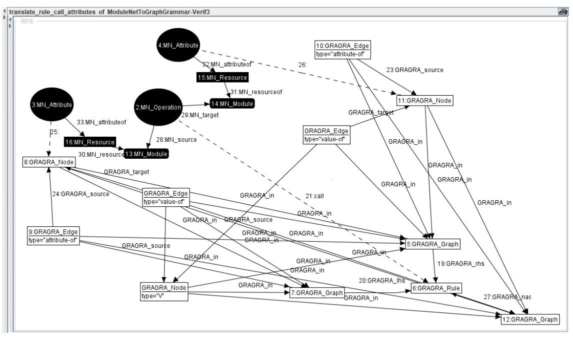
(e) TR24 left-handside


translate_rule_call_resources of ModuleNetToGraphGrammar-Verif3
RHS

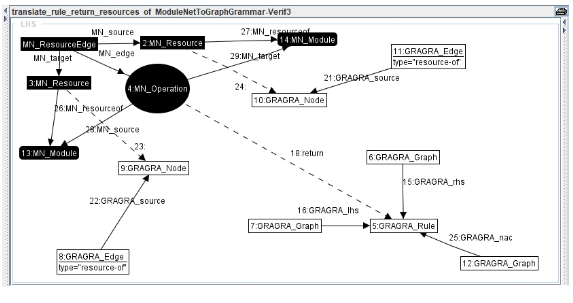
(f) TR24 right-handside


translate_rule_call_attributes of ModuleNetToGraphGrammar-Verif3
LHS

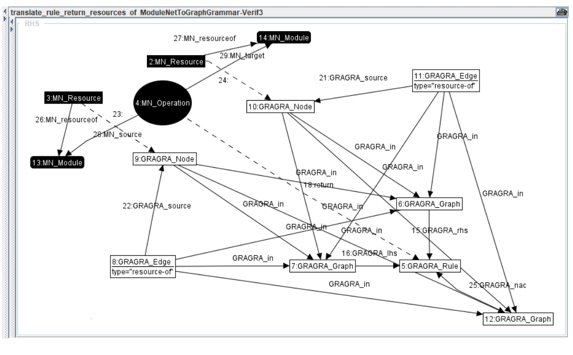
(g) TR25 left-handside


translate_rule_call_attributes of ModuleNetToGraphGrammar-Verif3
RHS

(h) TR25 right-handside

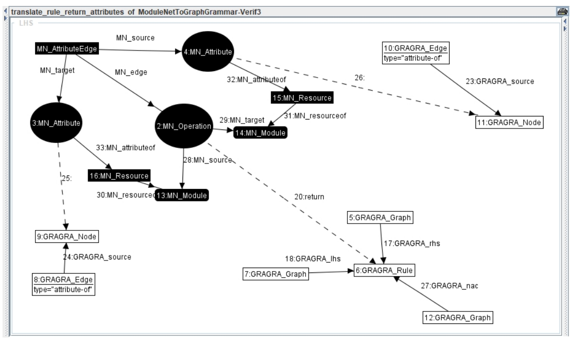

translate_rule_return_resources of ModuleNetToGraphGrammar-Verif3
LHS

(i) TR27 left-handside

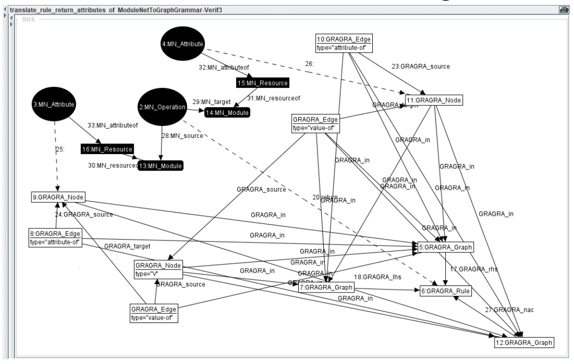

translate_rule_return_resources of ModuleNetToGraphGrammar-Verif3
RHS

(j) TR27 right-handside

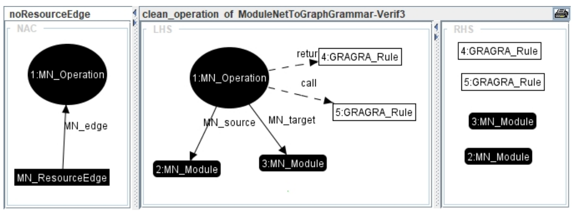
Figure A.4: Rules TR28, CL29 through CL33, AD34 and AD35 of translation grammar.



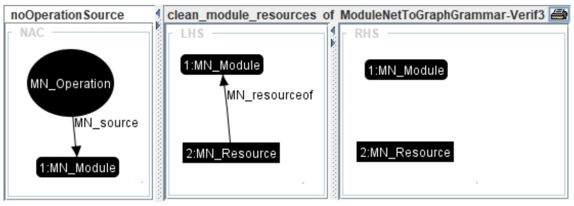
(a) TR28 left-handside



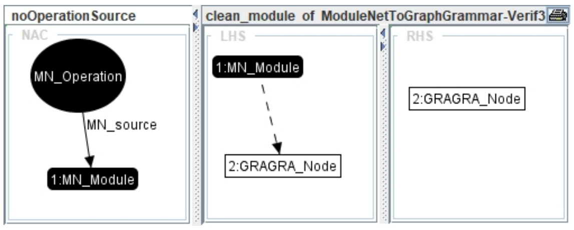
(b) TR28 right-handside



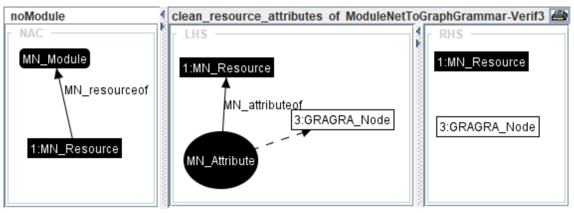
(c) CL29



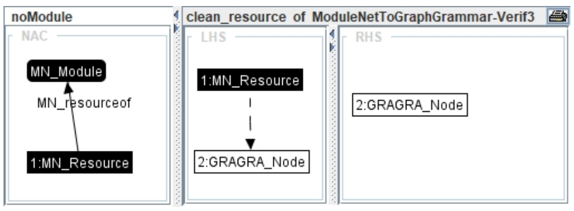
(d) CL30



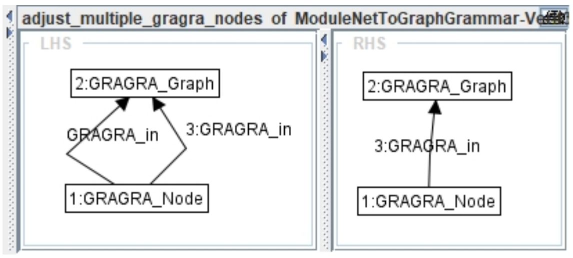
(e) CL31



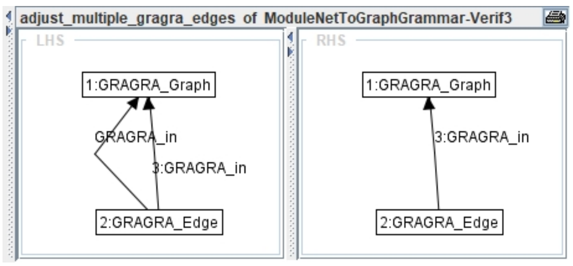
(f) CL32



(g) CL33

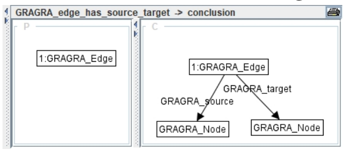


(h) AD34

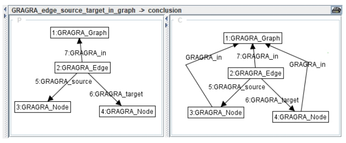


(i) AD35

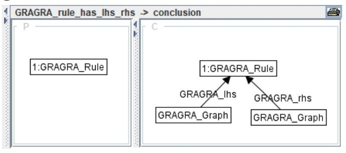
Figure A.5: Atomic constraints A1 through A8.



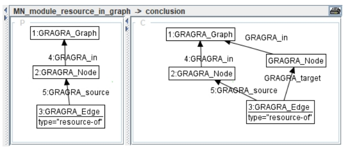
(a) A1



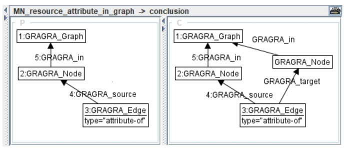
(b) A2



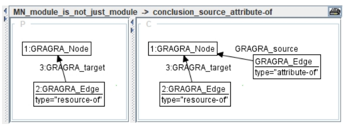
(c) A3



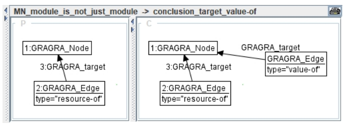
(d) A4



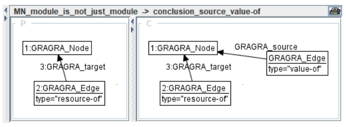
(e) A5



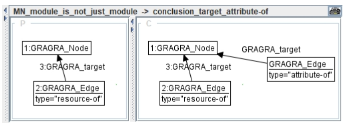
(f) A6 conclusion I



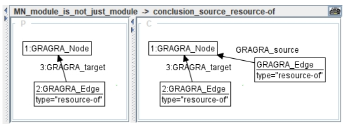
(g) A6 conclusion II



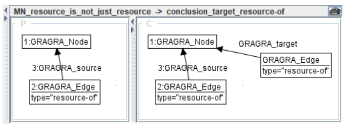
(h) A6 conclusion III



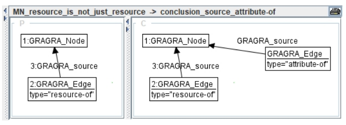
(i) A6 conclusion IV



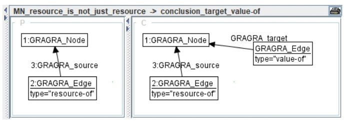
(j) A6 conclusion V



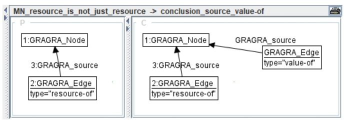
(k) A7 conclusion I



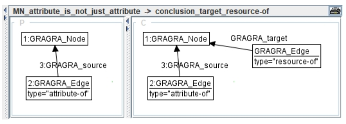
(l) A7 conclusion II



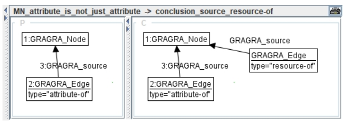
(m) A7 conclusion III



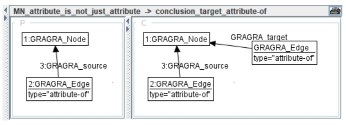
(n) A7 conclusion IV



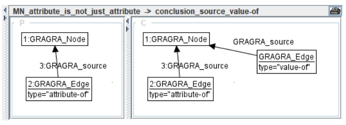
(o) A8 conclusion I



(p) A8 conclusion II

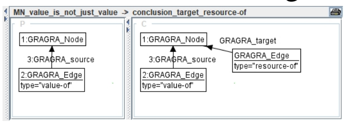


(q) A8 conclusion III

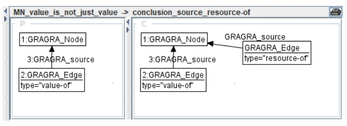


(r) A8 conclusion IV

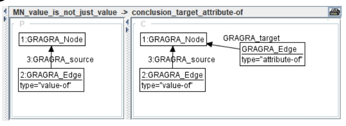
Figure A.6: Atomic constraints A9 through A12.



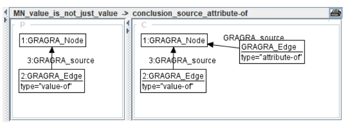
(a) A9 conclusion I



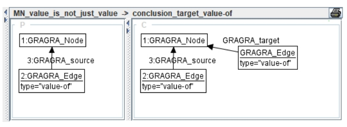
(b) A9 conclusion II



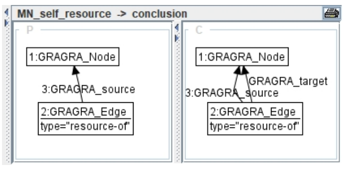
(c) A9 conclusion III



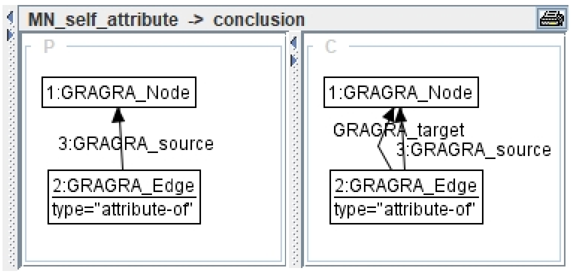
(d) A9 conclusion IV



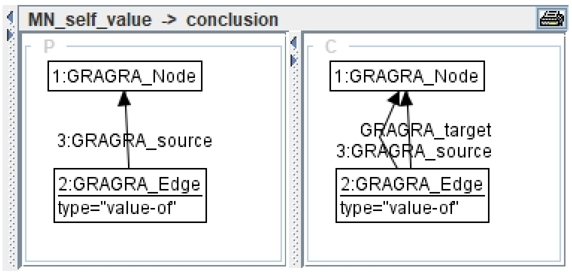
(e) A9 conclusion V



(f) A10



(g) A11



(h) A12

APPENDIX B — RESUMO ESTENDIDO

Neste trabalho planejamos resolver o problema de integração de módulos utilizando gramáticas de grafos. Nós criamos um framework de verificação que nos permite gerar avisos dizendo a desenvolvedores quais integrações necessitam de sua atenção. O framework, que chamamos de MIGRATE, é composto por quatro artefatos e três processos, que grifamos a seguir. Inicialmente, **artefatos de software** são fornecidos para um procedimento de **extração**, o qual produz uma **rede de módulos**. Redes de módulos são um formalismo que também definimos nesta dissertação, o qual captura como módulos de software se integram. Em seguida, a rede de módulos é fornecida para um procedimento de **tradução**, que utiliza uma gramática de grafos e produz uma gramática de grafos, à qual chamamos de **gramática de verificação**. Esta última é fornecida para o procedimento de **verificação**, que finalmente produz **avisos** úteis aos desenvolvedores interessados em encontrar falhas de integração de software. Estes avisos são: atributo, recurso, módulo ou operação opcional, atributo estritamente opcional, operação inalcançável, recurso perdido e atributo desatualizado. Nós fornecemos um protótipo com implementações para os procedimentos de tradução e verificação do framework MIGRATE e demonstramos em um estudo de caso.

O framewok MIGRATE impõe alguns requisitos sobre as gramáticas de grafos que analisa, as gramáticas de verificação, como ter nomes e padrões de regras específicos, além de nós e arestas que podem ser interpretados de volta para operações, módulos, recursos, atributos e valores. Nós precisávamos de uma maneira de garantir que as gramáticas de grafos que analisamos estivessem dentro desses requisitos, e ao mesmo tempo fornecer a desenvolvedores a possibilidade de fazerem mudanças diretamente nos modelos. Por estes motivos, nós criamos redes de módulos, que são um formalismo para expressar como módulos integram uns com os outros. No entanto, nós não definimos a semântica de redes de módulos, ao invés disso nós criamos um procedimento de tradução que encontra uma gramática de grafos (de verificação) que é semântica de uma rede de módulos. Isto nos proporcionou o reuso da teoria de análise de pares críticos, e também irá nos proporcionar o reuso em trabalhos futuros de todas as diferentes técnicas de análises que já foram desenvolvidas para gramáticas de grafos. Nós demonstramos como traduzir redes de módulos para gramáticas de verificação usando gramáticas de grafos. Nós provamos que nosso procedimento de tradução termina e argumentamos que ele gera gramáticas de verificação bem definidas. Nós reusamos análise de pares críticos de gramáticas de verificação para construir um procedimento de verificação que indica diferentes tipos de avisos relacionados às estruturas de redes de módulos.

Nós temos muitas limitações para endereçar no futuro. Nós fomos desafiados a expressar todos os diferentes tipos de operações que podem ser feitas sobre dados, enquanto nossas redes de módulos são limitadas nas operações que suportam. Nós não apresentamos nenhum algoritmo de extração de redes de módulos, visto que este trabalho ainda está em andamento. Nosso procedimento de tradução não pode lidar com transferência de informação dentro de módulos e ele exige que redes de módulos sejam tão pequenas quanto possível, para evitar problemas de memória e falta de tempo. Nós usamos um algoritmo de extração de pares críticos, que sabidamente leva muito tempo e consome muita memória. Finalmente, nosso procedimento de verificação produz poucos tipos de avisos, alguns dos quais não são úteis de forma alguma. Há duas grandes áreas que irão necessitar de nossa atenção ao estender o protótipo e construir uma ferramenta de verificação usando os conceitos deste trabaho: suporte a redes de módulos maiores em termos de memória e tempo, e significado de avisos.

Redes de módulos maiores são especialmente difíceis durante o procedimento de derivação aplicado na tradução e na computação de pares críticos. Considerando que nossa gramática de tradução é confluente, poderíamos melhorar a derivação evitando aleatorização de casamentos entre regra e grafo e sempre aplicando o primeiro casamento que encontrarmos. Pares críticos podem ser melhorados se restringirmos à geração de pares críticos essenciais ou iniciais, reduzindo muito a quantidade de pares gerados. Outra melhoria seria parar a busca assim que encontrarmos pares críticos que façam sentido, evitando geração de pares críticos redundantes que levariam aos mesmos avisos.

Para melhorar os avisos que suportamos, devemos melhorar cada um dos procedimentos apresentados, estendendo redes de módulos para suportar mais tipos de operações, e modificando os procedimentos do framework da mesma maneira. Nós também nos concentramos em encontrar problemas devido ao fluxo de informação, como atributos que não são necessários porque não carregam informação para atributos necessários. No entanto, nós desconsideramos problemas de fluxo de controle, que podem levar a todo tipo de defeitos quando modificados, mesmo que o fluxo de informação não mude. Expandiremos tipos de avisos em trabalhos futuros.