HeteroGenius a framework for hybrid analysis of heterogeneous software specifications

Manuel Giménez Mariano M. Moscato
Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires
{mgimenez, mmoscato}@dc.uba.ar

Carlos G. Lopez Pombo
Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires
CONICET
clpombo@dc.uba.ar [*]

Marcelo F. Frias
Departmento de Ingeniería de Software,
Instituto Tecnológico de Buenos Aires
CONICET
mfrias@itba.edu

Nowadays, software artefacts are ubiquitous in our lives being an essential part of home appliances, cars, cell phones, and even in more critical activities like aeronautics and health sciences. In this context software failures may produce enormous losses, either economical or, in the worst case, in human lives. Software analysis is an area in software engineering concerned with the application of diverse techniques in order to prove the absence of errors in software pieces. In many cases different analysis techniques are applied by following specific methodological combinations that ensure better results. These interactions between tools are usually carried out at the user level and it is not supported by the tools. In this work we present HeteroGenius, a framework conceived to develop tools that allows the user to perform hybrid analysis of heterogeneous software specifications.

HeteroGenius was designed prioritising the possibility of adding new specification languages and analysis tools and enabling a synergic relation of the techniques under a graphical interface satisfying several well-known usability enhancement criteria. As a case-study we implemented the functionality of `Dynamite` on top of HeteroGenius.

1 Introduction

Nowadays, software artefacts are ubiquitous in our lives being an essential part of home appliances, cars, cell phones, and even in more critical activities like aeronautics and health sciences. In this context software failures may produce enormous losses, either economical or, in the worst case, in human lives. Software analysis is an area in software engineering concerned with the application of diverse validation and verification techniques in order to prove the absence of errors in software pieces.

Several languages and notations have been made available to help analysts and designers capture and model different aspects involved in software applications. For instance, UML [9, 8] offers a range of diagrammatic notions, from class diagrams to state diagrams, collaborations diagrams, and so on. This proliferation reflects the need to reduce the complexity of developing large systems as each language allows (teams of) engineers to address a specific view or phase of the development process. The same happens at the level of the formalisms that can formally support the use of such languages and methods.

[*]The author would like to thank the MEALS project (EU FP7 programme, grant agreement No. 295261).

Submitted to:
LAFM 2013

In summary, we face a scenario in which there is a multitude of modelling languages and supporting logics, and tools for processing such languages by reasoning in the underlying logics. Thus, heterogeneity becomes a major source of complexity. We find heterogeneity at the level of languages because different languages serve different purposes in software behaviour specification, and at the level of analysis technique as different techniques provide different results.

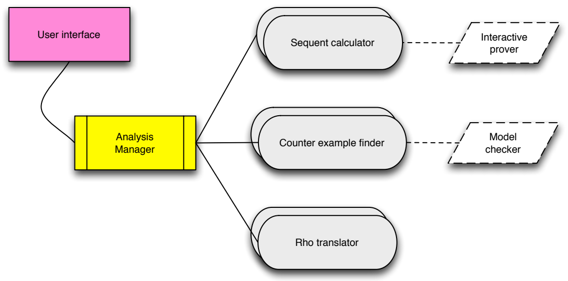
The field of institutions[7] grew as an effort in providing formal foundations for software specification languages and analysis techniques. In [14], Meseguer developed the categorical formalization of logical system by complementing the model theoretic view of a logic (institutions) with its deductive view (entailment system and proof calculus). In his work, Meseguer also introduced the notion of institution representations as a tool enabling reuse of proof systems, a limited view of heterogeneity. It was in [20] where both institution morphisms and representations (also called co-morphisms) were extensively studied. In [14, Defs. 23 and 27, Sec. 4], Meseguer extends the definitions of entailment and institution representation to work on theories. In [20, Prop. 5.2, Thm. 5.3 and Coro. 5.4, Sec. 5.1], Tarlecki proves general conditions under which a proof system for a richer logic[1] I' can be used to prove properties of specifications written in a poorer one I provided there exists a map of logics from I to I'. From now on we will call these operations *ρ-translations*.

Many combinations of different tools have been depicted as methodologies for software analysis. Formal methods are usually divided into two categories: heavyweight and lightweight. These names refer to the amount of mathematical expertise needed during the process of proving a given property. Modern software analysis methodologies departed long ago from the idea that heavyweight formal methods or lightweight ones are applied disregarding the relation between these tools. We claim that enforcing these methodological directives as part of the process of software analysis produces better results.

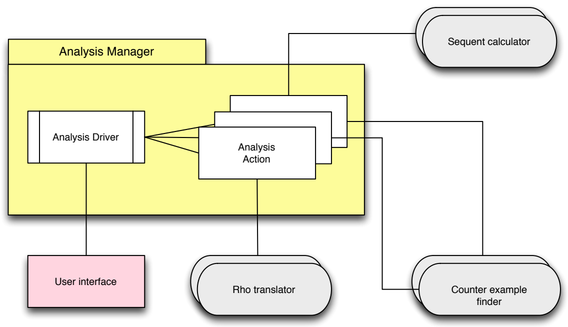
An example of this is `Dynamite` [5]. `Dynamite` is a theorem prover for Alloy [10] in which the critical parts of the proof (carried out in a theorem prover implemented on top of the semi-automatic theorem prover PVS [16]) are assisted by the Alloy Analyzer with the aim of reducing both the workload and the error proneness introduced by the human interaction with the tool. Another use of model theoretic tools in relation to the use of theorem provers is the fact that they provide an efficient method for: *a)* the gain of confidence in the hypothesis brought into a proof, *b)* the elimination of superfluous formulae appearing in a sequent, *c)* the removal of minor modelling errors, and even *d)* the suggestion of potential witnesses for existential quantifiers. All these actions are carried out by using the Alloy Analyzer in order to search for counterexamples for specific sets of conditions derived from the axioms in the specification and the property we want to prove. The result of this model-theoretical assistance for counterexample finding gives rise to a whole new class of analysis strategies resulting from a coordinated action of different tools over the same step of the proof. These actions can not be understood neither as pure proof commands, nor as pure model-theoretical commands. In such way, we call *hybrid* those kind of analysis that uses a coordinated mixture of the both approaches.

In this paper we will present the development of a general and scalable framework for building tools that allows the user to deal with hybrid analysis of heterogeneous specifications. To test the capabilities of the HeteroGenius framework we use it to build a new version of `Dynamite`.

[1]The word “logic” here is used as in [14, Def. 6].



(*a*)



(*b*)

Figure 1: (*a*) Global architecture of HeteroGenius. (*b*) *Analysis Manager* architecture.

2 HeteroGenius: A framework for hybrid heterogeneous analysis

Every analysis tool has to deal with four dimensions: the user interface, the conceptual analysis type, the languages and the analysis engine. Most softwares out there are built to work with just one conceptual analysis type, provide support for only one language, and use just one specific analysis engine. HeteroGenius aims to be an extensible multi-language and multi-engine analysis tool, so its design must decouple the four dimensions as much as possible. Is worth noting there is a natural coupling between some of them: i.e., every engine is bounded with a specific language and the user interface is probably designed to work with one kind of conceptual analysis type.

Figure 1 shows the global architecture of HeteroGenius. The *Analysis Manager* keeps track of the current heterogeneous analysis, and drives its evolution.

Abstracting external engines It is well known that service abstraction is a very useful tactic to accomplish modifiability. We use it also to prevent our design to be tied to specific external analysis engines, introducing some components that abstract services offered by those external softwares. We established three main families: sequent calculators, counter example finders, and ρ translators. Each family has a specific interface that captures and establishes the common behavior of its members. Our design enables other families to be added easily without code modification as we will see in the next sections.

A *sequent calculator* is an entity that has just one responsibility: given a sequent and some rule of the sequent calculus, it must return the result of applying that rule over that sequent. Naturally most sequent calculus interactive provers can be used as backends for this type of component. It is worth noting that each concrete calculator has its own set of rules. The intra language rules of the calculus actually implemented by HeteroGenius is going to be limited by the sum of all rules provided by the concrete sequent calculators. A *counter example finder* is a component that given a formula tries to answer wether or not a counter example exists for that formula. A ρ *translator* offers language translation services by translating sequents and specifications between languages. None of the mentioned components need to know anything about the current analysis, accomplishing in this way the desired decoupling between the engines and the analysis management.

Note that the same external engine can be used as backend for more than one concrete component. Our architecture does not force a specific way to communicate with external applications; each component is free but also responsible to chose the method it considers appropriate.

Analysis Manager The following are actions that the user might want to perform over the current analysis: apply certain sequent calculus rule over some analysis node, validate some sequent by searching for a counterexample, prune the analysis tree, change languages at some point of the analysis, etc. It is clear that the specific steps to perform each of the mentioned actions are very different: to apply the rule, a sequent calculator must be used and several new nodes may have to be added to the tree; on the other hand when validating the sequent no nodes will ever be added.

We introduced the idea of *analysis action* to model any kind of actions the user might want to perform over the analysis. Each *analysis action* is responsible for knowing and performing all the specific steps to actually apply the action it models. One *analysis action* may interact with none, one, or several of the external engine abstractions mentioned in the previous section.

As shown in figure 1, the *Analysis Driver* does not need to interact directly with any of them, thanks to the *analysis actions*, increasing the semantic coherence of our design. This component is responsible for keeping the current state of the analysis tree, but as we explained, the changes over it are made by *analysis actions*. The analysis tree design enables other languages to be added easily without code modification.

User interface The user interface of a software like HeteroGenius must present a clear outlook of the current analysis and provide an easy and intuitive way of interacting with it. According to [1], the way in which the current status of the proof is shown (formulas, sequents) is crucial for the usability. Our efforts were focused on achieving these objectives.

Visualization of tree structures is a research area on its own, so we decided to develop the interface of HeteroGenius using a mature framework: JUNG2[2]. The Java Universal Network/Graph Framework is a software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java and it is open source.

In HeteroGenius almost all interactions are done by *point & click*: just left click any analysis node, and a contextual menu will offer all the *analysis actions* applicable over the clicked node. In case that the *analysis action* being applied needs a formula from the user, she or he must use the keyboard to provide it.

3 Dynamite3: Implementing Dynamite on top of HeteroGenius

In order to implement a new version of Dynamite using the HeteroGenius framework, we needed to provide a *sequent calculator* for Alloy. As mentioned above, the previous version of Dynamite [**?**] is an interactive theorem prover for Alloy, based on sequent calculus. Then, we only had to wrap it with a new component that plays the role of an abstraction layer. The formal background of Dynamite can be found in [5] and [4], where it is proved that there exists a semantic preserving translation of the Alloy specification language to theories in an extension of fork algebras [2][3]. The class of algebraic structures considered for interpreting Alloy is the class of *point-dense omega closure fork algebras* (PDOCFA). Resorting to PVS's higher-order logic we constructed a semantics embedding of PDOCFA. Then, we could be able to use the PVS prover to provide a theorem prover for Alloy.

[2]`http://jung.sourceforge.net`

[3]The interested reader will find model-theoretic flavoured proofs based on the semantics of the languages in that work, but they can be easily restated within the framework of general logics by just bringing into the definitions of the translations their action on morphisms.

To test the HeteroGenius ability of supporting heterogeneous proofs, we also implement a concrete *sequent calculator* for PDOCFA. This calculator was developed by reusing much of the code of `Dynamite`, since the embedding of language of PDOCFA in PVS was already implemented in that tool.

We also implemented an Alloy *counter example finder* using the Alloy Analyzer as backend. Finally we developed an Alloy to PDOCFA ρ translator, which internally relies on the same translation used by the Alloy sequent calculator mentioned above.

Several new *analysis actions* were added: one for each sequent calculus rule provided by our *sequent calculators*, one to use the Alloy *counter example finder* to look for counter examples of the sequent of a given analysis node, one to switch languages during the analysis using the ρ translator, and a few more to manipulate the analysis tree (like pruning a subtree). We also added some *analysis actions* that makes use of interaction between different engines in order to provide the specific `Dynamite` commands, such as validated case or pruning of goals [**?**].

`Dynamite3` is available at `http://www.dc.uba.ar/dynamite/heterogenius`.

4 Conclusions, related and further work

In the line of the tools for heterogeneous analysis one remarkable piece of work is Hets. According to [**?**], Hets is intended for use as a proof manager of complex systems, specified using different formal languages. But once a specific obligation has to be proved, the proving process must be carried out using a specific external tool. Hets serves as an organizer of information related with which goals have been proved, which ones have not and how elements specified using different languages are related with each other. Instead, HeteroGenius is much more involved in the process of actually proving the goals. As explained above, it manages each proof of the specification, allowing the use of different external tools even when proving a single goal. Even more, HeteroGenius allows the interaction between different external tools and thus gives the user the possibility of performing hybrid analysis on the specification under study. The way in which each element is related with another one specified in a different language, is established through the ρ translators mentioned in previous sections. The management of the dependencies of the obligations being proved is one of the next steps in the development of HeteroGenius.

In this work we showed some of the relevant theoretical and practical issues behind the implementation of tool support for hybrid analysis of heterogeneous software specifications. Theoretic results were put in second place to leave enough space for the intuitions and the discussion on the development decisions. Finally we showed how the architecture of HeteroGenius enabled the reengineering of `Dynamite`, a heterogeneous analysis tool.

The reader should notice that even when the implementation of `Dynamite` requires certain level of heterogeneity in the language supporting the analysis, it fails in exemplifying how HeteroGenius can help in analyzing a software artifact described by the interactions of components described in different logical languages. Regarding this we are working on the implementation of an heterogeneous specification language presented in [12].

References

[1] Bernhard Beckert & Sarah Grebing (2012): *Evaluating the Usability of Interactive Verification Systems*. *Comparative Empirical Evaluation of Reasoning Systems.*

[2] Marcelo F. Frias (2002): *Fork algebras in algebra, logic and computer science*. *Advances in logic* 2, World Scientific Publishing Co., Singapore.

[3] Marcelo F. Frias, Gabriel A. Baum & Tomas S. E. Maibaum (2002): *Interpretability of first-order dynamic logic in a relational calculus*. In Harrie de Swart, editor: *Proceedings of the 6th. Conference on Relational Methods in Computer Science (RelMiCS) - TARSKI*, *Lecture Notes in Computer Science* 2561, Springer-Verlag, Oisterwijk, The Netherlands, pp. 66–80.

[4] Marcelo F. Frias, Carlos G. Lopez Pombo & Nazareno M. Aguirre (2004): *An equational calculus for Alloy*. In Jim Davies, Wolfram Schulte & Mike Barnett, editors: *Proceedings of the 6th. International conference on formal engineering methods (ICFEM)*, *Lecture Notes in Computer Science* 3308, Springer-Verlag, Seattle, Washington, United States, pp. 162–175.

[5] Marcelo F. Frias, Carlos G. Lopez Pombo & Mariano Miguel Moscato (2007): *Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications*. In Orma Grumberg & Michael Huth, editors: *Proceedings of the 13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, *Lecture Notes in Computer Science* 4424, Springer-Verlag, Braga, Portugal, pp. 587–601.

[6] Marcelo F Frias, Carlos G Lopez Pombo & Mariano M Moscato (2007): *Alloy Analyzer+ PVS in the analysis and verification of Alloy specifications*, pp. 587–601.

[7] Joseph A. Goguen & Rod M. Burstall (1984): *Introducing Institutions*. In Edmund M. Clarke & Dexter Kozen, editors: *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, *Lecture Notes in Computer Science* 184, Springer-Verlag, pp. 221–256.

[8] Object Management Group (2004): *Object Constraint Language Specification*. Object Management Group. Version 1.5.

[9] Object Management Group (2004): *OMG SysML specification coversheet*. Object Management Group. Version 1.0.

[10] Daniel Jackson (2002): *Alloy: a lightweight object modelling notation*. *ACM Transactions on Software Engineering and Methodology* 11(2), pp. 256–290.

[11] Daniel Jackson (2006): *Software Abstractions - Logic, Language, and Analysis*.

[12] Carlos G. Lopez Pombo (2007): *Fork algebras as a tool for reasoning across heterogeneous specifications*. Ph.D. thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Promotor: Marcelo F. Frias.

[13] Carlos G. Lopez Pombo, Sam Owre & Natarajan Shankar (2002): *A semantic embedding of the* $\mathbf{A_g}$ *dynamic logic in PVS*. Technical Report SRI-CSL-02-04, Computer Science Laboratory, SRI International.

[14] José Meseguer (1989): *General logics*. In Heinz-Dieter Ebbinghaus, José Fernandez-Prida, Manuel Garrido, Daniel Lascar & Mario Rodríguez Artalejo, editors: *Proceedings of the Logic Colloquium '87*, 129, North Holland, Granada, Spain, pp. 275–329.

[15] Mariano M Moscato, Carlos G López Pombo & Marcelo F Frias (2010): *Dynamite 2.0: new features based on UnSAT-core extraction to improve verification of software requirements*, pp. 275–289.

[16] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar & Mandayam Srivas (1996): *PVS: Combining specification, proof checking, and model checking*. In Rajeev Alur & Thomas A. Henzinger, editors: *Proceedings of the 9th. Computer Aided Verification (CAV)*, *Lecture Notes in Computer Science* 1102, Springer-Verlag, New Brunswick, NJ, pp. 411–414.

[17] Sam Owre, John Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*.

[18] Sam Owre, John M. Rushby, Natarajan Shankar & David Stringer-Calvert (1998): *PVS: an experience report*. In Dieter Hutter, Werner Stephan, Paolo Traverso & Markus Ullman, editors: *Proceedings of Applied Formal Methods – (FM-Trends) '98*, *Lecture Notes in Computer Science* 1641, Springer-Verlag, Boppard, Germany, pp. 338–345.

[19] Jens U. Skakkebæk & Natarajan Shankar (1993): *A duration calculus proof checker: Using PVS as a semantic framework*. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International.

[20] Andrzej Tarlecki (1996): *Moving between logical systems*. In Magne Haveraaen, Olaf Owe & Ole-Johan Dahl, editors: *Selected papers from the 11th Workshop on Specification of Abstract Data Types Joint with*

the 8th COMPASS Workshop on Recent Trends in Data Type Specification, *Lecture Notes in Computer Science* 1130, Springer-Verlag, pp. 478–502.