

A model of components with non-regular protocols

Mario Südholt

INRIA/EMN, LINA
OBASCO project, Département Informatique
École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cedex 3, France
www.emn.fr/sudholt

Abstract. Behavioral specifications that are integrated into component interfaces are an important means for the correct construction of component-based systems. Currently, such specifications are typically limited to finite-state protocols because more expressive notions of protocol do not support reasonable basic composition properties, such as compatibility and substitutability.

In this paper, we present first results of the integration into component interfaces of a notion of non-regular protocols based on “non-regular process types” introduced by Puntigam [17]. More concretely, we present three contributions: (i) a motivation of the usefulness of non-regular protocols in the context of peer-to-peer applications, (ii) a language for non-regular protocols and an outline of a suitable formal definition, (iii) a discussion of basic composition properties and an analysis of how to adequately integrate protocol-modifying operators in the model.

1 Introduction

Component-based programming promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. Explicit interfaces are commonly seen as being a fundamental means of components for this endeavor. Interfaces are intended to impose strong restrictions on components: they should make explicit all the means for communication and coordination of components. This requires a much stronger notion of interface than is common in object-oriented programming languages, where interactions may occur in a hidden fashion, *e.g.*, through a state global to two collaborating objects.

Interfaces of most component models, many academic ones (see, *e.g.*, [22, 2]) but in particular the major industrial ones, such as Sun’s Enterprise JavaBeans (EJB) [4], define component interfaces as sets of method signatures representing the services a component provides or requires. Such interfaces do not provide much information about component implementations and the correctness of their implementation is therefore very difficult to establish. Consequently, a large number of specification methods, such as Rational Rose [11] and State

Charts [9] enable specifications for component interfaces to be separated from component implementations.

The integration of behavioral specifications into component interfaces has been recognized early as a means to solve certain deficiencies of approaches relying on such specifications, in particular, the maintainability problem of separated evolution of component specifications and implementations. By far the largest class of approaches featuring integrated behavioral specifications are component models which include regular, *i.e.*, finite-state, protocols into interfaces (see, *e.g.*, [25, 24, 3, 15, 7]). Regular protocols enable the definition of several properties, such as compatibility and substitutability, useful for correctness proofs of components and can be implemented quite simply.

While more expressive notions of protocol in interfaces are highly interesting from a programming point of view, well-understood classes of protocols, such as protocols based on context-free languages, do not offer reasonable definitions of the above-mentioned basic composition properties. Consequently, more expressive protocols are rarely used in interfaces in order to construct components (two notable exceptions being counter-constrained finite state machines [19] and protocols based on symbolic transitions systems [14]).

In this paper, we present first results in the integration of a notion of non-regular protocols into component interfaces based on the “non-regular process types” introduced by Puntigam [17]. More concretely, we present three contributions. First, we motivate the usefulness of non-regular protocols in the context of peer-to-peer (P2P) applications, which, due to their distributed and large-scale nature, benefit from a component-based structure. Second, we present a protocol language in which such protocols can be defined and show how this language can be formally grounded in Puntigam’s calculus. Third, we discuss how basic composition properties are addressed in such a component model and present an analysis how protocol operators defined for regular protocols can be integrated into components with non-regular protocols.

The paper is structured as follows. In Sect. 2, we introduce trust management in P2P applications as a motivating problem for the use of non-regular protocols. Sect. 3 presents our component model, the corresponding protocol language, and sketch its formal definition. In Sect. 4, we investigate basic composition properties and the integration of protocol-modifying operators in our model. Sect. 5 discusses related work. Finally, we conclude and discuss future work in Sect. 6.

2 Motivation: trust management in P2P applications

In order to motivate that expressive protocols are useful and later illustrate our approach, we first consider peer-to-peer (P2P) architectures. P2P applications are distributed applications which are characterized by the importance of scalability and self-organization properties because of their typically very large user base, and the use of unstable and often low-bandwidth connections on the client but also server side [20].

The need for scalability and support for reorganization has led to the development of a large number of algorithms using P2P-specific protocols, among others, for routing, data replication, and trust management in such networks. Protocols are generally very useful for system-level applications (for recent work on protocols and protocol manipulations in system-level C applications and relevant references, see [5]). However, there is reason to believe that declarative protocol descriptions are particularly interesting in the context of P2P applications. In fact, these applications do not only require relations to be managed within a protocol but also among different instances of a protocol which are executing at the same time, *e.g.*, all instances of a protocol fetching different pieces of the same video. Furthermore, P2P applications are usefully implemented using components due to their size (note that components here may mean sets of strongly encapsulated C functions).

A specific algorithm for trust management in P2P networks to which protocols can usefully be applied has been proposed by Aberer and Despotovic [1]. Their approach can be summarized as follows. Trust is computed by a statistical analysis of past transactions of agents in a network. After each transaction, agents may register complaints which are stored in a distributed, partially-replicated, data structure which is organized into a virtual binary search tree. The evaluation of trustworthiness of an agent in the network is essentially done by searching for complaints about her. Moreover, the algorithm recursively searches for complaints about complaining agents in order to judge whether the latter could have attributed complaints maliciously. Without any further control, this recursive process would obviously traverse all of the network. The crucial property of this algorithm is that the search is “localized” by a cut-off heuristic consisting in judging an agent trustworthy when a certain number of agents yields a small enough number of complaints.

In order to illustrate our approach, we consider three specific (classes of) protocols which are part of this algorithm.

- *Optimizing data transfer.* A P2P application typically stores data (complaint data in the trust algorithm) in a partially-replicated manner. In such cases, data transfer may be sped up by first choosing faster connection links to duplicated data. (This is similar to the selection of the closest/fastest mirror before downloading a popular software distribution such as Debian Linux.) To this end, a protocol can be used which, optionally, first accesses a list of links suggested by the system to locations where data replicas are stored, and then repeatedly performs three operations: opening a connection, send/receive data to evaluate the available bandwidth to/from the current connection, and a close operation. Such a protocol can obviously be concisely described using a regular, *i.e.*, finite-state, protocol.
- *Trust computation.* The algorithm for trust computation above essentially relies on a sequence of send operations of queries for trust information about an agent a_1 and the corresponding responses containing the requested data. This computation must be performed recursively because a response involving information returned from another agent a_2 results in a query about the

- trustworthiness of a_2 . A correct evaluation of trust needs the reception of all information in the right order (because the cut-off heuristic may rely on that order). Such a protocol can be concisely defined in terms of a context-free language mechanism for the specification of well-balanced nested structures.
- *Cut-off heuristic*. It is frequently the case that heuristics which iteratively gather information from an (even small) number of neighboring nodes can be described (concisely) only using protocols of context-sensitive structure, *i.e.*, whose interaction structure is not even context-free. This is, *e.g.*, the case if an interaction structure involving four neighbors is equivalent to two interleaved nested structures involving different pairs of neighbors (because, in language-theoretic terms, the word $a^n b^m c^n d^m$ cannot be generated by a context-free grammar).

In the following we investigate support for the definition of protocols of such structure and their property-based manipulation.

3 Components with non-regular protocols

There are by now a number of proposals of component models with explicit protocols, *e.g.*, CwEP [7, 6] and SOFA [15]. These models augment traditional component interfaces consisting of sets of method signatures by one or several protocols. Furthermore, some additional protocol state may be present, *e.g.*, component identities in the case of CwEP.

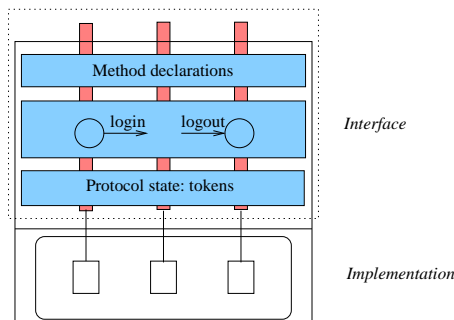


Fig. 1. Structure of components with non-regular protocols

In this paper, we consider an integration of non-regular protocols introduced by Puntigam [17]. Our components have the structure shown in Fig. 1. Components consist of an interface and a set of method implementations. The interface consists of three parts: a set of method signatures, a protocol declaration, and an additional protocol state consisting of a set of tokens used as guards governing method application. Besides being able to define more expressive protocols

than regular ones, we are also interested in providing declarative means for the description of such protocols.

Interface	::=	Signature Protocol Token
Signature	::=	(MethodSig)*
MethodSig	::=	Type Id (Formals) [Toks] [Toks]
Toks	::=	(Id[\times nat])*
Protocol, P	::=	Prim* P* P+ [P] (P \vee P) (P)
Prim	::=	Call NestedExp
Call	::=	Id (Actuals)
NestedExp	::=	[Id =] nestedExp (Call, P, Call) Id
Token	::=	(Id[\times nat] (Id = Id \times nat)*)*

Fig. 2. Syntax of non-regular protocols

3.1 Syntax and informal semantics

Concretely, we propose to define component interfaces in terms of the language generated by the grammar shown in Fig. 2. Such interfaces consist of a signature, a protocol, and token declarations. We will now discuss the form and informal semantics of these parts in turn.

The *signature* defines a set of method signatures each of which consists of five constituents: the first three are the standard return types, method identifier, and formal parameter declarations, respectively. The last two constituents define the method's dependencies on the token state: the fourth constituent specifies the tokens which must be present in the token state for the method to be applicable, in this case the specified tokens are removed from the state; the fifth constituent specifies the tokens to be added to the token state when the method is executed. Dependencies on method applicability expressed in terms of tokens enable non-regular relationships within a protocol: it is, in particular possible to make the application of a method call depend on a specific but arbitrary number of tokens, which includes ' ∞ ' representing an infinite number.

A *protocol* is a (parenthesized) sequence or choice of primitive constituents, which are method call expressions or nested expressions. A method call is of the standard form $id(params)$. Method calls are executed asynchronously, synchronization is to be performed by coordination between different components. A nested protocol consists of a protocol delimited by an entering call and an exit call. Note that nested expressions are the only protocol expressions which can be named. This enables balanced nesting to be expressed on the protocol level without forgoing automatic verification of composition properties. We will come back to this issue in the discussion of a formalization of the protocol language below.

Finally, the *token state* defines an initial set of tokens, *i.e.*, a multiset consisting of different token ids of different multiplicity. Furthermore, the declaration of the token state may define symbolic ids for tokens.

Note that while the increased expressiveness of the protocols generated by this language is due to the dependence of method acceptance on tokens, the language includes declarative support for regular and context-free structures.

3.2 P2P trust management revisited

We are now ready to formulate the protocols for P2P trust management introduced in Sect. 2 using our language:

- *Optimizing data transfer.* A protocol to evaluate the bandwidth of connections can be defined by the following protocol (to improve readability, we take the liberty in the following to separate primitive protocol expressions by semicolons.):

```
[ init; ] ( open; send ; receive ; close ; calculateBandwith)+
```

This protocol defines sequences of an optional initialization, followed by repeated sequences of bidirectional communications followed by a bandwidth computation using the finite-state constructions of our protocol language.

- *Trust computation.* Gathering of trust data up to a point when sufficient data has been collected (as determined by a suitable heuristic) can be defined as follows:

```
( G = nestedExp(sendQuery,G,getData); isSufficient )+
```

Through the use of the construct `nestedExp`, this protocol makes explicit that acquisition of trust-related data can be recursive (in which case it must be done in a well-balanced manner).

- *Cut-off heuristic.* Certain protocols which are not of context-free structure or which are difficult to describe using context-free structures can be defined (easily) using token-based protocols. As a simple example of the latter case consider a heuristic which prunes the trust data acquisition process by limiting the nesting depth to a fixed but arbitrary depth n . This heuristic can be defined by the following protocol:

```
t×n
void sendQuery(query) [t] []
( G = nestedExp(sendQuery,G,getData) ; isSufficient )+
```

This protocol first declares an appropriate number of tokens. The signature declaration of method `sendQuery` states that it can only be applied in presence of a token. Furthermore the token is removed since it is not re-injected in the token state (the last signature argument is empty). Therefore, the nested expression in the last line will be applicable exactly n times after which the token state will be empty.

3.3 Outline of a formal semantics

The protocol language shown in Fig. 1 has been designed such that its semantics can be defined by a translation into the notion of non-regular process types introduced by Puntigam [17]. This gives us a precise formal framework without reinventing the wheel. Furthermore, we can reuse notions of type correctness and subtyping on process types to investigate basic properties of component-based systems, in particular, compatibility and substitutability of components.

Since the contributions of this paper are the protocol language, motivation of its usefulness (rather than its formal definition), and mechanisms for the manipulation of such protocols, we limit the presentation of the formal underpinnings to the following two issues: a brief overview of non-regular process types as introduced by Puntigam, and an analysis of how the protocol language defined by Fig. 1 can be translated into non-regular process types.

Non-regular process types. The basic notions of non-regular process types (henceforth also referred to simply as types) can be summarized as follows. Types, denoted by $\{\bar{m}\}[s]$, consist of a set of (static) message signatures m and a (dynamic) token state s . The token state is composed of descriptors of the form $x^{p|q}$ indicating that p copies of token x exist and that the set of tokens x has been divided q times. Using divisions, tokens can be passed to collaborators, which at the same time updates the types of the sending and receiving component accordingly. The dividend p can mainly be a natural number, ‘ ∞ ’, which represents an infinite number of tokens, or an addition of two dividends. The latter case permits to gather new tokens from a collaborator; analogously to a division operation, this operation results in an update of the types of the sending and receiving component. A divisor $q = 0$ indicates that all tokens are present in the current token state.

Message signatures, denoted by $m(\bar{t})(\bar{p})[\bar{i}][\bar{o}]$, consist of an identifier m , type parameters t associated to the message, types p for formal parameters, and incoming and outgoing tokens \bar{i}, \bar{o} . Message acceptance is defined similarly as introduced previously: for a method to be acceptable, each state descriptor $x^{p|q} \in \bar{i}$ must be present in the dynamic token state of the enclosing object and if $q = 0$ that descriptor must be the only one relating to x in the token state. The token state is then updated by removing the tokens from \bar{i} and adding the tokens in \bar{o} . In this way types keep track of exact numbers of tokens and allow them to be passed around between objects.

The type-checking algorithm from [16] can also be used for the static checking of non-regular process types. Hence, conditions involving exact token numbers (including ∞) are statically checkable. Note, however, that the type system of [17] allows such conditions to be applied only to components which contain all token expressions of the types occurring in such conditions. In cases where the underlying language does not explicitly support such types, they can be passed along and checked during runtime.

Translation of the protocol language of Fig. 1. The protocol language shown in Fig. 1 has been designed in order to be based on the formalism of non-regular process types.

Our notions of token state and token manipulation through methods whose signatures make token manipulations explicit are essentially defined as in Puntigam’s approach.

Our language has two features which have no direct counterpart in non-regular types: regular expressions and nested expressions. Both of these can, however, be expressed using types. Two essential constituents of the corresponding translation are the following:

- Repetition of a method call can simply be defined using a token which is consumed from and immediately re-injected in the token state to enable the following call in the repetition.

$$m\star := \{m()(p)[x^{1|0}][x^{1|0}]\}[x^{1|0}]$$

Hence, m ’s token state initially contains one x . A call to m (with arguments conforming to p) requires one occurrence of x , which is removed from the token state as part of method acceptance but immediately re-injected.

- Nesting can be defined by a type using tokens to count method invocations. The main part of the corresponding translation is the following:

$$\text{nestedExp}(m, \epsilon, n) := \{m()(_)[x^{1|0}][x^{1|0}, y], n()(_)[x^{1|0}, y][x^{2|0}], n()(_)[x^{2|0}, y][x^{2|0}]\}[x^{1|0}]$$

Here, m generates tokens y which are consumed by n . Furthermore, once the first n has been accepted, m cannot be accepted anymore because no state $x^{1|0}$ is present in the token state anymore. Technically, this is achieved by using the dividends 1 and 2, which, intuitively speaking, separate execution of such a nested expression in two exclusive phases.

4 Protocol-based component composition

One of the main advantages of the introduction of explicit protocols in components is that composition of components can be defined in terms of protocol composition. This enables, in particular, reasoning about black-box component compositions which is almost impossible in the case of interfaces consisting of method signatures only (since almost no knowledge about component implementations is available).

In this section we present first results relating to (properties of) the composition of components based on non-regular protocols. Since there is very little directly related work, it is reasonable to start from results developed in the context of components with regular protocols. To this end we have studied the non-regular case by leveraging our previous work on the property-based composition of components based on finite-state protocols [8, 6]. Concretely, we present how two different issues related to properties of component composition carry over (or not) from the regular to the non-regular case:

- *Basic composition properties*: compatibility and substitutability, which are fundamental composition properties in component-based systems, have to be defined differently in the non-regular case compared to finite-state protocols.
- *Composition operators*: we consider the definition of operators modifying the structure of a protocol as well as operators modifying the protocol state.

4.1 Basic composition properties

In component-based programming two fundamental composition properties are traditionally considered: *compatibility* (the technical notion ensuring that components flawlessly work with each other) and *substitutability* (which addresses the question if a component can be substituted for another one without causing faulty service provision). Approaches which use finite-state protocols to make explicit part of the semantics of objects or components typically employ equivalence and containment relationships over sets of traces and failures to define such composition properties [13].

In contrast to finite-state languages, no decision procedures are known (or can even exist) for these correctness notions in the case of context-free or even more expressive languages. Non-regular process types enjoy, however, two decidable relations, *type equivalence* and *subtyping*, which can be used to similar effect. In fact, if t_1 is equivalent to (a subtype of) t_2 the set of traces generated by t_1 is equal to (larger than) the trace set of t_2 . Compatibility between collaborators can therefore be expressed in terms of compatibility of the manipulation of token states and type equivalence. Substitutability can be proven by using the subtype relation and type equivalence.

To give an example of the usefulness of these type-based relationships for substitutability of components, reconsider the cut-off heuristic introduced in the context of trust computation in P2P networks. Different cut-off heuristics which explore the complaint data base to different depths can be proved to obey subtype relationships, thus ensuring that deeper-reaching heuristics are substitutable for shallower ones (reflecting the fact that the former will yield better trust evaluations than the latter).

An important property of the type-based relationships is that they are defined for *deterministic types* only. Informally speaking, deterministic types are characterized by having a unique follow state for any application of a method declared within the type. The protocol language defined by Fig. 2 has been designed to yield only deterministic types. In particular, all types corresponding to protocols discussed in Sect. 3 are deterministic.

4.2 Composition operators for non-regular protocols

In a model of components with protocol-based interfaces, composition of components is naturally expressed through composition of protocols. Furthermore, component composition can then be supported by a set of protocol-composition operators which preserve (to a reasonable degree) fundamental composition properties, such as component substitutability.

In previous work [7, 6], we substantiated these claims for components with regular protocols. In particular, we have defined a set of operators for the modification of the static structure of finite-state automata and operators for the modification of a dynamic state associated to regular protocols. In the following we present an analysis of how such operators can be integrated into a model of components with non-regular protocols.

Before considering concrete operators, let us note that there is a fundamental difference between the regular case and the non-regular one: protocol structure is much more explicit in finite-state automata than in the non-regular process types. The former directly represent execution traces through the automata structure, while a non-regular type $\{\overline{m}\}[\overline{t}]$ represents traces only indirectly through an inductive construction. Furthermore, it would be very unwieldy to define the operators directly in terms of non-regular traces because the notions of equivalence and subtyping are constructively defined only on types not sets of traces. This issue is alleviated by using a protocol language such as that defined in Fig. 2 because its constructs directly correspond to trace sets; an example giving evidence for this claim can be found in the discussion of the definition of start states for the union composition operator below. (We strongly believe that declarative protocols even provide a reasonable solution to the issue. This is the subject of on-going work, though, and not further discussed here.)

Structural operators. We first consider the definition of three basic and fundamental structural operators, namely “union” (which, informally speaking, allows to add at a certain state new branches to protocols), “concatenation” of protocols, and general “insertion” of a protocol into another one.

Since process types do not represent the trace sets of protocols directly as explained above, the definition of structural operators based on non-regular process types therefore either has to be limited to structural properties directly expressible on the type level or includes rather complex proofs involving the trace sets generated by types.

It turns out that the three basic operators can be defined quite adequately in terms of non-regular process types:

- A union operation, denoted $\{\overline{m}_1\}[\overline{t}_1] \cup_s \{\overline{m}_2\}[\overline{t}_2]$, which adds a new protocol (*e.g.*, a new branch) to an existing protocol at a state s , can be directly defined on the type level. Informally, the method signatures and token states have to be merged, *i.e.*, the result can be defined as $\{\overline{m}_1 \oplus_m \overline{m}_2\}[\overline{t}_1 \oplus_t \overline{t}_2]$. However, the functions \oplus_m, \oplus_t cannot simply be defined to be multiset union because two technical problems have to be resolved regarding the merge:
 - Interference of the new parts and old parts of the resulting protocol has to be avoided: when the new part is executed the state related to the old part should not be affected. This can be achieved by an appropriate renaming of the methods and tokens of the newly added protocol *and* introduction of an adapter definition translating new names in case of communication with collaborators expecting old ones.
 - The definition of the starting state s is not obvious. In the case of regular protocols the state typically is one explicitly enumerated in the definition

of the corresponding finite-state automata and related to other states by the automata's transition relation. For non-regular process types, the state would have to be defined in terms of elements of the trace set generated by the original protocol, however, only the token state is directly accessible given the type definition and tokens do not represent states meaningful *w.r.t.* to positions in method sequences.

One approach to (partially) solve this problem is to introduce new tokens and modify methods of the type such that they emit these new tokens in order to mark specific positions as part of the type's token state.

Note that this technique is strongly supported by our declarative protocol language. The translation, *e.g.*, of a regular repetition such as $\star m$ into a non-regular type, can automatically yield position markers useful to define protocol states for composition operators.

- Concatenation can be treated similarly to the union operation in that it can be defined as a union operation on final states. Its definition thus reduces to the identification of final states which can frequently be done by using specific tokens as end markers.
- Insertion can also be treated using the means above, requiring mainly identification of start and end states within protocols.

In order to conclude the discussion of these operators, let us note that the properties of these operators carry over from the regular case (cf. [7, 6]) to the non-regular one. For instance, a protocol resulting from an application of the union operator above can be substituted for any of the two original protocols from the state s on.

State-manipulating operators. A second group of protocol-related operators allows the modification of the dynamic state of protocols. These include protocol-modifying operators directly working on the protocol state but also operators modifying the program execution and the protocol state at the same time. In the following, we briefly discuss the integration into our model of two (classes) of such operators which are analogous to operators put forward in the context of components with regular protocols [7, 8].

A first class of operators directly modifies the token state \bar{t} of a type $\{\bar{m}\}[\bar{t}]$. Some of these operators introduce new tokens, *e.g.*, to support the implementation of the structural operators discussed above. In this case, compatibility and substitutability properties of the protocol are preserved. Other such operators modify existing tokens, thus modifying the compatibility and substitutability properties of the corresponding protocols. The extent to which such properties still hold then have to be investigated on a case by case basis.

Another class of state-manipulating operators provide for the “spontaneous” emission of messages without directly modifying the underlying protocol. Such operators are useful, *e.g.*, in order to adapt a component temporarily. In the case of the trust management algorithm for P2P applications, such an operator could be used, *e.g.*, to temporarily augment the depth to which the underlying distributed complaint data base is explored. Composition properties of such operators also have to be proved on a case by case basis.

5 Related work

There is little directly related work, *i.e.*, work on protocols more expressive than regular ones and which, in particular, consider their constructive use as part of a component model. One notable exception is recent work by Puntigam [18] who proposes an integration of non-regular process types into a Java-like language and discusses issues of such types related to hot-swapping of components. However, he does not consider a more declarative language for protocol definition nor composition operators as discussed in this paper. Another interesting approach is Reussner’s work on counter-constraint finite state machines [19]. This approach is close to our approach in terms of expressiveness: such automata allow the definition of certain non-regular (even some context-sensitive) protocols, similar to the approach presented here. The exact relationship w.r.t. expressiveness between our approach and his approach is an interesting open research question. However, Reussner does not consider a user-level language like that presented here, does not provide an underlying typing discipline, and does not consider composition operators. Finally, another approach which goes beyond regular protocols is that by Pavel *et al* [14], who endow components with protocols based on symbolic transition systems (STS). However, that work does not include static property support as provided by non-regular process types and does not consider composition operators.

The work presented in this paper has been motivated by a lack of expressiveness of the (many) approaches using finite-state protocols. However, as exploited in this paper, work on regular protocols is still relevant for the non-regular case *w.r.t.* the kinds of properties useful for component-based programming, be it properties of protocol composition operators (*e.g.*, those presented in [7, 6]) or adaptation properties (see, *e.g.*, [25, 21, 24]).

There are several approaches using aspect-oriented techniques (in the sense introduced by Kiczales [10]) to manipulate protocols, which share many of the problems the present paper raises related to the definition of protocol-modifying operators. Walter and Viggers [23] propose an aspect language using context-free grammars to define patterns to be matched against Java source code. Their protocol language therefore is more restricted than ours. Furthermore, they do not consider any issues related to component encapsulation. Recently, we have worked [8, 6] on an aspect language for the manipulation of regular protocols. This work defines, in particular, an aspect language for protocol manipulations. Furthermore, it provides a discussion of several properties over regular protocols which are subject to aspect weaving: in particular, preservation of finitude of protocols in the presence of operators modifying the structure of protocols and techniques for the analysis of interaction properties of aspects over components with regular protocols.

There is a large body of work using specification means for non-regular protocols in the context of component-based programming. As two examples among many more let us cite the work by Braccialia *et al.*, who use protocol specifications based on the π -calculus in order to semi-automatically synthesize component adapters, and work using symbolic transition systems (STS) for component

analysis [12]. Such approaches feature notions of protocols which are even more expressive than that considered in this paper but cannot be used constructively and, for a large part, do not support automatic checking of composition properties.

6 Conclusion and further work

In this paper we have presented the integration of a notion of non-regular protocols in component interfaces. Concretely, we have presented three contributions. We have motivated that such protocols naturally arise in P2P applications and that our approach allows the concise formulation of the required protocols. We have defined a protocol language for non-regular protocols including declarative constructs for regular and context-free protocols. Furthermore, we have outlined a translation of the protocol language into Puntigam's calculus of non-regular process types. Finally, we have discussed composition properties in our setting as well as the integration of several protocol composition operators.

This paper presents a first step towards the definition of a component model with non-regular protocols and much interesting work remains to do. As to the protocol language, the current proposal provides a limited set of declarative constructs (nested and regular expressions). This set of constructs should be extended, thus reducing the use of token manipulations in protocol definitions. As to the composition properties, operators should be more deeply explored and more specific operators defined, *e.g.*, for component adaptation. Finally, there are open questions concerning the underlying formal framework, in particular its support for composition properties relevant for components and its efficient realization.

Acknowledgements. The author would like to thank the anonymous reviewers and his colleagues Jacques Noyé and Sebastien Pavel for their many helpful remarks.

References

1. K. Aberer and Z. Despotovic. Managing trust in a Peer-2-Peer information system. In Henrique Paques, Ling Liu, and David Grossman, editors, *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM-01)*, pages 310–317, New York, November 5–10 2001. ACM Press.
2. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In I. Crnkovic et al., editors, *Proc. of the 7th Int. Symposium on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *LNCS*, pages 7–22. Springer-Verlag, 2004.
3. L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proc. of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, September 10–14 2001. ACM Press.

4. L.G. DeMichiel et al. *Enterprise JavaBeansTM Specification*. SUN Microsystems, November 2003. Version 2.1, Final Release.
5. R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proc. of 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, March 2005.
6. A. Farías. *Un modèle de composants avec des protocoles explicites*. PhD thesis, École des Mines de Nantes/Université de Nantes, December 2003.
7. A. Farías and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2519 of *LNCS*, pages 995–1006, 2002.
8. A. Farías and M. Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, November 2004.
9. D. Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, March 1987.
10. G. Kiczales et al. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.
11. P. Kruchten. *Rational Unified Process: an Introduction*. Addison-Wesley, Reading, Massachusetts, USA, 1998.
12. O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In Z. Tari R. Meersman and al., editors, *Proc. of Distributed Objects and Applications (DOA'04)*, volume 3291 of *LNCS*, pages 1502–1519. Springer-Verlag, 2004.
13. O. Nierstrasz. Regular types for active objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28(10) of *ACM SigPlan Notices*, pages 1–15. ACM Press, October 1993.
14. S. Pavel, J. Noyé, P. Poizat, and J.-C. Royer. Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, LNCS. Springer-Verlag, April 2005. To appear.
15. F. Plasil and S. Visnovsky. Behavior protocols for software components. In *Transactions on Software Engineering*. IEEE, January 2002.
16. F. Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388. Springer-Verlag, 1997.
17. F. Puntigam. Non-regular process types. In P. Amestoy et al., editors, *Proceedings of the 5th European Conference on Parallel Processing (Euro-Par'99)*, number 1685 in LNCS, Toulouse, France, September 1999. Springer-Verlag.
18. F. Puntigam. State information in statically checked interfaces. 8th Int. WS on Component-Oriented Programming at ECOOP, July 2003.
19. R. H. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In B. Grosky, F. Plasil, and A. Krenek, editors, *Proc. of the 29th Annual Conference in Current Trends in Theory and Practice of Informatics (SOFSEM 2002)*, Milovy, Tschechische Republik, volume 2540 of *LNCS*. Springer-Verlag, November 2002.
20. M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE In-*

- ternet Computing Journal*, 6(1), September 25 2002. Special issue on peer-to-peer networking.
21. H. W. Schmidt and R. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proc. of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 213–229. Kluwer, March 2002.
 22. J. C. Seco and L. Caires. A basic model of typed components. In *Proc. of ECOOP*, volume 1850 of *LNCS*, pages 108–128. Springer-Verlag, 2000.
 23. R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.
 24. B. Wydaeghe. *PACOSUITE — Component Composition Based on Composition Patterns and Usage Scenarios*. PhD thesis, Vrije Universiteit Brussel (VUB), November 2001.
 25. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.