Light Analysis of Complex Systems

Massimo Marchiori
MIT
Laboratory for Computer Science
545 Technology Square, NE43-257
Cambridge, MA 02139, USA
max@lcs.mit.edu

Keywords: Verification, requirement, robustness.

ABSTRACT

For the majority of complex systems a satisfactory analysis is highly impractical, or even impossible at the current technological level. In this paper we employ a new methodology, called light analysis, to deal with such systems. Light analysis consists of extracting from a complex system a 'light' version, which is structurally simpler. This way, in order to analyze a complex system one can effectively analyze its simpler light version. We report on successful practical applications of the method to the analysis of systems that are out of the scope of all the current techniques.

1 INTRODUCTION

For many systems, the lack of automatic or semi-automatic verification tools is becoming a serious problem, since the analysis of the correctness with respect to a requirement can turn out to be extremely expensive. In this paper, we try to solve this problem by introducing a methodology, called light analysis, tailored for the verification of complex systems w.r.t. a requirement . The idea is to focus only on a restricted subclass of 'simple' system, i.e. systems for which the verification of the requirement is feasible. Then, one extracts from every complex system a 'light' version of it, which belongs to the class of simple systems (and can therefore be analyzed). The light version is a kind of lossy approximation of the original system, built in such a way that if it satisfies the requirement, then the original system satisfies the requirement as well. This way, in order to analyze a complex system using light analysis one can just analyze its simpler light version.

In this paper we illustrate the basic concepts of the light analysis methodology, and report on successful practical applications of light analysis. The fist class of complex systems we will consider is that of conditional term rewriting systems. These systems are a perfect candidate for light analysis since: 1) they are of primary importance, being the basic paradigm on which algebraic specifications, functional programming and narrowing rely, 2) there are practically no tools for their verification w.r.t. fundamental requirements like termination, 3) there is a 'simple' subclass (that of term rewriting systems), for which many tools and criteria for verification have been developed.

Thus, we will illustrate the main features and performances of a system for the light analysis of conditional term rewriting systems. Along the way, this specific exposition will also give the chance to contemporarily present and motivate some features that should be in general proper of a system for light analysis.

Next, we report on another more recent applications of light analysis, namely the study of Java programs: using light analysis we have been able to analyze Java programs with respect to various crucial liveness properties (like e.g. deadlock-freedom, absence of starvation, responsiveness); although at an early development stage, this system has already proved its utility in a variety of concrete applications.

We also show how the light analysis methodology has some nice applications in the field of impact analysis, since it can be used to define the 'robustness' of a system.

The paper is organized as follows.

In the first section, we present some general guidelines that every light analysis should follow. The next section presents the main features of the light analysis system developed for conditional term rewriting systems. A separate section reports on the effective performances of the system. We then report on the other important practical application of light analysis to the study of liveness properties for Java programs. Next, we illustrate how light analysis can be successfully employed for impact analysis. Finally, we end with a discussion of the scope of light analysis.

2 GUIDELINES

The methodology of light analysis must obey some general guidelines. The basic ingredients for light analysis are a class of complex systems $\mathcal{C}$, a requirement $\mathcal{R}$ that has to be tested[1], and a subclass $\mathcal{S}$ of $\mathcal{C}$ which is the considered class of 'simple' systems. In order to perform the *light analysis of $\mathcal{C}$ into $\mathcal{S}$ w.r.t. $\mathcal{R}$* one needs a *lightening* $\mathcal{L}$ that obeys the following guidelines. The first two conditions are compulsory:

1. $\mathcal{L}$ is a mapping from $\mathcal{C}$ to $\mathcal{S}$.

2. For every $P \in \mathcal{L}$, we have $\mathcal{L}(P) \in \mathcal{R} \Rightarrow P \in \mathcal{R}$ (this expresses the fact that, in order to verify if a system P in

[1] In the following, we will use the set-like notation "$P \in \mathcal{R}$" to indicate that a system P satisfies the requirement $\mathcal{R}$.

$\mathcal{C}$ satisfies the requirement $\mathcal{R}$, we can verify if the simpler system $\mathcal{L}(P)$ satisfies it).

The other guidelines are, theoretically, not strictly necessary, but they strongly contribute to the practical success of light analysis:

3. This condition is essential in the analysis of large and complex systems built in a modular way: $\mathcal{L}$ should respect the modular structure of a system. That is to say, for every operator modular composition operator $\mathbf{M}$ available in the systems, $\mathcal{L}(\mathbf{M}(P_1, \ldots, P_n)) = \mathbf{M}(\mathcal{L}(P_1), \ldots, \mathcal{L}(P_n))$. This means that all the information on the modular structure of a program is retained (and so $\mathcal{L}$ can act separately on each program submodule). This way, the analysis of the simple system obtained via $\mathcal{L}$ can resort on the modular information of the original system.
4. $\mathcal{L}$ should respect simple structures, that is $\mathcal{L}(P) = P$ if $P \in \mathcal{S}$. This condition expresses the fact that if we have already a simple system P, then it doesn't need to be 'lightened' any more. Besides intuitively rather natural, this condition helps to maintain readable the obtained lightened code, since it ensures that only the complex parts of a system are really modified, while the simple parts are left untouched,
5. $\mathcal{L}$ should be "reasonably efficient". This is, of course, a subjective condition. For instance, if the transformation $\mathcal{L}$ is computationally too expensive (e.g. it has exponential time complexity) it is rather difficult that it can be of some practical use. The same happens if the obtained lightened system is extremely long with respect to the original one. Thus, a minimal rationale for this condition to be satisfied is for instance:

a. $\mathcal{L}$ has polynomial complexity;

b. for a reasonable metric μ of the size of the programs, like Lines of Code (LOC), or Program Volume (V) etc., we have that $\mathcal{L}$ increases linearly, that is to say $\mu(\mathcal{L}(P)) \leq k \cdot \mu(P)$ (and k is preferably small). For example, if μ =V and $k = 2$ then the volume of a program can at most duplicate when $\mathcal{L}$ is applied.

6. $\mathcal{L}$ should be "reasonably powerful". In view of its approximation character, it may happen that light analysis is not successful in the 100% of the cases: that is to say, there may be a complex system P which satisfies $\mathcal{R}$, but such that its light version $\mathcal{L}(P)$ does not. Saying that $\mathcal{L}$ is "reasonably powerful" means that the probability of success of light analysis should be adequate to the user's needs. An approximate measure of this success can be computed for instance by selecting some sample systems satisfying $\mathcal{R}$, and computing the ratio between how many of these have a light version satisfying $\mathcal{R}$ (viz. they can be proven to satisfy $\mathcal{R}$ using light analysis), and the total number of these systems. We call such measure a *power estimation ratio* (p.e.r.) of $\mathcal{L}$. Therefore, a power estimation ratio of 0.4 indicates that, roughly, out of ten systems satisfying $\mathcal{R}$, four can be proven to satisfy $\mathcal{R}$ using light analysis. Of course, the p.e.r. can be calculated/updated "on the fly" during the use of $\mathcal{L}$, this way giving a better and better approximation of the power of the light analysis. Whether a power estimation ratio is satisfactory or not depends of course on the user's need. For some very complex systems, where the analysis is extremely heavy or impossible, even a low p.e.r. can lead to consider a lightening to be "reasonably powerful". One should also be aware that the p.e.r. is a parameter that should reflect the actual chance of success of $\mathcal{L}$: it can be the case that although a transformation has a certain 'abstract' p.e.r. (e.g. obtained by random testing), the effective p.e.r. on the class of systems the user's need is substantially different.

A limiting case of light analysis occurs when the lightening always preserves the requirement $\mathcal{R}$, that is to say every possible calculation of the power estimation ratio returns one as a result. Equivalently, this means that the first condition can be strengthened into $\mathcal{L}(P) \in \mathcal{R} \Leftrightarrow P \in \mathcal{R}$, i.e. a system satisfies the requirement if and only if its light version satisfies it. We call such a lightening *complete*. Note that complete lightenings are often used in practice, when $\mathcal{R}$ is the functional requirement (i.e. a functional specification of the program). Thus, completeness of $\mathcal{L}$ equals to say that the obtained light program calculates exactly the same things as the original one. Such complete lightenings are often used in the development and testing of complex systems, for instance to compile a program into a subkernel (e.g. operating systems), or even in the prototyping phase, where modules of the systems can be written in a high level executable specification, and tested using a complete lightening (cf. [22]). Other concepts that fits into the framework of 'complete lightening' are program development by transformation [15, 17], abstract virtual machines (cf. [6]), macro expansion, or some operations of software maintenance (see e.g. the cleaning process after restructuring described in [1]). In this paper we will not consider this limiting case, but instead focus on non complete lightenings, that is to say when light analysis really performs an approximation of a complex system, producing a 'light' system which may not be functionally equivalent to the original one. This is indeed the key motivation of light analysis: not being forced to translate all the features of the system, the analysis of the light system can be made easier and more effective.

3 PRACTICAL APPLICATIONS

Conditional Term Rewriting Systems (CTRSs for short, cf. [10]) are a perfect example of important paradigm for which there are no verification tools. On the one hand, CTRSs are of great importance, both in software implementation and specification. For instance, they: are the paradigm on which functional programming [18] and narrowing (see e.g. [14]) rely; are extensively used in computer aided proof systems (cf. [19, 23]); are the basic systems with which algebraic specifications are written (see e.g. [9, 5]); have been used as unifying paradigm to express other operational semantics [7]; are directly employed as program language in a variety of important systems like ASF, OBJ3, ACT TWO and many others (cf. [2]). On the other hand, their study is extremely difficult, and there are practically no systems able to analyze CTRSs, even w.r.t. basic requirements like termination. However, there is a subclass of CTRSs for which the situation is quite the opposite: that of Term Rewriting Systems (TRSs). TRSs are essentially CTRSs without conditions; being a much simpler paradigm, there are plenty of tools devoted to their analysis (for instance RRL, REVE, ReDuX etc.), and lot of powerful criteria to test their behaviour w.r.t. all the most important requirements. Thus, an extremely appealing task is to lighten CTRSs into TRSs: this would allow to effectively use for CTRSs all the powerful tools and criteria available for CTRSs, in a completely automatic way. The system we have developed is able to lighten CTRSs into TRSs. Two important features of the system are its flexibility and auto-tuning.

Flexibility: In order to perform light analysis, in principle one needs only a single lightening. However, if one aims

at general purpose analysis, this may be in some cases too restrictive: the problem is in the tradeoff between precision and effectiveness. On the one hand, the lightening could be too precise, in the sense that the obtained light system resembles too much the original one (so, a subsequent verification of a requirement can be rather expensive), even if a lighter version of the system could have been used. On the other hand, the opposite situation could happen, that is a too weak lightening could lose too much information of the original system, producing a system which is too lossy. It is therefore important, in practical applications of light analysis, to allow a certain degree of flexibility, providing the user with lightenings of variable power. In our system we have developed a whole family of lightenings, ranging smoothly from very 'lossy' ones to extremely 'precise' ones. A typical instruction in a CTRS is of the form $l \to r$ if *conditions*. which means that l is rewritten to r provided the condition is satisfied (the conditional part is not merely a collection of tests, since subtle parameter passing is possible, see e.g. [14]). TRSs instead, as said, are CTRSs without the conditional part, that is to say composed by much simpler rules of the form $t \to t'$ (t rewrites to t'). The system can lighten a conditional rule of a CTRS into a TRS, in a variety of ways. The conditional part has to be translated into the TRS. Thus, l must be actually rewritten into an object storing all the information on the condition and the parameters contained in l, and rewrite rules defining this object have to be introduced (cf. e.g. [13]). During this process, we can loose the evaluation of the conditional part in several ways: dropping sharing information, making some conditions less restrictive, dropping rules that prohibit interferences between the new objects introduced and the original objects (subtle interactions may lead to simplifications in the TRS that were not present in the original CTRS). Another important kind of approximation is directly on the data management: data can be approximated in such a way that very similar data can be considered to be equal; this may also be imposed to hold only for some time during the execution, and eventually the data come back to be different. The 'degree of similarity' between data can be smoothly specified via parameterized metrics (here, metric is used in the mathematical sense, that is to say the data endowed with the metric form a metric space, see e.g. [4]). The important thing to note is that all these are not only approximations but also *simplifications*, in the sense that not only the approximation w.r.t. the original CTRS becomes more and more lossy, but the structure of the TRS is increasingly made simpler.

Auto-tuning: In the current system, the user can directly choose the preferred degree of lightening, from the less powerful to the more powerful ones. This way the user can cope with the precision-effectiveness tradeoff seen above. The idea is that the user performs light analysis with a certain lightening, and if the analysis is not successful, a different lightening is tried. It would be nice, however, if this process could be automatized. The current version of the system uses a powerful auto-tuning feature, which allows to perform the above light analysis in a completely automatic way. When the auto-tuned analysis begins, the system tries to select a lightening which is in some sense appropriate for the given system, instead of simply starting from a fixed lightening (e.g. the less powerful one[2]). The system performs a static analysis of the current CTRS, and via a series of heuristics it tries to automatically select a suitable lightening. Essentially, the heuristics try to adhere to the following guidelines: 1) the lightening should produce a TRS which is simple (w.r.t. some representative metrics like program volume and other structural complexity ones), and 2) if a certain structure is already simple, the lightening should translate it in a faithful way (viz. not losing too much information). If the produced lightening is not successful for the light analysis of the requirement, the system starts an alternative search for a more suitable lightening, giving a more lossy lightening in case the verification of the requirements failed because the produced TRS was too complex, or a more precise lightening in case the produced TRS did not satisfy the requirement. The way this new refined lightening is produced is to select roughly the 'middle power' lightening between the less precise lightening (in case a more lossy lightening is required, otherwise the more powerful lightening is considered) and the last produced lightening. This way the search towards a suitable lightening proceeds rather fast (the worst number of iterations is approximately the base-2 logarithm of the total number of lightenings).

Available Requirements: Light analysis is dependent on the requirement $\mathcal{R}$ one wants to analyze. Currently, the system supports all the major requirements for CTRSs, namely termination, confluence, uniqueness of normal forms, normalization, and so on (for an exposition of all these requirements, see for instance [10]). It should be noted that, in many cases, a single lightening is able to cope with various requirements. For instance, termination and confluence can be studied using the same lightening. Indeed, it turned out that two main (families of) lightenings suffice to deal with all the aforementioned requirements. All these lightenings preserve the modular structure of a system (recall Point 3 seen in the Guidelines Section). Thus, each modular analysis for TRSs can be lifted via light analysis to a modular analysis for CTRSs, providing an extremely powerful and effective methodology for verification in the large (this is particularly useful from a practical point of view, since complex systems are likely to present an extensively modular architecture).

4 EXAMPLES

We have tested the system for the automatic verification of termination of several systems, written in a variety of languages that can be ultimately reduced to CTRSs, from functional languages like ML and Concurrent CLEAN, to specification languages like ASF. In all the cases, we employed the system with the auto-tuning feature, so requiring no manual intervention by the user for the choice of the lightening. In order to perform the verification of termination for TRSs we have employed ReDuX (a freely distributed system of average power and reasonably effective, cf. [3]) and the more powerful Semantic Labelling of [25]. We tested the system with programs ranging from some randomly chosen toy examples to effectively employed complex systems, obtaining a good percentage of successes (over 60%). In particular: We analyzed a compiler for the Linear Abstract Machine (LAM, cf. [12]), written in Concurrent CLEAN. By light analysis (via ReDuX), we were able to prove termination of the compiler. We analyzed a system performing the constraint generation for the constrained evolutionary training of Artificial Neural Networks (cf. [11]), written in ASF. Using light analysis (via Semantic Labelling), we were able to formally prove its termination. We analyzed a planning problem for civil engineering, written in ML; again, using light analysis (via ReDuX) we were able to successfully prove its termination.

[2]However, as an optional feature the user can select a fixed start lightening, disabling this initial auto-tuning step.

5 JAVA

More recently, we have applied the light analysis methodology also to the big problem of verification of liveness properties for Java programs. Here, the situation was different from the case of CTRSs, since there is no known automatic tool to study liveness properties of Java programs. So, we selected a subclass of Java, called regular Java, built a system for its liveness analysis, and then developed a lightening from Java to regular Java. It can presently cope with several crucial liveness requirements, like deadlock freedom, absence of starvation, responsiveness. The system performs various kinds of lightenings. For instance,

- it discards all the statements which are unrelevant for the verification of the requirement
- it recognizes all the Java built-in class libraries and eliminates those calls that do not give problems (to be more precise, those whose safety w.r.t. the requirement can be locally inferred via an analysis of the current module);
- it lightens 'if-then-else' constructs trying to disambiguate when possible (at the loosest level, it simply discards conditionals);
- it performs lightenings of the 'for' and 'while' constructs: the exit test conditions are analyzed, and the transformation function given by a single iteration of the for (or while) loop is calculated. Then, the tokens generating this transformation are lightened so that the final transformation is in a sufficiently tractable form. Essentially, it has to be proven that the transformation function will eventually make the exit test satisfied, which is reduced to proving that a certain function on the natural numbers decreases; 'sufficiently tractable' means that proving decreasingness is computable (there are intrinsic limits in the feasibility of this analysis due to deep algebraic reasons like Abel theorem, cf. [8], and so the lightening tries to approximate from the original function a suitable upper bound which is decreasing and computationally analyzable).

The system satisfies all the guidelines seen in the second section of the paper. As far as the power estimation ratio is concerned (6th guideline), testing the system with existing Java code freely available on the web, with code appearing on books and code developed in a software house has revealed a p.e.r. of over 90% (for instance, it can cope with all the programs in [21]). In addition, the system has full Java treatment, including all the built-in packages, multithreading, window toolkit. Another very interesting point is that the benefits of such approach have also had a big impact on software robustness, as we will see in the next section.

6 IMPACT ANALYSIS

Light analysis, for its approximation character, usually tests for requirements that are stronger than the original one. This gives nice connections with *impact analysis* (see e.g. [20, 24]), since the software verified with light analysis is stabler with respect to the given requirement $\mathcal{R}$: the losser the lightening, the stabler the software. One can also practically define using light analysis an *impact metric* measuring the *robustness* of a software w.r.t. the given requirement $\mathcal{R}$, according to how lossy is the used lightening $\mathcal{L}$: if the light analysis system is 'flexible' like the one we have reported on in this paper, then one can attribute numerical attributes of 'lossiness' to every lightening of the system, and computing the impact measure of a program in a way similar to the auto-tuning feature, looking for the lossiest lightening that makes light analysis successful). We defined impact metrics from the light analysis systems of this paper, and calculated the 'robustness' of the programs studied in the Examples Section w.r.t. the termination requirement, and of several Java programs w.r.t. various liveness requirements. Interestingly, the impact metrics so defined resembled a lot the expected behaviour: the general trend was that the more complex the system, the less stable it is; however, the difference between impact analysis and a simple complexity measure showed up in some cases, where some very simple programs turned out to have a very low robustness, while for instance a complex system like the constraint generator for the constrained evolutionary training of Artificial Neural Networks cited above turned out to have a rather high robustness (maybe thanks to its highly modular structure). In the Java case, it has been noticed by experience that programs little 'robust' according to the defined metric were less easy to analyze and debug than more robust programs. Moreover, in the great majority of the cases the lack of robustness was not due to a particular optimized code, but just to arbitrary programming choices. This led to the decision to directly enforce robustness of Java programs as a requirement: as an experiment, in the current Java software life cycle of a partner software house programmers are required to produce code that is robust according to the light analysis method. Experience has shown that imposing this requirement has led to an overall benefit in the production of correct code. This can be seen as the joint effect of several causes:

- first, requiring the code to be robust enforces the programmer to use simpler constructs in place of equivalent but less readable ones, which seems to help a lot also for the software construction process (in the long term this enhanced readability may be useful also for common team work, and for partly avoiding the legacy software problem);
- second, usage of the light analysis tool enabled to find right away many errors (mostly due to typos, like for instance mistyping the name of a control variable);
- third, once ensured certain liveness properties like deadlock freedom, absence of starvation and responsiveness, the whole debug of the program goes much faster: this, by reported experience, is especially true in the Java case, where the extremely slow execution speed often leads to the loss of big amounts of time in the cases where deadlock occurred, since programmers are led to assume the program is still correctly running, albeit very slowly (!).

7 SCOPE

We have seen that complete light analysis has already been used with success (recall the discussion in the Guidelines Section). However, it remains to clarify in what situations one should preferably try to use the general (possibly non-complete) light analysis approach. A first situation occurs when there are no verification tools for the whole class of considered complex systems, but there are some available for a subclass. An example of this situation is the CTRSs analysis presented in the paper. Another situation is when

the analysis is too complex for the actual level of knowledge. Then, searching to solve the verification problem for a restricted simpler subclass is a natural step even if one aims at a direct verification method for the whole class of systems. So, once such subclass has been found, light analysis becomes an obvious candidate if the effort to extend the analysis to the whole class seems very hard, or extremely time-consuming. An example of this case is the Java analysis previously mentioned. Light analysis is also suitable in cases where the verification tool is required to evolve with time, due to language evolution. To this respect, light analysis allows to limit the big problem of software aging (cf. [16]). The commonly used *program patches* only store the differences between the new and the old releases. Analogously, light analysis can support "verification patches": when a new version of the language (from $\mathcal{C}$ to $\mathcal{C}$') is released, we only have to provide a new lightening from $\mathcal{C}$' to $\mathcal{C}$, without rebuilding from the scratch a whole verification tool for each specific version of the language. This can be useful also to face the well-known problem of *language dialects*, where each vendor implements the language with its own particular extensions/differences: one only has to provide lightenings from each particular dialect to the common core of the language. For instance, this seems to be even more profitable in the case of Java, which is likely to undergo many evolutions and modifications[3], due to the extremely rapidly changing World Wide Web market: for instance, just to mention the major topics, Java should be soon extended in order to implement OpenDoc and be interfaced with the Virtual Reality Modeling Language; also, vendors like Netscape and Microsoft are studying how to make Java easily integrated with Javascript and ActiveX. In addition, light analysis can provide as a cost-free bonus a measure of the robustness of the software, as we have seen in the Impact Analysis Section. A final question to consider is: in case a direct verification method is available or easily possible, when is light analysis likely to be *more effective* than a direct verification method? The motivation for the success of light analysis is that if one has to verify a complex system w.r.t. a requirement, there is often no need to consider the precise structure of the system, since many details can be passed over. This is likely to happen if the requirement is not extremely precise, so that it doesn't exactly describe the behaviour of the system. For instance, if the requirement to verify is a very precise one like the functional requirement, it is unlikely that light analysis can be very superior to a direct analysis of the system (although, as seen, light analysis can be preferable for software reuse, software aging, impact analysis); on the other hand, if a requirement is less precise, like for instance termination, then it is correspondingly more likely that an approximation of the system would suffice, since many of the details are unnecessary. Thus, a reasonable answer is that the losser the requirement, the greater the chance that light analysis is more effective than another direct verification approach.

Acknowledgments

Thanks to the CWI of Amsterdam (The Netherlands) and to the University of Padua (Italy) for hosting much of this research project. Also, thanks to the anonymous referees.

[3] Like it has already happened, since in few months the Java Applet API changed between the Alpha and Beta releases, and the two releases are incompatible...

References

[1] Antonini, P., Canfora, G., and Cimitile, A. Re-engineering legacy systems to meet quality requirements: An experience report. *Proc. Int. Conference on Software Maintenance* (1994), IEEE Press, pp. 146–153.

[2] Bergstra, J., Heering, J., and Klint, P. *Algebraic Specification.* ACM Press, 1989.

[3] Bündgen, R. Reduce the redex → ReDuX. *5th RTA* (1993), vol. 690 of *LNCS*, Springer–Verlag, pp. 446–450.

[4] Folland, G. *Real Analysis: modern techniques and their applications.* John Wiley, 1984.

[5] Gaudel, M.-C. Algebraic specifications. *Software Engineer's Reference Book*, Butterworths, 1991, ch. 22.

[6] Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[7] Hannan, J., and Miller, D. From operational semantics to abstract machines. *Mathematical Structures in Computer Science 2* (1992).

[8] Herstein, I. *Topics in Algebra.* Xerox Corporation, 1988.

[9] Horebeek, I.V., and Lewi, J. *Algebraic Specifications in Software Engineering.* Springer–Verlag, 1989.

[10] Klop, J.W. Term rewriting systems. *Handbook of Logic in Computer Science*, vol. 2. Clarendon Press, Oxford, 1992, ch. 1, pp. 1–116.

[11] Kok, J.N., Marchiori, E., Marchiori, M., and Rossi, C. Evolutionary training of clp-constrained neural networks. *Proc. 2nd Int. Conf. on the Practical Application of Constraint Technology* (1996), The Practical Application Company Ltd., pp. 129–142.

[12] Lafont, Y. The linear abstract machine. *Theoretical Computer Science 59* (1988), 157–180.

[13] Marchiori, M. Unravelings and ultra-properties. *Proc. 5th Int. Conf. on Algebraic and Logic Programming* (1996), vol. 1139 of *LNCS*, Springer–Verlag, pp. 107–121.

[14] Middeldorp, A., and Hamoen, E. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing 5* (1994), 213–253.

[15] Morgan, C. *Programming from Specifications.* Prentice-Hall, 1990.

[16] Parnas, D.L. Software aging. *Proc. of the 16th Int. Conf. on Software Engineering* (1994), IEEE Press, pp. 279–287.

[17] Partsch, H.A. *Specification and Transformation of Programs: a formal approach to software development.* Springer–Verlag, 1990.

[18] Peyton Jones, S. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[19] Plaisted, D.A., Alexander, G.D., Chu, H., and Lee, S. Conditional term rewriting and first-order theorem proving. *Proc. of CTRS'92* (1992), vol. 656 of *LNCS*, Springer–Verlag, pp. 257–271.

[20] Queille, J.-P., Voidrot, J.-F., Wilde, N., and Munro, M. The impact analysis task in software maintenance: A model and a case study. *Proc. Int. Conf. on Software Maintenance* (1994), IEEE Press, pp. 234–242.

[21] Richtey, T. *Java!* New Riders Publishing, 1995.

[22] Sommerville, I. *Software Engineering*, fourth ed. Addison-Wesley, 1992.

[23] Stickel, M. Term rewriting in contemporary resolution theorem proving. *Proc. 6th RTA* (1995), J. Hsiang, Ed., vol. 914 of *LNCS*, Springer–Verlag.

[24] Yau, S.S., and Collofello, J.S. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering 6*, 6 (1980), 545–552.

[25] Zantema, H. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae 24* (1995), 89–105.