A Model for Integrating Knowledge into Component-Based Software Development

Agustín Cernuda del Río **Jose Emilio Labra Gayo** **Juan Manuel Cueva Lovelle**
Department of Computer Science - University of Oviedo
C/Calvo Sotelo S/N, 33007 Oviedo, Spain
+34 985 10 {50 94, 33 94, 33 96}
{guti, labra, cueva}@lsi.uniovi.es

ABSTRACT
Component-based software development allows to reduce development costs and shortening time-to-market, by using previously existing (or third-party) components to build software. But current component technologies usually solve cross-platform or low-level technical problems. There is a lack of available techniques for ensuring reliability during the integration process.

There is plenty of room for component misuses, which produce hidden errors which are difficult to detect. Also, the evolution and maintenance of systems can lead to new problems when active restrictions are forgotten and violated.

It is our opinion that components should incorporate additional information about assumptions and restrictions, and these statements should be automatically verifiable at construction time, in a completely static manner, prior to the testing stages.

In this paper, a component model is described which allows the integration of knowledge into component definitions, so as to check the suitability of any component combination at the earliest development stages. Then, the relationship between this abstract model and its implementation is considered. Also, the role of this model in system evolution and regressive verification is discussed.

PROBLEM DESCRIPTION
For several years, component-based software development has been presented as a silver bullet for many of the traditional problems of the software development industry. Components –self-contained pieces that can be used to build a system by assembling them, and reused across multiple projects- have been extensively used, with great success, in microelectronics or other engineering fields; however, the software development industry had not embraced this approach. Too often, computer programs are developed almost from scratch and in an artisan manner, with obvious disadvantages; higher costs (specially taking into account that much of the work has been done already by other teams), poorer quality (due to planning pressure and higher complexity of the systems) and a much longer time-to-market. Software development poses problems that are not present (or have been already solved) in other engineering fields; many of these problems have no solution by now, and they are expected to remain unsolved in the years to come [9, p.7].

The use of software components is expected to bring a shorter development cycle (by buying what can be bought instead of developing from the ground up), lower cost, and better quality (because components are supposed to be well-tested artefacts). In the last few years, a strong trend towards component-based software development has emerged [20].

However, we find that this paradigm is far from being mature, and the main problems have to do with proper knowledge management. Although components are expected to be robust, the integration process between different components can cause completely new defects [11, 12]. One reason is that component documentation is usually a bunch of human-readable usage instructions, written in natural languages and hence ambiguous; the only automatically-checkable specifications refer to the structure of component interfaces. This leaves plenty of room for component misuses; when joining components, a systems integrator can unintentionally violate the assumptions made by the component creator, be it calling order, concurrency issues, or use protocols. These defects can easily remain hidden until run time, with catastrophic consequences [15].

Moreover, the evolution and maintenance of systems can get even worse. As components and systems evolve, any change can also introduce new problems, and they can be very difficult to catch [16] . Even if a good regression test is done, there can be defects which have nothing to do with the global behaviour of the system, but with local requirements of individual components that are violated only in a few situations; in these cases, effective testing can be very difficult to achieve. So techniques are needed in order to guarantee that evolution does not induce a degradation of the system or affect its reliability [16, 19]. We think that design issues should be represented in a manner that can be processed by a machine [3] so that they can be verified whenever a change is made to the system. Pre-/post-conditions and invariants have been used for this purpose [17], but software must be executed in order for these

mechanisms to act, and in some sense, they suffer from the same difficulties as testing.

COMPONENTS AND KNOWLEDGE AS A MEANS OF VERIFYING SOFTWARE

Our research work is oriented towards this kind of problems. We think that the traditional testing stage is necessary, but many problems could be caught in a static analysis, at earlier development stages (at construction time). Many efforts have been made in the area of general-purpose static analysis of programs [2, 5, 6, 7, 14], but we think that another important key is the use of specific knowledge about the components and the codification of this knowledge in an automatically-checkable manner.

A component model can be used not only to physically organize the software, or to achieve a shorter or cheaper development cycle, but also to statically verify the software and to support the storing of knowledge about components. The term *statically*, in this context, implies that no execution of the program is needed. Our model proposes that a component's interface not only has a set of operation names and parameters, but also a set of restrictive expressions. These expressions state which conditions should be fulfilled when using the component, and also what the component can guarantee about its output. Assuming that components are *black boxes* that can only be used by interacting with their exposed interfaces, the only way of making a bad use of a component would be a bad use of these interfaces. So restrictive expressions about interfaces should detect –and hence help to prevent- any kind of misuse of the components.

As an elementary example, units of measurement are a traditional source of mismatches. Very frequently, type information does not allow distinguishing between different units; but type information is very often the only one that gets verified for proper matching. Should we decide to (automatically) verify measurement units mismatches in a system, a first solution would be the definition of a new, specific data type. But this is not always possible.

If components carry also restrictive expressions as proposed here, this problem can be addressed. The component designer/builder can write down all the relevant information, including component requirements. When two components are connected, each of the connection points can be tested for validity; the restrictive expressions associated with both components must match without leading to inconsistencies. If a component requires certain input value to be expressed in km, for instance, the element connected to that input value must offer some expression which states that the provided value is always expressed in km. Otherwise, the verification process will pinpoint the failed statement.

FIRST-ORDER LOGIC PREDICATES AND KNOWLEDGE BASE GENERATION

Our undergoing research project is focused in these ideas, and we are developing a component model for checking them. This component model (code-named Itacio [4, 13]) uses first-order logic predicates [18] in the form of Horn clauses as the means for expressing restrictions.

THE COMPONENT MODEL

Components: A *component* C is an entity which has a *frontier* F and a set of *restrictive expressions* E.

Given a component C, we will also use the notation F(C) and E(C) to designate the frontier and the set of restrictive expressions of that component. The set of all possible components is denoted by $\mathbf{C}$[1].

The *frontier* of a component is a finite set whose elements are called *connection points*. A connection point can be a *source* (*s*) or a *sink* (*k*), so F is in fact divided into two disjoint subsets:

$$F = S \cup K$$
$$S \cap K = \varnothing$$

where

$$S = \{s_1, s_2, \ldots, s_n\}$$ [2]
$$K = \{k_1, k_2, \ldots, k_m\}$$

The set of all possible connection point names is denoted by **A** (from "atoms").

We will make use of the notation S(C) and K(C) to designate respectively the set of sources and the set of sinks of a given component C. Also, notice that when several components are involved, indices are used to refer to components or their parts, and the *dot notation* may be used to qualify the connection points. Qualified names include the component name so that ambiguities are avoided; for instance, $s_i \in S(C_n)$ becomes $C_n.s_i$

Restrictive expressions are also divided into two disjoint sets: the set of *requirements*, R, and the set of *guarantees*, G. Both R and G are populated by first-order logic

---

[1] Domains will be expressed with bold fonts.

[2] Italic fonts will be used for connection points, so that they can be distinguished from indices.

predicates over the connection points of that component. The set of all possible predicates is denoted by **P**.

$$E = R \cup G$$
$$R \cap G = \varnothing$$
$$R = \{p_1(k_1, k_2, ..., k_m), \ldots, p_m(k_1, k_2, \ldots, k_m)\}$$
$$G = \{q_1(k_1, \ldots, k_m, s_1, \ldots, s_n), \ldots, q_r(k_1, \ldots, k_m, s_1, \ldots, s_n)\}$$

It can be seen that requirements refer only to sinks, whereas guarantees refer to both sinks and sources. For a given component C, G can have any cardinality, but R has always the same cardinality as K, and there is a one-to-one correspondence between requirements and sinks:

$$card(R) = card(K)$$
$$\forall\, k_i \in K\ \exists!\ p_i \in R$$

System: A *system* Ω is a finite graph (whose nodes ν are components and whose edges ε are source/sink pairs), together with a set of auxiliary predicates, $\boldsymbol{L}$ (from "library"). In this case we will use the *dot notation* in order to refer to the connection points (be it sources or sinks) of each component, as follows:

$$\Omega = \{\nu, \varepsilon, \boldsymbol{L}\}$$
$$\nu = \{C_1, \ldots, C_n\}$$
$$\varepsilon = \{(C_i.s_j, C_k.k_l)\}$$
$$\boldsymbol{L} \subset \mathbf{P}$$
$$\forall\, p \in \boldsymbol{L},\ p \text{ does not refer to any connection point}$$

We will also use the notation ν(Ω), ε(Ω) and $\boldsymbol{L}(\Omega)$ to refer to the nodes (components), the edges (connections between a source and a sink) and the auxiliary predicates of a given system Ω, respectively. The set of all possible systems is denoted by **S**. Also, whereas **P** and **A** are respectively the set of all possible predicates and atoms (and hence infinite sets), P and A are the set of all predicates and atoms used in a given system (hence finite sets). They are also denoted by P(Ω) and A(Ω).

Topological correctness: A system Ω is said to be *topologically correct* (and this fact is denoted with the predicate Tc(Ω)) if there is no isolated connection point and there is no connection point with multiple connections:

$$Tc(\Omega) \Leftrightarrow$$
$$\forall\, C_i \in \nu(\Omega),\ \forall\, s_j \in S(C_i)\ \exists!\ (C_i.s_j, C_k.k_l) \in \varepsilon(\Omega) \wedge$$
$$\forall\, C_k \in \nu(\Omega),\ \forall\, k_l \in S(C_k)\ \exists!\ (C_i.s_j, C_k.k_l) \in \varepsilon(\Omega)$$

From the previous definition, it can be proven that for a system to be topologically correct the total number of sources and sinks must be equal.

$$Tc(\Omega) \Rightarrow \Sigma\, card(S(C_i)) = \Sigma\, card(K(C_i))$$

THE VERIFICATION MODEL

Raw knowledge base: The *raw knowledge base* for Ω (denoted as $\boldsymbol{K}_r(\Omega)$) is the set of all predicates of all components in the system (provided that both sources and sinks are referred to with their qualified name to avoid ambiguities) and all auxiliary predicates. The subsets of requirements and guarantees keep separated and hence identifiable:

$$\boldsymbol{K}_r : \mathbf{S} \rightarrow \mathbf{P}^n \qquad \boldsymbol{R}_r : \mathbf{S} \rightarrow \mathbf{P}^n \qquad \boldsymbol{G}_r : \mathbf{S} \rightarrow \mathbf{P}^n$$
$$\boldsymbol{R}_r = \lambda\Omega\ .\ \{p \mid p \in R(C_i),\ C_i \in \nu(\Omega)\}$$
$$\boldsymbol{G}_r = \lambda\Omega\ .\ \{q \mid q \in G(C_i),\ C_i \in \nu(\Omega)\}$$
$$\boldsymbol{K}_r = \lambda\Omega\ .\ \boldsymbol{R}_r(\Omega) \cup \boldsymbol{G}_r(\Omega) \cup \boldsymbol{L}(\Omega)$$

At this point we should discuss the usefulness of the raw knowledge base. It has been said above that the properties of a system are not necessarily the mere addition of the properties of its components. The behaviour of a component can affect the system indirectly in remote places. On the other hand, fulfilling a component's requirements directly at its immediate neighbour component can be difficult or impossible; maybe these requirements are fulfilled indirectly by the previous combination of several components.

The raw knowledge base is easily generated for a given system, and it collects all the knowledge about each component. But it seems clear that the "local" knowledge of a component serves, by itself, a very limited purpose. In that sense, the aforementioned raw knowledge base $\boldsymbol{K}_r(\Omega)$ is not very useful. It is nothing but a concatenation of restrictive expressions of components, with no information about connections; it can be seen in its definition that $\boldsymbol{K}_r$ depends only on ν(Ω), and no reference is made to ε(Ω). It implies that any other system with the same components connected in a completely different way would lend the same raw knowledge base $\boldsymbol{K}_r(\Omega)$.

The following steps and definitions are headed towards the incorporation of all the information in ε(Ω), so that the resulting knowledge base contains all the relevant information of the system.

Atom renaming: The atom substitution function *subs* is defined as follows:

$$subs : \mathbf{P} \times \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{P}$$
$$subs(p(a_1, \ldots, a_n), a_i, b) = \sigma(p(a_1, \ldots, a_n))$$

where $\sigma = \{a_i / b\}$ is a substitution that, when applied to any predicate, produces a new predicate in which all occurrences of a_i have been simultaneously replaced by b.

Also, there is a *subsn* function which can be applied to a set of predicates:

$$subsn : \mathbf{P}^n \times \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{P}^n$$
$$subsn = \lambda P \lambda a \lambda b \,.\, \{q \mid p \in P,\ q = subs(p, a, b)\}$$

The atom alias generation function *alias* is defined as follows:

$$alias : \mathbf{P}^n \rightarrow \mathbf{A}$$
$$alias = \lambda P \,.\, b \in \mathbf{A} \,/\, b \text{ is not used in any predicate of } P$$

Knowledge base: Let $\varepsilon(\Omega) = \{e_1, e_2, \ldots, e_n\}$ be the set of edges of a given system Ω and let us suppose that $Tc(\Omega)$ holds. As said above, each element e_i will have the form (s_i, k_i) where *s* is a source of some component and *k* is a sink (dot notation has not been used in this case for the sake of readability). We define a series of transformations: for each edge, all the occurrences of both the source and the sink of that edge are substituted by the same alias, which is a generated atom name that did not exist previously in the set of expressions.

$$\boldsymbol{R}_0 = \boldsymbol{R}_r(\Omega) \qquad \boldsymbol{G}_0 = \boldsymbol{G}_r(\Omega) \qquad \boldsymbol{K}_0 = \boldsymbol{R}_0 \cup \boldsymbol{G}_0 \cup \boldsymbol{L}$$
$$f_i = alias(\boldsymbol{K}_{i-1})$$
$$\boldsymbol{R}_i = subsn(subsn(\boldsymbol{R}_{i-1}, s_i, f_i), k_i, f_i)$$
$$\boldsymbol{G}_i = subsn(subsn(\boldsymbol{G}_{i-1}, s_i, f_i), k_i, f_i)$$
$$\boldsymbol{K}_i = \boldsymbol{R}_i \cup \boldsymbol{G}_i \cup \boldsymbol{L}$$

The result of the last transformation, $\boldsymbol{K}_n$, is called the *knowledge base* for the given system Ω, and it is denoted by $\boldsymbol{K}(\Omega)$. Analogously, $\boldsymbol{R}_n$ is denoted by $\boldsymbol{R}(\Omega)$ and $\boldsymbol{G}_n$ is denoted by $\boldsymbol{G}(\Omega)$. In other words: $\boldsymbol{R}(\Omega)$ is the set of all requirements of the individual components, transformed so that if a source and a sink are connected, their names have been substituted (in all requirements) by the same atom name, a new name which is unique within the whole system. The same applies to $\boldsymbol{G}(\Omega)$. Finally, the knowledge base $\boldsymbol{K}(\Omega)$ contains the predicates of both $\boldsymbol{R}(\Omega)$ and $\boldsymbol{G}(\Omega)$.

Correctness: Let S be a system, $S \in \mathbf{S}$. S is said to be *correct* (and it is denoted by $correct(\Omega)$) $\Leftrightarrow$

$Tc(S) \quad \wedge$

$\forall\, p \in \boldsymbol{R}(\Omega)$ p is true based on the predicates in $\boldsymbol{K}(\Omega)$

Also, individual connection points can be tested for correctness; let us remember that there is a one-to-one relationship between elements of R(C) and K(C), so there is also a correspondence between each $p \in \boldsymbol{R}(\Omega)$ and some $k_i \in K(C_j)$. If p is false, then its associated sink has a non-acceptable connection.

The reason for this correction criterion is at follows. As said above, our verification strategy is based on verifying that each component has its requirements fulfilled. Since $\boldsymbol{R}(\Omega)$ contains a predicate for each component sink in the system, and that predicate is the set of requirements for that connection point, if this predicate holds, the requirements are in fact fulfilled.

At this point, the knowledge base incorporates all the knowledge we have about the system, including the information about the connection lattice. If an inference process is launched over a given predicate $p \in \boldsymbol{R}(\Omega)$, this inference process will be able to search through all the knowledge base, collecting all relevant information. Transitive relationships between requirements and guarantees are also taken into account; thanks to the substitution process described earlier, all the knowledge in $\varepsilon(\Omega)$ is implicitly included in $\boldsymbol{K}(\Omega)$.

IMPLEMENTATION ISSUES

As part of our research work, a prototype of the Itacio model has been developed to test its feasibility. In this chapter, several comments and learned lessons are collected regarding implementation.

The component notion: The above definition of component is general enough to catch any kind of requirement. The interface of the component is associated to the idea of *frontier* F, and this is nothing but a collection of sources and sinks. No type system is necessarily involved, no forceful identification between interfaces and object operations, as it usually happens in other component systems such as COM or CORBA [20]. However, it is still perfectly possible to adhere to such approaches.

This allows a huge degree of flexibility; the concept of component is intentionally vague, so any piece of software can be described as a component at any abstraction level as long as its frontier can be delimited.

Apart from this loose (and hence powerful) characterization of what a component is, the real usefulness of this model is that all components carry knowledge about their requirements and guarantees. This is not too difficult to do; in Itacio, a simple text file suffices for describing the frontier and restrictive expressions of a component.

Restrictive expressions: Because restrictive expressions are identified with first-order logic predicates (in the form of

Horn clauses), they can collect very diverse information. Also, correctness can be easily tested with an inference engine. In our prototype, a Constraint Logic Programming system is used [8, 10, 21]. This way, much of the initial goals of this project are fulfilled: the information about a component's intended use can be easily written and automatically verified.

Auxiliary predicates: Auxiliary predicates collect all the domain-specific knowledge of the environment the components are applied in. For instance, if components are identified with traditional objects or subroutines, and restrictive expressions are used to implement a usual type-checking system over them, $\boldsymbol{L}(\Omega)$ would contain all type-matching and casting rules of that type system. On the other hand, if components represent contracts between objects, $\boldsymbol{L}(\Omega)$ would probably contain the set of rules for well-formedness of contracts [16].

From local to global: As said above, the raw knowledge base lacks significant information and consists only of "local" statements, making impossible to make "transitive" reasoning. The knowledge base $\boldsymbol{K}(\Omega)$, however, implicitly carries all the information about connections. The iterative process of substitution by which $\boldsymbol{K}(\Omega)$ is built depends directly on each and every edge of $\varepsilon(\Omega)$; this renaming process implicitly relates sources and sinks by the coincidence of atom names.

This allows the verification process to be complete. All of the information contained in $\boldsymbol{K}(\Omega)$ (that is, all of the information of the system) is taken into account when verifying the fulfilment of some requirement. The inference process over a set of Horn clauses is well-known and this schema can take advantage of all this knowledge in a general, open, flexible manner, without the need of developing new complex and costly special-purpose algorithms such as the one described in [2, 6] unless it is really necessary.

Regarding implementation, $\boldsymbol{K}_r(\Omega)$ is easily generated by mere concatenation of strings or files, and the atom substitution process *subsn* is not difficult to achieve. In Itacio, atoms are marked with special characters, so that the substitution process is simply a substring search-and-replace algorithm.

Prototype: We have been working in the development of prototypes based on these ideas. After a first prototype based in a proprietary diagramming system [4] we have implemented a second one, based on a Java/XML core, a CLP system (ECLiPSe) for inferences, and a Web/VML interface (Fig. 1). All component and system specifications are collected in simple text files. A system can be verified according to the process described above; as a result, verification information is transformed into a graphical representation, rendered by means of VML (Fig. 2). The user can see the graph of components representing the system, and offending connections are pinpointed with special icons. Clicking on any connection point, the user can se an explanation about the correctness of that union.

DEVELOPMENT PROCESS ISSUES

How would this model fit in the software development process? The model is general enough to be applied in very different ways and at very different abstraction levels, from microcomponents [4] to architectural compositions [1]. The first step for the model to be used is a process of *instantiation*, for which a decision is made about what software artefacts will be identified with **C**. Next, the concepts of component sources and sinks must also be identified for the elements in **C**.

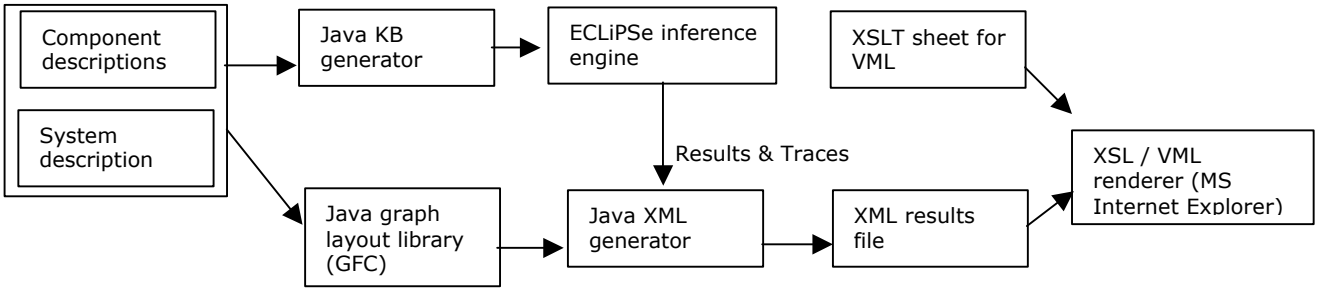

Fig. 1. *Itacio prototype architecture*

The construction of component-based software can be driven by any chosen methodology, and this is not the focus of this article. Essentially, components will be developed from scratch or reused from some other source, and then assembled. In both cases, the knowledge about each component (restrictions and guarantees) must be collected in

the component definition, which will be stored in some sort of database (and it has been said that simple text files will suffice as the simplest case). Also, any auxiliary predicates needed will also be stored in the corresponding library, $\boldsymbol{L}(\Omega)$. Of course, the process of collecting knowledge while keeping consistency and/or efficiency is not a trivial one, but it can be supported by work already done in the area of knowledge engineering.

During the assembly phase, verification can start very early. The model described here could even be slightly modified so that a knowledge base is generated upon a non-topologically-correct system; even if there are unconnected components, restrictions can be verified in a "check-as-you-add" basis. The only problem is that the knowledge base would be incomplete, so there would be the possibility that many restrictions fail, but this could be useful for the developer in practice, if he can isolate the interesting diagnostics and discard the others. When a restriction is not fulfilled, the system can pinpoint the offending connection and even explain the reason for the incorrectness, so that appropriate action (introduce additional components, modify component requirements) is taken.

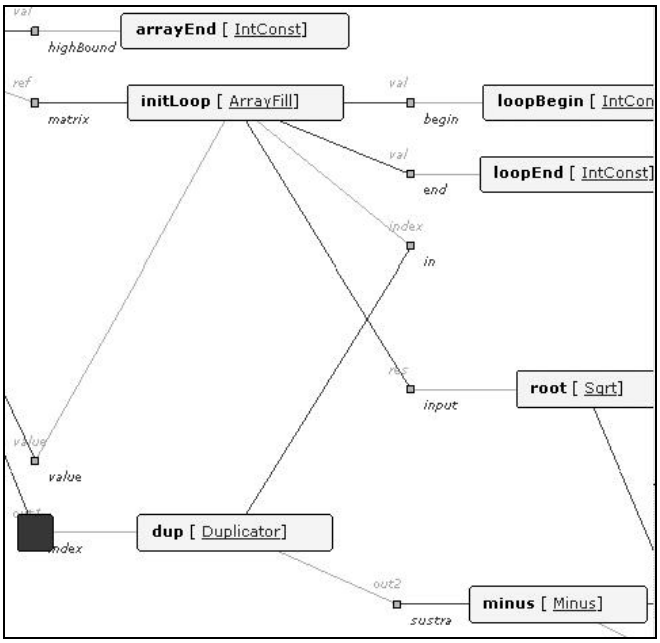

Fig. 2. *Results of a validation in Itacio (Internet Explorer screen fragment). The big square on the lower left corner is an invalid connection. Clicking on the invalid connection point it gives explanations about the failed requirements.*

This model has also a role in the development of a component in itself; the set of restrictive expressions offers an invaluable help regarding testing.Apart from the usual testing procedures based on general-purpose techniques, all of the restrictive expressions of the component should be taken into account. The problem of determining if a component behaves as it claims to behave (that is, if the restrictive expressions are, in fact, true) has no known universal solution, but it can ultimately be addressed by means of traditional testing and quality assurance techniques, by applying the Itacio model at inner abstraction levels, or by using any other verification-and-validation technique.

Evolution: The problem of system evolution can greatly benefit from this approach. Whenever a change is made to the system by substituting a component (be it with a newer version or with different components) the restrictive expressions for that component will be updated accordingly; in this case, a completely automatic process can generate again the whole knowledge base $\boldsymbol{K}(\Omega)$ and make all the corresponding tests over $\boldsymbol{R}(\Omega)$, detecting any violation of a requirement, even if that requirement is indirectly affected by the change as the result of a chain of influences.

If the change is being made inside a component C without substituting it, the knowledge stored in E(C) will serve again as a guide for proper testing; E(C) contains what a component promises to be. If these statements do not change (that is, the requirements/guarantees are not affected by the modification of the component) the test procedures developed when building the component should be reused to make a regression test (as automatic as it can be); if the behaviour of the component changes, E(C) will be updated, and the test cases will be updated accordingly, so that the component can be properly tested. Also, the impact of this behaviour change in the overall behaviour of the system can be assessed by generating $\boldsymbol{K}(\Omega)$ and verifying again if *correct*(Ω) holds.

CONCLUSIONS AND FUTURE WORK

The use of software components in industry is usually *ad-hoc* and many tools are focused on low-level issues. We think that many of the bugs that are produced when integrating components could be caught with a knowledge-based approach; much of the knowledge about restrictions or requirements is only collected in documentation and serves no purpose when it comes to automatic verification of systems, so its usefulness is simply lost.

The model presented here offers a method for integrating knowledge in the component-based software development

process, and its role in several development stages is discussed. This approach allows to enhance unit testing of components and also to make more complex deductions about the global behaviour of the system. This kind of analysis is not usually done, although relying exclusively on testing stages puts too much pressure on this activity.

As said above, we have built a second, web-based prototype of Itacio. We are working on specific applications of this prototype at different abstraction levels, such as microcomponents, contracts between objects, etc. We expect this system to grow in order to validate our ideas in a variety of fields. In the future, other research lines could be addressed, such as reverse engineering techniques for detecting components in legacy code (so that this verification model could be applied to non-component-based, legacy software), improving the use of constraint logic programming so that the system not only detects problems but also makes a reasoning to propose a solution (semi-automatic design systems), etc.

REFERENCES

1. Abd-Allah, A. *Composing Heterogeneous Software Architectures*. Doctoral Dissertation, Center for Software Engineering, University of Southern California, August 1996.
2. Bergeretti, J.F., Carré, B.A. *Information-Flow and Data-Flow Analysis of while-Programs*. ACM Transactions on Programming Languages and Systems, Vol. 7, Nº 1, Enero de 1985, págs. 37-61.
3. Biggerstaff, T., Richter, C. *Reusability framework, assessment and directions*. IEEE Software, págs. 41-49, Julio de 1987.
4. Cernuda, A., Labra, J. E., Cueva, J. M. *Itacio: A Component Model for Verifying Software at Construction Time*. III ICSE Workshop on CBSE. 5-6 June 2000, Limerick, Ireland. http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html
5. Cousot, P., Cousot, R. *Static Determination of Dynamic Properties of Programs*. Proceedings of the 2nd International Symposium on Programming. Editor: B. Robinet. Paris, 13-15 april, 1976.
6. Cousot, P., Halbwachs, N. *Automatic Discovery of Linear Restraints Among Variables of a Program*. Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. Tucson, Arizona, 23-25 january, 1978.
7. Cousot, P. *Abstract Interpretation*. ACM Computing Surveys, Vol. 28, Nº 2, Junio de 1996.
8. ECLiPSe Web site: http://www.icparc.ic.ac.uk/eclipse
9. Finkelstein, Anthony and Kramer, Jeff. *Software Engineering: A Roadmap. The Future of Software Engineering*, 22nd International Conference on Software Engineering, 2000. ACM ISBN: 1-58113-253-0.
10. Frühwirt, T. et al. *Constraint Logic Programming – An Informal Introduction*. Technical report ECRC-93-5. European Computer-Industry Research Centre, February 1993.
11. Gacek, C., Boehm, B. *Composing Components: How Does One Detect Potential Architectural Mismatches?* Position paper to the OMG-DARPA-MCC Workshop on Compositional Software. Center for Software Engineering (University of Southern California) 1998. http://sunset.usc.edu/TechRpts/Papers/usccse98-505.html
12. Garlan, D., Allen, R., Ockerbloom, J. *Achitectural Mismatch or Why it's hard to build systems out of existing parts*. IEEE Software, Noviembre de 1995, págs. 17-26.
13. Itacio project web page. http://i.am/itacio
14. Jones, N.D., Nielsen, F. *Abstract Interpretation: a Semantics-Based Tool for Program Analysis*. 30 june 1994.
15. Lions, J. L. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. Paris, July, 1996. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html
16. Lucas, C. *Documenting Reuse and Evolution with Reuse Contracts*. Ph.D. Dissertation, Vrije Universiteit Brussel, Belgium. September 1997.
17. Meyer, Bertrand. *Object-Oriented Software Construction* (2nd edition). Prentice Hall, 1988.
18. Smullyan, Raymond M. *First-Order Logic*. Dover Pubns, February 1995. ISBN: 0486683702.
19. 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.
20. Szyperski, Clemens. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
21. Wallace, Mark et al. *ECLiPSe: A Platform for Constraint Logic Programming*. William Peney Laboratory, Imperial College, London. August 1997.