Development Tools for Rule-Based Coordination Programming in LINC

Maxime Louvel[1(✉)], François Pacull[2], Eric Rutten[3], and Adja Ndeye Sylla[1]

[1] Univ Grenoble Alpes, CEA, Leti, 38000 Grenoble, France
{maxime.louvel,AdjaNdeye.Sylla}@cea.fr
[2] Bag-Era, Saint-Martin, France
francois.pacull@bag-era.fr
[3] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
Eric.Rutten@inria.fr

Abstract. During the last decades a lot of coordination models and languages have been proposed in the literature. These approaches have proven that they can greatly improve the development of distributed applications that are now common. However, to be used by many developers, there is still a gap regarding the available tools.

This paper details a set of tools that have been built to develop applications in LINC, a coordination environment rooted in Linda tuple spaces and Gamma chemical machine approaches. These tools allow developers to design better coordination rules, to monitor and update a running distributed application. The tools proposed here include design and debugging tools.

Keywords: Coordination environment · Development tools · Distributed systems

1 Introduction

Coordination models and languages [31] have been around for decades [28]. Initial works on tuple spaces with Linda [17] have paved the way to space and time decoupling between data producer and data consumer through a very simple API. The Gamma approach [3] brought the chemistry inspired programming to move away from sequential programming. Since these early works, many proposals have been made to improve them. An important trend has been to use a distributed tuple space [5,16,22,25,27,30]. Another interesting trend has been to extend the chemistry based programming to design self adaptive systems [32,37]. Coordination models have been used in many areas including mobile computing [8,27], context aware applications [19], cyber-physical systems, wireless sensor networks [10] or scientific applications [15]. Several works have tried to formalise the coordination aspects to bridge the gap with the formal methods [2,12].

One of the main challenges that is left open for the adoption of coordination paradigms by software engineers is the lack of tools [28]. There is a need

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
J.-M. Jacquet and M. Massink (Eds.): COORDINATION 2017, LNCS 10319, pp. 78–96, 2017.
DOI: 10.1007/978-3-319-59746-1_5

for development tools that better capture the coordination models instead of relying only on the understanding of software engineers. There is also a need for debugging tools and tools to monitor, introspect and update applications without restarting them.

This paper presents the development tools of the coordination environment LINC [23]. LINC uses distributed tuple spaces, a rule based language and transactional reactions implementing the chemical machine. LINC is used for distributed applications, possibly large scale, that may include cyber-physical systems or the Internet of Things (IoT). This paper details the tools available to developers of LINC applications. This includes rule generation tools as well as analysis and debugging tools. The former generate correct rules based on domain specific or formal models. The latter offer run time monitoring, run time debugging and analysis of the coordination rules executions. To the best of the authors knowledge there exist no equivalent set of development tools for tuple space programming environment. This paper introduces several tools tailored to LINC but that can be of interest, at least partially, to other coordination environments, under some assumptions.

This paper is organised as follows. Section 2 briefly introduces the LINC model. Section 3 details the modelling and analysis tools that are available to developers of LINC applications. Then Sect. 4 presents a short summary of several applications built with LINC and these tools. Section 5 presents related works. Finally, Sect. 6 concludes the paper and presents future works.

2 LINC Model

To make this paper self-contained, this section presents an overview of the coordination environment LINC. More details on LINC can be found in [23]. Technical information can also be found on the LINC wiki[1].

2.1 Tuple Space Implementation

Bags: LINC uses a distributed tuple space [7] implemented as a set of bags. In a bag all the tuples have the same number of fields. Bags are accessed through three operations:

- `rd()`: takes a partially instantiated tuple as input parameter and returns a stream of fully instantiated tuples matching the input pattern;
- `put()`: takes a fully instantiated tuple as input parameter and inserts it in the bag;
- `get()`: takes a fully instantiated tuple as input parameter, verifies if a matching tuple exists in the bag and consumes it.

[1] http://linc.middlewares.info/.

Objects: LINC objects, or simply objects, are the deployment units of LINC. An object contains one or more bags. An object has a type that defines its bags and its internal implementation. For instance a Sensor object has a Sensor bag (with the last sensors' values) and a thread that periodically polls the sensors and puts their values in the Sensor bag. Objects' type can be inherited to include new bags or to modify the object implementation. Bags are grouped within objects according to application logic. For instance, all the bags managing a set of devices that communicate with the same communication technology can be grouped in the same object. Any object may execute coordination rules that manipulate tuples in its own bags or in the bags of any other objects. From a functional point of view, it makes no difference which object manipulates the tuples of a bag. However from a run-time performance point of view, this may have a strong impact on CPU, memory or network resources.

2.2 Coordination Rules

The three operations `rd()`, `get()` and `put()` are used within production rules [9]. A production rule is composed of a precondition phase and a performance phase.

The precondition phase is a sequence of `rd()` operations which detect or wait for the presence of tuples in several given bags. The tuples are, for instance, values from sensors, external events, or results of service calls. The output fields of a `rd()` operation can be used to define input fields of subsequent `rd()` operations. The precondition builds an inference tree with right propagation. A `rd()` is blocked until at least one tuple corresponding to the input pattern is available. In addition to read in the tuple spaces it is possible to call `ASSERT` and `COMPUTE` functions. Both can execute any Python code. `ASSERT` functions return a boolean value that stops the inference tree if false and triggers the subsequent token (`rd`, `ASSERT` or `COMPUTE`) if true. `COMPUTE` functions can take as input any previously instantiated variables; they return a tuple of variables. These variables can be used by subsequent tokens or in the performance phase. When the last token of the precondition is reached, the performance is triggered.

The performance phase combines the three `rd()`, `get()` and `put()` operations to respectively verify that some tuples are present (e.g. the one(s) found in the precondition phase), consume some tuples, and insert new tuples. In this phase, the operations are embedded in one or multiple distributed transactions [4], executed in sequence. Each transaction contains a set of operations that are performed in an atomic manner. Hence, LINC guarantees that for operations belonging to the same transaction, either all are executed successfully or none.

A LINC rule example is given in Listing 1.1. The precondition (before the symbol `::`) first checks that "$c1$" is "*true*" (`rd` at line 1 on bag `Condition` belonging to object `O2`). Then it waits for events (line 2). Then it reads the parameters associated to the event (`rd` in line 3 with `evt` variable instantiated with the value return by the previous `rd`). For every branch of the inference tree reaching

the last `rd`, a performance is triggered. This performance contains 1 transaction (between curly brackets). It first checks that the condition is still valid (i.e. ("*c*1", "*true*") is in the bag `Condition`), then it consumes the event (line 6 with variables instantiated in the precondition), then it adds this event in the bag `Stats` of the object `O1` (line 7) and puts the command "*cmd*1" with the parameters $p1$ and $p2$ in the bag `Cmd` of the object `O3` (line 8). Note that the performance does not read again the `Param` bag, because the rule assumes that this value does not change.

```
["O2", "Condition"].rd("c1", "true")&                                    1
["O1", "Event"].rd(evt, time)&
["O3", "Params"].rd(evt, p1, p2)&                                        3
::
{ ["O2", "Condition"].rd("c1", "true")&                                  5
  ["O1", "Event"].get("evt1", p1, p2)&
  ["O1", "Stats"].put("evt1", time, p1, p2)&                             7
  ["O3", "Cmd"].put("cmd1", p1, p2)& }.
```

Listing 1.1. Example of LINC rule

Linda - LINC: `read` and `out` operations of Linda are respectively implemented in LINC by `rd` in precondition and `put` in performance. The `in` operation of Linda takes partially instantiated tuples and removes the matching tuples from the tuple space. If no tuple matches, the `in` is blocked. In LINC, this is implemented by a `rd` in the precondition and a `get` in the performance. In the precondition, the `rd` uses a partially instantiated tuple and blocks when no tuple matches. A performance is triggered for every matching tuple. In each performance, a `get` is done with the fully instantiated tuples created by the precondition `rd`.

3 Coordination Rules Development and Debugging

Being based on the tuple spaces paradigm LINC has a very small API (rd, get and put). Most of the complexity thus resides in how to use them best, or in other words how to write coordination rules. Two sets of tools are now presented, first to generate rules and second to analyse and debug them.

3.1 Rules Generation Tools

Two kinds of tools are available to developers. The first one consists in building domain specific tools, the second one relies on formal languages to ensure the correctness of the system. In both cases, rules are automatically validated and generated from the models built by the developers. Hence the developers can focus on the application logic and not the writing of rules. This paper presents a synthesis of several tools, how they are related and how they can help developers. Details on each tools can be found in papers cited in this section.

Domain Specific Tools. One solution to help developers is to provide them a domain specific design tool. Such tool limits the expressiveness offered by the coordination language to a valid subset. In addition, the tool uses knowledge on the domain to build advanced validation of the rules written by developers. The latter need only to be experts of their domains and not experts in coordination models or distributed systems.

An example of such tool is the Coordination Scheme Editor (CSE) [24] to design scenarios in building automation systems. CSE provides a drag an drop interface to design rules for sensors and actuators. The precondition is a set of rd on sensors and thresholds or ranges checks. The performance is a set of action on actuators. Examples of verification done when generating rules are: valid ranges in sensors, valid types of commands sent to actuators or invalid thresholds.

High-Level Reactive Language Support for Behavioural Specification

Motivation: As explained above, LINC provides applications with transactional guarantees that ensure their reliability at run-time. But, although LINC ensures the consistent execution of individual rules, it does not prevent from writing erroneous rules. Individual rules can present bugs, as in all programming languages, but more importantly, because more difficult to detect and avoid, the set of rules may contain design errors such as conflicts between rules, and violations of applicative constraints that can bring the system in unsafe or undesirable states. More precisely, a rule, written to achieve a given objective, can violate one or several other objectives of other rules. An example of conflict is between rule A which opens the window to decrease the CO2 level in the room and rule B which closes the window and turns on the AC to decrease the room temperature. Moreover, the execution of a sequence of rules can bring the system in an undesirable state or into an undesired circularity or endless loop of rules having the effect of firing each other.

Therefore, when writing rules one has to control their resulting behaviours. Developers have to manually avoid conflicts and violations of constraints by adding the testing of numerous additional conditions on specific rules, and adding new rules. For large applications, programmed by large sets of rules, manually controlling the rules is a difficult combinatorial problem and is error prone.

Our Approach: To tackle this problem, we adopt the direction of high-level language support for behavioural specification. We consider high-level languages from the domain of reactive systems, related to transition systems, because they can be equipped with formal verification or synthesis tools, based on models for the behaviours of programs. On the other hand, we want them to be implemented in such a way as to be concretely executable in the LINC environment. We propose to combine the two reactive programming approaches in order to support the safe design and execution of control systems, in the application domain of smart environments. As illustrated in Fig. 1, data is gathered from the controlled environment through the tuple spaces. A transition system takes

the correct decision and produces the commands executed by LINC transactions. LINC transactions update both the actual system and the state of the transition system. Hence if an action on the actual system fails (e.g. due to a communication error or a hardware failure), the state of the transition system is not updated and stays consistent with the actual system. For instance if a window failed to open due to a communication error, the transition system will not wrongly assume that it has been opened.

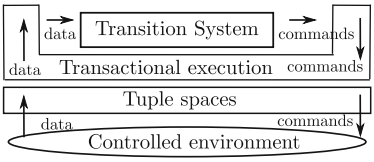


Fig. 1.

One approach, presented in [34], first models the application logic using coloured Petri nets. Then, it verifies the designed model using a model-checker [33], and finally, it generates the corresponding LINC rules. This approach enables the verification of several properties (e.g. absence of conflicts and violations of constraints) before execution. When generating LINC rules from the coloured Petri nets specification, the state of the system is embedded in a LINC bag. The transition and update of the state is performed in a transaction together with the actions on the system. However, modelling the application logic also requires to manually specify how to avoid conflicts and violations of constraints, in the model. Moreover, when a target property is not satisfied, the model has to be manually modified and verified again.

To overcome these limitations, we explore the use of a high-level language based on transition systems to access another formal technique: Discrete Controller Synthesis (DCS) [35]. Similarly to model-checking, DCS is an algorithmic technique that performs an exploration of the reachable state-space of a transition system. However, whereas model-checking verifies the satisfaction of a property in a programmed solution, DCS produces a solution from a more declarative specification. This specification contains the possible behaviours and properties to be enforced by control. More precisely, DCS takes (i) a transition system modelling possible behaviours of a system, (ii) a partition of the set of variables between controllable and uncontrollable ones, and (iii) a property to be enforced (e.g. invariance of a predicate). Given this, DCS synthesises, if it exists, the constraint on controllable variables in function of current values of state and uncontrollable variables, so that the resulting constrained transition system satisfies the property. This constitutes a controller that will avoid transitions leading to incorrect states (i.e. violating the property), or to correct states from which a sequence of uncontrollable conditions and transitions could lead to an incorrect state. Concretely, LINC has been combined with the automata-based

language Heptagon/BZR [11]. Heptagon/BZR enables the verification and the control of the system behaviour at compilation time. The generated executable code is called from the execution of LINC rules. A related approach was used in the context of ECA rules [6], different from tuple spaces, but with a similar goal as a mean to coordinate the execution of rules so as to avoid inconsistencies, circularities, and environment and application related constraints.

This way designers are provided for a support for both design-time and run-time reliability. At design time the developers have language support and DCS to generate a control function that can avoid conflicts, circularities and violations of constraints. The generated function is then reliably executed using LINC which enforces, through distributed transactions, the consistency between the states of the automata and the states of the actual system. Advantages of our reactive language-based approach are as follows:

- high-level language support for controller specification, less tedious and error-prone than at lower level;
- correctness of the controller, w.r.t. specified properties, hard to enforce manually, obtained by using formal methods and algorithms like model checking;
- automated synthesis of controllers, using DCS with declarative objectives, which is more constructive than classical verification of a hard to write solution; our approach also ensures that developers need only to be expert of their domain and not expert in formal models;
- automated generation of executable code compatible with LINC.

Extension to Other Coordination Languages. The tools presented in this section generate LINC rules. Generating code or rules for other tuple spaces based coordination languages is possible under certain conditions. Most of the tuple spaces based languages will allow a level of abstraction similar to LINC. However the tools presented above assume the following:

- the consistency between the state of the actual system and the state of the transition system will be ensured only if a distributed consensus mechanism (transactions in LINC) is available;
- a reactive environment that calls the transition system when necessary;
- LINC combines tuple spaces, production rules and transactions. This allows to only consider the important states. For instance, consider a rule consuming events A and B and producing C and D. The state where only A is consumed or only C is produced is not possible in LINC. This simplifies the generation of LINC rules and might not be directly available in other environments.

3.2 Analysis and Debugging Tools

The tools developed for rules generation are naturally not meant to completely replace the hand writing of rules, but as a complement. The domain specific tools provide a solution limited to a domain and a subset of the coordination model.

The tools based on formal models help to ensure the correctness of the coordination in the system. They can avoid conflict between rules, within the parts of the systems modelled in the high-level languages. Moreover, advanced features of LINC such as guards, alternative transactions, graceful degradation (see [23] for more details) are not always available to developers. Hence a set of tools are now described to monitor, analyse and debug applications. Note that these tools can be used in combination with the modelling tools. Indeed, the latter produce correct rules according to the assumptions made in the formal model or the domain hypothesis. However, if an error occurs outside these assumptions, the rules generated from the modelling tool will behave incorrectly. Therefore rules have to be validated by analysis and debugging tools. This section now presents tools to monitor, modify and analyse the execution of an application.

Monitor. When developing and debugging a distributed application, it is not always easy to access the information of all the elements. Indeed they are spread out, possibly in different networks. Furthermore, it is possible that they move (e.g. due to load balancing, system update or failure). Hence the developer has to keep track of what is where and how to access it. Then another challenge is to know what to observe and how. In tuple spaces based models, the important information is stored in the tuple spaces. It is thus fundamental to be able to access them easily.

To answer these challenges, every object in LINC provides a web interface to monitor its content and access the monitor interface of the other objects in the application. The interface can be accessed from any web browser. It allows to observe the content of any bags of any objects. This is particularly useful when a rule is blocked. The developer can connect to the monitor and check if the required tuples are there or not. In addition, the monitor interface allows to modify the content of the bags of the object. Hence a developer can unlock an application by adding a missing tuple. At the development stage, this can help to move forward if a bug occurred. At the production stage, this allows the maintenance to unlock the application while tracking the cause of the missing tuple. Rules in LINC are also modelled as tuples. Hence it is possible, from the monitor, to start, stop or update rules or group of rules by adding or removing the corresponding tuples.

Debugging and Information Traces. In a distributed application it might be a daunting task to gather all the traces generated in one application. To help developers, a tracer is built for every LINC object. Instead of using print on various files, the developer simply calls the tracer to add a trace. The tracer contains all the traces that might be useful for the application (e.g. communication error, debug traces or traces defined by developers when writing new LINC objects). For instance, when a developer creates an object to support a new sensor protocol he/she will use traces to debug his/her own code. The traces are saved to disk. The trace files are accessible through the Monitor interface of each object; they can also be accessed offline, e.g. by downloading them in a

development computer. Trace files are generated incrementally. Hence they can be periodically archived, possibly remotely, if needed.

Traces may have an important cost, in CPU and disk usage. To limit this, several levels of traces are available, from −1 (error) to 3 (debug information). The level of trace is set when starting an object. Then every trace with a level higher than the trace level are ignored. This is actually very efficient because the actual trace string is passed via a dictionary and is not evaluated if the trace is ignored. This mechanism allows to leave traces in the code and activate them only when necessary. Traces level can be dynamically changed. For instance the level can be increased when an error occurred. If the application behaviour is periodic or event based, the next loop or event handling will be traced with a higher level. Once the problem is understood, the trace level can be decreased again.

In the monitor, the traces are presented in a HTML table, sorted by date. The last line contains the most recent trace. Inside the LINC object, the tracer can either be used directly or cloned. All the clones of the tracer share the same file. Two filters criteria are provided to find specific traces in the table. The *tracer clone* displays traces of the selected clones only. The *Level* displays traces with a level lower or equal to the filter. The tracer clones allow to filter on functionality (e.g. communication traces or traces related to specific actions). Both filters can be combined to display the traces of one or more clones which are lower than a given level. It is then possible to search within the traces. This creates a new filter that can be combined with the clone or level filters. A second level of search is provided to filter again the traces.

Exploring Coordination Rules Executions. A LINC application is composed of a set of rules defining the behaviour of the application. When executing these rules, the LINC objects build inference trees (precondition) and execute performances. All the preconditions and performances are logged in several log files. Logs are organised by rule: for each rule there is one log. These log files are then explored to help the developer understand the behaviour of its application. The logs are dedicated to rules executions and capture the application behaviour. They are complementary to traces.

Logger Interface. For each rule, the inference tree built can be walked through a web interface. Three types of elements are presented for the precondition:

- `rd`: contains the precondition `rd()` executed (e.g. `rd("evt-a", ts)` to read all the events `"evt-a"`;
- result of `rd`, `COMPUTE`: contains the tuples returned (e.g. `rd("evt-a", "X32")`;
- result of `ASSERT`: successful calls are shown as a specific tuple, failed calls are presented in red because they stop the inference tree.

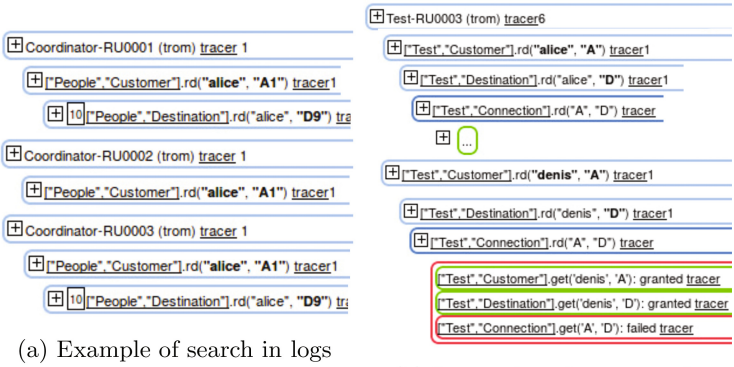
When a performance is reached, successful transactions are displayed in green without details. If the transaction failed, it is automatically expended to see

which operation caused the failure (displayed in red). Hence the developer can understand why this performance did not execute.

The logger may help the developer to see where an execution is blocked. Typically, when a `rd` has not returned matching tuples. From the logger interface, the developer can be redirected to the monitor of the object and inspect the bags' content. From there the developer can see for instance that the error comes from:

- an erroneous spelling and add the missing tuple to continue the execution;
- a missing tuple, meaning that another transaction (in the same rule or in another rule) consumed the tuple or that it has never been produced.

Filters. Looking at all the inference tree might not be possible when its size increases. However, most of the time the developer is only interested in some particular tuples. Hence, the logger interface allows developer to search the logs of interest. The search can be done in object, bag and/or tuple name. The search can be done on one or more rules. If matching logs are found, they are displayed in the interface. The developer can then browse them. For instance if it filters on tuples matching `"evt-a"` only the actions including matching tuples will be displayed (branches of the inference tree and transactions). If a rule handles all the events (`rd(evt, ts)`), only the interesting ones (i.e. handling `evt-a`) will be displayed. Figure 2a shows an example of log search on tuples containing `"alice"`. This example contains only preconditions for this tuple, i.e. no performance with a tuple containing `"alice"` has been triggered. The problem here can be that a tuple is not added as expected by the rule.



(a) Example of search in logs

(b) Logger interface example

Fig. 2. Logger interface and search (Color figure online)

Logger Interface Example. To illustrate the use of the logger interface, let's consider an application which models people travelling with a metro. This application has one object (Test) containing three bags. The bag `Customer` for customer locations (containing the tuples `("alice", "A")`, `("denis", "A")`); the

bag `Arrival` for customer destinations (containing `("alice", "D")`, `("denis", "D")`) and the bag `Connection` for metro connections (containing `("A", "D")`). The precondition of the rule Listing 1.2 reads the customer location and arrival of each person and checks that a connection exists between the two stations. The performance then gets the customer, its destination, the connection and puts the customer in its arrival station. After executing the rule, the developer notices that only alice reached its destination, denis is still in its initial location.

```
["Test","Customer"].rd(name, departure) &
["Test","Destination"].rd(name, arrival) &                    2
["Test","Connection"].rd(departure, arrival)&
::                                                            4
{ ["Test","Customer"].get(name, departure) ;
  ["Test","Destination"].get(name, arrival) ;                 6
  ["Test","Connection"].get(departure, arrival) ;
  ["Test","Customer"].put(name, arrival) ; }.                 8
```

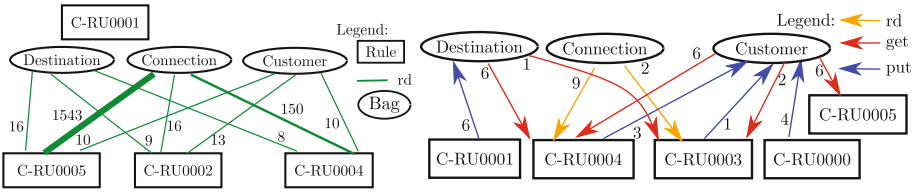
Listing 1.2. Travel LINC rule

Figure 2b shows the log of the rule. The rule is executed twice (one starting with the tuple `("alice","A")` and the other starting with the tuple `("denis","A")`). In this example, the two people make the same journey, and the same connection tuple is used. The first transaction that executes succeeds and gets the tuple `("A","D")` which makes the second transaction fail because the tuple seen in the precondition is not there anymore. In Fig. 2b, the third action of the transaction (line 7 of Listing 1.2) for the second transaction (`get("A","D")`) has failed which is shown in the log displayed. The solution here is to use a `rd()` instead of a `get()` on the connection to avoid removing the tuple.

Current Inference Tree. The Logger interface contains all the coordination actions done since the application started. To decrease the resources used (CPU, memory), LINC objects periodically execute a garbage collector to remove unnecessary branches in their inference tree. The garbage collection consists in walking the tree and checking that the tuples seen are still in the bags. If not, this means the performances that will be triggered by this branch will fail. To avoid this, the branch is removed from the tree. It thus may be useful to know the current inference tree of a LINC object. For that, an interface is provided that displays the current tree in a textual form. This interface can help to understand why the garbage collection has not been performed, for instance because the same tuple is always produced (e.g. `("evt-a")`). Here a solution is to use different tuples to model different events (e.g. `("evt-a", "XX...")`).

Analysing Coordination Rules Executions. The logger interface allows to understand what has been done by the rule. This can solve functional errors or highlight some performances issues. However it is not easy to draw high level conclusion by looking at all the coordination actions. For that, two analysis tools are provided; the first one focuses on data flow observations; the second one focuses on a global view of the logs.

Data Flow. This tool provides a data flow view of the coordination actions done. The information comes from the log files but is presented differently. This tool gives a picture of the rules' executions to link the data (tuples contained in bags) to the control flow (coordination actions done by the LINC objects). Two kinds of pictures are built, one for the preconditions and one for the performances. Figure 3a and b respectively show the preconditions and the performances for the execution of the rule in Listing 1.2. These pictures show the data flow between rules and bags (rd, get and put). The size of the line is proportional (logarithmic) to the number of actions done. The number of actions is also added in small on the line. By comparing the two pictures, one can see that there are more preconditions operations ($\sum_{prec\ op} = 1775$) than performances operations ($\sum_{perf\ op} = 46$). This means lots of preconditions never reach the performance phase, in other words big inference trees are built for nothing.



(a) Rules analysis - all preconditions

(b) Rules analysis - all performances

Fig. 3. Rules analysis

It is also possible to build a bag centred picture. This time only one bag is displayed with all the rules accessing this bag. This is particularly useful to find out which rule gets or puts tuples in a bag. For all analysis, it is possible to display only the successful performances, only the failed performances or both. Figure 4a is centred on the bag Customer. Here one can see that rule RU0001 is missing (compare to Fig. 3b) and that RU0002 does not access the bag. RU0002 is shown in the picture because it is connected to the Customer bag in the precondition. Figure 4b shows only the successful performances. Here only RU0003 and RU0004 update tuples in the bag Customer. Finally, Fig. 4c shows a picture of a rule that only copies data from a bag to another one.

All representations are also available in a table format with numbers instead of arrows of different size. Both views can be interesting depending on the type of patterns to observe, the number of bags, rules or tuples.

Error prone patterns that may be found with this tool are typically:

- overloading bags (a bag where more `put` than `get` are done);
- unused bags;
- performances which fail most of the time (meaning a lot of coordination is done just to fail afterwards).

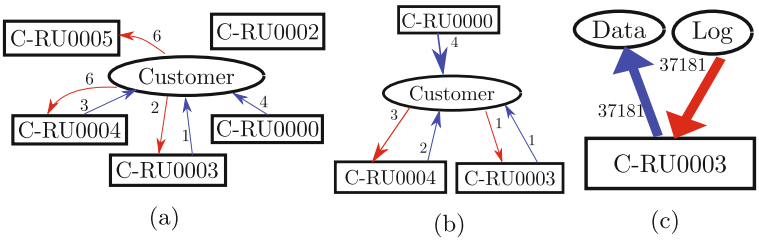


Fig. 4. Rules analysis - bag centered

This tool might also be useful to find or check some patterns such as a producer/consumer relationship between two or several rules. Hence better choices can be made for the deployment decision. Indeed, nothing prevents a rule to access only bags physically far from the object executing the rule. However, from a run-time performance point of view this may have a strong impact.

Global View on the Logs. The second tool for log analysis provides a global view of the logs. For that, the tool relies on the D3JS javascript framework[2]. This framework allows to easily build data driven document and to display them in a web browser. Several interesting points may be analysed with this tool. For instance, it is possible to view the size of the inference tree, if all the branches of the tree lead to a performance, or the ratio of successful performances over the number of performances tried. This tool can help developers to understand how the rules they wrote perform in the system. The goal is to optimise the rules in order to decrease the CPU and memory used by the LINC object executing them. Such optimisation may be achieved by reducing the size of the inference tree, the number of failing performances or the number of branches that lead to no performance.

Figure 5 shows the overview of three rules with a radial tree display[3]. At the centre is found the root node of the inference tree. The nodes of the inference tree leading nowhere are displayed in grey, the nodes leading to successful (resp. failing) performances are displayed in green (resp. red) and the nodes leading to both successful and failing performances are displayed in orange. Failing `ASSERT` in the precondition are displayed as failing performances. It is possible to click on each node to have the information of the action done (e.g. `rd("evt-a")`, `put("action-b")`). Figure 5 contains examples of log analysis for several rules. Here we can see that rule Fig. 5a is more efficient with a smaller inference tree and more green nodes than rule Fig. 5b which does a lot of work for nothing. Rule Fig. 5c shows a rule that simply copies data from a bag to another one; hence it is mainly green. The few orange points are failing `ASSERT` for tuples that should not be processed by this rule.

[2] https://d3js.org/.
[3] https://bl.ocks.org/mbostock/4063550.

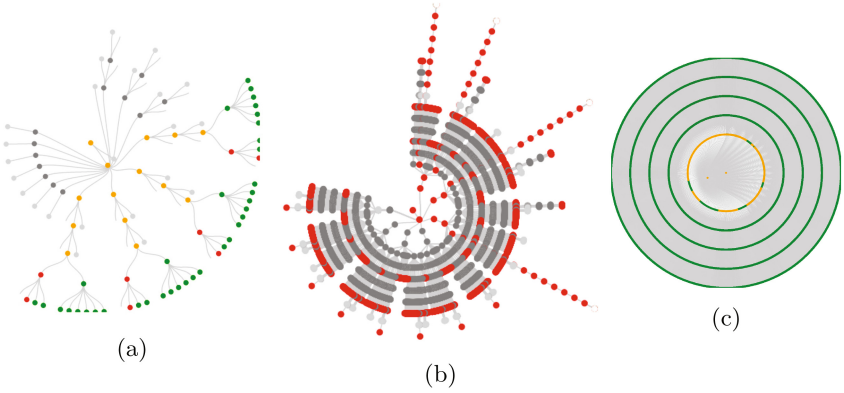


Fig. 5. Examples of global views (Color figure omline)

It is possible to filter the global view of the logs on tuples, objects or bag names and on time interval. This can help developers understand when a rule did more or less work, or if some tuples (e.g. modelling events) triggered a lot of work or none.

Extension to Other Coordination Languages. The tools presented in this section help to design, analyse and debug LINC rules. They could be adapted to other coordination environments with some assumptions.

The monitor is well suited for any tuple space environment because important information is stored in tuple spaces. This tool provides a central point of view to access all the information of the coordination entities (objects in LINC). To be as useful as in LINC, the monitor requires full introspection and intersession capacities of the coordination entities and tuple spaces. For the tracer tool, its main interest is to provide relevant filters and being accessible from the monitor.

The analysing tools require to log all the coordination actions. This requires to instrument the code of the coordination entities to log all relevant actions. This is of particular interest for exogenous coordination languages [1] that do not mix the coordination code within the computation code. For these languages it seems possible to extract, as in LINC, the relevant coordination actions to analyse them. Indeed the application state is stored in the tuple spaces and the application logic is done by the coordination entities.

The interfaces of the analyses tools are relevant if an inference tree is used. Some display are specific to LINC that uses rules with precondition and transactions in performances (typically the logger interface). The data flow and global view of the log could be adapted even if the rules execution paradigm is different.

4 Applications Developed with LINC

More than a dozen of demonstrators have been developed with LINC, within European or national projects, with industrial partners, in several domains.

LINC is now transferred to industry through the start-up Bag-Era[4] which is responsible for its support and commercialisation. This section details a few examples showing what LINC and its tools can be used for.

In the building automation domain, several demonstrators have been built during the H2020 TOPAs and FP7 SCUBA projects. In these demonstrators buildings have been controlled automatically. Some experiments ran for several months. Data collection, on several buildings in Ireland and France, is ongoing to continuously analyse the buildings' behaviour (with ≈1000 data points updated every minutes or ever 10 min). In such applications, with remote components, the monitoring has proven really useful. The logger observation and analysis tools have been useful to developers during the development stage.

In the Artemis Arrowhead project, a LINC application has been deployed across three different and remote Local Area Networks [14]. A web interface to control the application has been developed and used across Europe. Some of the LAN provide no external access. The monitor was available on some parts of the application. For the rest, the traces and log files were downloaded and analysed locally, with the same tools executed by local objects.

In the frame of French project IRT Nanoelec, a smart parking has been developed [13]. LINC allowed to coordinate a dozen of independent products, some off-the-shelf, some still under development. In this demonstrator, approximately one hundred rules were written. Part of them have been generated from Coloured Petri Nets [34]. The monitoring and log analysing tools, with their filtering and search features have proven useful to developers.

Finally, LINC has been used by control scientists to develop advanced control strategies [36]. In this work, LINC bridged the gap between the control scientists assumption and devices of wireless sensor networks.

5 Related Works

In [21] the authors propose the Peer Model, a design tool for parallel and distributed applications. The model assumes a tuple space middleware. The authors focus on some specific patterns commonly used in distributed applications such as split or join of data flow. This approach is extended in [20] where the author introduces design patterns dedicated to coordination. Similarly to our design tools, this helps developer using coordination models. However, no help is provided at run time. In addition, our design tools offer verification at design time.

REO [2] is a coordination model focusing on the interaction between components. A lot of efforts have been put to provide a formal model of REO and connections with formal languages [12]. This is similar to our design tool based on formal languages. However, REO focuses only on the interaction between components whereas our approach provides developers with a complete model of the environment and the application.

[4] http://www.bag-era.fr/.

ReSpecT nets [38] is a language to map ReSpecT programs to Petri nets. ReSpecT is a reactive language based on TuCSoN [29] and tuple spaces. However, this fails to bridge the gap between (i) the large expression capabilities of tuple space coordination languages and (ii) the set of Petri nets on which interesting formal analysis can be done. Whereas, in our approach verification are done on the model and then the coordination code is generated. It is thus possible to limit in the model the expressiveness to provided formal guarantees. However this language could be a good target for our rule generation tools.

Finally, in [26] the authors propose a software engineering methodology to develop multiagent systems in the SAPERE project which target self awareness in pervasive systems. The developers use an API and a development methodology. However it is not clear if the authors provide any tools to help developers understand how their applications are evolving and how to debug them.

6 Conclusion

Design and debugging tools are essential to make coordination models and languages accepted by a broader range of developers. This paper has presented the development tools put in place with LINC, a coordination environment based on tuple spaces and implementing the chemical machine paradigm. The design tools detailed in this paper allow developers to generate rules that are validated with domain specific knowledge or by the use of formal methods. In addition several tools allow developers to monitor, analyse and update a distributed application.

To the best of the authors knowledge, there exists no other tuple space coordination environment providing an equivalent set of tools. The tools developed for LINC could be useful for other tuple spaces based coordination environments. The paper initiates the analysis of what might be reused and what assumptions, on the coordination environment, are made by the different tools.

Future works will focus on extending analysis tools to allow to re-execute an application with the same scheduling as the original execution. This is not guaranteed by default because most of the rules executions are not deterministic. We means that the same result may be achieved with different histories if we consider elementary actions (rd, get and put) order. In case of bug it could be possible that a particular history is the cause of the bug. Thus restarting the application is not enough to ensure that we will have the same sequence during the second execution and thus the bug is not reproducible. By forcing at the replay the order of the rd, get and put we can ensure that the order of the different event is kept and thus this helps in reproducing the bug. Regarding modelling tools, perspectives involve generalising the approach by integrating the potentials of having multiple controllers and control loops. These loops will require to be coordinated themselves, following a related approach involving coordination controllers for autonomic loops [18].

Acknowledgment. This work is funded by the H2020 TOPAs (grant 676760).

References

1. Arbab, F.: What do you mean, coordination. Bull. Dutch Assoc. Theor. Comput. Sci. NVTI **1122**, 1–18 (1998)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
3. Banâtre, J.-P., Fradet, P., Métayer, D.: Gamma and the chemical reaction model: fifteen years after. In: Calude, C.S., PĂun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2000. LNCS, vol. 2235, pp. 17–44. Springer, Heidelberg (2001). doi:10.1007/3-540-45523-X_2
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, vol. 370. Addison-Wesley, New York (1987)
5. Cabri, G., Leonardi, L., Zambonelli, F.: Mars: a programmable coordination architecture for mobile agents. IEEE Internet Comput. **4**(4), 26–35 (2000)
6. Cano, J., Delaval, G., Rutten, E.: Coordination of ECA rules by verification and control. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 33–48. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43376-8_3
7. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32**, 444–458 (1989)
8. Collins, J., Bagrodia, R.: Mobile application development with MELON. In: Guo, S., Lloret, J., Manzoni, P., Ruehrup, S. (eds.) ADHOC-NOW 2014. LNCS, vol. 8487, pp. 265–278. Springer, Cham (2014). doi:10.1007/978-3-319-07425-2_20
9. Cooper, T., Wogrin, N.: Rule-based Programming with OPS5, vol. 988. Morgan Kaufmann, San Fransisco (1988)
10. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Teenylime: transiently shared tuple space middleware for wireless sensor networks. In: Proceedings of the International Workshop on Middleware for Sensor Networks, pp. 43–48. ACM (2006)
11. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. SIGPLAN Not. **45**(4), 57–66 (2010)
12. Dokter, K., Jongmans, S.-S., Arbab, F., Bliudze, S.: Combine and conquer: relating BIP and Reo. J. Logical Algebr. Methods Program. **86**(Ice), 3–20 (2016)
13. Ducreux, L.F., Guyon-Gardeux, C., Louvel, M., Pacull, F., Thior, S.R., Vergara-Gallego, M.I.: Rapid prototyping of complete systems, the case study of a smart parking. In: 2015 International Symposium on Rapid System Prototyping (RSP), vol. 2016, February, pp. 133–139, Amsterdam (2015)
14. Boutin, V., et al.: Energy optimisation using analytics and coordination, the example of lifts. In 19th IEEE Conference on Emerging Technologies and Factory Automation (2014)
15. Fernandez, H., Tedeschi, C., Priol, T.: Rule-driven service coordination middleware for scientific applications. Future Gener. Comput. Syst. **35**, 1–13 (2014)
16. Garnock-Jones, T., Felleisen, M.: Coordinated concurrent programming in syndicate. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 310–336. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49498-1_13
17. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. (TOPLAS) **7**(1), 80–112 (1985)
18. Gueye, S.M.K., Palma, N., Rutten, E.: Component-based autonomic managers for coordination control. In: Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 75–89. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38493-6_6

19. Julien, C., Roman, G.-C.: Egospaces: facilitating rapid development of context-aware mobile applications. IEEE Trans. Softw. Eng. **32**(5), 281–298 (2006)
20. Kühn, E.: Reusable coordination components: reliable development of cooperative information systems. Int. Jo. Cooper. Inf. Syst. **25**(4), 1740001 (2017)
21. Kühn, E., Craß, S., Joskowicz, G., Marek, A., Scheller, T.: Peer-based programming model for coordination patterns. In: Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 121–135. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38493-6_9
22. Kuhn, E., Riemer, J., Mordinyi, R., Lechner, L.: Integration of XVSM spaces with the web to meet the challenging interaction demands in pervasive scenarios. Ubiquit. Comput. Commun. J. CPE, 20–31 (2008). SI (Special issue of Coordination in Pervasive Environments)
23. Louvel, M., Pacull, F.: LINC: a compact yet powerful coordination environment. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 83–98. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43376-8_6
24. Louvel, M., Pacull, F., Vergara-Gallego, M.I.: Coordination scheme editor for building management systems. In: IECON 2016 42nd Annual Conference of the IEEE, pp. 7052–7057. IEEE (2016)
25. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the TOTA approach. ACM Trans. Softw. Eng. Methodol. **18**(4), 15 (2009)
26. Molesini, A., Omicini, A., Viroli, M., Zambonelli, F.: Engineering pervasive multi-agent systems in SAPERE. In: Cossentino, M., Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS, vol. 8245, pp. 196–214. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45343-4_11
27. Murphy, A.L., Picco, G.P., Roman, G.-C.: Lime: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. **15**(3), 279–328 (2006)
28. Omicini, A., Viroli, M.: Coordination models and languages: from parallel computing to self-organisation. Knowl. Eng. Rev. **26**(01), 53–59 (2011)
29. Omicini, A., Zambonelli, F.: TuCSon: a coordination model for mobile information agents. In: Proceedings of the 1st Workshop on Innovative Internet Information Systems, vol. 138 (1998)
30. Omicini, A., Zambonelli, F.: Coordination for internet application development. Auton. Agents Multiagent Syst. **2**(3), 251–269 (1999)
31. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Adv. Comput. **46**, 329–400 (1998)
32. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with alchemist. J. Simul. **7**(3), 202–215 (2013)
33. Schmidt, K.: LoLA a low level analyser. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000). doi:10.1007/3-540-44988-4_27
34. Sylla, A.N., Louvel, M., Pacull, F.: Coordination rules generation from coloured petri net models. In: PNSE@ Petri Nets, pp. 325–326 (2015)
35. Sylla, A.N., Louvel, M., Rutten, E.: Combining transactional and behavioural reliability in adaptive middleware. In: Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware, p. 5. ACM (2016)

36. Vergara-Gallego, M.I., Mokrenko, O., Louvel, M., Lesecq, S., Pacull, F.: Implementation of an energy management control strategy for WSNs using the LINC middleware. In: Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks, pp. 53–58 (2016)
37. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. CM Trans. Auton. Adapt. Syst. **6**(2), 14:1–14:24 (2011)
38. Viroli, M., Omicini, A.: Respect nets: towards an analysis methodology for respect specifications. Electron. Notes Theor. Comput. Sci. **180**(2), 123–144 (2007)