Journal of Logical and Algebraic Methods in Programming 87 (2017) 67–91

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A language-based approach to modelling and analysis of Twitter interactions ☆

Alessandro Maggi [a], Marinella Petrocchi [b], Angelo Spognardi [c,*], Francesco Tiezzi [d]

[a] IMT Institute for Advanced Studies, Lucca, Italy
[b] CNR, Istituto di Informatica e Telematica, Pisa, Italy
[c] DTU Compute, Lingby, Denmark
[d] School of Science and Technology, University of Camerino, Italy

ARTICLE INFO

Article history:
Received 1 January 2016
Received in revised form 25 September 2016
Accepted 21 November 2016
Available online 1 December 2016

Keywords:
Twitter interactions and communications
Formal semantics
Verification
Model checking

ABSTRACT

More than a personal microblogging site, Twitter has been transformed by common use to an information publishing venue, which public characters, media channels and common people daily rely on for, e.g., news reporting and consumption, marketing, and social messaging. The use of Twitter in a cooperative and interactive setting calls for the precise awareness of the dynamics regulating message spreading. In this paper, we describe Twitlang, a language for modelling the interactions among Twitter accounts. The associated operational semantics allows users to precisely determine the effects of their actions on Twitter, such as post, reply-to or delete tweets. The language is implemented in the form of a Maude interpreter, Twitlanger, which takes a language term as an input and explores the computations arising from the term. By combining the strength of Twitlanger and the Maude model checker, it is possible to automatically verify communication properties of Twitter accounts. We illustrate the benefits of our executable formalisation by means of an application scenario inspired from real life. While the scenario highlights the benefits of adopting Twitter for a cooperative use in the everyday life, our analysis shows that appropriate settings are essential for a proper usage of the platform, in respect of fulfilling those communication properties expected within collaborative and interactive contexts.

1. Introduction

Online social networks are widespread means to enact interactive collaboration among people by, e.g., planning events, diffusing information, and enabling discussions. With about 500 million of tweets sent per day [2], Twitter[1] provides one of the most illustrative examples of how people can effectively interact without resorting to traditional communication media.

☆ This work is a revised and extended version of [1], presented in the Proceedings of the 13th edition of the International Conference on Software Engineering and Formal Methods (SEFM). The work has been partially sponsored by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, the Italian PRIN 2010LHT4KM CINA, and the Registro.it project MIB (My Information Bubble).

* Corresponding author.
E-mail addresses: alessandro.maggi@imtlucca.it (A. Maggi), marinella.petrocchi@iit.cnr.it (M. Petrocchi), angsp@dtu.dk (A. Spognardi), francesco.tiezzi@unicam.it (F. Tiezzi).

[1] Twitter web site: http://twitter.com.

http://dx.doi.org/10.1016/j.jlamp.2016.11.003
2352-2208/

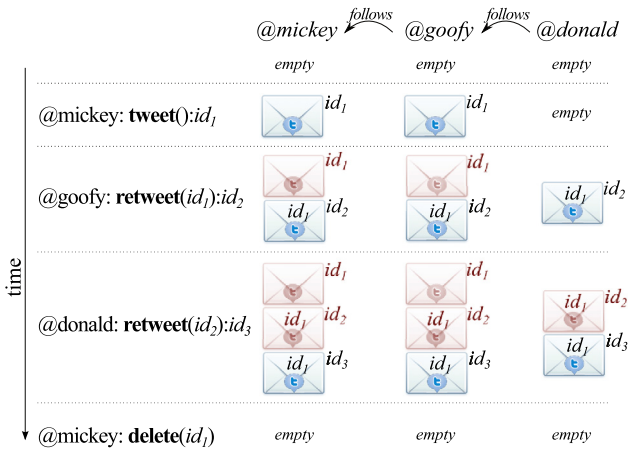

Fig. 1. Effects of an example Twitter interaction on users' accounts (for interpretation of the references to colour in this figure, the reader is referred to the web version of this article).

The platform is used as a marketing tool where people can easily promote retail stores and business services [3], while traditional mass media companies, such as broadcasters and newspapers, currently leverage Twitter as a new media channel. Noticeably, it has also been used for spreading alerts and activity information messages by civil protection departments and the most well-known humanitarian driving forces (see, e.g., [4]).

One of the keys for the success of this socially-centric platform consists in its ease of use. Basically, Twitter users interact by posting *tweets*, i.e., textual messages up to 140 characters. Tweets can also carry pictures, URLs, or *mentions* to other users, the latter triggering *notifications* to the mentioned users. There are three types of possible relationships between Twitter users A and B: either A *follows* B, meaning that the tweets posted by B are directly visible to A (more precisely, they appear on A's Twitter *timeline*), or B follows A (with the complementary meaning), or they follow each other. Of course, there is also the case of no relationship between A and B. Users may also *reply* to any tweet, or even do a *retweet*, in order to spread to their followers what they think particularly worth of notice.

Recently, researchers have focused their attention on several aspects of Twitter, from modelling the number and nature of follow relationships (see, e.g., [5]), to applying sentiment analysis and natural language processing techniques to tweets. These techniques allow, for example, to discover trending topics and their correspondence to real events (see, e.g., [6]) or, by relying on both supervised and unsupervised learning, to detect malicious accounts. In this paper, we focus on what probably represents one of the core aspects of the platform, that makes it so popular and widespread: the Twitter communication model and interaction network. All those who like to use Twitter for socialising, being informed, interact within the community, must precisely know the dynamics of their tweets, say, e.g., which accounts are directly reachable by their tweets, or what happens if a tweet is deleted. A conscious usage of Twitter becomes even more crucial when it is used as a communication media to support (possibly critical) collaborative work.

While the potentialities of the knowledge value provided daily by Twitter to its subscribers are undeniable, the achievement of a full user experience-awareness should not be given for granted. Indeed, the effects of (a sequence of) Twitter interactions may be not clearly known by all the Twitter users. As simple examples, we invite the reader to consider the following three sequences of actions:

1. post a tweet t – reply to t – delete t;
2. post a tweet t – retweet t – undo the retweet;
3. post a tweet t – retweet t – retweet the retweet – delete t.

Without introducing here a formal notation, we give the intuition for such sequences. Sequences 1 and 2 involve two users, say *@mickey* and *@goofy*, while sequence 3 involves also a third user, say *@donald*. In sequence 1, *@mickey* posts a tweet t and *@goofy* replies to that tweet, then *@mickey* deletes t. In sequence 2, *@mickey* posts a tweet t, *@goofy* retweets t, and successively *@goofy* cancels his retweet. In sequence 3, *@mickey* posts a tweet t, *@goofy* retweets t, then *@donald* retweets *@goofy*'s retweet, and finally *@mickey* deletes the original tweet t. The effects of the removal actions in these three sequences of interactions are quite different. In the first case, t is removed from any timeline, while the reply still exists. In the second case, the fact that *@goofy* cancels his retweet does not cause any effect to t, that still exists. Finally, in sequence 3, deleting t leads to the disappearance of the original tweet together with of all its retweets. Fig. 1 gives a pictorial representation of sequence 3, from the point of view of the messages received by the three considered accounts. For the sake of modelling, each tweet/retweet is labeled by a unique identifier id_j. As a matter of notation, we use an envelope to represent a message (in this case a tweet or a retweet). The message identifier is written beside the envelope, in the right-corner; in case of a retweet, inside the envelope it is reported also the identifier of the retweeted tweet. Moreover, we use a blue colour to distinguish the last message received by an account from the others, which are coloured by a light brown colour.

The previous interactions are just some of many possible interactions that users can engage on Twitter. Even these simple examples have effects that could be not fully intuitive for the community. In the following of the paper, we will show examples of interactions leading to more subtle and counterintuitive effects. This motivates the need for designing a rigorous model to trace, and hence analyse, Twitter interaction patterns.

In [1], the authors proposed Twitlang, a specification language describing a network of Twitter accounts and their behaviour. This was the first attempt to formally model the basic interactions resulting from users communicating on Twitter. The Twitlang formal semantics clearly determines the effects of the actions of a Twitter account, with respect to all the other accounts (including subtle and counterintuitive effects). This is determined "a priori", without the need of experimenting interactions on the Twitter platform and their effects case by case. Besides being interesting per se, the Twitlang formal semantics has been implemented in the form of a Maude[2] interpreter, called Twitlanger. It takes a language term, i.e., a specification of a network of Twitter accounts, as an input, and performs an automatic or interactive exploration of the computations arising from the term.

It is worth noting that the language is able to capture the core aspects of Twitter communications, i.e., standard behavioural patterns, like posting a tweet, replying to or retweeting a particular tweet. We focus on such aspects of the Twitter communication network, since we argue that they are already enough to enable the kind of analyses which we intend to carry out. For the sake of clarity, we point out that the model of Twitter, as caught by Twitlang, is not complete. Indeed, syntax and semantics of some more specific features, as direct messages, the blocking of an account, likes, and tags in pictures are not modelled. Another modality that is not covered is the one related to privacy settings, which, when turned on, substantially modify the way tweets are delivered. In our work, in fact, we focus on modelling and analysing communication properties and communication configurations of the Twitter network in a "cooperative" setting. Thus, we have mainly considered a collaborative use of Twitter, where accounts are not protected and where Twitter plays the role of a public repository of messages. However, we acknowledge that, in a more general perspective, considering protected accounts is particularly relevant, especially for modelling and analysing privacy properties of the platform. Therefore, we envisage an extension of Twitlang including, besides some of the features mentioned above, the management of the privacy settings.

This paper proposes an optimised version of Twitlanger, which allows to significantly improve the performance analysis with respect to the version developed in [1]. Most of all, we integrate here the model checking facilities offered by the Maude toolset. This integration enables the automatic verification of the propagation of messages within a network of Twitter accounts, where the latter may be linked together by follow relationships and perform basic operations, such as tweeting, replying, retweeting, deleting tweets. We also introduce a real life-inspired scenario, which represents a collaborative environment where people need to inform and be informed on everyday activities. Quite naturally, Twitter represents a friendly and ready-to-use channel for communication tasks. Our analysis explores whether different configurations of a network of accounts on Twitter are appropriate or not to fulfil the communication goals dictated by the scenario tasks. In the negative cases, the Maude model checker automatically outputs counterexamples, thus helping to find solutions for fixing the communication fails.

From the practical point of view, we envision two classes of potential users for Twitlang/Twitlanger: *(i)* researchers working on formal methods, who can use the formal semantics, and possibly its Maude implementation, for developing other analysis techniques for Twitter interactions; and *(ii)* ICT managers, who intend to use Twitter as a communication media for collaborative work in their organisations and want to ensure that their applications enjoy desired communication properties. In this second case, in particular, our approach can be used to properly design a Twitter sub-network according to the goals and properties to achieve (e.g., in a university, if a professor has to issue a communication regarding an exam, then all students interested in it must receive the message). Then, the realisation of such a sub-network simply consists of possibly creating some new Twitter accounts (e.g., the one of the student office) and requiring the involved Twitter users (professors, administrative staff, etc.) to respect a disciplined use of Twitter (e.g., establishing some follow relationships, retweeting all tweets having a given hashtag, etc.).

Road map. The remainder of this paper is organised as follows. The next section introduces the syntax of Twitlang, focusing on specifying a simple Twitter interaction pattern, while Section 3 presents the semantics of the language. Then, we describe in Section 4 a sequence of Twitter interactions among three parties, which is peculiar for its counterintuitive visible outcome. We show that the semantics of the language is capable to precisely capture that subtle outcome, without the need for setting up empirical experiments. Section 5 describes the basic Maude modules of the Twitlanger interpreter, shows the results of the experimental evaluation of the optimised version of Twitlanger, presents the integration with the Maude model checker, and finally defines a set of properties valuable for the Twitter platform. In Section 6, we introduce a university environment where the staff and the students communicate via their Twitter accounts: we show how our approach is able to verity the fulfilment of communication properties among the actors at stage. Section 7 is devoted to the related work in the area of Twitter modelling and analysis techniques. Finally, in Section 8, we conclude the paper.

Main novelties. The current paper represents a revised and extended version of the work in [1]. In particular, novelties with respect to the predecessor are as follows:

[2] Maude System web site: http://maude.cs.illinois.edu/.

Table 1
Twitlang: syntax.

(*Networks*)	$\mathcal{N}$	::=	$u : T : N : F : B \mid \mathcal{N}_1 \parallel \mathcal{N}_2$
(*Timelines*)	T	::=	$\epsilon \mid m \mid T_1, T_2$
(*Notification lists*)	N	::=	$\epsilon \mid m \mid N_1, N_2$
(*Messages*)	m	::=	$\langle id_{cur}, id_{ret}, id_{rep}, text, u_a, u_l, u_s \rangle$
(*Following lists*)	F	::=	$\epsilon \mid u \mid F_1, F_2$
(*Behaviours*)	B	::=	$\mathbf{nil} \mid a.B \mid B_1 + B_2 \mid B_1 \mid B_2 \mid K$
(*Actions*)	a	::=	$\mathbf{tweet}(text, x) \mid \mathbf{delete}(x) \mid \mathbf{find}(\mathcal{P}, z)@t \mid \mathbf{retweet}(z, y)$ $\mid \mathbf{undo}(y) \mid \mathbf{reply}(z, text, U, x) \mid \mathbf{follow}(u) \mid \mathbf{unfollow}(u)$
(*Targets*)	t	::=	$u \mid \mathbf{all}$

- Section 5 presents an optimised version of the Twitlanger interpreter, shows the results of its experimental evaluation, and introduces the integration of the Maude model checker, with related properties specifically defined for Twitter.
- A more complex, real life inspired application is newly introduced in Section 6 and it is modelled with Twitlang. The case study intends to show the benefits of the adoption of Twitter for everyday life.
- We leverage the Maude model checker to check a set of communication properties on the newly introduced case study (Section 6). The properties have been defined with the aim of analysing the access to the available information in a considered Twitter (sub)network, where the accounts perform basic actions, such as tweet, retweet, reply, delete and undo. Such an investigation is beneficial to assure that the Twitter network (or a portion of that) is configured in such a way to achieve a sound adoption of the platform, when used for making everyday life goals easier (e.g., assuring that tweets are received by the expected recipients).
- The Twitlang syntax and semantics have been slightly revised to more strictly adhere to the real Twitter settings and usage (Sections 2 and 3).
- The related literature has been revised and our contribution has been critically positioned within the existing research efforts (Section 7).
- For the reader's convenience, a full account of the operational semantics is given in Section 3.

2. Twitlang: a formal language for modelling Twitter interactions

In this section, we introduce the syntax of Twitlang, the formalism we propose to model interactions among Twitter accounts.

Twitlang syntax is reported in Table 1. A *network* $\mathcal{N}$ is a composition, by means of parallel operator $\parallel$, of *accounts* of the form $u : T : N : F : B$, where:

- u is a *username* that uniquely identifies the account;
- T is the *timeline*, i.e. the list of messages received from the account's followings or sent by the account;
- N is the list of *notifications* of the account, containing the messages where the account's username is mentioned and the replies to account's messages;
- F is the list of *followings* of the account;
- B is a model of the account's *behaviour*, expressed as a process performing Twitter actions.

Symbol ϵ is used to denote an empty list.

A *message* is a data tuple of the form $\langle id_{cur}, id_{ret}, id_{rep}, text, u_a, u_l, u_s \rangle$, where:

- id_{cur} is the identifier of the (current) message;
- id_{ret} is the identifier of the original tweet the current message is a retweet of;
- id_{rep} is the identifier of the message the current message is a reply to;
- *text* is the textual content of the message;
- u_a is the username of the author of the (retweeted or replied) original message;
- u_l is the username of the sender of the last retweet in a retweet chain;
- u_s is the username of the sender of the current message.

We will use the *null* symbol _ to leave unspecified a field of a message, as, for example, in the case of a new tweet, where the fields id_{ret}, id_{rep}, u_a and u_l are irrelevant. Moreover, we will exploit a *projection* function $m \downarrow_i$ that returns the i-th field of the message m. It is worth noticing that the identifiers used in a message act as *links* to other messages.

Thus, given a message $\langle id_1, id_2, id_3, t, u_1, u_2, u_3 \rangle$, the identifier id_1 is a link to access all messages produced as replies to this message (i.e., the set of messages $\{m \mid m \downarrow_3 = id_1\}$), while the identifier id_3 can be used to access the previous message in the conversation (i.e., the message m such that $m \downarrow_1 = id_3$). Other messages can be iteratively retrieved from the already accessed ones. The navigation among messages via links can be done in the Twitter platform by means of the functionalities *expand* and *view conversation*. As an example, let us consider the case of a reply to a reply of a tweet; the message m corresponding to the reply of the tweet permits accessing both the tweet message (by means of the id in the third field of m) and the second reply message (by means of the id in the first field of m).

Account *behaviours* are modelled by means of terms of a simple process algebra (actually, this is a simple variant of the well-known process algebra CCS [7], with specialised actions). Each process is built up from the *inert* process **nil** via *action prefixing* ($a.B$), *nondeterministic choice* ($B_1 + B_2$), *parallel composition* ($B_1 \mid B_2$), and *process invocation* (K). We assume that K ranges over a set of *process constants* that are used in (recursive) process definitions. We assume also that each constant K has a single definition of the form $K \triangleq B$.

Processes can perform eight different kinds of *actions*. We use the following pairwise disjoint sets of variables: the set of *tweet variables* (ranged over by x), the set of *retweet variables* (ranged over by y), and the set of *message variables* (ranged over by z). Variables in Twitlang are sort of write-once variables that, when instantiated with a value, disappear from the term and cannot be reassigned. Moreover, we use U to denote a set of usernames. We define three action prefixes $\mathbf{tweet}(text, x).B$, $\mathbf{retweet}(z, y).B$ and $\mathbf{reply}(z, text, U, x).B$ used to send messages to other accounts; they bind variables x and y in B. The receivers of such messages are determined according to follower-following relationships and presence of mentions in the content of messages, as formally described by the language semantics (Section 3). In particular, action $\mathbf{tweet}(text, x)$ produces a new tweet with content *text*, whose fresh message identifier is bound to the tweet variable x. Action $\mathbf{retweet}(z, y)$ permits retweeting a message identified by z; the fresh identifier of the retweet message is bound to the retweet variable y. Action $\mathbf{reply}(z, text, U, x)$ produces a message in response to the message identified by z; the produced message has content *text*, inherits all mentions from the replied message but for those specified in the set U of usernames,[3] and its identifier is bound to variable x. Tweet and reply messages can be removed by means of action $\mathbf{delete}(x)$, while retweet messages by means of action $\mathbf{undo}(y)$. Actions $\mathbf{retweet}(z, y)$ and $\mathbf{reply}(z, text, U, x)$ act on a message that, at runtime, will replace the message variable z. This message is retrieved from the Twitter network by means of the (blocking) action $\mathbf{find}(\mathcal{P}, z)@t.B$, which indeed binds variable z in B. The action relies on a *predicate* $\mathcal{P}$ for selecting a message among those stored in a given account u (target $t = u$) or among all messages in the network (target $t = \mathbf{all}$). Predicates are boolean-valued expressions obtained by logically combining the evaluation of (comparison) relations between message fields and values. Intuitively, the **find** action represents all activities performed by a human user leading to the reading of a message, such as looking at her own timeline, looking at the profile pages[4] of other users, reading messages embedded in websites (e.g., online newspapers), or reading messages retrieved via the *search* functionality of Twitter. An account can add/remove a username u to/from its following list F by means of actions $\mathbf{follow}(u)$ and $\mathbf{unfollow}(u)$, respectively.

We conclude the presentation of the syntax by showing how the examples shown in Fig. 1 are rendered in our formalism.

Example 1 *(Tweet–retweet–retweet–delete).* Let us consider a network of three accounts with usernames u_m (*@mickey*), u_g (*@goofy*) and u_d (*@donald*), with empty timelines and notifications lists and such that u_g follows u_m and u_d follows u_g:

$$u_m : \epsilon : \epsilon : \epsilon : B_m \parallel u_g : \epsilon : \epsilon : u_m : B_g \parallel u_d : \epsilon : \epsilon : u_g : B_d$$

Account u_m posts a tweet, waits for a local message indicating that u_d has retweeted it, and then deletes it. Account u_g (resp. u_d) reads a local message from u_m (resp. u_g) and retweets it. This is rendered by the following behaviours:

$$B_m = \mathbf{tweet}(Hello, x).\mathbf{find}(\downarrow_7 = u_d, z)@u_m.\mathbf{delete}(x).\mathbf{nil}$$

$$B_g = \mathbf{find}(\downarrow_7 = u_m, z')@u_g.\mathbf{retweet}(z', y).\mathbf{nil}$$

$$B_d = \mathbf{find}(\downarrow_7 = u_g, z'')@u_d.\mathbf{retweet}(z'', y').\mathbf{nil}$$

Predicate $\downarrow_7 = u$ is verified by a message m if its sender (i.e., $m \downarrow_7$) is the username u.

3. Twitlang operational semantics

The operational semantics of Twitlang is given in the SOS style [8] in terms of a structural congruence and of a labeled transition relation. Notably, the semantics is only defined for *closed* terms, i.e. terms without free variables. Indeed, we consider the binding of a variable as its declaration (and initialisation), therefore free occurrences of variables at the outset in a term must be prevented since they are errors similar to uses of variables before their declaration in programs. Notice also that the semantics is defined over an *enriched syntax* that also includes those auxiliary terms resulting from replacing

[3] For the sake of simplicity, the set U is statically defined. This is adequate for the purpose of our study; a more dynamic definition of the set could be considered in further developments.

[4] The *profile page* of a Twitter account contains all messages (i.e., tweets, retweets and replies) produced by the account. This is the part of the account's timeline visible to other users.

Table 2
Twitlang: structural congruence.

$B + \mathbf{nil} \equiv B$	$B_1 + B_2 \equiv B_2 + B_1$	$(B_1 + B_2) + B_3 \equiv B_1 + (B_2 + B_3)$
$B \mid \mathbf{nil} \equiv B$	$B_1 \mid B_2 \equiv B_2 \mid B_1$	$(B_1 \mid B_2) \mid B_3 \equiv B_1 \mid (B_2 \mid B_3)$
$\dfrac{K \triangleq B}{K \equiv B}$	$\dfrac{B \equiv_{alpha} B'}{B \equiv B'}$	$\dfrac{B \equiv B'}{u : T : N : F : B \equiv u : T : N : F : B'}$
$\mathcal{N}_1 \parallel \mathcal{N}_2 \equiv \mathcal{N}_2 \parallel \mathcal{N}_1$	$(\mathcal{N}_1 \parallel \mathcal{N}_2) \parallel \mathcal{N}_3 \equiv \mathcal{N}_1 \parallel (\mathcal{N}_2 \parallel \mathcal{N}_3)$	

Table 3
Twitlang: labeled transition relation (behaviours).

$\mathbf{tweet}(text, x).B \xrightarrow{\mathbf{tweet}(text,id)} B[id/x]$	[B-TWEET]
$\mathbf{delete}(id).B \xrightarrow{\mathbf{delete}(id)} B$	[B-DELETE]
$\mathbf{find}(\mathcal{P}, z)@t.B \xrightarrow{\mathbf{find}(\mathcal{P},z)@t} B$	[B-FIND]
$\mathbf{retweet}(m, y).B \xrightarrow{\mathbf{retweet}(m,id)} B[id/y]$	[B-RETWEET]
$\mathbf{undo}(id).B \xrightarrow{\mathbf{undo}(id)} B$	[B-UNDO]
$\mathbf{reply}(m, text, U, x).B \xrightarrow{\mathbf{reply}(m,(m\downarrow_7 \cdot m\downarrow_5 \cdot mentions(m\downarrow_4))\setminus U \cdot text,id)} B[id/x]$	[B-REPLY]
$\mathbf{follow}(u).B \xrightarrow{\mathbf{follow}(u)} B$	[B-FOLLOW]
$\mathbf{unfollow}(u).B \xrightarrow{\mathbf{unfollow}(u)} B$	[B-UNFOLLOW]
$\dfrac{B_1 \xrightarrow{\alpha} B_1'}{B_1 + B_2 \xrightarrow{\alpha} B_1'}$ [B-CHOICE]	$\dfrac{B_1 \xrightarrow{\alpha} B_1'}{B_1 \mid B_2 \xrightarrow{\alpha} B_1' \mid B_2}$ [B-PAR]

(free occurrences of) variables with the corresponding identifier. Finally, it is worth noticing that not all processes allowed by the syntax in Table 1 are meaningful. Indeed, in a general term of the language, the messages stored in the accounts may not be consistent; for example, we could have a message representing a retweet whose reference to the original tweet does not correspond to any tweet message in the network. Thus, to ensure consistent terms, we only consider *reachable* networks (whose formal characterisation is provided later in Definition 1), which are networks obtained by means of reductions from networks with no stored messages.

3.1. Structural congruence

The *structural congruence*, written $\equiv$, is defined as the smallest congruence relation on networks that includes the laws shown in Table 2. Almost all laws are standard laws of process algebras. The first six laws are the (abelian) monoid laws for $+$ and $\mid$ (i.e., they are associative and commutative, and have **nil** as identity element). The seventh law permits to replace a process invocation with the corresponding process behaviour (in case of recursive definitions, this allows recursion unfolding). The eighth law equates alpha-equivalent behaviours, i.e. behaviours only differing in the identity of bound variables (alpha-equivalence is denoted by $\equiv_{alpha}$); as an example, using this law the term $\mathbf{tweet}(text, x_1).\mathbf{delete}(x_1).\mathbf{nil}$ can be safely converted in $\mathbf{tweet}(text, x_2).\mathbf{delete}(x_2).\mathbf{nil}$, as variable x_1 and x_2 are just placeholders for message identifiers. The ninth law permits lifting the structural congruence from behaviours to nets. The last two laws state that $\parallel$ is commutative and associative.

3.2. Behaviour semantics

To define the labeled transition relation, we rely on an auxiliary relation on behaviours, which is defined as the smallest relation on behaviours generated by the rules in Table 3. We write $B \xrightarrow{\alpha} B'$ to mean that "B can perform a transition labeled α and become B' in doing so". Transition labels are generated by the following production rule

$$\alpha ::= \textbf{tweet}(text, id) \mid \textbf{delete}(id) \mid \textbf{find}(\mathcal{P}, z)@t \mid \textbf{retweet}(m, id) \mid \textbf{undo}(id) \mid \textbf{reply}(m, text, id) \mid \textbf{follow}(u) \mid \textbf{unfollow}(u)$$

Basically, each action gives rise to the corresponding label. When a **tweet**, **retweet** or **reply** is executed, a fresh message *id* is generated and used to replace the corresponding variable x or y via a *substitution*, i.e. a function $[v/k]$ mapping variable k to value v. Application of a substitution to a behaviour, written $B[v/k]$, has the effect of replacing every free occurrence of k in B with v. As clarified later, the freshness of identifiers is ensured by operational rules at the network level.

The message text produced by a **reply** action extends the reply message with the mentions inherited from the replied message m, except for those indicated in the set U. To this aim, we exploit a *mention retrieval* function $mentions(text)$ and the *removal* operator $text\backslash U$: the former returns the set of usernames mentioned in *text*, while the latter removes from *text* all mentions to accounts belonging to the set U. Thus, in the label generated by the **reply** action, the text of the message consists of a mention to the sender of message m, a mention to the author of the original tweet, all mentions included in the text of m except those in U and, of course, the text of the reply. They are composed by means of the *concatenation* operator $\cdot$. Notably, new mentions can be added by means of the text of the reply.

Execution of an action permits to take a decision between alternative behaviours (rule [B-CHOICE]), while execution of parallel actions is interleaved (rule [B-PAR]). Notably, symmetric versions of such rules are not necessary, as structural congruence (applied via rule [N-STR], introduced later) ensures associativity and commutativity properties of operators $+$ and $|$.

3.3. Network semantics

At network level, the *labeled transition relation* is the smallest relation on closed reachable networks generated by the rules in Tables 4 and 5. We write $\mathcal{N} \xrightarrow{\lambda} \mathcal{N}'$ to mean that "$\mathcal{N}$ can perform a transition labeled λ and become $\mathcal{N}'$ in doing so". Transition labels are generated by the following production rule

$$\lambda ::= m \mid \textbf{delete}(id) \mid \textbf{undo}(id) \mid u : \textbf{found}(m) \mid u : \textbf{added}(u') \mid u : \textbf{removed}(u')$$

meaning that a message m has been transmitted, the tweet/reply identified by *id* and its related messages have been deleted, the retweet identified by *id* has been deleted, a message m is retrieved by u, the account u' has been added to the following list of u, and the account u' has been removed from the following list of u, respectively.

Rule [N-TWEET] transforms a **tweet** label into a network label m representing the message generated by the action. The message is inserted in the timeline of the account. Notably, premise $id \notin \text{ids}(T, N, B)$ checks that the message *id* is fresh in the considered account (in fact, function $\text{ids}(\cdot)$ returns all identifiers used in the terms passed as arguments).

Rule [N-RETWEET] is similar; the extra premise $m \downarrow_7 \neq u$ permits blocking a retweet of a message generated by the same account u (indeed, this is not allowed in Twitter). Notice that this time the second field of the produced message records the identifier of the original tweet. If m is a retweet, this information is retrieved from the second field of m, while in case of tweet or reply it is retrieved from the first field. This is achieved by resorting to function $origId(m)$ that returns $m \downarrow_2$ if $m \downarrow_2 \neq \_$, otherwise $m \downarrow_1$. Similarly, the fifth field is determined by means of function $author(m)$ that returns $m \downarrow_5$ if $m \downarrow_2 \neq \_$ (i.e., m is a retweet), otherwise (i.e., m is a tweet or a reply) it returns $m \downarrow_7$. Moreover, the text of the retweet is the same of that of the retweeted message (in Twitter, indeed, the **retweet** action does not allow to modify the text of the retweeted message).

Rule [N-REPLY] is similar; the rule properly records the identifier and author of the replied message m in the third and fifth fields of the generated message, respectively.

Rule [N-DELIVER] takes care of delivering a new message to all the accounts of the network that have to receive it. In particular, this rule should be repeatedly applied in order to consider one by one all the accounts. For each account is checked if the identifier of the message is fresh. In this way, at the end of the inference of the transition, the global freshness of the identifier is ensured. Notably, this does not require to use a restriction operator à la π-calculus [9], because the scope of the identifiers is always global, i.e. each user potentially can access every tweet in the network (recall that in Twitter it is possible to access the messages sent by any user by visiting her Twitter profile page). The possible insertion of the message in the timeline and notification list of the considered account is regulated by the following *insertion operators*:

- *tweet insertion*: a message m is inserted in the timeline T of an account only if the sender of m is in the following list F of this account

$$T \oplus^F m = \begin{cases} (T, m) & \text{if } m \downarrow_7 \in F \\ T & \text{otherwise} \end{cases}$$

- *notification insertion*: a message m is inserted in the notification list N of an account with username u only if u is mentioned in the text of m, or m is a retweet whose original tweet message has been sent by u, or m is a reply to a message sent by u

$$N \oplus^u m = \begin{cases} (N, m) & \text{if } u \in mentions(m \downarrow_4) \vee m \downarrow_5 = u \vee m \downarrow_6 = u \\ N & \text{otherwise} \end{cases}$$

Table 4
Twitlang: labeled transition relation (networks).

$$\frac{B \xrightarrow{\mathbf{tweet}(text,id)} B' \qquad id \notin \mathrm{ids}(T,N,B)}{u:T:N:F:B \xmapsto{\langle id,\_,\_,text,\_,\_,u\rangle} u:(T,\langle id,\_,\_,text,\_,\_,u\rangle):N:F:B'} \text{[N-TWEET]}$$

$$\frac{B \xrightarrow{\mathbf{retweet}(m,id)} B' \qquad id \notin \mathrm{ids}(T,N,B) \qquad m\downarrow_7 \neq u}{u:T:N:F:B \xmapsto{\langle id,origId(m),\_,m\downarrow_4,author(m),m\downarrow_7,u\rangle} u:(T,\langle id,origId(m),\_,m\downarrow_4,author(m),m\downarrow_7,u\rangle):N:F:B'} \text{[N-RETWEET]}$$

$$\frac{B \xrightarrow{\mathbf{reply}(m,text,id)} B' \qquad id \notin \mathrm{ids}(T,N,B)}{u:T:N:F:B \xmapsto{\langle id,\_,m\downarrow_1,text,m\downarrow_7,\_,u\rangle} u:(T,\langle id,\_,m\downarrow_1,text,m\downarrow_7,\_,u\rangle):N:F:B'} \text{[N-REPLY]}$$

$$\frac{\mathcal{N} \xmapsto{m} \mathcal{N}' \qquad m\downarrow_1 \notin \mathrm{ids}(T,N,B)}{\mathcal{N} \parallel u:T:N:F:B \xmapsto{m} \mathcal{N}' \parallel u:(T \oplus^F m):(N \oplus^u m):F:B} \text{[N-DELIVER]}$$

$$\frac{B \xrightarrow{\mathbf{delete}(id)} B'}{u:T:N:F:B \xrightarrow{\mathbf{delete}(id)} u:(T \ominus id):(N \ominus id):F:B' \not\downarrow_{id}} \text{[N-DELETE]}$$

$$\frac{\mathcal{N} \xmapsto{\mathbf{delete}(id)} \mathcal{N}'}{\mathcal{N} \parallel u:T:N:F:B \xmapsto{\mathbf{delete}(id)} \mathcal{N}' \parallel u:(T \ominus id):(N \ominus id):F:B \not\downarrow_{id}} \text{[N-DELPROPAG]}$$

$$\frac{B \xrightarrow{\mathbf{undo}(id)} B'}{u:T:N:F:B \xrightarrow{\mathbf{undo}(id)} u:(T \oplus id):(N \oplus id):F:B' \not\downarrow_{id}} \text{[N-UNDO]}$$

$$\frac{\mathcal{N} \xmapsto{\mathbf{undo}(id)} \mathcal{N}'}{\mathcal{N} \parallel u:T:N:F:B \xmapsto{\mathbf{undo}(id)} \mathcal{N}' \parallel u:(T \oplus id):(N \oplus id):F:B \not\downarrow_{id}} \text{[N-UNDOPROPAG]}$$

$$\frac{B \xrightarrow{\mathbf{find}(\mathcal{P},z)@u} B' \qquad \exists m \in (T \cup N) \;:\; \mathcal{P}(m) = true}{u:T:N:F:B \xmapsto{u:\mathbf{found}(m)} u:T:N:F:B'[m/z]} \text{[N-FIND-U]}$$

$$\frac{B \xrightarrow{\mathbf{find}(\mathcal{P},z)@t} B'' \qquad (t = u' \vee t = \mathbf{all}) \qquad \exists m \in T' \;:\; \mathcal{P}(m) = true \wedge m\downarrow_7 = u'}{u:T:N:F:B \parallel u':T':N':F':B' \xmapsto{u:\mathbf{found}(m)} u:T:N:F:B''[m/z] \parallel u':T':N':F':B'} \text{[N-FIND-T]}$$

To clarify the message delivery mechanism of Twitlang, we show in Fig. 2 a simple example of a transition inference, where a user (u_1) tweets a message that is delivered in the timeline of a follower (u_2) and in the notification list of a mentioned user (u_3). The effect of this transition is graphically shown in Fig. 3. Notably, the rules for the propagation of the effects of actions **delete** and **undo** work similarly.

Rule [N-DELETE] deletes the tweet identified by *id* and all its retweets from the account that performed the **delete** action (which is the account that emitted such a tweet). The deletion is then propagated to the other accounts by rule [N-DELPROPAG]. The deletion of a message from a list L (which denotes either a timeline or a notification list) is defined by the following operator:

- *tweet deletion*: a message m is deleted from the list L of an account only if *id* is its identifier or m is a retweet of a message identified by *id*

$$L \ominus id \;=\; L \backslash \{m \in L \mid m\downarrow_1 = id \;\vee\; m\downarrow_2 = id\}$$

Moreover, retweeting and replying of deleted messages (which may happen when a **delete** is executed after a **find**) are prevented by means of the *block* operator $B \not\downarrow_{id}$, which replaces prefixes $a.B'$ in B by **nil** when a is a **retweet** or a **reply**

Table 5
Twitlang: labeled transition relation (networks), cnt.

$$\frac{B \xrightarrow{\textbf{follow}(u')} B'' \qquad u' \notin F}{u : T : N : F : B \parallel u' : T' : N' : F' : B' \xrightarrow{u:\textbf{added}(u')} u : (T, \{m \in T' \mid m \downarrow_7 = u'\}) : N : (F, u') : B'' \parallel u' : T' : N' : F' : B'} \text{[N-FOLLOW1]}$$

$$\frac{B \xrightarrow{\textbf{follow}(u')} B'' \qquad u' \in F}{u : T : N : F : B \parallel u' : T' : N' : F' : B' \xrightarrow{u:\textbf{added}(u')} u : T : N : F : B \parallel u' : T' : N' : F' : B'} \text{[N-FOLLOW2]}$$

$$\frac{B \xrightarrow{\textbf{unfollow}(u')} B''}{u : T : N : F : B \parallel u' : T' : N' : F' : B' \xrightarrow{u:\textbf{removed}(u')} u : (T \backslash \{m \in T \mid m \downarrow_7 = u'\}) : N : (F \backslash u') : B'' \parallel u' : T' : N' : F' : B'} \text{[N-UNFOLLOW]}$$

$$\frac{\mathcal{N}_1 \xrightarrow{\lambda} \mathcal{N}_1' \qquad \lambda \in \{u : \textbf{found}(m), u : \textbf{added}(u'), u : \textbf{removed}(u')\}}{\mathcal{N}_1 \parallel \mathcal{N}_2 \xrightarrow{\lambda} \mathcal{N}_1' \parallel \mathcal{N}_2} \text{[N-PAR]}$$

$$\frac{\mathcal{N} \equiv \mathcal{N}_1 \xrightarrow{\lambda} \mathcal{N}_2 \equiv \mathcal{N}'}{\mathcal{N} \xrightarrow{\lambda} \mathcal{N}'} \text{[N-STR]}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\textbf{tweet}(\text{“}foo\ u_3\text{”}, x).\textbf{nil} \xrightarrow{\textbf{tweet}(\text{“}foo\ u_3\text{”}, id_1)} \textbf{nil}}\ [\text{B-TWEET}]}{u_1 : \epsilon : \epsilon : \epsilon : \textbf{tweet}(\text{“}foo\ u_3\text{”}, x).\textbf{nil} \xrightarrow{m} u_1 : m : \epsilon : \epsilon : \textbf{nil}}\ [\text{N-TWEET}]}{u_1 : \epsilon : \epsilon : \epsilon : \textbf{tweet}(\text{“}foo\ u_3\text{”}, x).\textbf{nil} \parallel u_2 : \epsilon : \epsilon : u_1 : \textbf{nil} \xrightarrow{m} u_1 : m : \epsilon : \epsilon : \textbf{nil} \parallel u_2 : m : \epsilon : u_1 : \textbf{nil}}\ [\text{N-DELIVER}]}{u_1 : \epsilon : \epsilon : \epsilon : \textbf{tweet}(\text{“}foo\ u_3\text{”}, x).\textbf{nil} \parallel u_2 : \epsilon : \epsilon : u_1 : \textbf{nil} \parallel u_3 : \epsilon : \epsilon : \epsilon : \textbf{nil} \xrightarrow{m} u_1 : m : \epsilon : \epsilon : \textbf{nil} \parallel u_2 : m : \epsilon : u_1 : \textbf{nil} \parallel u_3 : \epsilon : m : \epsilon : \textbf{nil}}\ [\text{N-DELIVER}]$$

Fig. 2. Example of a transition inference (where m stands for $\langle id_1, \_, \_, \text{“}foo\ u_3\text{”}, \_, \_, u_1 \rangle$, and conditions on id_1 in rules [N-DELIVER] and [N-TWEET] are omitted because they are straightforwardly satisfied).

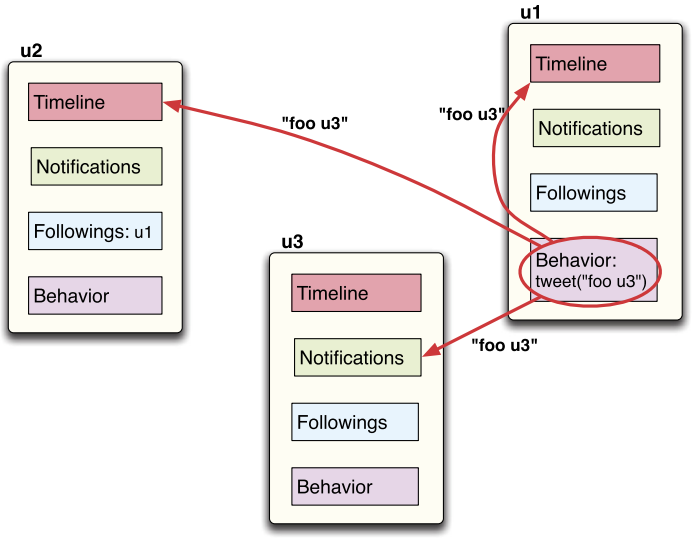

Fig. 3. Graphical representation of the delivery of a tweet.

action having a message m as parameter with $m\downarrow_1$, $m\downarrow_2$ or $m\downarrow_3$ sets to id.[5] To clarify the use of the block operator, let us consider the following example:

$$u_1 : \epsilon : \epsilon : \epsilon : \mathbf{tweet}(\text{“}foo\text{”}, x).\mathbf{delete}(x).\mathbf{nil}$$
$$\parallel\ u_2 : \epsilon : \epsilon : u_1 : \mathbf{find}(\downarrow_7 = u_1, z)@u_2.\mathbf{retweet}(z, y).\mathbf{nil}$$

The execution of the **tweet** action by u_1 and of the **find** action by u_2 leads to the following network

$$u_1 : m : \epsilon : \epsilon : \mathbf{delete}(id).\mathbf{nil}$$
$$\parallel\ u_2 : m : \epsilon : \epsilon : u_1 : \mathbf{retweet}(m, y).\mathbf{nil}$$

with $m = \langle id, \_, \_, \text{“}foo\text{”}, \_, \_, u_1\rangle$. Now, if user u_1 performs the **delete** action, then user u_2 cannot perform the corresponding **retweet** action, as the original tweet has been removed. To achieve this, when m is deleted, $\mathbf{retweet}(m, y).\mathbf{nil}$ is replaced by **nil** by means of the block operator.

Retweets are undone by means of rules [N-UNDO] and [N-UNDOPROPAG], that are similar to rules for the **delete** action except for the deletion operator:

- *retweet deletion*: a message m is deleted from the list L of an account only if m has id as identifier of the current message

$$L \oplus id \ = \ L \backslash \{m \in L \mid m\downarrow_1 = id\}$$

In this case, it is considered only the first field of the message. Thus, only the retweets identified by id are removed, while other retweets of the same tweet and the tweet itself are not affected by the deletion.

Rule [N-FIND-U] allows account u to look for a message satisfying predicate $\mathcal{P}$ in its timeline and notification list. If a message is found, say m, the label produced at network level is $u : \mathbf{found}(m)$. Rule [N-FIND-T] is similar, but it looks for a message in the profile page of another account u', which either is specifically indicated in the target ($t = u'$) or is anyone of the rest of the net ($t = \mathbf{all}$). Once a message is found, rule [N-PAR] permits terminating the search without affecting the other accounts of the network.

Rule [N-FOLLOW1] extends the followings list of account u with username u' when the former is not a follower of the latter; consequently, extends the timeline T with messages (i.e. tweets and retweets) sent by u'. Rule [N-FOLLOW2] is used to let the **follow** action pass without affecting the timeline and the following list of u when this account is already a follower of u'. Rule [N-UNFOLLOW] performs the inverse operations, i.e. it removes u' from F and the messages sent by u' from T. Rule [N-PAR] is used when an action **find**, **follow** or **unfollow** has taken place in a part of a network, in order to allow the whole network to evolve accordingly. This rule cannot be used instead when other kinds of action are executed, i.e., when a message is sent or deleted/undone. Indeed, as explained before, in such cases specific rules for the propagation of action effects must be used. Notably, for the sake of simplicity, if the argument u' of actions **follow** and **unfollow** does not correspond to an account of the network, or it is the same account performing such actions, rules [N-FOLLOW] and [N-UNFOLLOW] cannot be applied and, hence, the actions are blocked. In fact, this kind of situations cannot take place in Twitter, where only existing accounts can be object of actions **follow** and **unfollow**.

Finally, rule [N-STR] states that structural congruent nets have the same transitions.

We can now formally define the class of reachable networks, and conclude with an example.

Definition 1 *(Reachable networks).* The set of reachable networks is the closure under $\overset{\lambda}{\longmapsto}$ of the set of terms generated by the following grammar:

$$\mathcal{N} \ ::= \ u : \epsilon : \epsilon : F : B \quad \mid \quad \mathcal{N}_1 \parallel \mathcal{N}_2$$

Example 2 *(Tweet–retweet–retweet–delete).* Let $\mathcal{N}$ be the network defined in the Example 1. The behaviour B_m of the account u_m can evolve as follows:

$$B_m \xrightarrow{\mathbf{tweet}(Hello, id_1)} B'_m$$

Now, by applying rule [N-TWEET], the message $m_1 = \langle id_1, \_, \_, Hello, \_, \_, u_m\rangle$ is produced. Then, by applying rule [N-DELIVER], m_1 is delivered to u_g (since u_g is a follower of u_m). Thus, the resulting transition is:

$$\mathcal{N} \xrightarrow{\langle id_1, \_, \_, Hello, \_, \_, u_m\rangle} \mathcal{N}' = u_m : m_1 : \epsilon : \epsilon : B'_m \parallel u_g : m_1 : \epsilon : u_m : B_g \parallel u_d : \epsilon : \epsilon : u_g : B_d$$

Similarly, u_g and u_d perform their actions as follows:

$$\mathcal{N}' \xrightarrow{u_g:\mathbf{found}(m_1)} \xrightarrow{m_2} \xrightarrow{u_d:\mathbf{found}(m_2)} \xrightarrow{m_3} \mathcal{N}'' = u_m : m_1 : (m_2, m_3) : \epsilon : B'_m \parallel u_g : (m_1, m_2) : m_3 : u_m : \mathbf{nil} \parallel u_d : (m_2, m_3) : \epsilon : u_g : \mathbf{nil}$$

[5] The definition of more sophisticated solutions, e.g. based on exception handling, to deal with actions involving deleted messages is left for future investigation. Indeed, the blocking solution used here properly suits the study carried out in this paper.

where messages m_2 and m_3 are $\langle id_2, id_1, \_, Hello, u_m, u_m, u_g\rangle$ and $\langle id_3, id_1, \_, Hello, u_m, u_g, u_d\rangle$, respectively. Finally, u_m performs the find and delete actions:

$$\mathcal{N}'' \xrightarrow{u_m:\mathbf{found}(m_3)} \xrightarrow{\mathbf{delete}(id)} \mathcal{N}''' = u_m : \epsilon : \epsilon : \epsilon : \mathbf{nil} \parallel u_g : \epsilon : \epsilon : u_m : \mathbf{nil} \parallel u_d : \epsilon : \epsilon : u_g : \mathbf{nil}$$

As in Fig. 1, the action produces a domino-effect that removes all messages from the timelines and notification lists.

4. An example interaction with counterintuitive effects

Twitter provides users with a basic set of simple features to communicate each other over the platform. Despite the apparent simplicity of such features, the combination of some communication actions can lead to counterintuitive effects.

We consider three Twitter accounts, say *@mickey*, *@donald*, and *@goofy*. We suppose that the three accounts belong to three distinct researchers, *Mickey Mouse*, *Donald Duck*, and *Goofy*, respectively. *Mickey* and *Donald* are colleagues and follow each others on Twitter, while *Goofy* is neither a follower nor a following of both. This scenario is rendered in our formalism as the following network (for the sake of presentation, we consider empty the timelines and notifications lists of the accounts at the beginning of the interaction):

$$u_m : \epsilon : \epsilon : u_d : B_m \;\parallel\; u_d : \epsilon : \epsilon : u_m : B_d \;\parallel\; u_g : \epsilon : \epsilon : \epsilon : B_g$$

Mickey is attending a conference on Social Informatics and listens with interest to *Goofy*'s talk on his recent results on using formal methods for the specification of the Twitter interaction patterns. Since *Mickey* and *Donald* are performing research on very related topics, *Mickey* sends an enthusiastic tweet mentioning both *Donald* and *Goofy*, with the following text: "*@donald great work by @goofy on # formalmethods and Twitter! Let's start a collaboration!*". Thus, the behaviour of the *Mickey*'s account is:

$$B_m = \mathbf{tweet}(\text{“}u_d \textit{ great work by } u_g \textit{ on\#formalmethods and Twitter!} \ldots\text{”}, x).\, B'_m$$

Such a tweet, called hereafter the original tweet and denoted by m_1, appears on (1) *Donald*'s user timeline, since *Donald* follows *Mickey*, and on *Donald*'s notifications list, since *Donald* has been mentioned; (2) *Goofy*'s notifications list, since *Goofy* has been mentioned, but *Goofy* does not follow *Mickey*; and (3) *Mickey*'s user timeline:

$$u_m : m_1 : \epsilon : u_d : B'_m \parallel u_d : m_1 : m_1 : u_m : B_d \parallel u_g : \epsilon : m_1 : \epsilon : B_g$$

It happens that *Donald* has listened some rumours on *Goofy*'s professional reputation. Quite recklessly, he replies to the original tweet, although removing the mention to him: in that reply, called hereafter the replying tweet and denoted by m_2, *Donald* writes the following "*@mickey don't go for it, waste of time*". Note that mention to *@mickey* is automatically inserted in the replying tweet, being it a reply to the original tweet sent by *Mickey*. By default, the reply contains all the mentions included in the original tweet, thus, in this case, it automatically contains *@goofy*. However, *Donald* manually removes the mention to *@goofy* from the reply, before sending it. Thus, the behaviour of the *Donald*'s account is:

$$\begin{aligned} B_d = {} & \mathbf{find}(\downarrow_7 = u_m \wedge \#\textit{formalmethods} \in \textit{hashtags}(\downarrow_4), z)@u_d. \\ & \mathbf{reply}(z, \text{“}u_m \textit{ don't go for it, waste of time}\text{”}, \{u_g\}, x').\, B'_d \end{aligned}$$

Notably, the reply is triggered by the presence in the *@donald* account of a message whose sender is *@mickey* and whose text contains the hashtag *#formalmethods* (in fact, function $\textit{hashtags}(\cdot)$ returns all hashtags in the text passed as argument).

Donald's reply (1) appears on *Mickey*'s user timeline, since *Mickey* follows *Donald*, and on *Mickey*'s notifications list, since *Mickey* has been mentioned; (2) appears on *Donald*'s user timeline; and (3) quite surprisingly, is added to a conversation on *Goofy*'s notifications list, even if the mention to *Goofy* has been explicitly removed. In particular, the reply is tied to the original tweet, and it is visible on *Goofy*'s notifications list upon clicking on the "expand" button. Fig. 4 shows the screenshot of *Goofy*'s notifications list, upon clicking on the "expand" button. Formally, we have:

$$u_m : (m_1, m_2) : m_2 : u_d : B'_m \parallel u_d : (m_1, m_2) : m_1 : u_m : B'_d \parallel u_g : \epsilon : m_1 : \epsilon : B_g$$

where m_1 at u_g now allows *Goofy* accessing the message m_2. In fact, as explained in the section devoted to the presentation of our formalism, the identifiers in a message can be thought of as *links* to retrieve other messages. In our example, the identifier of m_1 (i.e., its first field) can be used to retrieve m_2, because $m_2 \downarrow_3$ is set to the m_1's identifier (since m_2 is a reply to m_1).

Finally, having seen the message of *Donald*, *Mickey* decides to remove his tweet, which is expressed in our formalism as an action $\mathbf{delete}(x)$. This removes all occurrences of m_1, leaving untouched those of m_2:

$$u_m : m_2 : m_2 : u_d : B''_m \parallel u_d : m_2 : \epsilon : u_m : B'_d \parallel u_g : \epsilon : \epsilon : \epsilon : B_g$$

However, we can notice that, even if the reply message is still around, *Goofy* now has no direct link to it.

This example also shows that our formalisation mainly focuses on studying the ways to access the available information in the considered Twitter network. Indeed, despite the fact that in Twitlang all messages in the network are potentially accessible by visiting user profile pages, usually only a restricted set of messages are relevant for a given user. Indeed, we may restrict our study (and we typically do that) to messages that are 'directly reachable' to a user, as they appear in her timeline or notification list, or are 'indirectly reachable' via the *expand* facility.

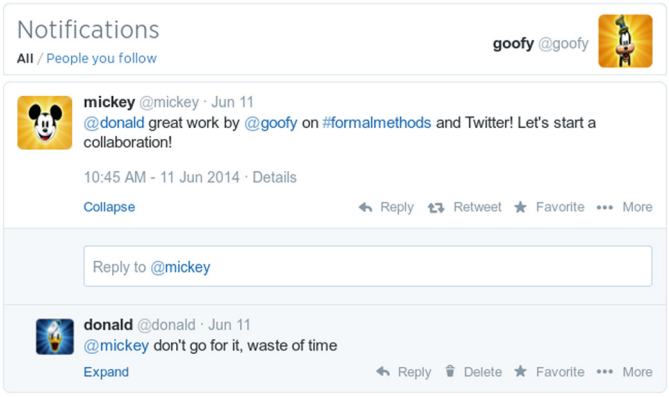

Fig. 4. Donald's reply is visible on Goofy's notification list.

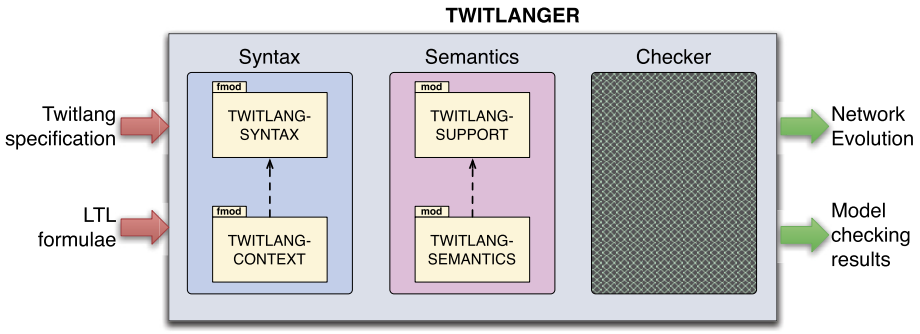

Fig. 5. Twitlanger architecture (for interpretation of the references to colour in this figure, the reader is referred to the web version of this article).

5. Twitlanger: executable Twitlang in Maude

Maude is "a programming language that models (distributed) systems and the actions within those systems" [10]. The systems are specified by defining algebraic *data types* axiomatising system states, and *rewrite rules* axiomatising local transitions of the system.

In this section, we present Twitlanger, the interpreter for Twitlang written in Maude. Four basic Maude modules represent the core of Twitlanger:

- TWITLANG-SYNTAX provides declarations of sorts, e.g., networks, messages, actions and behaviours, and operators on those sorts that are defined in the language syntax. It also defines subsort relationships which are mainly used to capture the hierarchy between sets and respective elements. This module also provides reserved ground terms representing the names of actions (tweet, delete, find, etc.) and network-level labels (found, added, etc.). Given the similarities between behaviours in Twitlang and processes in CCS [7], we used Verdejo and Martí-Oliet state-of-the-art implementation of CCS in Maude [11] as a foundation for operators definition.
- TWITLANG-CONTEXT defines the top-level behaviours' context that supports behaviour definition in terms of bindings to identifiers.
- TWITLANG-SUPPORT defines equations that provide support operators used in rewrite rules for behaviour unfolding and network transitions.
- TWITLANG-SEMANTICS defines rewrite rules, alongside additional operators and equations introduced to allow for a more compact and readable definition of the transition rules. The latter represent the operational semantics rules for behaviours and networks defined in Tables 3, 4 and 5 of Section 3.

At architecture level, the Maude modules described above are logically organised in components, as shown in Fig. 5 (where red arrows represent inputs, green arrows represent outputs, and dashed arrows represent use dependences among modules). The Syntax component allows Twitlanger users to interface with the tool, by providing as input a Twitlang specification. The Semantics component permits studying the evolution of the Twitlang network defined in the input specification, by exploring its computations. This information can be directly returned as output (*interpreter* facility of Twitlanger) or given as input to the Checker component. This latter component (whose Maude modules are described in Section 5.3) takes as a

Table 6
Experiment 1 completion times.

No. passive	No. 1 active		No. 2 active	
	v1.5 (s)	*v2.6 (s)*	*v1.5 (s)*	*v2.6 (s)*
1	0.047	0.040	0.748	0.043
2	0.059	0.040	4.389	0.044
3	0.127	0.040	25.526	0.052
4	0.445	0.040	146.125	0.047
5	1.973	0.040	788.612	0.046

further input the LTL formulae to be verified over the specification and performs the verification (*analysis* facility), whose results are returned as output.

Maude uses appropriate strategies for rule application. The default strategy is implemented by the *rewrite* command, that explores one possible sequence of rewrites, starting by a set of rules and an initial state [10]. To prevent undesirable looping caused by recursive rewrites inside operator arguments, we have adopted an approach similar to the one described in [11], where operators defining the evolution of behaviours and networks are declared as "frozen".

5.1. From Twitlanger 1.5 to version 2.6

One of the main goals of the first version of the interpreter (1.x), presented in [1], was to provide a form of "validation" to Twitlang. This objective guided the development towards a Maude specification as faithful as possible to the theoretical semantics, by matching transition rules to rewrite rules almost at a 1:1 ratio. However, one of the drawbacks of early versions of Twitlanger was the impossibility of producing more than a one-step successor of a given state using Maude's *rewrite* command. With the updated version 2.x of Twitlanger this restriction has been lifted in order to correctly integrate the Maude model checker with the interpreter. As a result of the shift in priorities, additional efforts have been dedicated to optimise the code for analysis efficiency and performance.

Besides small tweaks and bug-fixes, the prominent modification introduced moving to version 2.x of the interpreter was the removal of all the rewrite rules used to realize propagation effects over the network (i.e., transition rules [N-DELIVER], [N-DELPROPAG], [N-UNDOPROPAG] and [N-PAR]). The remaining rewrite rules have been redesigned to propagate their effects on the rest of the network using equations, which are in general much less computationally taxing. At the same time, this change "hides" intermediary propagation states to the Maude Model Checker – reasoning at the level of rewrite rules – that are not relevant for the properties of interest, thus delivering better performances without loss of meaningful information.

It should be noted that these modifications do not affect the correctness of the interpreter as the equational rules used for effect propagation are *confluent* (i.e., effects are "local" within each single user's network, thus the order of evaluation does not affect the outcome) and *terminating* for a finite set of users in the system.

To evaluate the improvements achieved in the new version of Twitlanger, we have realised two simple benchmarks that consist in measuring the time needed to compute all the configurations of two parametric test systems.[6] Each system is characterised by a network definition that is parametric in the number of users of specific "classes".

Experiment 1. This simple experiment consists in a system characterised by two classes of users, *Active* and *Passive*, defined in Twitlang by the following accounts:

- $Active : \epsilon : \epsilon : \epsilon : \mathbf{tweet}(\text{"}Hello\text{"}, x).\mathbf{delete}(x).\mathbf{tweet}(\text{"}Bye\text{"}, y).\mathbf{nil}$
- $Passive : \epsilon : \epsilon : Active : \mathbf{nil}$

While the behaviour of *Active* users is very simple, it can give a good indication on the performance improvement achieved by managing tweet/delete effects propagation with equations instead of rules. *Passive* users, on the other hand, are quite simply users that do nothing but follow *Active* users, thus receiving their tweets.

The obtained results are shown in Table 6 and in the respective graph of Fig. 6.

Although the performance gap is evident, it is worth pointing out how the first, unoptimised version of the interpreter is extremely sensitive even to very small variations on the number of users, to the point that with just two *Active* ones the completion time increases by almost one order of magnitude for each additional *Passive* user.

On the other hand the newer and optimised version of the interpreter is seemingly not affected by such small variations on the populations of the benchmark experiment.

Experiment 2. This second experiment consists in a system characterised by three classes of users: *Main*, *Active* and *Passive*. As in Experiment 1, the only classes of users that are subject to changes in population to evaluate the performance of the

[6] Notice that the following results are mainly intended to show how the two versions of the interpreter compare, while they do not necessarily reflect the achievable performances in a dedicated environment as all the benchmarks have been conducted on a common portable computer with limited processing power (CPU Intel Core i5-6200U, RAM 8 GB, operating system Linux Xubuntu kernel 4.4).

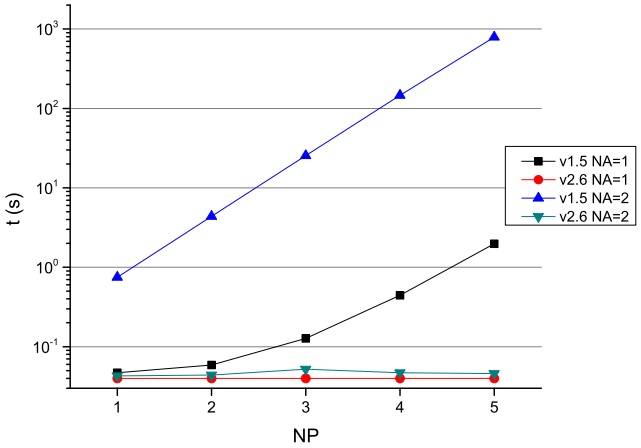

Fig. 6. Experiment 1 completion times (NA/NP being the number of *Active* and *Passive* users, respectively).

Table 7
Experiment 2 completion times.

No. passive	No. 1 active		No. 2 active	
	v1.5 (s)	*v2.6 (s)*	*v1.5 (s)*	*v2.6 (s)*
1	0.210	0.040	16.648	0.050
2	0.916	0.041	145.212	0.050
3	5.290	0.041	1057.778	0.051
4	29.673	0.043	6877.158	0.050
5	155.773	0.041	41160.268	0.051

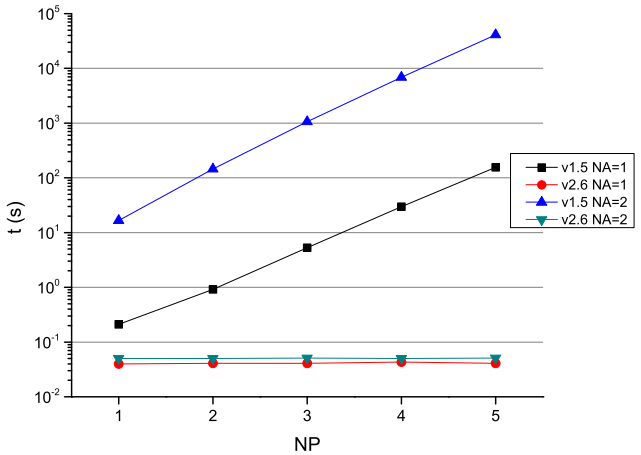

Fig. 7. Experiment 2 completion times (NA/NP being the number of *Active* and *Passive* users, respectively).

interpreters are *Active* and *Passive*, as class *Main* has always a fixed population of one instance. The characteristics of the three classes are defined in Twitlang as follows:

- $Main : \epsilon : \epsilon : \epsilon : \textbf{tweet}(\text{“}Hello\text{”}, x).\ \textbf{find}(\downarrow_7 = Active, z)@Main.\ \textbf{retweet}(z, y).\ \textbf{nil}$
- $Active : \epsilon : \epsilon : Main : \textbf{find}(\downarrow_7 = Main, z')@Main.\ \textbf{reply}(z', \text{“}Bye\text{”}, \{\}, x').\ \textbf{nil}$
- $Passive : \epsilon : \epsilon : Active : \textbf{nil}$

The obtained results are shown in Table 7 and in the respective graph of Fig. 7.

Despite having a slightly more complex scenario compared to Experiment 1, the overall results are roughly the same, showing how the new version of Twitlanger is not affected by the tested variations in the population of *Active* and *Passive* user classes, while the first release of the interpreter exhibits execution times exponentially proportional to the system's population.

Table 8
Experiment 1 completion times for Twitlanger 2.6.

NP	NA				
	1	2	3	4	5
10	0.049 s	0.047 s	0.183 s	4.674 s	376.464 s
100	0.044 s	0.065 s	0.516 s	16.435 s	1124.375 s
200	0.046 s	0.085 s	0.974 s	29.542 s	1948.732 s
300	0.048 s	0.110 s	1.344 s	41.424 s	2731.477 s
400	0.050 s	0.130 s	1.637 s	54.872 s	3579.970 s
500	0.051 s	0.152 s	2.148 s	67.180 s	4346.298 s
600	0.055 s	0.161 s	2.430 s	80.736 s	5166.600 s
No. of states	4	29	436	10761	372594

Table 9
Experiment 2 completion times for Twitlanger 2.6.

NP	NA				
	1	2	3	4	5
10	0.070 s	0.052 s	0.133 s	0.710 s	4.585 s
100	0.044 s	0.074 s	0.330 s	1.987 s	12.676 s
200	0.047 s	0.103 s	0.547 s	3.514 s	21.733 s
300	0.050 s	0.126 s	0.704 s	4.660 s	29.920 s
400	0.055 s	0.161 s	1.000 s	6.085 s	39.012 s
500	0.056 s	0.186 s	1.159 s	7.355 s	48.635 s
600	0.061 s	0.215 s	1.397 s	8.841 s	56.090 s
No. of states	6	32	190	1057	5479

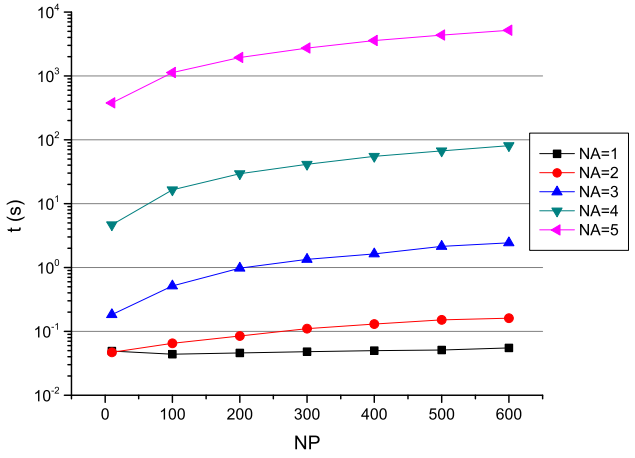

Fig. 8. Experiment 1 completion times for Twitlanger 2.6 (NA/NP being the number of “Active” and “Passive” users, respectively).

While these preliminary results are encouraging, the tested populations have been restrained in order to obtain “reasonable” execution times on version 1.5 of Twitlanger, but evidently they are not big enough to be representative of real-world scenarios nor they tax the new version of the interpreter to the point of allowing us to assess its actual limitations.

To partially address this, we made Twitlanger 2.6 run the previous benchmarks again with a larger variation in the population of both *Active* and *Passive* users. The results are shown in Tables 8 and 9, and in the respective graphs of Figs. 8 and 9.

The results of these benchmarks outline a performance figure indicating that Twitlanger 2.6, in its current state, can handle well Twitlang definitions of systems with a large number of “passive” users following a limited number of “active” ones. This kind of configuration, referred to as “hubs” in network theory, can fit many interesting real-world use cases.

5.2. Twitlanger: an example

The example presented in Section 4 can be used to demonstrate how a Twitlang specification can be coded to be accepted by Twitlanger, and how to use it to elaborate all the reachable states.

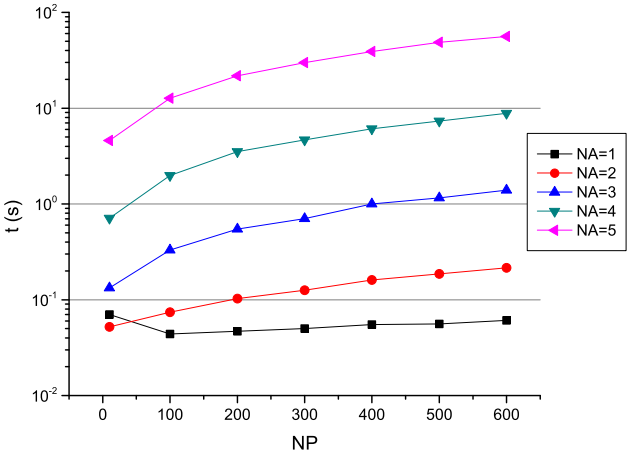

Fig. 9. Experiment 2 completion times for Twitlanger 2.6 (NA/NP being the number of "Active" and "Passive" users, respectively).

The representation of each state in Twitlanger has the following structure:

```
!(id-counter,
{network-label_1}
{network-label_2}
...
{network-label_N}
(user_1 : timeline_1 : notification-list_1 : following-list_1 : behaviour_1)
||
...
||
(user_M : timeline_M : notification-list_M : following-list_M : behaviour_M)
```

The `id-counter`, as the name suggests, is an integer counter that is stored in the state representation and indicates the next "fresh" *id* that can be assigned to a message. Each time an *id* is assigned, the `id-counter` is accordingly incremented by 1. This is a very simple implementation that ensures extremely low overhead on the interpreter and it effectively realises the pre-condition $id \notin \text{ids}(T, N, B)$ in Table 4 if the value of `id-counter` in the initial specification is chosen strictly greater than all the *ids* of messages already present in the system.

The network realising the example can be encoded in the machine-readable syntax of Twitlang as follows:

```
!(1,
(Donald : empty : empty : Mickey :
find(predP7(Mickey) & predHashTag(# 'formalmethods),z)@ Donald) .
reply(z, @ Mickey 'dont 'go 'for 'it 'waste 'of 'time, @ Goofy, y) . nil)
||
(Goofy : empty : empty : emptyfl : nil)
||
(Mickey : empty : empty : Donald :
tweet(@ Donald 'great 'work 'by @ Goofy 'on # 'formalmethods 'and
'Twitter, x) . find(predP7(Donald),z')@ Mickey . delete(x) . nil)
```

No network label is initially present, as no user has yet performed any action. Then, the interpreter can be used to evaluate the evolution of the network, verifying that the exploration yields the expected outcome. Indeed, provided that the specification includes an equation stating that the term `example` corresponds to the given initial state, by issuing the following command:

```
search example =>* T:Twitter .
```

we obtain a full unfolding of the possible rewrite traces, up to the final state (from now on, for the sake of compactness, we will use `M1` and `M2` to refer to the original tweet from Mickey and the reply message from Donald, respectively having *id* equal to 1 and 2):

```
!(3,
{M1}
{(Donald :Nfound(M1))}
{M2}
{(Mickey :Nfound(M2))}
```

```
{(Mickey :Ndelete(1))}
(Donald : M2 : empty : Mickey : nil)
||
(Goofy : empty : empty : emptyfl : nil)
||
(Mickey : M2 : M2 : Donald : nil)
```

Note that, contrary to `Nfound`, the `Ndelete` network label takes as second argument only the *id* of the tweet instead of the whole message, coherently with the transition labels λ of Twitlang defined in Section 3.3.

This last state cannot be further unfolded as all the behaviours of the users in the network have been reduced to the inert behaviour **nil**.

Further analyses of the interactions can be performed by invoking `search` with the `such that` clause, effectively introducing a condition that the solutions have to fulfil. For instance, we may use the auxiliary operator `expand`, which evaluates accessible messages through direct linking (without resorting to the **find** action) from a specific user's perspective:

```
search example =>* T:Twitter such that ( M2 in expand(Goofy,1,T:Twitter) ) .
```

The command basically says "find all states of the system in which user *Goofy* can access message M2 via a one-hop link". The output produced by the interpreter in this case is comprised of two solutions, the first one describing the trace and the state:

```
!(3,
{M1}
{(Donald :Nfound(M1))}
{M2}
(Donald : M1 ; M2 : M1 : Mickey : nil)
||
(Goofy : empty : M1 : emptyfl : nil)
||
(Mickey : M2 ; M1 : M2 : Donald :
find(predP7(Donald),z')@ Mickey . delete(1) . nil)
```

which represents the system configuration after *Donald* replies to *Mickey*. It shows that indeed *Goofy* is able to easily access M2 as soon as the message is published, even though it carries no mention of him. On the other hand, the only other solution found by the interpreter that satisfies the clause is the subsequent state in which *Mickey* has performed the **find** action. These results confirm that, after deleting `M1`, *Goofy* loses his only direct link to `M2` and, thus, he cannot access it without resorting to an explicit **find**.

5.3. Integration with the Maude LTL model checker

The Maude LTL model checker [12] is an *on-the-fly explicit-state* model checker provided as a module that can be imported into a Maude specification and used to verify properties written in *linear temporal logic*.

Given a set of *atomic propositions* AP, formulae of the *propositional linear temporal logic* $\mathrm{LTL}(AP)$ can be inductively defined as follows:

- **True**: $\mathsf{T} \in \mathrm{LTL}(AP)$; this formula holds always.
- **Atomic propositions**: if $p \in AP$, then $p \in \mathrm{LTL}(AP)$; this formula holds at a given state of the system whenever the proposition p is satisfied by this current state.
- **Next operator**: if $\varphi \in \mathrm{LTL}(AP)$, then $\circ\varphi \in \mathrm{LTL}(AP)$; this formula holds at a given state when the subformula φ holds at the next state.
- **Until operator**: if $\varphi, \psi \in \mathrm{LTL}(AP)$, then $\varphi \,\mathcal{U}\, \psi \in \mathrm{LTL}(AP)$; this formula holds at a given state when the subformula ψ holds at the current or a future state and φ continuously holds until then.
- **Boolean connectives**: if $\varphi, \psi \in \mathrm{LTL}(AP)$, then the formulae $\neg\varphi$ and $\varphi \vee \psi$ are in $\mathrm{LTL}(AP)$; these operators have the standard meaning of logical negation and disjunction.

Additional LTL connectives can be defined from the above minimal set and are included in Table 10 along with their syntax in the Maude model checker; we refer to [12] for a more complete list of derived operators supported by the Maude LTL model checker.

As detailed in [12], by fixing a distinguished sort *State*, the initial model of a Maude's rewrite theory $\mathcal{R}$ has an underlying Kripke structure $\mathcal{K}(\mathcal{R}, State)$ given by the total binary relation extending its one-step sequential rewrites. Since Kripke structures are models of temporal logic, in order to obtain a language of LTL properties for the rewrite theory $\mathcal{R}$, the only additional ingredients required are atomic predicates for $\mathcal{K}$, which can be specified as *equationally-defined computable state predicates*.

Table 10
Linear temporal logic operators and their syntax in Maude LTL model checker.

Operator	Syntax	Syntax (Maude)	Primitive rep.
Next operator	$\circ\varphi$	`O_`	–
Until operator	$\varphi\ \mathcal{U}\ \psi$	`_U_`	–
Boolean OR	$\varphi \vee \psi$	`_\/_`	–
Boolean NOT	$\neg\varphi$	`~_`	–
Boolean AND	$\varphi \wedge \psi$	`_/\_`	$\neg(\neg\varphi \vee \neg\psi)$
Implication	$\varphi \rightarrow \psi$	`_->_`	$\neg\varphi \vee \psi$
Eventually	$\Diamond\varphi$	`<>_`	$\mathsf{T}\ \mathcal{U}\ \varphi$
Henceforth	$\Box\varphi$	`[]_`	$\neg\Diamond\neg\varphi$
Leads-to	$\varphi \rightsquigarrow \psi$	`_|->_`	$\Box(\varphi \rightarrow (\Diamond\psi))$

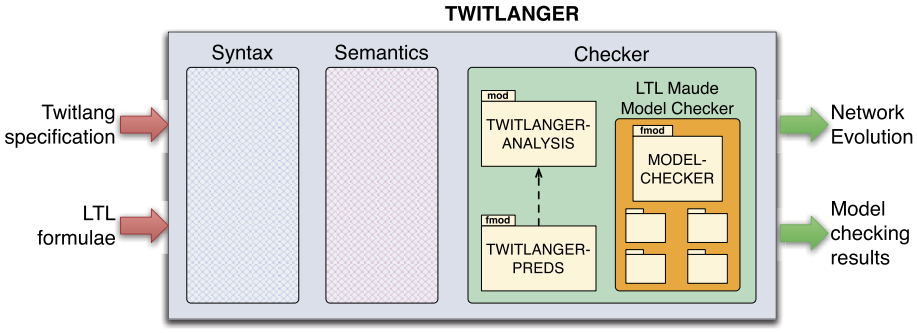

Fig. 10. Twitlanger architecture: checker component in detail.

In practice, integrating and using the Maude LTL model checker in Twitlanger is quite straightforward (see Fig. 10). A new module, which we called `TWITLANGER-PREDS`, has to be defined to include the predefined `MODEL-CHECKER` module and specifying:

- the *subsort declaration* that identifies which sort in the specification will be used as *State* of the Kripke structure: in our case this declaration corresponds to `subsort Twitter < State .` (where `State` is a key sort of the `MODEL-CHECKER` module);
- the *syntax of the state predicates* relevant for the analysis through operators and constants of sort `Prop` (which is a subsort of `Formula` in the `MODEL-CHECKER` module);
- the *semantics of the state predicates* via equations involving the operator `|=` defined in the `MODEL-CHECKER` module as follows:

```
op _|=_ : State Prop -> Result [special ...] .
```

Once this module is set up, it can be used to model check any LTL formula `form` that involves the defined state predicates, given an initial state `init`, by issuing the following Maude command:

```
reduce modelCheck(init,form) .
```

Assuming that the set of reachable states is finite, Maude will either return `true` if `form` holds, or a counterexample, if it does not hold.

To simplify the definition of the module, operations supporting the analysis of Twitlang specifications (e.g., to access the messages in a user timeline or managing the message linking) are provided in the `TWITLANGER-ANALYSIS` module.

Moreover, for the purpose of performing analyses on basic message delivery/visibility over Twitter accounts, we have defined a small set of state predicates:

- `op tweetSent : Predicate -> Prop .`
  given a predicate P over messages (just like the one used for the **find** action, not to be confused with state predicates), this property holds in any state reached after a tweet matching P has been sent;
- `op tweetFound : Predicate User -> Prop .`
  given a predicate P over messages and a username U, this property holds in any state reached after a tweet matching P has been found (via action **find**) by U;
- `ops tweetInTimeline tweetInNList tweet@User : Predicate User -> Prop .`
  given a predicate P over messages and a username U, these three properties hold in any state where a tweet matching P is present respectively in U's timeline, notification list, or either of them;

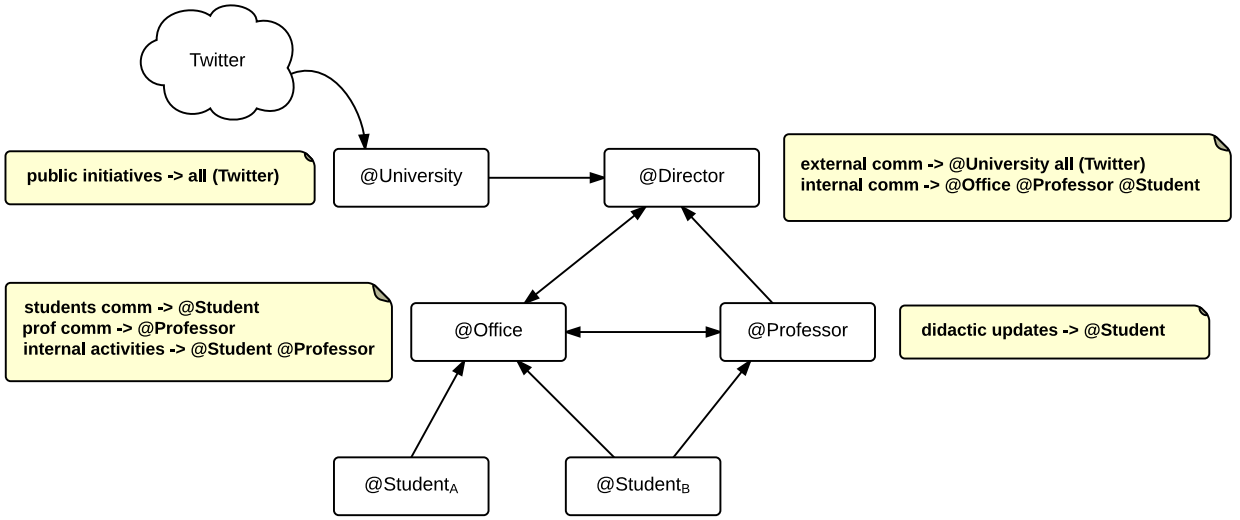

Fig. 11. Accounts and their relationships in the university case study.

- `op tweet@Users : Predicate UserList -> Prop .`
  given a predicate P over messages and a list of usernames L_U, this property holds in any state where a tweet matching P is present – either in the timeline or the notification list – in all the accounts whose usernames are in the list L_U;
- `op tweetDeleted : Id User -> Prop .`
  given a message identifier *id* and an account username u, this property holds in any state reached after the tweet with identifier *id* has been deleted by u;
- `op retweetUndone : Id User -> Prop .`
  given a message identifier *id* and an account username u, this property holds in any state reached after the retweet with identifier *id* has been undone by u.

In Section 6 we will show how these state predicates can be used to specify LTL formulae encoding relevant properties for analysing interactions among Twitter accounts via Twitlanger.

A more comprehensive overview of Twitlanger, alongside the access to the complete Maude implementation of the Twitlanger modules and examples discussed in this paper are available at http://sysma.imtlucca.it/tools/twitlanger/.

6. Twitlang(er) at work on a case study from the academic domain

To better understand the benefits of employing Twitlang to model communications in Twitter, in this section we present and analyse a richer and more concrete case study. The reference domain is an academic one, constituted by a small group of users representing the staff and students in a university who are involved (both actively and passively) in several communications with different topics and intended audiences.

We selected the following set of user accounts: *University*, *Director*, *Office*, *Professor*, and $Student_i$. Fig. 11 shows the pictorial representation of Twitter accounts and relationships possibly involved in such a scenario. In this schematic representation, rounded rectangles represent those Twitter accounts significant for the case study. Notice also that we included a "Twitter cloud" in the schema, representing a subset of the entire Twitter network that follows the official *University* account. Incoming arrows represent follow relationships (e.g., *University* follows *Director*, which in its turn follows *Office*, etc.). It is worth noting that the depicted configuration has been defined on the base of reasonable hypotheses, but it is obviously not meant to be exhaustive and other relationships could have been introduced. Finally, in yellow we have highlighted possible communications that the accounts at stage are interested to generate, and the related expected receivers. For example:

- *Professor* is mainly interested in sending notices about didactic updates to $Student_i$ accounts;
- *Director* produces both external communications targeted at *University* and the rest of Twitter, as well as internal communications meant for *Office*, *Professor* and $Student_i$.

To demonstrate some basic operations with the Maude LTL model checker in combination with Twitlanger, we can start defining a simple behaviour for the *University* account, by adding the following definition to the `context`:

```
'Bu = def ( tweet('Academic 'Year 'Inauguration '2016 'tomorrow,x) .
            tweet('Open 'call 'for 'researchers,y) . nil )
```

Then we can verify that the user representing the Twitter cloud eventually receives both messages sent from *University*. We can formalise this property with the following LTL formula:

```
eq form1 =
  ( tweetSent(predP4(tweet-inaug)) |-> tweet@User(predP4(tweet-inaug),TwtCloud) )
  /\
  ( tweetSent(predP4(tweet-call)) |-> tweet@User(predP4(tweet-call),TwtCloud) ) .
```

where the predicate `predP4` yields `true` when evaluated on a message whose textual content (i.e., the fourth field of the message tuple) matches the argument. For the sake of compactness, we used the terms `tweet-inaug` and `tweet-call` to refer to the contents of the two tweets sent by *University*.

We can thus ask the model checker to see whether the specification `case-study` satisfies the formula by issuing the command:

```
reduce modelCheck(case-study,form1) .
```

which unsurprisingly returns `true`. Indeed this property is trivially satisfied considering that the *TwtCloud* account follows *University*, thus all tweets sent by it are bound to appear on *TwtCloud*'s timeline.

Similarly, we can check that, in this context, at no point *Director* will have tweets sent by *University* in his timeline or notification list, by means of the formula:

```
eq form2 = [] ~ ( tweet@User(predP7(University),Director) ) .
```

that the Maude model checker proves again to be `true`.

Supposing that *Director* needs to issue an internal communication directed at the faculty, we might want to verify that the Twitter cloud does not receive such a message in its timeline or notification list, despite its indirect following connection to *Director* through *University*. We can thus enrich the initial state by defining the behaviour of *Director* as:

```
'Bd = def ( tweet('Scientific 'Board 'meeting 'canceled,x) . nil )
```

and then express the relevant property as the formula:

```
eq form3 = [] ~ ( tweet@User(predP4(tweet-meeting),TwtCloud) ) .
```

Furthermore, we could modify the Twitter cloud behaviour in order to capture the very common user practice of looking for tweets browsing another account's profile page:

```
'Btwt = def ( (find(predP4(tweet-meeting),z)@University) . nil )
```

and verify that the message sent by *Director* cannot be accessed from the profile page of *University* by testing the formula:

```
eq form4 = [] ~ ( tweetFound(predP4(tweet-meeting),TwtCloud) ) .
```

As expected, formulae `form3` and `form4` turn out to be satisfied.

Now, consider a different scenario within the same context where *Professor* has to issue a communication regarding an exam and he would like all $Student_i$ accounts to receive it. We can initially define *Professor*'s behaviour as follows:

```
'Bp1 = def ( tweet(#('exam) 'will 'take 'place 'in 'classroom 'A5,x) . nil )
```

Then, we can express the desired property as the LTL formula:

```
eq form5 = tweetSent(predP4(tweet-exam)) |->
           ( tweet@Users(predP4(tweet-exam),(StudentA ;; StudentB)) ) .
```

where `tweet-exam` is again a compact notation for the content of the tweet sent by *Professor*, while `;;` denotes the concatenation operator among list elements. Invoking the model checker with this formula over the initial state enriched with the behaviour `'Bp1`, it returns a *counterexample*, meaning that the property does not hold. The counterexample provided by the Maude model checker is a pair consisting of two lists of transitions: the first one is a finite path starting from the provided initial state, and the second one describes a loop. In our case, the second element is simply a *deadlock* state that is reached once the behaviour associated to each account reduces to `nil` and thus the system can no longer evolve:

```
!(4,
{< 1 null null (tweet-inaug) none none University >}
{< 2 null null (tweet-call) none none University >}
{< 3 null null (tweet-exam) none none Professor >}
(University : < 2 null null (tweet-call) none none University > ;
              < 1 null null (tweet-inaug) none none University >
            : empty : Director : nil)
||
(Director : empty : empty : Office : nil)
||
(Office : < 3 null null (tweet-exam) none none Professor >
        : empty : Director ;; Professor : nil)
||
(Professor : < 3 null null (tweet-exam) none none Professor >
           : empty : Office ;; Director : nil)
||
(StudentA : empty : empty : Office : nil)
||
(StudentB : < 3 null null (tweet-exam) none none Professor >
          : empty : Office ;; Professor : nil)
||
(TwtCloud : < 2 null null (tweet-call) none none University > ;
            < 1 null null (tweet-inaug) none none University >
          : empty : University : nil)
```

The outcome is not surprising, as $Student_A$ does not follow *Professor* (see Fig. 11) and thus it did not receive the tweet related to the exam.

Considering Twitter's communication characteristics, there is a variety of alternative configurations that can fulfil `form5`, some of which are as follows:

1. *Professor* mentions $Student_A$ and $Student_B$ in the tweet:

   ```
   'Bp2 = def ( tweet(@(StudentA) @(StudentB) tweet-exam,x) . nil )
   ```

   This solution has the advantage of making the tweet to appear in both students' notification lists, which in Twitter makes such messages more evident to the receiver compared to tweets present in the account's timeline, as they are collected separately. On the other hand, this approach does not scale well, and quickly becomes infeasible with the growing number of students (if anything, because a tweet is naturally enclosed in 140 characters, including mentions).
2. Each student interested in the exam must follow the professor account:

   ```
   'Bs = def ( follow(Professor) . nil )
   ```

   This solution has the benefit of decentralising the responsibility of defining the recipients of the communication from the sender and let only the interested students be involved in the related tweets. However, this approach requires a change in the behaviours of users in the system who are not necessarily controllable by the university (i.e., students) and may not be willing to timely update their following list.
3. Assuming that, like in our initial configuration, each $Student_i$ follows the *Office*, the latter could adopt the policy of retweeting tweets sent by *Professor* having hashtag `#exam`:

   ```
   'Bo = def (find(predHashTag(#('exam)) & predP7(Professor),z)@Office .
             retweet(z,y) . nil )
   ```

   Of course, this third solution has the implication that all the students – following *Office* – will receive the tweet about the exam, independently from their interest in it.

Giving as input to the Maude model checker different initial states realising the three proposed solutions, we obtain a positive result for the satisfaction of `form5` in all instances.

It should be noted that the way the behavioural context is defined in Twitlanger allows to define recursive behaviours. This would support a modification of the third solution in order to model a simple *bot* that keeps retweeting matching messages from *Professor*. To make it work properly, though, this would require some additional "tricks" in the behaviours of the accounts to prevent *Office* from retweeting the same tweet indefinitely, and most importantly it would be problematic to use in conjunction with the Maude model checker as it cannot handle an infinite set of reachable states.

We conclude this section by illustrating further properties relevant for the case study, which also allow us to show other LTL operators at work. First, let us consider the case in which *Office* tweets a communication with all students as expected target; when the communication will not be considered relevant anymore, *Office* will delete it. We can formalise the property that the communication is available to $Student_i$ accounts, until it is not deleted, as follows:

```
eq form6 = tweetSent(predP4(tweet-communication) & predP1(1)) |->
           ( tweet@Users(predP4(tweet-communication),(StudentA ;; StudentB))
             U tweetDeleted(1,Office) ) .
```

This property is naturally expressed by resorting to the Until operator. As expected, the formula is satisfied.

Consider now the case in which the *Director* updates an internal communication previously sent, by changing the time of a meeting. We can formalise two properties as follows:

- whenever the updated communication is sent, the old communication does no longer appear:

```
eq form7 = [] tweetSent(predP4(tweet-new-communication)) ->
               ( [] ~(tweet@User(predP4(tweet-old-communication),Director)) ) .
```

- the old communication appears until the new one is sent:

```
eq form8 = [] tweetSent(predP4(tweet-old-communication)) ->
               ( tweet@User(predP4(tweet-old-communication),Director) )
                 U tweetSent(predP4(tweet-new-communication) ) .
```

This pair of properties is interesting because it may seem reasonable that both of them are satisfied. However, as demonstrated by their verification, this is not the case. Indeed, assuming that the *Director* deletes the old communication before emitting the updated one, `form7` holds while `form8` does not. This happens because there are some system states in which the old communication is cancelled but the new one has not been tweeted yet. Instead, assuming that the *Director* deletes the old communication after emitting the updated one, `form8` holds while `form7` does not. Therefore, the two properties never hold at the same time.

Finally, let us consider a communication tweeted by a *Professor* and retweeted by *Office*. We may expect that a message with this content is not available to $Student_i$ accounts after *Office* undoes the retweet. This property can be formalised as follows:

```
eq form9 = tweetSent(predP4(tweet-prof-communication) & predP1(2)) |->
           ( retweetUndone(2,Office) ->
             [] ( ~ tweet@User(predP4(tweet-prof-communication),StudentA)
                 /\ ~ tweet@User(predP4(tweet-prof-communication),StudentB) ) ) .
```

This formula, however, is not satisfied. In fact, in this case, since the deletion action acts on a retweet rather than a tweet, the effect involves only the retweet messages. In particular, since the account $Student_B$ directly follows *Professor*, it has received the original tweet, besides the retweet. Therefore, the predicate `tweet@User(predP4(tweet-prof-communication),StudentB)` evaluates to true and, hence, the formula argument of the Henceforth operator, as well as the overall formula `form9`, is false.

7. Related work

Recent studies have put a spotlight on Twitter in a variety of research areas. Remarkably, efforts have been spent towards the characterisation of those social dynamics, which are inferred through the analysis of the platform and have an impact on real life (and vice versa). Usually, the existing studies concern two important dimensions: the Twitter network and the tweets content.

Focusing on network aspects, the general idea is to characterise significant users' behaviours in terms of kind and frequency of their Twitter interactions. Authors of [5] provide a characterisation of the topological features of the Twitter follow graph, mainly aiming at answering questions related to the inner nature of the platform, e.g.: "Is Twitter a social network or an information network?". From the analysis they carried on, conclusions are that Twitter evolves towards a social network. Indeed, even if the "follow" relationships is primarily about information consumption, many relationships are instead "built on social ties". Similar issues are addressed in [13], where two Twitter networks are identified: a network made of followers and friends that shows a certain level of stability and a "topical" network, characterised by a high level of contingency. The work investigates how the two networks influence each other (for example, whether the participation in the same hashtag-based conversation changes the follower list of the involved accounts). Finally, work in [14] models information propagation through different social networks (among them, Twitter). While the mentioned works concern information and social aspects of Twitter, we are interested in the effects of user interactions in terms of message spreading. In order to properly address aspects concerning the dynamic evolution of social relationships with Twitlang, we should probably consider an extension of the language including username passing, i.e., the capability of binding variables to dynamically discovered usernames, in the style of name passing in π-calculus [9].

The second prominent area of research is the analysis of the tweets content, motivated by a number of goals, such as, e.g., personalised message recommendation, breaking news detection, and sentiment analysis. As examples, aiming at making tweets useful for recommendations, authors of [15] propose a method for enriching the semantics of tweets, by

identifying and detailing, e.g., topics, persons, events mentioned in tweets. The usefulness of the platform for real-time crisis management has been tested by various work, see, e.g. [16,17], where technologies were investigated for understanding the semantic meaning of Twitter messages. Authors of [18] study the Twitter hashtags ability to represent real-world entities, by comparing hashtags characteristics with Semantic Web "strong identifiers" features. By analysing a dataset of Twitter conversations, work in [19] measures the "economy of attention" in the Twitter world. As predicted by Dunbar's theory, Twitter users can entertain a maximum of 100–200 stable relationships and are limited by cognitive and biological constraints as well as in the real world. Revealing the sentiment behind a tweet is motivated by several reasons. For example, sentiment analysis may help in the assessment of a global overview, i.e., to figure out how many opinions on a certain topic are positive or negative (see, e.g., the series of work in [20,21] for a polarity evaluation of tweets). Still related to polarity detection and sentiment analysis, both works in [22,23] detect the "public sentiment" over real tweets-sets and associate its fluctuations with a timeline of notable events that took place in the period tweets were collected. The authors of [24] address the problem of using text-mining tools to understand tweets (whose restricted length may prevent such tools from being employed to their full potential). The authors propose several schemes to train standard tools and compare their quality and effectiveness.

In our work, we have mainly strived to put the basis for analysing the communications among a (sub)network of Twitter accounts. For achieving the goal, instead of exclusively focusing on the content of tweets and/or on social aspects of the Twitter interactions, we carried on a novel study on the effects of such interactions, from the point of view of the Twitter user, with a special care on understanding and characterising the communication mechanisms underlying the message spreading. To the best of our knowledge, there is no previous attempt to rigorously formalise Twitter interaction patterns. Instead, a series of blogs offer to the general public some useful, yet informal, tips on tweets, retweets, and replies, see, e.g., [25]. We based our formalisation on experiments that we have extensively carried out to properly define the Twitlang semantics. In this task, we were also supported by the explanation offered by the Twitter staff and available online.[7]

Proposing a syntax and associated semantics able to describe the cause-effect relationships among communicating Twitter accounts should not be considered as a standalone work. Indeed, our formalisation aims at putting rigorous basis for a uniform approach to Twitter accounts' properties specification and analysis. The first, significant step in this direction is given by the implementation of the Twitlanger tool. In Section 6 we focused our attention to the evaluation of communication properties, mainly based on follow relationships among a set of Twitter accounts. We could easily extend our approach, still towards the verification of communication properties, but exploiting a richer set of conditions over additional characteristics of the accounts. As instances of properties verifiable with minimal modification to Twitlang, we could verify that account *@X* actually receives tweets from all the accounts that twitted the hashtag #hashtag. Furthermore, let the reader consider that account *@Y* is representative of company *W*; the account *@Y* would like to be sure that his tweets will reach all the accounts that have retweeted (or have replied to) tweets from the *W*'s competitors. As a further example, account *@Z* would aim at not receiving tweets from accounts that, e.g., use to reply with a certain frequency, have no image in their profiles, use to tweet more URLs than plain text, etc.

Noticeably, the last example reminds behaviours typical of malicious Twitter accounts, as pointed out in, e.g., [26]. Indeed, Twitter versatility and spread of use have made it the ideal arena for proliferation of anomalous accounts, that behave in unconventional ways. Literature has focused its attention on *spammers*, that is, those accounts actively putting their efforts in spreading malware, sending spam, and advertising activities of doubtful legality (see, e.g., [27]) as well as on *fake followers*, corresponding to Twitter accounts specifically exploited to increase the number of followers of a target account (see, e.g., [28,29]). Interestingly, a brand new branch on research in the area is the one of anomalous groups detection. As spammers evolve, standard classification techniques may not be effective anymore and new ones are being proposed to catch behavioural commonalities of groups of accounts, see, e.g., work in [30,31]. One of our research goals for the future is to leverage the Twitlang(er) approach also for distinguishing genuine accounts from anomalous ones, by making use of the analysis techniques enabled by the formal semantics and based on behavioural characteristics, like, e.g., the frequencies of tweets and retweets and the massive presence of URLs in tweets.

We think that our work can be extended in several directions, in order to enable some of the analyses mentioned above. In fact, our formalism could serve as a uniform, common formal ground for modelling and analysing Twitter accounts' behaviour. For example, quantitative information could be added to model the frequency of actions (by resorting, e.g., to a stochastic approach). Also, the current semantics of our language assumes that all the accounts under investigation are public. As anticipated in the Introduction, considering protected accounts would have led to modifying the modalities through which tweets flow in the network. In this regard, we acknowledge the work in [32–34], where we assist to a paradigm shift on modelling privacy settings and access control policies on social networks. In fact, we see how, from standard access control mechanisms, the research trend moves forward towards considering the relationships among the accounts in the network. Other models, like the one in [35], consider a variant of Dynamic Epistemic Logic to model both public and private announcements on Facebook. We do not exclude to update Twitlang to deal with protected accounts, even if, in the current work, we have been more interested in modelling and analysing settings where the accounts are public.

[7] https://support.twitter.com/articles/119138 and https://support.twitter.com/articles/164083.

We conclude the section by comparing Twitlang with some of the closely related works from the process calculi literature, which are not specifically devised for Twitter but have nevertheless inspired some features of our formalism. The network layer of Twitlang and, in particular, the tuple-based format of messages, take inspiration from Klaim [36]. However, the communication between Klaim network nodes takes place via Linda-like primitives and is only dyadic, while a Twitlang account can atomically send messages via Twitter-like primitives to multiple accounts. A similar form of multicast communication is provided by SCEL [37], which anyway is established on a generic attribute-based approach, specifically devised for dealing with dynamic formation of autonomic component ensembles. The attribute-based communication of SCEL could be exploited to model the delivery of tweets to their target accounts, but it is not suitable for atomically removing messages from multiple accounts as required by actions **delete** and **undo**. Moreover, with respect to SCEL, and other formalisms based on π-calculus [9], Twitlang is not equipped with the restriction operator, which is indeed not necessary for the scope of our study. Finally, Twitlang behaviours are defined by composing Twitter actions by means of some operators borrowed from CCS, i.e. action prefixing, nondeterministic choice, parallel composition and invocation of process definitions.

8. Concluding remarks

In this work, we have extended Twitlang, a formal language to model communication interactions on Twitter. The operational semantics of the language allows to know in advance which are the effects of the basic actions that Twitter users daily perform, without the need of setting up experiments (which, of course, we have extensively carried out to properly define our formal semantics). On top of the formal semantics, we have improved the computational performances of Twitlanger, the interpreter of the language written in Maude. As a further progress, the Maude model checker supporting automatic analysis has been incorporated into Twitlanger, thus enabling verification of Twitter interactions properties. We tested the benefits of the whole approach over a non-trivial case study inspired by an application of Twitter to everyday communication tasks.

It is worth noting that the language is currently able to capture the core aspects of Twitter communications, i.e., standard behavioural patterns, like posting a tweet, replying to, or retweeting a particular tweet. However, it could be easily extended by giving both the syntax and the semantics rules for more specific features, as direct messages, retweets including new text, and blocking of an account. Concerning peculiar behaviours, an example, which perhaps not everyone is aware of, is as follows: putting a mention at the very beginning of a tweet implies that the tweet is sent only to the intersection of the author's followers and of the mentioned account's followers. This and other peculiarities, if considered relevant for specific analyses, could be incorporated within our approach.

We also plan to carry out a comprehensive study of interesting properties for Twitter networks. In this paper, indeed, the verification of properties relevant for the considered case study is used to illustrate the feasibility and effectiveness of the approach we propose, while a more complete report of properties is out of its scope.

Concerning our tool, Twitlanger, we aim as future work at improving its performance and usability. Indeed, for a practical use in large case studies, the performance of the tool needs to be enhanced. As discussed in Section 5.1, a significant gain in this respect has been already achieved passing from version 1.x to 2.x. Further improvements could be made by still working on the Maude code, or by integrating other analysis tools, such as MultiVeStA [38] or the Maude LTLR model checker [39]. On the other hand, for achieving better performances, a more drastic reengineering of the tool could consist in the production of a new implementation using a classic programming language (e.g., C, C++, Java). In this case, the Maude implementation of Twitlanger would serve as guideline for the new implementation and, most of all, as reference implementation to be used as a means of comparison to ensure the faithfulness of the new implementation to the operational semantics of Twitlang. Regarding the usability of the tool, we aim at developing a user-friendly, on-line service, based on Twitlanger. In particular, our intention is to make the tool more accessible via a graphical user interface. On the one hand, it will allow users to define (parts of) Twitlang specifications by means of a graphical editor. On the other hand, it will allow to perform analysis by means of simple questions and easy-to-understand answers. This latter point would simply rely on the use of structured questions, or even on techniques for automatically converting natural language to LTL logic.

Acknowledgements

We thank Rocco De Nicola for his fundamental contribution to the definition of Twitlang. We also thank the anonymous referees for their useful comments and suggestions.

References

[1] R. De Nicola, A. Maggi, M. Petrocchi, A. Spognardi, F. Tiezzi, Twitlang(er): interactions modeling language (and interpreter) for Twitter, in: Proceedings Software Engineering and Formal Methods – 13th International Conference, SEFM 2015, York, UK, September 7–11, 2015, 2015, pp. 327–343.
[2] C. Smith, 170 amazing Twitter statistics and facts. DMR, http://goo.gl/SK9CmP, November 2016.
[3] Brandwatch.com, Brands on Twitter 2013: brandwatch report, https://goo.gl/ucHyKK, 2013, last checked December 18, 2015.
[4] Save the Children, It's hurricane season. Is your family prepared in case of an emergency?, http://goo.gl/vZynkt, 2014, last checked December 18, 2015.
[5] S.A. Myers, A. Sharma, P. Gupta, J. Lin, Information network or social network?: The structure of the Twitter follow graph, in: WWW, ACM, 2014, pp. 493–498.
[6] A. Ritter, C. Cherry, B. Dolan, Unsupervised modeling of Twitter conversations, in: HLT-NAACL, 2010, pp. 172–180.

[7] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.
[8] G. Plotkin, A structural approach to operational semantics, J. Log. Algebraic Program. 60–61 (2004) 17–139.
[9] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, Inf. Comput. 100 (1) (1992) 1–77.
[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-performance Logical Framework, Springer, 2007.
[11] A. Verdejo, N. Martí-Oliet, Implementing CCS in Maude 2, in: WRLA, in: ENTCS, vol. 71, Elsevier, 2002, pp. 239–257.
[12] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, Electron. Notes Theor. Comput. Sci. 71 (2002) 162–187.
[13] L. Rossi, M. Magnani, Conversation practices and network structure in Twitter, in: Sixth International Conference on Weblogs and Social Media ICWSM, 2012.
[14] M. Magnani, L. Rossi, The ML-model for multi-layer social networks, in: ASONAM, 2011, pp. 5–12.
[15] F. Abel, Q. Gao, G.-J. Houben, K. Tao, Analyzing user modeling on Twitter for personalized news recommendations, in: User Modeling, Adaption and Personalization, Springer, 2011, pp. 1–12.
[16] F. Abel, C. Hauff, G.-J. Houben, R. Stronkman, K. Tao, Twitcident: fighting fire with information from social web streams, in: WWW, 2012, pp. 305–308.
[17] M. Mendoza, B. Poblete, C. Castillo, Twitter under crisis: can we trust what we RT?, in: SOMA, ACM, 2010, pp. 71–79.
[18] D. Laniado, P. Mika, Making sense of Twitter, in: ISWC, vol. 1, 2010, pp. 470–485.
[19] B. Gonçalves, N. Perra, A. Vespignani, Modeling users' activity on Twitter networks: validation of Dunbar's number, PLoS ONE 6 (8) (2011).
[20] S. Rosenthal, P. Nakov, S. Kiritchenko, S. Mohammad, A. Ritter, V. Stoyanov, SemEval-2015 task 10: sentiment analysis in Twitter, in: 9th International Workshop on Semantic Evaluation, Association for Computational Linguistics, 2015, pp. 451–463.
[21] P. Basile, N. Novielli, UNIBA: sentiment analysis of English tweets combining micro-blogging, lexicon and semantic features, in: 9th International Workshop on Semantic Evaluation (SemEval 2015), Association for Computational Linguistics, 2015, pp. 595–600.
[22] J. Bollen, H. Mao, A. Pepe, Modeling public mood and emotion: Twitter sentiment and socio-economic phenomena, in: ICWSM, 2011.
[23] A. Pak, P. Paroubek, Twitter as a corpus for sentiment analysis and opinion mining, in: Seventh Conference on International Language Resources and Evaluation, ELRA, 2010.
[24] L. Hong, B.D. Davison, Empirical study of topic modeling in Twitter, in: SOMA, ACM, 2010, pp. 80–88.
[25] D. Larson, 9 strange things about tweets, retweets and DMs every Twitter user must know, http://goo.gl/XyvAO, 2011, last checked December 18, 2015.
[26] G. Stringhini, C. Kruegel, G. Vigna, Detecting spammers on social networks, in: ACSAC, ACM, 2010, pp. 1–9.
[27] C. Yang, R. Harkreader, G. Gu, Empirical evaluation and new design for fighting evolving Twitter spammers, IEEE Trans. Inf. Forensics Secur. 8 (8) (2013) 1280–1293.
[28] S. Cresci, R. Di Pietro, M. Petrocchi, A. Spognardi, M. Tesconi, A criticism to society (as seen by Twitter analytics), in: 34th International Conference on Distributed Computing Systems Workshops, ICDCS 2014 Workshops (DASec), IEEE, 2014, pp. 194–200.
[29] S. Cresci, R. Di Pietro, M. Petrocchi, A. Spognardi, M. Tesconi, Fame for sale: efficient detection of fake Twitter followers, Decis. Support Syst. 80 (2015) 56–71.
[30] B. Viswanath, M.A. Bashir, M.B. Zafar, S. Bouget, S. Guha, K.P. Gummadi, A. Kate, A. Mislove, Strength in numbers: robust tamper detection in crowd computations, in: ACM on Conference on Online Social Networks, COSN '15, ACM, 2015, pp. 113–124.
[31] S. Cresci, R.D. Pietro, M. Petrocchi, A. Spognardi, M. Tesconi, DNA-inspired online behavioral modeling and its application to spambot detection, IEEE Intell. Syst. 31 (5) (2016) 58–64, http://dx.doi.org/10.1109/MIS.2016.29.
[32] P.W.L. Fong, M. Anwar, Z. Zhao, A privacy preservation model for Facebook-style social network systems, in: Proceedings Computer Security – ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21–23, 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 303–320.
[33] P.W. Fong, Relationship-based access control: protection model and policy language, in: Proceedings of the First ACM Conference on Data and Application Security and Privacy, CODASPY '11, ACM, 2011, pp. 191–202.
[34] R. Pardo, G. Schneider, A formal privacy policy framework for social networks, in: Proceedings Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1–5, 2014, Springer International Publishing, 2014, pp. 378–392.
[35] J. Seligman, F. Liu, P. Girard, Facebook and the epistemic logic of friendship, in: Proceedings of the 14th Conference on Theoretical Aspects of Rationality and Knowledge, TARK 2013, Chennai, India, January 7–9, 2013, 2013.
[36] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: a kernel language for agents interaction and mobility, IEEE Trans. Softw. Eng. 24 (5) (1998) 315–330.
[37] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: the SCEL language, ACM Trans. Auton. Adapt. Syst. 9 (2) (2014).
[38] S. Sebastio, A. Vandin, MultiVeStA: statistical model checking for discrete event simulators, in: ValueTools, ICST/ACM, 2013, pp. 310–315.
[39] K. Bae, J. Meseguer, Model checking linear temporal logic of rewriting formulas under localized fairness, Sci. Comput. Program. 99 (2015) 193–234.