On the Use of Formal Specifications as Part of Running Programs

Giovanna Di Marzo Serugendo

Abstract. Issues related to large scale systems made of autonomous components encompass *interoperability* among independently developed software and *adaptability* to changing environmental conditions. Formal specifications are traditionally used at design time for software engineering tasks. However, recently, several attempts of using formal specifications at run-time have been realised that let envisage a future use of formal specifications at run-time that will enhance interoperability and adaptability of autonomous components.
This paper intends to highlight the potentialities of the use of formal specifications at run-time as a support for the correct execution of such components. This paper reviews and discusses the use of formal specifications at run-time from different perspectives: software engineering, run-time code evolution, adaptive middleware, trust and security, or business applications. It highlights the potentialities of the use of formal specifications at run-time as a support for interoperability and adaptability of interacting autonomous components. It identifies as well application domains and open issues related to the combination of specifications and code in the framework of large scale systems.

1 Introduction

Formal methods are traditionally used at design time as a tool for defining systems, for analysis tasks, and for model checking.

However, current and future applications' needs are different than those of traditional software. Indeed, computing paradigms such as ubiquitous, pervasive computing, or service-oriented computing imply the use of a large number of autonomous components, services or agents interacting at run-time, possibly with decentralised control, independently developed, and acting on behalf of self-interested users. Off-line verification in these cases is impossible or of limited utility. Therefore, several works have emerged that combine the use of formal methods and programming languages at run-time, in order to benefit of some functional and quality assurances at run-time. This paper reviews (non-exhaustively) some of these works: from traditional ones allowing exception handling, to more recent ones supporting interactions among independently developed components. Even though this area of research is rather young and not yet mature enough for direct and efficient application into actual systems, this paper advocates that there is a large potential of interest and benefit of using formal methods at run-time, essentially due to run-time reasoning and decoupling of code from specification. Focus is given on large scale systems made of autonomous interacting components.

Section 2 briefly reviews different domains where formal methods are used at run-time and for different purposes. Section 3 describes the potential interest of having

simultaneously formal specifications and executable code. Section 4 lists several domains where formal specifications at run-time may prove useful. Section 5 identifies several issues related to the use of formal methods at run-time. Finally, section 6 mentions some advantages and drawbacks of the use of formal methods as part of running programs.

2 Current and Emergent Practice

2.1 Design by Contract.

The most popular and probably the earliest work using formal specifications inside programs is the ”Design by Contract” paradigm of Meyer [24, 25]. The idea is to attach to each function or routine of the program a list of pre- and post-conditions. Pre- and post-conditions are assertions or logical conditions that have to hold at the entry, respectively at the output of the corresponding routine. In addition to pre- and post-conditions applying to specific routines, invariants applying to a class as a whole, i.e. which have to hold for all instances of a class, can also be defined.

The ”Design by Contract” paradigm serves different purposes. At design-time, it is used for testing, debugging and for quality assurance of the related software. The program runs are checked against the pre- post-conditions and the invariants. At runtime, it is used for exception handling. Exceptions occur when a routine cannot fulfil its contract: post-condition or invariant are violated, a called sub-routine fails, or the underlying hardware or operating system indicates an abnormal condition. Exceptions are handled by exception handlers whose goal is to restore the objects in a state where the invariants hold. If they cannot, the routine fails and throws an exception to its caller. The Eiffel programming language [23] has built-in features supporting the Design by Contract paradigm.

Trusted Components. Built on the notion of Design by Contract, the initiative of Trusted Components launched by Meyer et al. [27, 26] aims at providing software components equipped with ”specified and guaranteed quality properties”. This notion covers both assessing properties of existing components and producing proofs of correctness of some properties (specified by the contract) for newly developed components.

2.2 Proof-Carrying Code (PCC)

With similar goals to the above notion of Trusted Components, but intended for runtime decisions instead of design time implementations, the Proof-Carrying Code mechanism [29, 28] allows a host system to determine if it is safe to execute a newly received untrusted binary program. The program comes with a proof that it validates some safety properties agreed in advance. The code producer creates such a proof, the code consumer (e.g. the host system that has to execute it) then simply checks that the proof is valid given the received binary program.

More precisely, a *safety policy* is specified in advance by the consumer, and expresses the conditions under which the consumer considers the program execution to be safe. The safety policy is made of: safety rules specifying operations and their preconditions; and interface calling conventions describing post-conditions and invariants

that the code must establish. It is expressed with first-order predicate logic. The code producer performs a verification that the code she intends to furnish respects the safety policy, and provides a *proof* of the successful verification, realised through theorem proving. On receipt of the code, the consumer *validates* the proof received along with the code, through proof checking.

2.3 Run-Time Verification

Run-time verification encompasses both the use of lightweight formal methods at run-time to complement traditional methods for proving programs correctness at design-time, and the use of formal techniques for dynamic program monitoring [16, 17, 32].

Dynamic monitoring usually consists in executing the program and checking whether it conforms to a requirement specification. The most popular languages for expressing such specifications are either temporal logics, or state machines. Among the different proposals made in this field, we can mention [4], who provide a specification method for expressing the semantics (not only the syntax) of components' interfaces. The program runs concurrently with its specification and deviations from the expected specified behaviour reveals incorrectness in the program. This technique is realised without any instrumentation of the program. The interesting point here is that the component's interface specification not only describes the signature but it specifies the component's behaviour. This technique uses executable specifications written with the Abstract State Machine Language (AsmL), and the COM infrastructure for monitoring the execution of a component and checking the behavioural equivalence of the component and the concurrently executing specification.

An interesting verification tool for the logic-based alternative is provided by [10] for Java programs. In this approach, the programmer specifies Linear-Time Temporal Logic (LTL) formulae directly in the code under the form of metadata annotations. These annotations are compiled into Java bytecode as attributes; they are thus available together with the program, and subsequently used by verification tools. Another runtime verification system for Java programs, the Java PathExplorer, requiring instrumentation of the code, is provided by [18]. This tool monitors Java programs execution traces by checking them against a provided requirement specification written with Maude, a specification and verification system allowing implementation of rewriting logic. The instrumentation of the code serves to insert additional bytecode that send relevant events to an observer. The observer, which may reside on another machine, actually checks the event trace against the provided specification.

The commercial tool Temporal Rover [14] allows to insert LTL temporal assertions into a Java program under the form of comments. The Temporal Rover tool generates a new program file where these assertions are implemented, so that the validation of the temporal properties is executed as part of the program.

In an attempt to provide a kind of unifying logic encompassing the different proposals, [5] propose the temporal finite trace monitoring logic EAGLE and its Java implementation.

2.4 Ontologies

An ontology for a given domain is a description of some *shared* concepts and relationships among these concepts. Ontology usually defines a set of keywords for expressing the concepts, and for expressing the relationships among them. However, expressivity of ontology may vary from very large vocabularies to complete formal theories [15].

Ontologies are currently used as an interoperability tool for knowledge management in business applications, for autonomous agents, and for semantic Web services.

Meta-Ontologies. Meta-ontologies are algebra allowing definition of type theories, operations, and axioms. From that perspective, category theory [19], higher-order logics that define terms, operators, axioms, and provable or checkable theorems are meta-ontologies.

2.5 Trust-Based Management Systems

Trust management systems deal with security policies, credentials and trust relationships (e.g., issuers of credentials). Most trust-based management systems combine higher-order logic with a proof brought by a requester that is checked at run-time. Those systems are essentially based on delegation, and serve to authenticate and give access control to a requester [34]. Usually the requester brings the proof that a trusted third entity asserts that it is trustable or it can be granted access. Those systems have been designed for static systems, where an untrusted client performs some access control request to some trusted server [2, 6]. Similar systems for open distributed environment have also been realised, for instance [22] proposes a delegation logic including negative evidence, and delegation depth, as well as a proof of compliance for both parties involved in an interaction. The PolicyMaker system is a decentralised trust management systems [3] based on proof checking of credentials allowing entities to locally decide whether or not to accept credentials (without relying to a centralised certifying authority).

More recently, an operational model for trust-based access control in highly dynamic environment has been defined by [11]. Interacting parties maintain trust values about each other. These trust values are updated dynamically depending on positive or negative behaviour of the corresponding principal. This schema allows trust to *evolve* with time as a result of evidence, and allows to adapt the behaviour of principals consequently.

2.6 Smart Labels/Smart Tags

Smart tagging systems are already being deployed for carrying or disseminating data in the fields of healthcare, environment, and user's entertainment. For instance, in the framework of data dissemination among fixed nodes, [8] propose a delivery mechanism, based on the local exchange of data through smart tags carried by mobile users. Mobile users or mobile devices do not directly exchange smart-tags; they only disseminate data to fixed nodes when they are physically close to each other. Data information vehicled, by smart tags, is expressed as triples indicating the node being the source of

the information, the information value, and a time indication corresponding to the information generation. Smart tags maintain, store, and update these information for all visited nodes. A Bluetooth implementation of these Smart Tags has been realised in the framework of a vending machine [7]. In smart tagging systems, data remain structurally simple, and understandable by human beings, and does not actually serve as a basis for autonomous local decisions.

2.7 Self-Configuring Systems

In the field of self-configuring systems, [9] propose a model based on a service-oriented middleware able to perform dynamic binding of components (or services) based on behavioural specifications *and* on contextually non-functional requirements. The selection and binding of the component is performed at run-time and is based on the adequacy of its functional description to the user's requirements. Once a component is selected, the underlying infrastructure allocates the resources necessary for the component to execute, based on the component's non-functional requirements. Several components can be composed together (sequentially, conditionally, or in parallel) based on an execution sequence specified by the user under the form of a dependency graph.

The component's functional description is expressed in IOPE format: Input, Output, Pre-condition and Effects. Input and Output serve to describe the parameters types of the interface, while pre-condition and effects are similar to pre-condition and post-conditions of the Design by contract paradigm.

Self-configuration is obtained through adaptation to changing user's requirements and changing environmental/contextual information, which is realised thanks to the decoupling of code from those requirements and information.

This is an ongoing work: the formal language to express the IOPE information and the implementation of the middleware are under way.

2.8 Specification-Carrying Code

Specification-Carrying Software. The notion of specification-carrying software is being investigate since several years [31, 1]. This idea has been proposed initially for software engineering concerns, essentially for ensuring correct composition of software and realising correct evolution of software. Algebraic specifications and categorical diagrams are used for expressing the functionality, while co-algebraic transition systems are used to define the operational behaviour of components. The visions of this team include as well run-time generation of code from the specifications.

Alternatively, [30] propose a version where the behaviour of a component is not fully specified in all its operational details, but sufficiently in order to be used for correct self-assembly of software at run-time. Indeed, moving from the traditional use of formal methods for testing and debugging, this approach intends to replace traditional APIs with full formal specifications, understood and checked at run-time by the different components or services involved in a computation. The specification becomes the primary element and the basis for communication and interaction. This approach is currently supported by a service-oriented middleware architecture implemented in

Java, supporting specifications written either as regular expressions or in Prolog. Components offering services publish their specification, while components requesting services submit specification requests. The middleware then checks services specifications with service requests and seamlessly binds the service provider and the service consumer.

This approach has been applied to run-time code evolution [30] and as a potential solution to autonomic computing [13].

2.9 B2B Interoperability

At a larger scale, the Web-Pilarcos middleware [21] allows independently developed business applications to interoperate. The business applications are grouped into what the authors call a ”eCommunity” whose structure is defined by roles and interactions between the roles. A business application is assigned a given role if it fulfils the corresponding conformance rules. A Business Network Model (BNM) semantically describes the collaboration rules requested by each partner and defines the structure of the eCommunity. A eCommunity contract, expressed as an XML-schema, comprises the BNM as well as additional information related to the format of messages, functional and non-functional (trust, QoS, security) aspects of the different services. The Web-Pilarcos middleware supports eCommunities by providing discovery of services, eCommunity's contract management and monitoring. It checks interoperability of the different business applications, their adherence to the BNM, and maintains interoperability at the collaboration, semantic and technical levels. The Web-Pilarcos approach goes beyond traditional unified virtual enterprise systems for B2B, where all business applications have to share the same interoperability model.

2.10 Summary

We can see from the different paradigms and approaches discussed above, that the range of use of formal methods at run-time varies greatly. We will compare them from the point of view of dynamic interactions of components at run-time.

The use of design by contract at run-time is currently limited to exception handling. Both parties of the contract have to share it in advance. For trusted components, proof of properties are based on contracts, however they do not serve interoperability purposes.

Proof-Carrying code is useful for checking safety properties, agreed in advance. Usually these properties are low-level properties; they do not express functional or non-functional requirements. The code consumer needs to know the kind of program it receives. However, as advocated by [12], proofs are not the ultimate solution, since even if a proof has been positively checked, a component may nevertheless fail due to changing environmental conditions (particularly in highly volatile environment). Therefore, a more adaptable schema, as one based on evolving trust, can be more efficient.

Run-time verification is essentially meant for checking deviations of the program execution from its expected execution. In addition, dynamic monitoring of program usually reveals only errors (as traditional model checking) but cannot guarantee that the program is correct in all cases, but only in the particular traces that have been checked against the specification.

Moving from purely software engineering concerns to interacting components or agents, ontologies serve interoperability purposes. They are based on a common shared domain of concepts. They act as a powerful tool for independently developed software provided there is a common ontology.

At a more dynamic level, trust-based management systems allow the different interacting components to take security decisions based on the evolving trust values.

Self-configuring systems, specification-carrying code are attempts to replace traditional well-agreed (in advance) APIs with formal specifications understood at run-time by some middleware infrastructure. This avoids the need of having shared ontologies, or agreed contracts, thus allowing a high-degree of interaction among heterogeneously designed components.

Following the same ideas, but a larger level of granularity, B2B middleware for interoperable business applications, are addressing similar concerns: allowing interaction and run-time evolution of independently developed business applications.

As a summary, we can observe that there is a shift from pure software engineering concerns to new communication paradigms for distributed systems based on formal specifications. In addition, we can observe that in the above described approaches, the more the specification is decoupled from the code, the more they apply to coarse grain components, and the more they allow dynamic interactions among the components.

3 Potential Interest

The potential interest, we foresee of the use of formal specifications at run-time, resides essentially in the *semantic interoperability* and *adaptability* possibilities they offer for large scale systems made of autonomous independent components. The potentiality resides in the one hand on the run-time reasoning that can be performed on the specification, and on the other hand on the decoupling of concerns between the code and the specification information.

3.1 Semantic Interoperability

Formal specifications allow going far beyond interface descriptions or shared keywords or concepts. Ideally, they allow: run-time understanding of the functionality of the components they represent (useful for self-assembly of components), on-the-fly deduction of component's properties, as well as compositions of properties on which to base composition of components for obtaining new functionalities (useful for automating the composition of components).

Design by contract, and similarly proof-carrying code techniques, allow a limited form of semantic interoperability: pre- and post-conditions allow run-time checking of expected properties, but APIs must be shared among the different components. Run-time verification tools essentially serve dynamic monitoring purposes (i.e., checking deviations from a requirement's specification), and therefore have a limited utility for supporting dynamic interaction among unknown components.

Ontology-based systems provide a semantic interoperability based on the sharing of common concepts, essentially keywords. Smart-tags provide an infrastructure for disseminating and handling tags at run-time among autonomous components. The tag is

the support for interactions, however the type of tags remains limited to numerical or textual values, and do not benefit yet from richer descriptions based on formal specifications.

The most advanced techniques for realising semantic interoperability are those based on service-oriented computing, such as self-configuring systems (Subsection 2.7), specification-carrying code or B2B interoperability techniques (Subsection 2.9). An underlying middleware handles the decoupling of functional and non-functional formal specifications from services codes; of roles description from business applications. The middleware seamlessly retrieves corresponding services and applications based on the specified descriptions.

3.2 Adaptability

In addition to functional adaptability, captured by the above notion of semantic interoperability, formal methods may prove useful for satisfying non-functional requirements at run-time, particularly for systems evolving in changing environments, and needing to constantly adapt themselves.

Dependability. Covering several issues, from exception handling, to resilience to unexpected environmental conditions, dependability can be dealt with formal specifications. Indeed, as already mentioned, the design by contract favours exception handling at the level of classes. At a coarser level of granularity, non-functional requirements such as QoS, constraints, CPU requirements expressed as formal specifications may serve to guide the component's execution in order to maintain the component's requirement level of functionality. In the techniques reviewed above, Design by Contract, proof-carrying code, and run-time verification techniques allow to detect violations of expected conditions or properties, and support exception handling. More advanced techniques, such as those based on service-oriented techniques provide resilience to unexpected environmental conditions.

Uncertainty. Independently designed and developed components necessarily interact with unknown software, and necessarily deal with uncertainty in both the peer components and their environment. Proof-carrying code techniques allow executing a code only if a proof of correctness has been furnished for well specified agreed properties. Trust-based systems help components in taking run-time decisions related to both peers' or executing environment's behaviour. Those decisions are based on observations and experience. Specification-carrying code supports interaction with unknown software based on formal specifications only, and not on agreed APIs.

Security Issues. In a world where a high number of components have to interact together, do not know each other in advance, cannot fully or durably rely on peers, hosts or servers, a dynamic trust-based management system allows entities to take decisions on the basis of recent, own or shared, experiences. Such a framework allows run-time and autonomous adaptation of entities to insecure situations.

Run-time Code Evolution. Software that cannot be stopped nevertheless needs to be updated. Service-oriented computing combined with formal specifications of component's requirements and functionality provide a powerful tool for offering a 24/7 service while performing code changes.

Run-time policies. Individual components or whole workflow processes may define run-time policies or protocols related to: security, mode of operation, constraints, etc. Decoupling policies from the code, and having the policies expressed as formal specifications allows reasoning about the policies, on-the-fly understanding and checking of those policies, and more importantly allows run-time modification of the policies. For instance, in eSociety applications, such as eGovernment services, software is submitted to laws changes. Any change in the law, affects the way services have to work. For large software as those we can find in public administration, changing the software code to be compliant with the new laws, while still offering the service to the citizens, may become an impossible task. However, if policies are specified independently of the underlying code which is simply assembled so as to adhere to the policy given a user's requirement, a change in the law turns out to be a change in the corresponding policy, without any modification of the code. Service-oriented computing techniques such as self-configuring systems, specification-carrying code or the Web-Pilarcos middleware are among the techniques that better support the application of run-time policies through the decoupling of code and specifications provided as a built-in feature.

4 Applications Domains

Application domains that most likely will benefit the most from approaches based on the use of formal specifications at run-time are those made of a large number of autonomous components or devices, evolving in dynamic environments, and under uncertainty conditions. Among the techniques described in this paper, service-oriented computing techniques directly support these requirements, since the different components are independently equipped with all the necessary information (described through a formal specification) to interact with unknown software.

4.1 Ambient intelligence.

Ambient intelligence scenarios envisage devices and software agents, running in devices, that organise themselves for the wellness of their respective users: software agents interoperate and share knowledge or experiences, they gather information (e.g., road traffic), they automatically pay amount of money from e-purses, they customise rooms lights and temperature, requests for references, or build user profiles.

These applications are supported by an unobtrusive and invisible technology, which is able to take decisions, and initiatives, make proposals to the user, and negotiate. In addition, in order to fully support human beings without overloading them with requests and information, the underlying technology (devices, and agents) needs advanced means of communication for: understanding each other, gather and share knowledge, information and experience among each other, ensure their own security (data integrity, confidentiality, authentication, access control), and resources management. In distributed and decentralised environments, as those in which ambient intelligence systems will evolve, interoperable policies are closely linked with authorisation policies, or resource management.

Entities evolving in ambient intelligence systems will need to deal with different kinds of information. They are autonomous and not always able to rely on a central control entity dictating its behaviour. Therefore they must be provided with means for *understanding* and *adapting* their behaviour to changing situations and environment. Such a technology needs an infrastructure enabling agents' mutual understanding, and knowledge sharing for handling interoperability, security support, and resource management. Formal specifications provide an interoperability basis for ambient intelligence systems founded on *semantic* information exchange.

4.2 Autonomic Computing.

There is currently a growing interest in biologically inspired systems, not only from researchers but also from industry. Recent interest by IBM, as part of their *Autonomic Computing* [20] program, and by Microsoft, as part of the *Dynamic Systems Initiative*, indicates the importance of self-organisation and self-adaptation for managing distributed resources. Formal specifications provide solutions addressing *self-management of autonomic components*. Indeed, coupled with the corresponding infrastructure, they enhance *self-protection* by checking proofs of access control or interoperable compatibility, or to refuse or accept an interaction with a component that appears to be faulty or malicious. Based on a provided or collected user profile (expressed as a theory), components can *self-configure* to customise their appearance or behaviour to the user. *Self-optimisation* and *self-healing* are made possible by observation, experiences, and recommendations that allow, for instance, components to optimise the use of a pool of printers, or to alert users that faulty printers should be restarted, or refilled with paper or toner.

4.3 Services

On the one hand Web services represent a first step towards software services composition through the Web. On the other hand, efforts towards automating Web tasks have lead to the Semantic Web research works. Combined together, Semantic Web services are under investigation for allowing automating service composition on the Internet. Current Semantic Web services architectures rely on ontology for realising these automation tasks and on specific repositories. Replacing ontologies with more powerful formal specifications could allow any individual user to publish its own service on the Web (described through the specification) in a similar way as today Web pages are published, and any other system or user to use it (maybe anonymously) on the basis of required properties matching the ones of the published service. This would give rise to what could be called "Google-like" services, where instead of searching data, the user or the underlying software system searches for a particular service on the Web through a "Google-like" service browser.

5 Issues

We have identified the following issues related to the use of formal specifications at run-time.

Content. Functional description encompasses interfaces, signatures, contracts, operational behaviour. Non-functional descriptions encompass a larger range of information from QoS to constraints, to policies, to protocols, etc. The richer the information, the more powerful interactions can be envisaged, but the more power consuming the computation becomes.

Languages. From the above described approaches, we can observe that languages for expressing the information vary from simple keywords, to more structured ontologies, to algebraic specification, to different kinds of logics (temporal logics, descriptive logics, higher-order logics, etc.), and even to category theory. Here again the more expressive the language, the more powerful the management of the specifications can be, but the more difficult the corresponding automated tools are. In addition, there is no consensus yet or any emerging formal specification language allowing powerful reasoning with a reasonable need for specification processing power.

Specification Checker / Theorem Proving. In addition to the language for expressing the specification, it is necessary to have run-time efficient tools for processing them, either specification checkers or theorem provers.

Run-Time Infrastructure. Finally, it is necessary to define a run-time infrastructure supporting both the processing of formal specifications and the corresponding code execution. From the above described approaches, solutions seem to come from service-oriented architectures allowing varying degree of granularity for components (from classes to business applications), as well as a decoupled processing of the corresponding specifications (functional descriptions, policies, protocols, etc).

6 Advantages / Drawbacks

Formal specifications and automated reasoning solve interoperability problems: there is no need for compatible interfaces or exact declarations and queries. Specifications may express as well non-functional properties, (re)configuration policies, and interaction protocols allow tackling issues related to dynamic large scale systems such as adaptability to uncertain environments.

Formal specifications at run-time provide several advantages for run-time execution of decentralised autonomous software in general, for ambient intelligence scenarios and for autonomic computing systems. Among them we can cite interaction and interoperability with unknown entities, seamless integration of new entities and functionalities, possible combination of services, robustness against errors or failures.

However, there is a need for additional mechanisms and automated tools for checking the adequacy of a code with its published specification, for discovering errors, and propagating information about erroneous code, for correlating information and detecting malicious attacks.

In addition, the use of formal methods at run-time is currently slowed down because the tools (specification checkers or theorem provers) for dealing with formal methods are not efficient enough for a run-time computation of a program, or not enough automated (they still need human assistance). However, research in this field is advancing and we can foresee some advances in the use of formal methods at run-time.

7 Conclusion

This paper has reviewed different works from different domains and driven by different concerns, but with a common ”conviction” that formal specification can be helpful if used at run-time: for designing correct software, for guiding executable software, for composing services and middleware services, as a powerful tool for autonomic computing, etc. Focus has been given on interactions among independently developed autonomous components. Current service-oriented computing techniques based on a middleware supporting a decoupling of code from specifications, describing functional, non-functional, or contextual information, seem the more promising for realising future efficient systems. As advocated as well by [33] in the context of middleware services, formal semantics and reasoning will most likely be the key to ensure the interactive management of resources and services, of large-scale interactive systems, all systems that are naturally exposed to dynamic changing conditions.

The different attempts at using specifications at run-time described in this paper show an increased interest in this field from different communities. This area of research is rather young; consequently there is currently no satisfying efficient solution. Tools dealing with formal specifications are becoming more powerful; this lets presuppose that the efficient processing of formal specifications at run-time will soon become possible.

Acknowledgements

This work is supported by Swiss NSF grant 200020-105476/1.

References

1. M. Anlauff, D. Pavlovic, and D. R. Smith. Composition and refinement of evolving specifications. In *Proceedings of Workshop on Evolutionary Formal Software Development*, 2002.
2. A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, 1999.
3. M. Balze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Conference on Security and Privacy*, 1996.
4. M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Workshop on Specification and Verification of Component-Based Systems*, 2001.
5. H. Barringer, A. Goldberg, K. Havelund, , and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004*, volume 2937 of *LNCS*, pages 44–57. Springer-Verlag, 2004.
6. L. Bauer, M. A. Schneider, and E. W. Felten. A proof-carrying authorization system. Technical Report TR-638-01, Princeton University Computer Science, 2001.
7. A. Beaufour. Using Bluetooth-based Smart-Tags for Data Dissemination. In *Pervasive Computing 2002*, 2002.
8. A. Beaufour, M. Leopold, and P. Bonnet. Smart-tag based data dissemination. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, 2002.

9. U. Bellur and N. Narendra. Towards a Programming Model and Middleware Architecture for Self-Configuring Systems. In *The First International Conference on Communication Systems Software and Middleware*, 2006.
10. E. Bodden. A Lightweight LTL Runtime Verification Tool for Java. In J. Vlissides and D. Schmidt, editors, *OOPSLA Companion*, pages 306–307, 2004.
11. V. Cahill and al. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing Magazine, special issue Dealing with Uncertainty*, 2(3):52–61, 2003.
12. G. Di Marzo Serugendo and M. Deriaz. A social semantic infrastructure for decentralised systems based on specification-carrying code and trust. In D. Hales and B. Edmonds, editors, *Socially-Inspired Computing*, 2005.
13. G. Di Marzo Serugendo and M. Deriaz. Specification-Carrying Code for Self-Managed Systems. In *International Workshop on Self-Managed Systems & Services*, 2005.
14. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer-Verlag, 2000.
15. D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, 1998.
16. K. Havelund and G. Rosu, editors. *Proceedings of The Run-Time Verification Workshop (RV'01)*. Electronic Notes in Theoretical Computer Science 55 (2). Elsevier Science B. V., 2001.
17. K. Havelund and G. Rosu, editors. *Proceedings of The Run-Time Verification Workshop (RV'02)*. Electronic Notes in Theoretical Computer Science 70(4). Elsevier Science B. V., 2002.
18. K. Havelund and G. Rosu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
19. M. Johnson and C. N. G. Dampney. On Category Theory as a (meta) Ontology for Information Systems Research. In *International Conference On Formal Ontology In Information Systems (FOIS'01)*, 2001.
20. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
21. L. Kutvonen, T. Ruokolainen, J. Metso, and J. Haataja. Interoperability middleware for federated enterprise applications in Web-Pilarcos. In D. Konstantas, J.-P. Bourrires, M. Lonard, and N. Boudjlida, editors, *Interoperability of Enterprise Software and Applications*, pages 185–196, 2005.
22. N. Li, J. Feigenbaum, and B. N. Grosof. A logic-based knowledge representation for authorization with delegation. In *12th IEEE Computer Security Foundations Workshop*, 1999.
23. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
24. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.
25. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
26. B. Meyer. The grand challenge of trusted components. In *ICSE*, pages 660–667. IEEE, 2003.
27. B. Meyer, C. Mingins, and H. Schmidt. Providing trusted components to the industry. *IEEE Computer*, 31(5):104–105, 1998.
28. G. Necula. Proof-carrying code. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, 1997.
29. G. Necula and P. Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, September 1996.
30. M. Oriol and G. Di Marzo Serugendo. A disconnected service architecture for unanticipated run-time evolution of code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 2004.
31. D. Pavlovic. Towards semantics of self-adaptive software. In *Self-Adaptive Software: First International Workshop*, volume 1936 of *LNCS*, pages 50–65. Springer-Verlag, 2000.

32. O. Sokolsky and M. Viswanathan, editors. *Proceedings of The Run-Time Verification Workshop (RV'03)*. Electronic Notes in Theoretical Computer Science 89 (2). Elsevier Science B. V., 2003.
33. N. Venkatasubramanian. Safe "Composability" of Middleware Services. *Communications of the ACM*, 45(6):49–52, June 2002.
34. S. Weeks. Understanding trust management systems. In *2001 IEEE Symposium on Security and Privacy*, 2001.