Itacio: A Component Model for Verifying Software at Construction Time

Agustín Cernuda del Río
R&D Department
Seresco, S.A.
C /Matemático Pedrayes, 23
33005 Oviedo, Spain
+34 8 5235364
guticr@seresco.es
guticr@arrakis.es

Jose Emilio Labra Gayo
Department of Computer Science
University of Oviedo
C/Calvo Sotelo S/N
33007 Oviedo, Spain
+34 8 5103394
labra@lsi.uniovi.es

Juan Manuel Cueva Lovelle
Department of Computer Science
University of Oviedo
C/Calvo Sotelo S/N
33007 Oviedo, Spain
+34 8 5103396
cueva@lsi.uniovi.es

ABSTRACT

In spite of the continuous improvement of the development processes and methodologies, the generalised construction of reliable software is still to come. Not only because many of the recommended engineering practices are not sufficiently applied, but also for the lack of available techniques to achieve the desired robustness. In many real-world cases, the formal specification and verification techniques developed so far seem too abstract and intimidating to developers.

In this paper, a component-based approach for improving the reliability of software development is presented. First, the problem and its proposed solution are stated: a discussion is made about the role of components as a potential new way for building *correct-by-construction* software.

A generic component model for this purpose is defined, under the restrictions of conceptual simplicity and real-world potential applicability. This generic model is then instantiated at different abstraction levels related to the development process.

The model serves as a means of expressing constraints for the components; a specific component lattice produces a constraint lattice which can be statically evaluated to find mismatches and potential errors.

Keywords

Components, component model, software verification, component specification, contracts.

1 INTRODUCTION

The Problem

Building reliable software is a very difficult task, a task whose achievement by the software development community is still far. Even in an ideal scenario where requirement collection or requirement changing problems (and any other human-factor ones) are completely ruled out, the technical construction process itself is a challenge. Due to the intrinsic high complexity of software [2], and due also to the discrete nature of software systems, a small mistake (in any of the development stages) which goes beyond the different testing filters can lead to global system crashes or unpredicted behaviour.

It is well known that many of the quality problems associated with software arise from human and economic factors, such as changing requirements, excessive time pressure, bad engineering practices, poor quality management and the like [8]. But the intrinsic difficulty of the technical process of building software (from coding up to system integration) should not be underestimated.

Traditional Solutions

Usually, the possibility of defects arising from the construction process itself is undertaken by means of testing techniques. Test cases are carefully prepared to verify the behaviour of complete, running elements, with the help of sound methods and heuristic criteria, in order to verify anticipated error-prone situations.

In some sense, this is essentially a brute-force approach. Not because test cases are randomly generated –in fact, they are not; they are very specific artefacts-, but because test cases verify the behaviour of finished entities, and this has proven to be difficult; many defects can remain unnoticed. In combination with testing, some way of detecting defects early, during the construction process itself, should be provided. It is theoretically impossible to completely verify a computer program, but some work can be done to improve early error detection.

Taking a computer program and automatically looking through it in order to verify its behaviour is, stated that way, an extremely difficult task. But similar problems are

solved in other engineering disciplines. The main difference is that software systems are usually poorly described and specified at construction time. The key to early error detection lies in specification models that allow the automation of the verification, but whose use is suitable in a real-world scenario.

An example of this is the well-known mechanism of signature checking in compilers. Calling a function or method is a complex process; in traditional stack-based machines, the caller must push parameters in the system stack. Then, the called routine must be able to pop and use them (this implies previous knowledge about the number and size of parameters), and the result must be pushed on the stack. The caller must use this return value. In addition, the stack must be cleaned up.

There are several steps where fatal errors can be made: pushing an incorrect number of parameters, mismatching size of expected and actual parameters, and so on. Luckily, many modern programming languages and compilers incorporate static signature checking, a mechanism that allows the detection of mismatches at compiling time. Usually, no program with this kind of mismatches goes beyond this phase, because the programmer or designer is forced to the specification of *signatures* as a part of the programming process itself.

The programmer has a high level of protection in this area. But there are no similar mechanisms for other critical areas. The internal structure of the code, the combination of objects, the proper use of an object's expected calling protocol... these facets of the development process are entirely left to the designer or programmer, who has to rely on human-readable documentation (usually limited to the description of *signatures*, something clearly incomplete). As a protection mechanism, several resources, such as pre/post-conditions and invariants [9], can be used. But these mechanisms are run-time ones, and in that sense, they are conceptually not that different from testing.

Components... What For?

What can a component-oriented approach bring to this issue? Components are successfully used in other engineering disciplines (think about electronics), and one of the advantages of using components in these cases is that components are a means to support specifications. These specifications can be verified and matched against each other at early stages of the development. Individual quality control for components benefits also from these specifications, because they give a starting point upon which to build test cases, quality control procedures and acceptance criteria.

A component abstraction, focused on conceptual simplicity (few concepts), can bring a simple, flexible, powerful foundation on which to develop a static (as opposed to *run-time*) verification system for software. This abstraction (and the special use of the word *component* in this work) starts from the problem as stated before, and not from the commonly accepted idea of *component*, limited to a coarse-grained, binary entity. Usually, components are seen only as a means of accelerating the development process (leveraging the functionality already built) or from a market-based point of view [12]. Although these approaches are in no doubt interesting, the component metaphor can open new ways of building reliable software, just as the use of signatures did (nowadays, there are a very limited number of situations in which stack corruption can be caused for parameter passing).

2 A COMPONENT MODEL

The component model presented here, code-named Itacio (pronounced e-tuh-theo) offers a sort of skeleton which is the foundation to build practical uses. As in any other abstract model, some definitions must be made.

Defining what a software component is can be a long topic in itself. Many acknowledged authors have explained their use of that word. In addition, there is a mainstream school of thought about software components, and market and packing possibilities are a central factor [12]. In this particular case, however, reliability and early verification are the main concerns.

Yet Another Definition of Software Component

Even at the risk of critics, we are forced to give yet another interpretation of the word. As stated above, the first step to be done is to describe a generic, simple and flexible component model to be applied at different levels of abstraction. Here, a component will be any *software fragment*, with well-defined frontiers or bounds. It will be something so vague -by now-, but it must be noticed that although the concept is intentionally vague, it calls for a component to have clearly defined limits.

The frontier of a component has a permeable part, which is discrete, divided into connection points. These connection points are the only way of interaction for a component. There are two kinds of connection points: sources (which supply information from the component to outside) and sinks (which receive information and carry it towards the component).

Instantiation

This definition extends the use of the component concept far beyond the binary entity to which it is usually applied. So, by means of an *instantiation* process, the concept can be translated to refer to a library, a whole executable program, a class definition, a procedure, a loop, or an operator. This is an essential advantage of this open approach. More on this later.

Verifying Component Composition

It is obvious that components become especially interesting when they are composed. Components are for composition [12, p.3]. Starting from the previous definition of software

component, there is no other way to compose software components than using their connection points, by joining the sources of the frontier of a component with the sinks of the frontier of another component, and vice versa.

Apart from the enumeration of components and their frontiers, with sources and sinks, where is the promised specification? It resides on the component itself, and the specification refers to the connection points. Each connection point is associated with several *restrictive expressions*. Sources offer *guarantees*, and sinks pose *requirements*.

Restrictive expressions should be statically, automatically, unambiguously verifiable. Starting from a component connection lattice, the generic or parameterised restrictive expressions associated with connection points are instantiated as appropriate, and a sort of *knowledge base* is generated. This is, in fact, a formal description of the system under analysis; a rigorous specification that can be easily tested for correctness. Sink requirements and source guarantees must match when considered as a whole; the result of their evaluation should be a definite verdict about the suitability of the composition. In addition, the verification process can pinpoint the invalid unions, offering specific hints about the reasons of mismatches and how to correct them.

Static verification is a very important feature of this model, as stated in the problem description. *Executable* guarantees and requirements, such as pre/post-conditions, cannot be verified at design time; they have effect only at run-time.

The Role of Logic Programming

In the research activities currently in progress, restrictive expressions are supported by means of first-order logic predicates [10], restricted to the form of Horn clauses (such as the ones used in the Prolog language). These predicates offer several important advantages:

- There is a lot of accumulated experience in using their underlying mathematical foundation (first-order logic) and its real-world implementation (logic programming, inference engines).
- Logic language is appropriate for humans to express declarative ideas such as requirements and guarantees.
- A logic specification can be evaluated and a Boolean (true or false) correctness conclusion can be obtained, with no ambiguity.
- First-order predicates (specifically, Horn clauses) offer a flexible way to express almost any kind of restriction.
- The verification process described here –generating a sort of *knowledge base* at design time and then inferring- is nicely supported by a logic programming system.
- Advances in logic programming, such as Constraint Logic Programming (CLP [4]) extend the application field to new domains. They are especially suitable for stating restrictive expressions over values and domains, providing a powerful, very suitable tool for component specification as described here.

  In this scope, a high percentage of restrictions can refer to values (numeric or not) and valid ranges. In fact, passing invalid numeric values to subroutines which cannot operate with them is a usual source of errors. A traditional logic programming system, such as Prolog, would be unable to handle this kind of restrictions, whereas a CLP system is specially designed to do it.

3 MODEL INSTANTIATION

The model described before offers significant advantages. Component specifications include the enumeration of their connection points and the restrictive expressions associated with them. Starting from component specifications (or building new ones as needed) and combining them in the design/programming process results in a component lattice, which can be translated into a knowledge base –in fact, a verifiable specification-.

This way of working is very general, and can be scaled to fit several development stages and abstraction levels. The instantiation of the model (mapping the concepts of components and connection points with real software entities) fills the gap between the theory and the real development. As an example, we expect this model to be applied at the following levels:

Micro-components

A high percentage of the defects of a program is originated at the coding level. However, coding is usually left to the programmer's cleverness. There seems to be no way of verifying programs correctness at coding level, covering topics such as: fulfilling parameter-passing restrictions, checking return values, using loops properly, avoiding overflow or out-of-range conditions, etc.

As stated above, the component model is a good way to support specifications. Examining real source code details, one can find that most of the constructions are easily identifiable by a programmer, and there is a limited number of them which are used to build a program. It is not very usual that a programmer needs to write a completely new structure; rather than that, a mature programmer has a finite, although wide, set of resources (what is called *idioms*).

Considering (and building) code as a set of micro-components, and applying the previous ideas, we can go back to the old "integrated circuit" metaphor. The following instantiation of the model applies:

Component	Connection points
Function	Sinks: in-parameters Sources: out-parameters, return value
Operator	Sinks: operands Sources: result
User input	Sinks: human-readable message Sources: variable value
User output	Sinks: human-readable message, variable value Sources: none
Array reading loop	Sinks: array size, array indexing Sources: array element
etc.	

By now, we have made the first experiments on a limited prototype, based on functions, operators and user input/output. The user of this prototype can draw component lattices, and C++ source code is generated. This allows building very simple calculation programs. But these programs are automatically verifiable; all of the specifications of the involved elements (components) must be fulfilled.

As an example, classical programming mistakes are detected. If a value is passed to the "square root" function, and it is not guaranteed that this value is positive, the system pinpoints the offending connection and explains the reason of the error. The same applies to zero denominators or other invalid values.

A great advantage of this system relies in its ability to perform a global analysis starting from purely local specifications. For instance, if a sink requires a non-zero value as its input, it does not mean that the source directly connected to that sink must explicitly offer exactly that guarantee. That approach would be too rigid. Instead, based upon the whole connection lattice, the system can infer that the source always offers a non-zero value, due to the combined behaviour of all the components directly or indirectly involved.

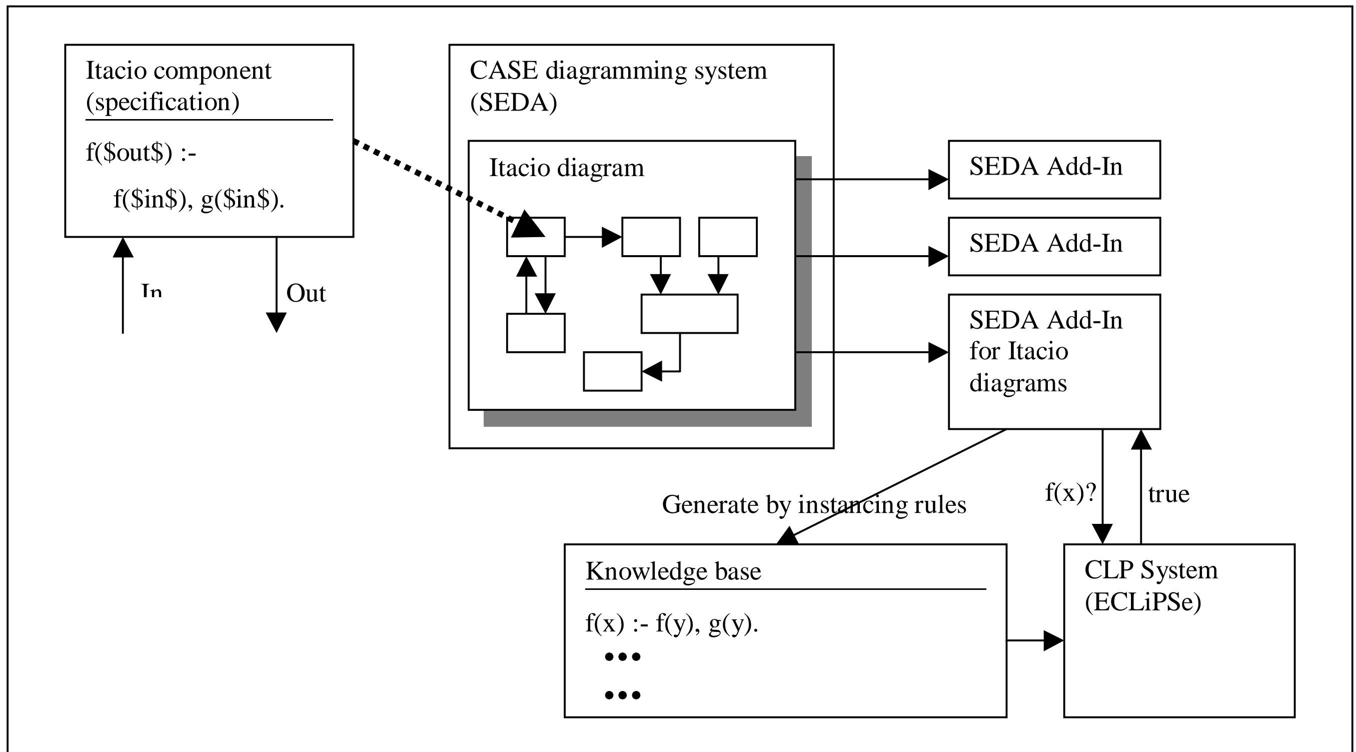

Fig. 1 – Prototype architecture

Objects and Contracts

At a higher abstraction layer, a common issue is the combination of objects or their equivalents. Several approaches have been taken in this area. The notion of *contract* has been around for a long time, and it has suffered a continuous evolution. Bertrand Meyer's [9] notion of contract refers to the unilateral commitment that a class adopts under the form of preconditions, postconditions and invariants. Wirfs-Brock, Wilkerson and Wiener [14] promote the notion of contract as a set of

unidirectional (but bilateral) collaborations between *two* participants, a client and a server. Steyaert, D'Hondt et al [11] talk about *reuse contracts*, which involve several participants and their interaction. The members of the Demeter Research Group [7] have defined the Contracts language for specifying and implementing interactions between the participants in a contract.

Nevertheless, all of these kinds of contracts have things in common. All of them are, in some way, *executable* contracts. Pre- and post-conditions and invariants have been mentioned before; other kind of specifications are useful for important purposes, but their use as an automated verification tool is also difficult.

A purely declarative specification mechanism as the one described here is, in our opinion, better suited to achieve the pursued early error detection. First, because the ideas applied in the aforementioned contract specification techniques can also be stated with restrictive expressions. Secondly, because declarative specifications are much easier to match and check.

Patterns

We believe that patterns are a specially appropriate environment for specifying and using components. Several works on architectural mismatches [1, 6] have promoted the use of domain-specific software architectures (DSSA) as a proper reuse framework. Basically, patterns and DSSA allow the components to be functionally identified; interaction patterns, collaborations, roles, etc. are easily recognised.

With patterns as a foundation, role and collaboration constraints can be identified and expressed by means of first-order logic predicates. The main concerns when integrating components at this level are:

- Proper use of interfaces. Expected method calling sequence, re-entrance constraints.
- Proper implementation of required interfaces. Method availability, parameters, types.
- Dependencies. Methods used as a support for implementing other methods, assumptions about the dependencies between other objects.

This is the kind of problems addressed by the Contracts language [7]. But, as we said, specifications written in Contracts are difficult to use as a static-checking mechanism, and static-checking is the preferred way for early error detection.

In addition, patterns offer a valuable common language for describing systems. As well as being useful as ideas for component design, patterns can even be used directly into specifications; for instance, it may be required for a component to be able to play the known role of "observer" [5] and this requirement can be expressed almost exactly like that. If components are classified taking into account these "known roles" and patterns, verification systems can also discard specific component combinations. Of course, this specific kind of specification would be useful only in some situations, provided that the model is being applied at a very high abstraction level.

Other abstraction levels

The vagueness of the initial Itacio model allows its use in other scopes by means of the appropriate instantiation process; for instance, at the architectural level, leveraging previous work in this field [1] and using the established vocabulary for stating restrictions.

4 RESEARCH STATUS

At the moment, research is still in progress. Our initial ideas have been captured in the model described before. This theoretical model must be further formalised and refined.

After that, we have developed a prototype which includes a diagramming facility for describing component lattices. This tool has been built upon an existing generic diagramming system; it offers a rudimentary support for component repositories, allows the topological definition of the connection between sinks and sources, and also the specification of restrictive expressions for components. We have made some experiments in the scope of micro-components and simple calculus programs, in order to refine the ideas presented here.

Starting from a component lattice, this system is also capable of generating a knowledge base (a specification of the whole system), instantiating the restrictive expressions as appropriate, and asking questions to this knowledge base, graphically pinpointing the invalid connections between components and giving hints about the cause of the error. This is done by interfacing with a CLP (Constraint Logic Programming) system, ECLiPSe [3, 13].

The prototype generates also a C++ program, equivalent to the component lattice, although this program is not optimal C++ code (in fact, code generation is still outside the scope of this project). Constraint logic programming is also used at this stage, to find an appropriate sequence for code generation based on the dependencies between components.

We are working on the scope of micro-components, and we also expect to apply this model at other abstraction levels. The development of these instantiation activities is currently the main task being carried on, and we expect it will be so in the near future.

5 PROS AND CONS

To summarise, the main benefits of using this schema can be stated as follows:

- Itacio offers a simple, clear model which can be easily understood by programmers and engineers without a long, specialised training process.
- Programs are built by joining components in a real composition process. This process can be validated at early times, without involving testing and run-time phases.
- Verification is a formal and automatic process, and its results are unambiguous. The developers do not have to rely on their particular interpretation of some human-readable, ambiguous, incomplete documentation.
- If a union is not guaranteed to be correct it is automatically considered to be incorrect. This has an important impact in quality and error detection. By contrast, testing techniques can only show the presence of errors, not giving much confidence about their absence.
- Detected errors can be self-explained if restrictive expressions are written with the purpose of doing so.
- This is a real-world technique. As well as no special skills being required, the underlying concepts and their implementation do not call for revolutionary new techniques to be developed; depending on the desired level of automation, even a simple logic programming system can support the basic requirements.
- With some additional work in code generation and reverse-engineering, this schema could reach higher productivity degrees.
- This model does not leave out previous advances in contract specification. All of the previous work –IDL, Contracts, DSSA- can be integrated in Itacio, simply by incorporating it in a first-order-logic, declarative notation.
- The specification of a component is a good starting point for generating test cases and measuring quality as the conformance of the component with its specification.
- We must admit that a component may be defined by a specification that it does not actually fulfil. The problem of verifying if a program implements what it promises has no solution. However, at least specified components would *promise something* clearly and explicitly; the vast majority of current commercial component platforms do not characterise components but with signatures and data types, which are obviously insufficient.

Of course, there are also some issues to be taken into account:

- The biggest problem for this model to work is notation and standardisation. First-order logic is a very flexible expression mechanism; this is an advantage for expressing all kinds of restrictions and guarantees, but this also makes difficult to mix and match specifications from different sources. The same restriction can be expressed in many completely different ways. This problem, however, is a constant in all facets of computer systems integration.
- Although this model has been developed with real-world applicability as a main concern, a considerable amount of work needs to be done for a complete Itacio-based tool to ship. This would be mostly routine development work –an integrated development environment, graphical tools, specification repositories and the like- but the advantages of the model can be difficult to appreciate without a visible working tool.
- There are several challenges to face before the model can be considered fully functional. The most important one is to develop proposed component specification sets; the model does not require the specifications to be a universal standard, but practical guidelines would be a key success factor. Some of these sets could be a complete micro-component coverage of the structures usually used to write code, and a good integration of patterns and contracts into the model.
- It would be a great step to enrich the model with a well-defined code generation system, and with round-trip engineering tools for easier integration between the design/verification system and traditional coding tools.

6 FUTURE RESEARCH AREAS

This verification system is a goal in itself, but it opens also very important issues that are worthwhile investigating in the future. Some of them are:

- A reverse-engineering system for detecting micro-components in existing code. This is, in our opinion, an achievable goal, and it would be of great interest for improving the quality of legacy code. Reverse engineering from code to analysis/design is a very complex problem, but the gap between code and micro-components is much narrower. Even if the complete translation from source code to micro-components happens to be impossible, a partial coverage would be a great advance to detect hidden defects.
- Constraint logic programming is a powerful tool. The specification model described here can go beyond verification purposes; the inference engine could not only detect a problem, but also propose a solution, describing what requirements should be fulfilled for a validity expression to become true. At higher abstraction levels, this could be the basis for a

semiautomatic design system, able to complete unfinished designs and give advice to the designer.

- Itacio-style specifications can also be used as a means for searching components in a library, a problem often presented as a key for the success of code reutilization. Other kind of formal specifications have been used for this purpose, and of course other works have focused on signatures as a means of finding existing components [15]. An Itacio-like specification of the needed component can be matched against the specifications of the available components to find a suitable one. This is also a research field for the future.

ACKNOWLEDGEMENTS
The authors wish to thank Seresco, S.A. for giving permission to build the prototypes upon the CASE diagramming system SEDA, a subsystem of their AIDA development platform.

REFERENCES

1. Abd-Allah, A. Composing Heterogeneous Software Architectures. Doctoral Dissertation, Center for Software Engineering, University of Southern California, August 1996.
2. Booch, Grady. Object-Oriented Analysis and Design with Applications, 2nd edition. Addison-Wesley Object Technology Series, February 1994. ISBN: 0805353402.
3. ECLiPSe Web site: http://www.icparc.ic.ac.uk/eclipse
4. Frühwirt, T. et al. Constraint Logic Programming – An Informal Introduction. Technical report ECRC-93-5. European Computer-Industry Research Centre, February 1993.
5. Gamma, E. et al. Design Patterns : Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing). Addison-Wesley Pub Co., October 1995. ISBN: 0201633612
6. Gacek, C. Exploiting Domain Architectures in Software Reuse. ACM-SIGSOFT SSR'95, 1995.
7. Holland, I. The Design and Representation of Object-Oriented Components. Doctoral Dissertation, College of Computer Science, Northeastern University, 1992.
8. McConnell, Steve. Rapid Development : Taming Wild Software Schedules. Microsoft Press, July 1996. ISBN: 1556159005.
9. Meyer, Bertrand. Object-Oriented Software Construction (2nd edition). Prentice Hall, 1988.
10. Smullyan, Raymond M. First-Order Logic. Dover Pubns, February 1995. ISBN: 0486683702.
11. Steyaert, Patrick et al. Reuse contracts: Managing the evolution of reusable assets. Proceedings of OOPSLA'96, vol. 31(10) of ACM Sigplan Notices, pages 268-285. ACM Press, 1996.
12. Szyperski, Clemens. Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 1997.
13. Wallace, Mark et al. ECLiPSe: A Platform for Constraint Logic Programming. William Peney Laboratory, Imperial College, London. August 1997.
14. Wirfs-Brock, Rebecca et al. Designing Object-Oriented Software. Prentice Hall, 1990.
15. Zaremski, A. M. and Wing, J. M. Signature Matching: A Key to Reuse. School of Computer Science, Carnegie Mellon University, 1998.