RIVERtools: an IDE for RuntIme VERification of MASs, and Beyond

Angelo Ferrando*

DIBRIS, University of Genova, Italy
angelo.ferrando@dibris.unige.it

Abstract. This work introduces RIVERtools, an IDE supporting the use of the "trace expressions" formalism by users that want to perform runtime verification of their own system.

Keywords: RIVERtools, Trace expressions, Engineering Multiagent Systems, Runtime Verification (RV), Integrated Development Environment for RV

1 Introduction

During the development of software systems, it is not rare having the necessity to state that the implemented system meets our requirements. Generally this happens in all kind of software systems, but becomes more difficult when we increase the system complexity. In particular, if we focus on the development of Multiagent Systems (MASs), it is extremely common to have the need for checking the agents' behaviour in order to verify if what we expect *in theory* is also correctly achieved by our implementation *in practice*. When we work on small and centralized systems we have the chance to use standard formal approaches, such as Static Verification (for instance Model Checking). In the MAS community there are many ad-hoc model checking tools such as AJPF (Agent JavaPathFinder) [11], MCMAS (Model Checking MAS) [15], MABLE [17], just to cite some.

When the MAS becomes larger, model checking it (as well as the environment where it is immersed) becomes quickly intractable. In these scenarios, a valid alternative is Runtime Verification (RV). The main difference with respect to standard static verification is the stage when it is applied, which is *execution time*. In fact, in RV we do not need to *simulate* all possible paths that the system may generate during its execution, but we limit the analysis directly to the paths exposed and generated by the system during its *real* execution. As a consequence, RV could be more suitable and applicable than static verification in *black-box* scenarios, where there is no access to the source code of the system we want to verify.

Differently from static verification, very few tools expressly meant to runtime verification of MASs exist. Besides [1, 7, 10, 16] and a few others, the only works

* https://angeloferrando.github.io/website/assets/videos/RIVERtoolsDemo.mp4

on runtime verification of MASs are those that lead to the development and refinement of the trace expression formalism[1].

As it happened for tools such as Jason[2], Cartago[3], and Moise[4], where the need of a more modular, flexible and standardized framework brought to the creation of JaCaMo[5], also for the RV of MAS there is the pressing need of a general purpose framework, with the objective to be modular with respect to the target system (the system must be seen as a black-box) and "programmer-friendly" focusing on the reuse of the developed software. Following this aim we developed RIVERtools, an Integrated Development Environment (IDE) for supporting the automatic generation of code to be used to implement the black-box RV engine of a generic software systems - focusing on challenging scenarios where the target system is a MAS.

RIVERtools is proposed as a generic, modular and domain independent IDE to achieve the runtime verification of any target system. Rather than other solutions proposed in literature, RIVERtools tries to born as a "multi-target" IDE for the generation of monitors that can be used to achieve the runtime verification of different systems. Obviously we can achieve the same results without using an IDE, but using RIVERtools we have the great advantage to write our specifications once and for all (supported by highlights, syntax and type checking), leaving the technical details at a lower level that can be implemented just one time for each new target system of interest (updating the compiler without changing the high level specification).

The RIVERtools main features can be summarized as follows:

- support during the definition of our properties (the formalism presented in Section 2);
- abstraction from the domain specific target, with RIVERtools we can define our specification leaving all technical details to the IDE compiler (presented in Section 4);
- automatic integration of the properties inside the runtime verification process;
- extensibility since its structure is based on the presence of "connectors-to-something" (for each new target system to be verified, we define a new "connector-to-target" updating the RIVERtools compiler).

The main objective of RIVERtools is to separate and generalize the writing of monitors to achieve the runtime verification of any possible target system. In this paper we focus on its initial exploitation in the MAS context, in particular in the Jade framework.

First of all we have to present the formalism that we can use inside RIVERtools, in the next section we present briefly its syntax giving the semantics by intuition.

[1] https://www.google.it/search?q=runtime+verification+of+multiagent+systems
[2] http://jason.sourceforge.net/wp/index.php
[3] http://cartago.sourceforge.net
[4] http://moise.sourceforge.net
[5] http://jacamo.sourceforge.net

2 Trace Expressions

Trace expressions are a specification formalism expressly designed for RV; they are an evolution of multiparty global session types [4], initially proposed for RV of agent interactions in MASs [2, 3, 8, 9, 13]; and, more recently, extended with the notion of parameters to enhance their expressive power [6].

A trace expression τ represents a set of possibly infinite event traces, and is defined on top of the following operators:

1. ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ;
2. $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type[6] ϑ ($e \in \vartheta$), and the remaining part is a trace of τ;
3. $\tau_1{\cdot}\tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2;
4. $\tau_1 \wedge \tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2;
5. $\tau_1 \vee \tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2;
6. $\tau_1 | \tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces of τ_1 with the traces of τ_2;
7. $\vartheta \gg \tau$ (*filter*), a derived operator denoting the set of all traces contained in τ, when deprived of all events that do not match ϑ;
8. $<X;\tau>$ (*binder*), binds the free occurrences of X in τ; accordingly, the trace expression $\sigma\tau$ obtained from τ by substituting all free occurrences of $X \in dom(\sigma)$ in τ with $\sigma(X)$.

The semantics of trace expressions is specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. As it is usual, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$. If the trace expression τ_1 specifies the current valid state of the system, then an event e is considered valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 will specify the next valid state of the system after event e. Otherwise, the event e is not considered to be valid in the current state represented by τ_1.

Without loss of generality, in the rest of the paper we focus only on communicative events (message exchange).

3 Runtime Verification using Trace Expressions

We consider the SWI-Prolog[7] implementation of the trace expression formalism; the RV pipeline can be summarized as:

1. the implementation of the operational semantics modeled by δ in SWI-Prolog;
2. the definition of a trace expression in SWI-Prolog;

[6] To be more general, trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$.

[7] http://www.swi-prolog.org

3. the implementation of a library to allow the communication between SWI-Prolog and the target system we want to verify (in the following we will refer to it as the connector);
4. the writing of a "main" file which uses this library;
5. the execution of the "main" file to perform the runtime verification of the target system.

What is extremely important to note is that the components involved in the RV process are: a SWI-Prolog library (1), a trace expression (2), a library to integrate SWI-Prolog with the system (the connector) (3) and a file to start everything correctly (4). But among these, only the trace expression (2) needs to be defined each time by the programmer. In fact, when the target system is chosen, (1) and (3) can be implemented once and for all; and the "main" file (4) can be automatically generated starting from the trace expression (2). Following this intuition, we created an IDE which supports the automatic generation of as much components as possible among the needed ones, and that helps the programmer in correctly representing the trace expression that models the interaction protocol to be verified.

4 RIVERtools

RIVERtools has been developed using *Xtext*[8], a framework for development of programming languages and domain-specific languages which can be integrated as an Eclipse[9] plugin. With respect to other frameworks, Xtext has been chosen above all for its support to the *Xtend*[10] language (a dialect of Java).

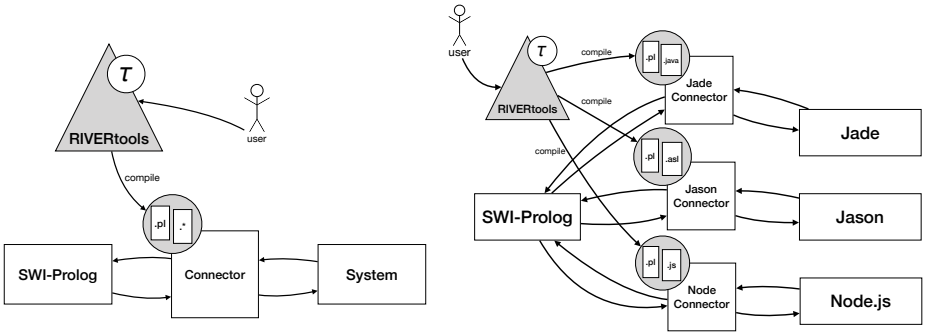

Fig. 1. RIVERtools general representation (left) and its exploitation in three different scenarios: Jade, Jason and Node.js (right).

[8] https://eclipse.org/Xtext/
[9] https://www.eclipse.org
[10] http://www.eclipse.org/xtend/

In Figure 1(left), a high level summarization of how the RIVERtools IDE works is presented. First of all, the programmer (the user) can write the trace expression inside RIVERtools. This phase is supported by:

- syntax checking,
- type checking (contractiveness, decentralization),
- roles, event types and events definition, and so on.

Referring to the second point, a trace expression is contractive if all its infinite paths contain the *prefix* ':' operator [5] and can be decentralized (for decentralized runtime verification purposes) only if it satisfies a set of good properties: *connectedness for sequence* and *unique point of choice* [14]. Even if we cannot go into the technical details for space constraints, both contractivennes and decentralizability checks are provided by RIVERtools.

Keeping in mind the summarization made in Section 3, we have a trace expression defined using RIVERtools, the SWI-Prolog library and the connector to the target system. From these, we need to create the SWI-Prolog representation of the trace expression to be used inside the SWI-Prolog library (where the trace expression operational semantics is implemented), and the "main" file to properly initialize and run the connector. These two files are automatically generated by RIVERtools by compiling the trace expression.

In more detail, the new RV pipeline using the trace expression formalism with the RIVERtools integration can be summarized as:

1. the user writes the trace expression inside RIVERtools and chooses the target system;
2. RIVERtools checks the correctness of the trace expression and, if it is correct, generates the SWI-Prolog representation of the trace expression and the "main" file for the target system;
3. the "main" file, using the SWI-Prolog library and the connector to the target system, can be used by the user to start the RV process.

Figure 1(right) shows the flexibility of the proposed approach. As an example, if the target system is Jade, RIVERtools compiles the trace expression τ in one Prolog file and one Java file: the second file is indeed dependent on the target system; for Jade it is a Java file, for Jason it would be an ASL file and for Node.js it would be a Javascript one.

4.1 The grammar

The trace expression structure that is recognized and managed by RIVERtools is expressed by the grammar below.

⟨*trace_ expression*⟩ ::= '`interaction_trace_expression`' '`{`'
'`id:`' ⟨*atom*⟩
'`target:`' ⟨*target*⟩
'`body:`' ⟨*term*⟩+

‘`roles:`’ ⟨*role*⟩*
‘`types:`’
(⟨*type*⟩ ‘`:`’ ‘`{`’ (⟨*role*⟩ ‘`=>`’ ⟨*role*⟩ ‘`:`’ ⟨*content*⟩ (‘`[`’ (⟨*condition*⟩)+ ‘`]`’)?)+ ‘`}`’
‘`[`’ ⟨*channel*⟩ ‘`]`’)*
(‘`threshold:`’ ⟨*reliability*⟩)?
(‘`channels:`’ (⟨*channel*⟩ (‘`[`’ ⟨*reliability*⟩ ‘`]`’)?)+)?
‘`}`’

We have seven different fields:

- *id:* is the name of the protocol;
- *target:* is the target system to verify;
- *body:* is the collection of terms representing the body of the protocol, it follows the trace expression syntax presented in Section 2;
- *roles:* is the set of roles involved in the event types of the protocol;
- *types:* is the set of event types used in body;
- *threshold:* (optional) is the minimum level for the reliability allowed by the protocol (it will be more cleare after);
- *channels:* (optional) is the set of channels available for the communication between the roles.

We can better understand the syntax through an example.

```
interaction_trace_expression {
  id: ping_pong
  target: jade
  body:
    pingPong <- ping : pong : pingPong
  roles:
    alice
    bob
  types:
    ping : { alice => bob : hello } [email]
    pong : { bob => alice : world } [sms]
  threshold: 0.7
  channels:
    email [0.8]
    sms
}
```

This is a simple representation of a ping-pong protocol. Since the target system is set to `jade` the roles involved are automatically considered as agents, in particular here we have two agents involved: `alice` and `bob`. The protocol identifier is `ping_pong`. The protocol we want to verify (the body) consists in one `ping` followed by one `pong` followed by one `ping` and so on infinitely. The event types involved in the protocol are explicitly defined: `ping` corresponds to the interaction event `hello` sent by `alice` to `bob` using the `email` channel and `pong` corresponds to the interaction event `world` sent by `bob` to `alice` using the `sms` channel. These two channels must be also defined with their respective reliabilities, for instance in this case we have that the `email` channel is more reliable than the `sms` channel. The `threshold` level for the channels is also defined.

The threshold field is used to set the level of reliability under which we do not trust a channel. When the reliability of a channel is under the threshold

RIVERtools automatically considers "optional" all the event types using this channel. For instance, if we have a channel `sms` with reliability `0.3`, the threshold set to `0.7`, and an event type `ping:{alice => bob : hello}[sms]`, each time we use `ping` inside the body field, *i.e.* `ping:t` (where `t` is the continuation of the trace expression after the prefix operator), RIVERtools automatically compiles it in `((ping:epsilon)\/(epsilon))*(t)` meaning that the event matching `ping` can be missing. If we set the threshold to a smaller value, for instance `0.2`, we are relaxing our reliability requirements and since the `sms` channel has a greater reliability value, it will not be modified by RIVERtools because we are trusting all the channels with reliability greater than `0.2`. In case of the limit scenario where the reliability of a channel is `0`, RIVERtools will remove from the protocol all the occurrences of the event types using it. This particular case becomes interesting when we focus on distribution features, because we can write a correct protocol in which we explicit all the event interactions but when we exploit it in a real world we discover that not all events are observable. This issue can bring to have missing information during the decentralization of our specifications. Thanks to the explicit representation of such absence of reliability, RIVERtools can analyze the trace expression considering these kind of unavailability and can achieve a correct decentralization without losing information (for more details about how to decentralize properly an agent interaction protocol defined using a trace expression see [12, 14]).

The channel integration inside the runtime verification process is a task for the connector and it is totally domain dependent.

When a channel is not indicated for an event type, the default one is chosen whose reliability is `1`. The same happens for the channels, if the reliability is not given, the reliability is set to `1` (in the `ping_pong` example the `sms` channel has reliability `1`).

Starting from the trace expression, RIVERtools generates its SWI-Prolog implementation and the system dependent "main" file (in this case a Java file since the target is Jade) to use the Jade-Connector and the SWI-Prolog library. The programmer will provide its own MAS implemented in Jade and using the "main" file it will be possible to perform RV on it. In more details, since Jade is a Java framework used to implement MASs, the Jade-Connector is a Java library containing both all the primitives to sniff the events generated by the agents developed by the programmer and the code necessary to use the SWI-Prolog library containing the trace expression semantics implementation. In this way, the only part that RIVERtools must generate is the SWI-Prolog implementation of the trace expression, all the rest is fixed and can be reused. It will be enough to run the "main" file (Java) providing:

- the agents developed by the programmer;
- the Jade-Connector library (in the build path);
- the SWI-Prolog library (used by the Jade-Connector).

5 Example

Let us suppose to have a MAS implemented in Jade representing a book-shop scenario. The involved agents are: `alice`, `barbara`, `carol`, `dave`, `emily` and `frank`. The protocol that we want to verify at runtime can be summarized in natural language as follow:

1. the agent `alice` sends a `whatsApp` message to the agent `barbara` asking to buy a book;
2. the agent `barbara` sends an `email` message to the agent `carol` asking to reserve the book in the bookshop;
3. the agent `carol` sends a `whatsApp` message to the agent `dave` asking to check the availability of the book;
4. the agent `dave` checks the availability of the book, and if the book is available
   (a) it sends a `whatsApp` message to the agent `emily` asking to take the book in the bookshop;
   (b) the agent `emily` sends an `email` message to the agent `barbara` saying that the book is available in the bookshop.

   otherwise
   (a) it sends an `email` message to the agent `frank` asking to order the book;
   (b) the agent `frank` sends a `whatsApp` message to the agent `barbara` saying that the book will be availabel in the bookshop in two days.

The corresponding trace expression representation inside RIVERtools is:

Listing 1.1. Book-shop trace expression written inside RIVERtools.

```
trace_expression {
  id: book_purchase
  target: jade
  body:
    purchaseBook <- buy : reserve : checkAvail :
                    (take2Shop : availNow : epsilon \/
                     order : avail2Days : epsilon)
  roles:
    alice, barbara, carol, dave, emily, frank
  types:
    buy : { alice => barbara : buy_me_book} [whatsApp]
    reserve : { barbara => carol : reserve_me_book } [email]
    checkAvail : { carol => dave : is_available? } [whatsApp]
    take2Shop : { dave => emily : send_me_book } [whatsApp]
    availNow : { emily => barbara : book_available } [email]
    order : { dave => frank : order_book } [email]
    avail2Days : { frank => barbara : book_in_2_days } [whatsApp]
  threshold: 0.6
  channels:
    email [0]
    whatsApp [1]
}
```

In this example we have two kinds of channels used by the agents to communicate: `email` and `whatsApp`. We point out the reliability of the two corresponding channels. The `whatsApp` channel has reliability set to `1`, while the `email` channel has the reliability set to `0`. This means that, during RV, we expect the monitor not to be able to observe messages passed on the `email` channel. This may be due

to any reason, for example lack of permissions or unavailability of information. Thanks to the presence of explicitly declared channels, RIVERtools can take into account the channel reliability during the correctness checking for the trace expression. Channel reliability may impact on contractiveness and decentralization of the original trace expression (as observed in Section 4.1): RIVERtools takes channel reliability into account when automatically performs those checks.

Compiling this trace expression, RIVERtools generates the two files `book_purchase.pl` and `BookPurchase.java`. Providing the connector library for Jade and the MAS implementation of the book-shop, it will be enough to execute the main method of the `BookPurchase` class to achieve the RV of a Jade MAS. Once obtained this Java class and the trace expression SWI-Prolog representation, the programmer can simply execute the verifier without adding a single line of code.

Listing 1.2. BookPurchase.java automatically generated by RIVERtools.

```
public class BookPurchase {
  public static void main(String[] args)
    throws StaleProxyException, IOException {

  /* Call at the SWI-Prolog library */
  JPLInitializer.init();

  /* Registration of the trace expression generated by RIVERtools */
  TraceExpression tExp = new TraceExpression("book_purchase.pl");

  /* Initialize JADE environment */
  jade.core.Runtime runtime = jade.core.Runtime.instance();
  Profile profile = new ProfileImpl();
  AgentContainer container = runtime.createMainContainer(profile);

  /* Create the custom defined agents (default are instances of Agent class) */
  List<AgentController> agents = new ArrayList<>();
  Agent alice = new Agent();
  AgentController aliceC = container.acceptNewAgent("alice", alice);
  agents.add(aliceC);

  // the same for barbara, carol, dave, emily and frank
  // ...

  /* Create a single centralized monitor verifying the trace expression tExp*/
  SnifferMonitorFactory.createAndRunCentralizedMonitor(tExp, container, agents);

  /* Channels creation (here are simulated)*/
  Channel.addChannel(new SimulatedChannel("email", 0));
  Channel.addChannel(new SimulatedChannel("whatsapp", 1));

  /* Run the agents */
  for(Agent agent : agents) {
    agent.start();
  }
}
```

5.1 Screenshots

In the following we reported screenshots showing some of the typical errors handled by RIVERtools.

The book-shop scenario in RIVERtools: In Figure 2 we have defined inside RIVERtools the book-shop trace expression presented in Section 5. Even though the trace expression reported is the same, we can note two new fields that have been omitted previously, `decentralized` and `partition`. These two fields are used by RIVERtools to generate `BookPurchase.java` that exploits a decentralized set of monitors rather than only one centralized (for more details about how the decentralized algorithm works see [14]).

Partition not valid: In Figure 3 we change the reliability of the `email` channel from `1` to `0`. Unfortunately, doing in this way we cannot distribute the runtime verification on each single role because we could lose information. This is handled by RIVERtools that communicates to the programmer that the current proposed partition is not allowed (because two critical points are not satisfied [14]).

Roles existence: In Figure 4 we remove a role from the `roles` set. Since this role is used inside the definition of two event types, RIVERtools informs the programmer about an existence problem, since it is not able to find the corresponding role inside the `roles` set.

Event types existence: In Figure 5 we remove an event type from the `types` field. After that, RIVERtools finds the corresponding use inside the terms consisting the protocol body, and it communicates to the programmer about the use of an event type undefined.

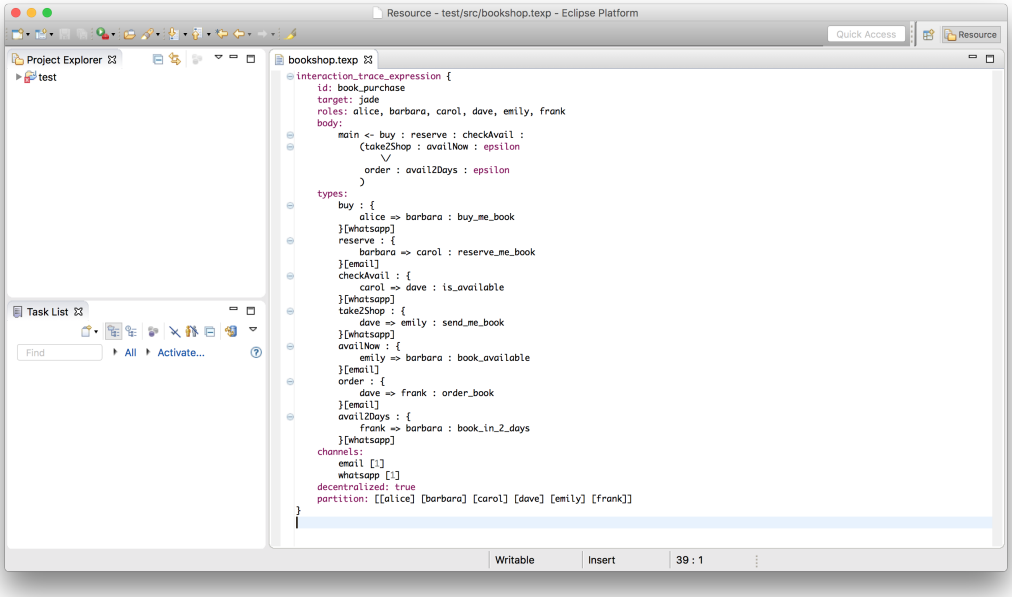

Fig. 2. The book-shop scenario presented in Section 5.

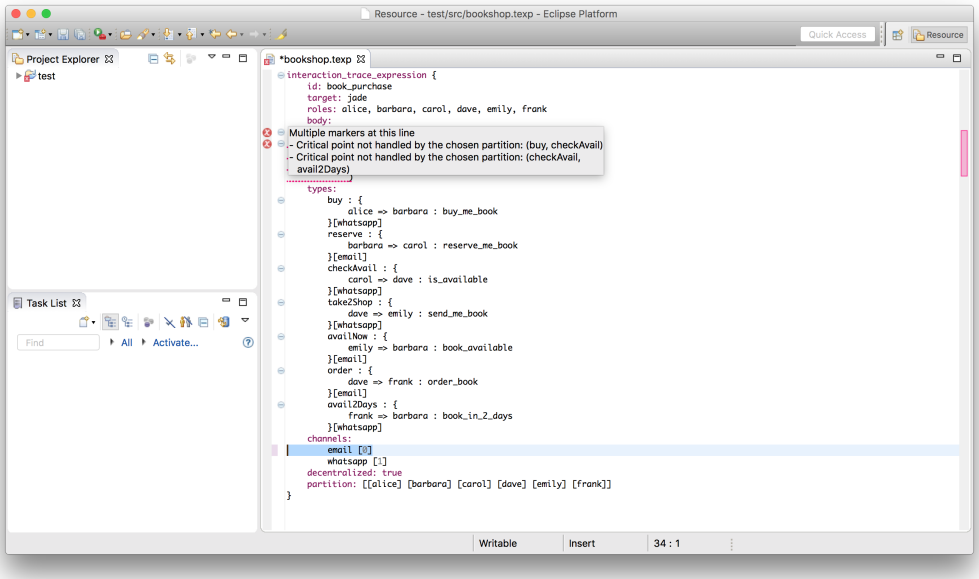

Fig. 3. Error: Partition not valid

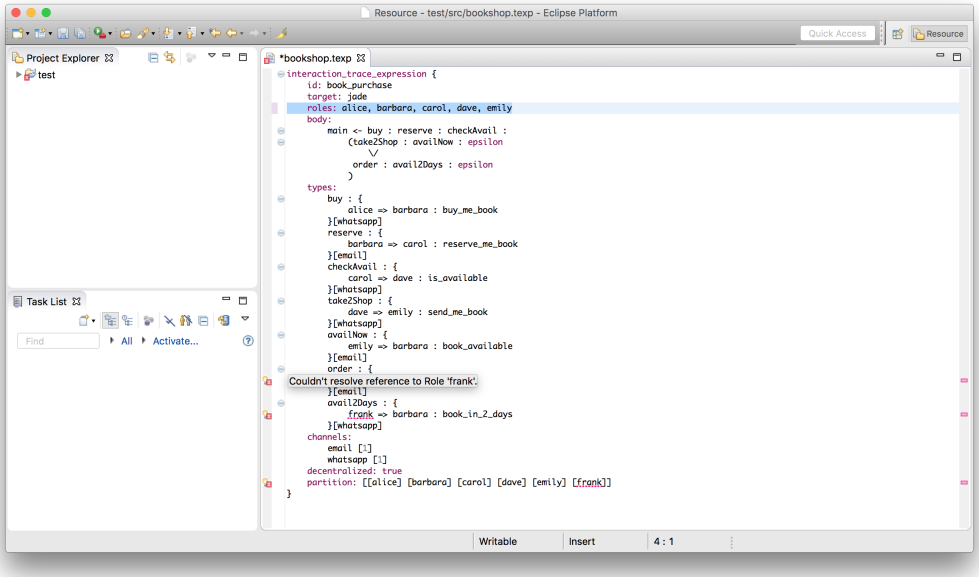

Fig. 4. Error: After having removed the role “frank”, we have an existence error.

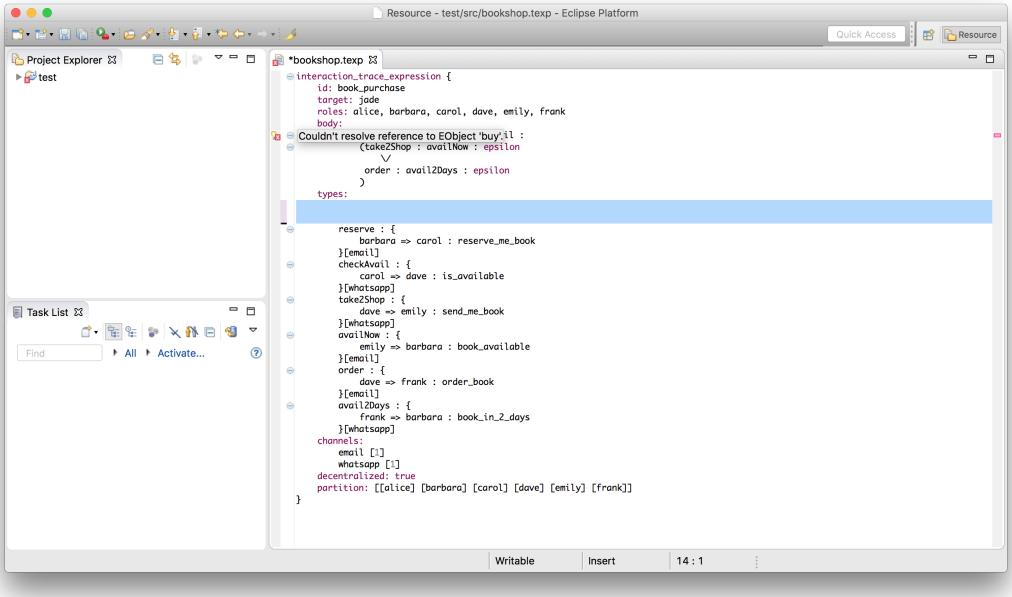

Fig. 5. Error: After having removed the event type buy, we have an existence error.

6 Conclusions and Future Work

In this demo paper we have presented RIVERtools, an IDE that can be used for the integration of the trace expression formalism with any possible target system. Thanks to the presence of connectors that handle all the domain dependent issues, we have showed that RIVERtools allows focusing on a more abstract level leaving all technical details to the connector implementation. In this demo paper we have presented the RIVERtools general structure and we have analyzed its possible use through an example involving a `book-shop` scenario developed as a MAS in Jade. Since RIVERtools is at the beginning, we have not already tested it on a real scenario, but it is our intention to deploy it in some challenging context where will be necessary to exploit the expressivity of our formalism.

The future directions of our work will include to implement connectors to other target systems. In the MAS context, we plan to add a connector to Jason. Outside the MAS community, we will explore the possibility of an integration with some general purpose system, as Node.js[11], which has already been used in fascinating scenarios like the Internet of Things (IoT).

[11] https://nodejs.org/

References

1. H. Alotaibi and H. Zedan. Runtime verification of safety properties in multi-agents systems. In *Proc. of ISDA 2010*, pages 356–362. IEEE, 2010.
2. D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global protocols as first class entities for self-adaptive agents. In *Proc. of AAMAS 2015*, pages 1019–1029. ACM, 2015.
3. D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach. *Intelligenza Artificiale*, 9(2):131–148, 2015.
4. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *Proc. of DALT 2012*, volume 7784 of *LNAI*, pages 76–95. Springer, 2012.
5. D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In *Theory and Practice of Formal Methods*, volume 9660 of *LNCS*, pages 47–64. Springer, 2016.
6. D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In S. Das, E. Durfee, K. Larson, and M. Winikoff, editors, *Proc. of AAMAS 2017*. IFAAMAS, 2017.
7. N. A. Bakar and A. Selamat. Runtime verification of multi-agent systems interaction quality. In *Proc. of ACIIDS 2013*, pages 435–444, 2013.
8. D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE and Jason multiagent systems with Prolog. In L. Giordano, V. Gliozzi, and G. L. Pozzato, editors, *Proc. of CILC 2014*, volume 1195 of *CEUR Workshop Proceedings*, pages 319–323. CEUR-WS.org, 2014.
9. D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE multiagent systems. In D. Camacho, L. Braubach, S. Venticinque, and C. Badica, editors, *Proc. of IDC 2014*, volume 570 of *Studies in Computational Intelligence*, pages 81–91. Springer, 2014.
10. F. Chesani, P. Mello, M. Montali, and P. Torroni. Commitment tracking via the reactive event calculus. In *Proc. of IJCAI'09*, pages 91–96. Morgan Kaufmann Publishers Inc., 2009.
11. L. Dennis, M. Fisher, M. Webster, and R. Bordini. Model checking agent programming languages. *Automated Software Engineering*, pages 1–59, 2011. 10.1007/s10515-011-0088-x.
12. A. Ferrando. Automatic partitions extraction to distribute the runtime verification of a global specification. In *Proceedings of the Doctoral Consortium of AI*IA 2016 co-located with the 15th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2016), Genova, Italy, November 29, 2016.*, pages 40–45, 2016.
13. A. Ferrando, D. Ancona, and V. Mascardi. Monitoring patients with hypoglycemia using self-adaptive protocol-driven agents: A case study. In M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern, editors, *Proc. of EMAS 2016, Revised, Selected, and Invited Papers*, volume 10093 of *LNCS*, pages 39–58. Springer, 2016.
14. A. Ferrando, D. Ancona, and V. Mascardi. Decentralizing MAS monitoring with DecAMon. In *Proc. of AAMAS 2017*, pages 239–248, 2017.
15. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *STTT*, 19(1):9–30, 2017.
16. D. Meron and B. Mermet. A tool architecture to verify properties of multiagent system at runtime. In *Proc. of PROMAS*, volume 4411 of *LNCS*, pages 201–216. Springer, 2006.

17. M. Wooldridge, M. Huget, M. Fisher, and S. Parsons. Model checking for multi-agent systems: the Mable language and its applications. *International Journal on Artificial Intelligence Tools*, 15(2):195–226, 2006.