Combining Transactional and Behavioural Reliability in Adaptive Middleware

Adja Ndeye Sylla[1,2]
Adjandeye.sylla@cea.fr

Maxime Louvel[1,2]
maxime.louvel@cea.fr

Éric Rutten[1,3]
Eric.rutten@inria.fr

[1]Univ. Grenoble Alpes, F-38000 Grenoble, France
[2]CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
[3]Inria, CNRS, LIG, F-38000 Grenoble France

ABSTRACT

Adaptive systems behaviours can be intuitively programmed, using rule based middleware, as a set of rules. The rules verify conditions and perform actions in order to achieve a set of objectives. However, this raises several problems. First, inconsistencies may result from the fact that an action is not actually performed due to a communication error or a hardware failure. Second, the rules may be conflicting and their sequential chaining may lead to undesirable behaviour. This paper proposes an approach that combines transactional and behavioural reliability (i.e. consistency and no conflict) in adaptive middleware. This approach is implemented using the middleware LINC and the automata based language Heptagon/BZR. A case study, in the field of building automation, is presented to illustrate the approach.

CCS Concepts

•Software and its engineering → Middleware; Formal methods; Software reliability;

Keywords

Rule based middleware; Transaction; Transition systems

1. INTRODUCTION

Today's systems are not only distributed but also adaptive: they have to react to their environments in order to achieve a set of objectives. In this context, policy or rule based middleware is convenient. It allows to build the considered system as a set of rules that verify conditions and perform actions in order to achieve the objectives. However, this raises several problems. First, it may happen that the action of a rule is not actually performed due to a communication error or a hardware failure. Assuming that the action is performed leads to an inconsistency: the system state and its logical representation differ. Second, the rules may be conflicting and may lead to an undesirable behaviour. A conflict occurs between several rules when they are activated at the same instant and have contradictory actions. For instance, a rule that opens a window to ventilate a room can be conflicting with another rule that limits the noise level at a given threshold. Therefore, when writing rules, one has to manually avoid conflicts. This is done by verifying several conditions on potentially conflicting rules in order to prevent them from being activated at the same instant. For large systems, manually avoiding conflicts between rules is tedious and error prone.

In distributed systems, inconsistencies are avoided through distributed consensus [8] or distributed transactions [2]. Conflicts between rules can be detected or avoided using transition systems [3, 15]. For instance, the solution proposed in [15] is to first model the system using coloured Petri net; then, to verify the absence of conflicts in the designed model with a model-checker; and finally, to generate the corresponding rules. However, this approach requires to manually avoid conflicts in the coloured Petri net models.

This paper proposes to combine transactional and behavioural reliability (i.e. consistency and no conflict), as shown in Figure 1, in the feedback loop execution scheme of adaptive middleware. Transactional reliability is achieved using a middleware that supports transactions. Behavioural reliability is achieved using a transition system.

The paper is structured as follows. Section 2 presents background materials. Then, Section 3 details the composition of execution schemes of both a transactional middleware (LINC [10]) and a transition system (H/BZR [7]). Section 4 presents a case study, in the field of building automation to illustrate the proposed approach. Section 5 discusses related work. Finally, Section 6 concludes and gives future works.

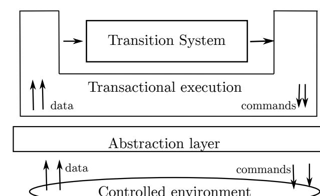


Figure 1: Transactional and behavioural reliability in adpative middleware feedback loop execution scheme

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ARM 2016, December 12-16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-4662-7/16/12...$15.00
DOI: http://dx.doi.org/10.1145/3008167.3008172

2. BACKGROUND

2.1 LINC

LINC [10] is a rule based middleware used to develop and deploy reactive applications over distributed systems. LINC is based on three paradigms:

- **Associative Memory** [4]: It consists in modelling the system as a distributed set of tuple spaces containing tuples. In LINC, tuple spaces are called bags and are distributed over different locations. Tuples are manipulated through three operations: *rd*, *get* and *put*. The *rd* is used to verify the presence of a given tuple in a bag. The *get* allows to remove a tuple from a bag and the *put* is used to insert a tuple. These operations are used as basic primitives of production rules.
- **Production Rules** [5]: A production rule consists of two parts: a *precondition* and a *performance*. The precondition uses the operation *rd*, with a partially instantiated tuple as parameter, to verify specific conditions in the system. If these conditions are true, the performance is triggered. The performance uses the three operations. The *rd* is used to verify conditions. The *get* and the *put* are used to perform actions on the system and update its logical state. The tuples manipulated in the performance must be fully instantiated.
- **Distributed Transactions** [2]: They are used in the performance part to ensure the all-or-nothing property. A transaction allows to group as one operation: the verification of conditions (*rd*), the realisation of actions (*put*), and the update of the system logical state (*get*, *put*). Thus, the performance part may abort if, for instance, the verification of a condition through a *rd* is no longer true. The performance also aborts if a *put* fails because the corresponding action cannot be performed for a given reason (e.g. hardware failure).

2.1.1 Example of LINC rule

```
["Sensors"].rd("pr1","true") &                              1
["Locations"].rd("pr1","presence",location) &
["Locations"].rd(lamp1,"lamp",location) &                   3
["States"].rd(lamp1,"off")
::                                                          5
{ ["Sensors"].rd("pr1","true");
  ["Actuators"].put(lamp1,"switchon");                      7
  ["States"].get(lamp1,"off");
  ["States"].put(lamp1,"on");}.                             9
```

Listing 1: LINC rule example

Listing 1 presents an example of rule that switches on the lamp of a room when a presence is detected. The rule is composed of two parts: a precondition (before the symbol ::) and a performance (between brackets). The precondition consists of three *rd* operations applied on different bags to verify the presence of tuples. This allows to know: if a presence is detected by the sensor with the id `pr1` (line 1) and if the lamp of the same room is off (lines 2, 3, 4). If these conditions are true, the performance is triggered. The performance consists of one transaction that first verifies if the room is still occupied (line 6), then it switches on the lamp (line 7) and changes its logical state (lines 8, 9). The performance succeeds if all its operations succeed. For instance, if a communication error occurs in the lamp actuator, the *put* at line 7 fails and the performance fails. The logical state of the lamp is unchanged and is consistent with the lamp actual state.

2.1.2 Execution of a LINC application

The rules of a LINC application are executed, in parallel, by distributed rule engines. An engine can execute several rules. For each rule, the associated engine first executes the precondition. To do this, the engine evaluates the *rd* operations and builds an inference tree with instantiation and propagation of the variables.

Let us consider the rule example presented in Listing 1. The engine starts by evaluating the first *rd*. This *rd* returns, one by one, all the tuples of the bag `Sensors` which match the pattern `("pr1","True")`. If no matching tuple exists, the *rd* is blocked and waits for new matching tuples. For every returned tuple, the engine creates a new branch and evaluates the next *rd* (line 2). In the same way, the result is all the tuples of the bag `Locations` matching the pattern `("pr1","presence",location)`. For each tuple returned, a new branch is created, the variable `location` is instantiated and its value is used to evaluate the third *rd*. This also returns all the tuples of the bag `Locations` matching the pattern `(lamp1,"lamp",location)`. A new branch is created, the variable `lamp1` is instantiated and its value is used to evaluate the fourth *rd*. Whenever a new matching tuple is added in one of the considered bags, it is returned and a new branch is created. This allows to react to all the events occurring in the system when they occur.

The performance is triggered for each branch with a length equal to four (i.e. number of *rd* in the precondition). Several instances of the same performance can be triggered in parallel for different branches. The performance is composed of one transaction executed with a two phase-commit. In the first phase, pre-operations are performed (*pre_rd*, *pre_get*, *pre_put*) to see if the operations can actually be done. When a *pre_rd* or a *pre_get* succeeds, the tuple is locked. Another transaction which wants to use the same tuple has to wait until it becomes unlocked. If all the pre-operations succeed, the second phase of the transaction performs a *confirm* to consume (*get*) (or release (*rd*)) the locked tuples and insert the new tuples (*put*). If one pre-operation fails, the second phase releases all the locked tuples and nothing is done.

2.2 Heptagon/BZR

Heptagon/BZR or H/BZR [7] is a reactive programming language used to build systems that react to their environments. It belongs to the same family as Lustre and Signal [1]. These languages rely on the synchrony hypothesis [1]: a reaction is assumed faster than the system dynamics. The reactive behaviour of the system can be sample driven or event driven [1]. In the first case, a reaction is triggered periodically (e.g. every 5 seconds). In the second case, a reaction is triggered each time an event occurs.

2.2.1 Automata-based H/BZR programs

In H/BZR, systems are modelled using automata or equations. When the considered system is composed of several entities, each entity is first modelled using an automaton. Then, the automata are composed, in parallel or in hierarchy, to obtain a global automaton that models the whole system. In this case, a reaction of the system involves a reaction of all the different entities, at the same instant. It

corresponds to one transition of the global automaton.

H/BZR enables, through a contract mechanism, Discrete Controller Synthesis (DCS) [12]. DCS is a formal method used to enforce given objectives on a model. Given a model that represents all the possible behaviours of a system, DCS inhibits those that violate the objectives. To do this, DCS requires to partition the variables in two sets: controllable and uncontrollable variables. For a given objective (e.g. avoid undesired or unsafe states), the DCS algorithm explores the state space of the system and gives appropriate values to the controllable variables so that the objective is enforced, whatever the values of the uncontrollable variables.

A basic H/BZR program consists of one block called node. A node has input and output parameters. It contains the automaton modelling the system. A node has a contract part in which are defined the objectives to enforce.

2.2.2 *Execution of a H/BZR program*

The compilation of a H/BZR program generates a sequential code in C or Java. In both cases, the generated code includes a variable that stores the state of the automaton modelling the system and a `step` function. The `step` takes as parameter a set of inputs, computes the outputs and updates the state of the automaton. One execution of the `step` corresponds to one reaction of the system. Therefore, the `step` must be correctly executed, by ensuring the synchrony hypothesis, every time the system has to react to its environment. This can be event driven or sample driven.

2.3 Comparison of LINC and H/BZR

To compare LINC and H/BZR, let us consider a room equipped with two actuators (i.e. a shutter and a lamp). The room must be controlled to achieve two objectives:

1. if presence, luminosity must be in [500, 600] lux;
2. if confidential meeting, room must be completely closed.

In LINC, the first objective is achieved using two rules: R_1 and R_2. R_1 opens the shutter if a presence is detected and the outdoor luminosity is between 500 and 600 lux. R_2 switches on the lamp if a presence is detected and the outdoor luminosity is not between 500 and 600 lux. The second objective is achieved using a rule, R_3, that closes the shutter during a confidential meeting. Each rule first verifies a set of conditions. Then, it sends a specific command to the target actuator and changes its logical state. LINC, through transactions, ensures that if a command cannot be sent, the logical state of the corresponding actuator is not changed. This prevents from inconsistencies. However, LINC does not prevent from conflicting rules. For instance, let us consider the following scenario: presence detected, outdoor luminosity between 500 and 600 lux and confidential meeting held. In this case, both R_1 and R_3 are activated to open and close the shutter at the same instant, leading to a conflict.

In H/BZR, each actuator is first modelled as an automaton by specifying its effects on the environment. Then, the automata are composed to obtain a global automaton in which the objectives are enforced through DCS. When executed, the generated `step` function returns the commands to send to the actuators and changes the state of the global automaton. These commands allow to reach both objectives without conflict. The conflict on the shutter is avoided by returning the command `switch on` for the lamp and `close` for the shutter. However, the `step` changes the state of the automaton before the commands are applied. If the commands cannot be applied for a given reason (e.g. communication error), the `step` becomes inconsistent. Indeed the state of the automaton is different from the actual state of the actuators.

LINC and H/BZR can be combined to enhance adaptive systems reliability. LINC ensures the consistency of an action and H/BZR prevents from conflicting actions.

3. COMBINING LINC AND H/BZR

The logical behaviour of the considered system is first controlled using Heptagon/BZR and DCS to reach the target objectives without conflict. Then, the generated `step` function is executed by LINC to avoid inconsistencies.

3.1 Design flow

Designers are required to model each entity of the system as an automaton, by specifying its states, its transitions and its effects on the environment. Designers also have to formalise the target objectives as logical propositions and define a contract in order to enforce them on the system global automaton and generate a `step` function. The `step` requires specific data in order to compute the commands. Providing the `step` data may require to transform data collected from the environment, to aggregate them or use them to estimate the required data. For instance, a CO_2 data collected from the environment can be used to estimate a presence. Data transformation, aggregation and estimation are not subject to conflicts and then, can be done using a set of rules. These rules, as well as the generated `step` will be executed in LINC.

3.2 Step Execution in LINC

The `step` function takes as parameter a set of inputs, computes the outputs and changes the state of the automaton modelling the system. In the same way, the performance of a LINC rule is used to change the system state. Hence, the `step` should be invoked in the performance of a LINC rule. This rule is written to be triggered each time an event occurs in the system and then, enables the reactive execution of the `step`. The precondition of this rule first collects, from the environment, data corresponding to the input parameters of the `step`. Then, the performance invokes the `step`. This is done by applying an operation *put* on the bag `Step` that encapsulates the `step` function. The pattern of this *put* includes instantiated variables (input parameters of the `step`) and non instantiated variables to store the `step` outputs. This does not comply with the logic of LINC which requires the variables used in the performance to be instantiated.

To overcome this limitation, the proposed solution is to first invoke the `step` in the precondition. This does not change the state of the automaton, this only instantiates the variables used to store the `step` outputs i.e. the commands to send. Then, to invoke the `step` again in the performance in order to change the automaton state. In this case, it is necessary to ensure that the two invocations give the same result. Indeed, if the current state of the automaton changes before the execution of the performance (due to the execution of another instance of the same rule), the second invocation of the `step` will give another result, different from the previous one. Hence, the computed outputs will be no longer valid and must be computed again.

As example, let us consider a room equipped with a lamp

and H/BZR program that maintains the luminosity of this room between 500 and 600 lux when a presence is detected. The step generated by the compilation of this program takes as parameter the value measured by the room presence sensor, returns the command to send to the lamp and changes the state of the lamp automaton. This step is invoked in the LINC rule presented in Listing 2.

The precondition of this rule first reads the value measured by the room presence sensor in the variable presval. Then, at line 4, the precondition applies a *rd* operation, with the partially instantiated tuple (presval, currentState, command), on the bag Step. This *rd* invokes the step function with the instantiated value of presval to compute the command to send to the lamp. In this invocation, the step does not change the state of the lamp automaton. It returns the automaton current state and the computed command respectively in the variables currentState and command.

At line 7, the performance applies a *rd* operation on the bag StepAcess with the pattern ("allowed"). This locks the tuple ("allowed") during the execution of the transaction. Another transaction that wants to use the same tuple has to wait until it becomes unlocked, at the end of the current transaction execution. This ensures the synchrony hypothesis: the step function that changes the state of the automaton will not be invoked in parallel, by several rules. Finally, the performance applies the operation *put* with the pattern (presval, currentState, "") on the bag Step. This will first verify if the state of the automaton is still equal to currentState. Then this will invoke the step function with the value of presval. In this invocation, the step produces the same command and changes the automaton state. The performance is embedded in one transaction. It succeeds if all its operations succeed. If one operation fails, the performance fails and nothing is done. Otherwise, the command is sent to the lamp and the step that changes the lamp automaton state is executed. Invoking the step at the end of the performance ensures that this function is executed only if all the previous operations succeed. This prevents from cancelling the execution of the step when the performance fails because another operation fails.

```
["Sensors"].rd("pr1",presval) &                          1
["Locations"].rd("pr1","presence",location) &
["Locations"].rd(lamp1,"lamp",location) &                3
["Step"].rd(presval, currentState, command)
::                                                        5
{ ["Sensors"].rd("pr1",presval);
  ["StepAcces"].rd("allowed");                            7
  ["Actuators"].put(lamp1,command);
  ["Step"].put(presval, currentState, "");}.              9
```

Listing 2: Step execution in LINC

3.3 Step encapsulation and rule generation

The step function of a H/BZR program is encapsulated in a LINC bag named Step. This bag can be manipulated using the operations *rd* and *put* as shown in Listing 2. To implement the *rd* and the *put* of the bag Step, the H/BZR program is first compiled in C. The generated C code includes a struct named memory (storing the state of the automaton), the name and the type of each input and output parameter of the step function. The execution of the step produces the commands and updates the value of the struct memory in order to change the state of the automaton.

When applied on the bag Step, the *rd* copies the struct memory in another struct named memorycopy and executes the step function using memorycopy. This allows computing the commands without changing the state of the automaton modelling the system behaviour. The computed commands and the current state of the automaton are returned in two variables currentState and command. The *put* on the bag Step first compares the current value of the struct memory to currentState (returned by the *rd*). If they are identical, the *put* executes the step using the struct memory to change the automaton state. Otherwise, the *put* fails and the transaction is not executed. A script is used to automatically encapsulate the step of a H/BZR program (implement the *rd* and the *put*) and generate the rule that invokes it.

3.4 Discussion on design cost and runtime cost

Controlling the logical behaviour of a system in LINC requires to manually achieve the objectives by first thinking about all the possible cases and then, writing a set of rules. These rules can then be distributed for performance or geographical constraint. However, to avoid conflicts between rules, it will be required to add additional conditions on several rules. At the end, several rules read the same inputs at the same instant. The rules seem independent but are not.

In LINC combined with H/BZR, controlling the system logical behaviour just requires to design an automaton for each entity of the system and formalise the objectives as logical propositions. The DCS algorithm is in charge of exploring all the possible cases to automatically achieve the objectives without conflict. LINC combined with H/BZR enforces all the objectives without conflict using one single rule that invokes the step. This rule can replace the set of rules in which several rules read the same values at the same instant.

Controlling the logical behaviour of a large system using one single rule would be limiting. A solution is to first decompose the system into sub-systems and then, design a global automaton for each sub-system. If the sub-systems share objectives, their automata have to communicate and will be composed. This can cause expensive DCS. To deal with this problem, the potential modular DCS in H/BZR [6] can be exploited.

In LINC, communication errors and hardware failures are handled by transactions that perform alternative actions and raise alerts only if no alternative action is possible [10]. In addition, *put* actions on actuators may use different communication protocols if several are available. Hence, a *put* fails only if all the communication links to the target actuator are down. To deal with such situations, it is possible to configure the rule such that it will try several times to see if a communication has become possible.

4. CASE STUDY

Let us consider an office that consists of a room with several actuators (i.e. a window, a shutter, a door, a lamp, a mechanical ventilation and a reversible air conditioner) and a set of sensors (i.e. presence, luminosity, CO_2, noise, temperature). Other sensors are installed outside the room to enquire outdoor conditions. Information about the meetings (day, time, features) that will be held in the office can be obtained through a specific agenda. The aim of the case study is to control the office, to achieve the following objectives:

- For comfort, when a presence is detected, the lumi-

nosity must be between 500 and 600 lux and the noise level must be lower than 80 dB.

- For comfort, when a presence is detected and the temperature is lower than 17°C (resp. greater than 27°C), the room must be heated (resp. cooled).
- For air quality, the room must be ventilated when a presence is detected and the CO_2 exceeds 800 ppm.
- For energy savings, natural lighting, ventilation, heating and cooling are preferred to artificial lighting, ventilation, heating and cooling.
- For confidentiality, the office must be completely closed during a confidential meeting.
- For air quality, the room must be quickly ventilated between two meetings separated by less than 30 min.
- For air quality, the room must not be polluted by pollen or outdoor CO_2.

To demonstrate the interest of our approach, the case study is first implemented in LINC without H/BZR. Then, our approach is used and a comparison is done.

4.1 Development Using LINC

32 rules were first written to control the office. These rules are grouped in five sets: `Temperature`, `Air quality`, `Noise`, `Luminosity` and `Confidentiality`. The rules of each set verify relevant conditions and send appropriate commands to specific actuators. The set of rules written to control the office contains twenty nine potential conflicts. Three examples of conflicts are:

- Window: open for cooling and close for confidentiality;
- Window: open for heating and close for outdoor CO_2;
- Shutter: open for light and close for confidentiality.

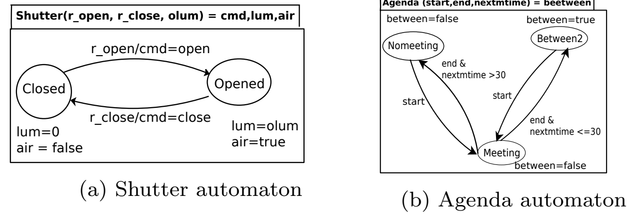
All the conflicts were manually avoided. For every potential conflict, one or more *rd* operations were added in the precondition of the involved rules to ensure that they will not be triggered at the same time. For instance, let us consider the following rules. R_1 closes the window to reduce noise (from outdoor) in the room. R_2 opens the window for natural ventilation. R_3 switches on the HVAC if natural ventilation is not possible (e.g. polluted outdoor air). If the room is occupied, the CO_2 level is high, the outdoor air is not polluted and the outdoor noise is high, there will be a conflict between R_1 and R_2 on the window. To solve this conflict, R_2 must verify if the outdoor noise is normal before opening the window. Another rule R_4 is then required to switch on the HVAC, if there is noise outside, because the window will not be opened and R_3 will not be triggered (i.e. natural ventilation is possible).

At the end, 41 additional conditions were added on 11 rules (34.37%) and 14 rules were added (43.75%).

4.2 Development using H/BZR and LINC

The office is first designed using H/BZR and discrete controller synthesis. Then, the H/BZR program is compiled and the `step` is invoked in a LINC rule.

The automaton of the office is obtained by the parallel composition of automata describing the behaviours of a



(a) Shutter automaton

(b) Agenda automaton

Figure 2: Shutter and Agenda automaton

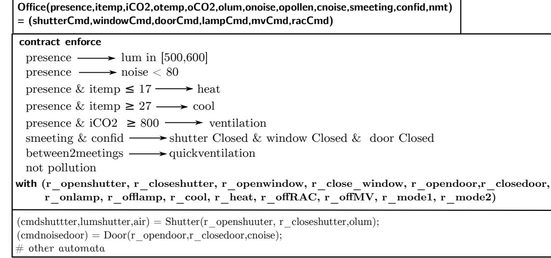


Figure 3: Office node for DCS

room, a lamp, a shutter, a door, a reversible air-conditioner (RAC), a mechanical ventilation (MV), a window and an agenda. Fig. 2a presents the automaton of the shutter. This automaton has two states `Closed` and `Opened` and two transitions. At each state, two variables `lum` and `air` are associated to respectively specify the luminosity provided by the shutter and if it allows air to pass. For instance, when the shutter is opened, it provides a luminosity equal to the outdoor luminosity (`olum`) and allows air to pass. When the variable `r_open` is true, the automaton goes from the state `Closed` to `Opened` and produces the command `open`. Fig. 2b presents the agenda automaton. This automaton has three states: `Nomeeting`, `Meeting` and `Between2`. These states allow to know: if there is a meeting or no and if a meeting will be held in less than 30 min, after a previous meeting.

Fig. 3 presents the main node of the H/BZR program: the office automaton. This automaton takes as parameter all sensor values and meetings information. It returns the commands to send to the different actuators. This automaton is used with DCS to enforce the objectives. The objectives are first formalised as logical propositions. For instance, the first proposition (`presence` $\rightarrow$ `lum in [500,600]`) means that when presence is true, luminosity must be in the range [500,600]. Then, a `contract` is defined to `enforce` these logical propositions with a set of controllable variables associated to automata transitions. For instance, in shutter automaton, `r_open` and `r_close` are controllable variables. H/BZR was able to enforce the objectives without conflict.

The `step` function generated by the compilation of the H/BZR program was invoked in one LINC rule following the execution scheme presented in Listing 2.

4.3 Discussion on the case study

Controlling the office in LINC combined with H/BZR just required to design 8 automata and formalise the objectives as logical propositions. This generated one rule that invokes the `step` and enforces the objectives without conflict.

In LINC, the office was first controlled using 32 rules that can be distributed. However, to avoid conflicts between the rules, it was required to add additional conditions on several

rules. For instance, five conditions were added on the rule that opens the window and the shutter to heat the room. Avoiding conflicts also required to add several rules and make sure that all the necessary rules were added.

In this case study, the luminosity is affected by two actuators involved only in few other objectives (i.e. lamp, shutter). Hence, the office control could be divided in two subsystems (i.e. `luminosity`, `Temperature-CO2-Noise`) that could be controlled separately. However, these sub-systems are not independent. They share an objective (related to confidentiality) and a variable (air provided by the shutter). Therefore, the they must communicate. This could be done with a parallel composition of their automata with DCS or through modular DCS [6] which is less expensive.

5. RELATED WORK

In [9], the authors enhance the verification capabilities of the iLAND middleware, through Petri nets, to increase the reliability of adaptive real time distributed systems. Our approach, improves the reliability of the middleware LINC with discrete controller synthesis (DCS) instead of verification. The advantage of DCS compared to verification is it prevents developers from manually enforcing properties on the system behaviour and verifying if they are satisfied. Indeed, DCS automatically constrains the system behaviour and enforces the target properties, if a solution exists.

In [14], the authors detect and solve conflicts among a set of rules in order to provide behavioural reliability in a service oriented middleware named aWESoME. Transactional reliability should be added to their approach in order to ensure that the services will be properly delivered.

In [13], the authors propose a model-checking based approach for the reliability of IoT middleware. The system behaviour is modelled using automata and a set of properties are verified. If a target property is not satisfied, the counterexample given by the model-checker is used to modify the model and verify it again. In our approach, properties are automatically enforced through DCS, if feasible.

In [11], the authors propose behavioural reliability in a policy based system. A constraint solver is used to detect if a new policy and an existing one can be triggered at the same instant. For each pair of policies that can be triggered simultaneously, their effects on the environment are compared to detect a potential conflict that has to be solved by the user before execution. In our approach, users are not required to solve conflicts (this is automatically done) and transactional execution is also provided to avoid inconsistencies.

6. CONCLUSION

This paper has proposed an approach that combines transactional and behavioural reliability in adaptive middleware. This prevents from inconsistencies, resulting from communication errors or hardware failures, and conflicting actions. The approach was implemented using the middleware LINC and the automata based language H/BZR. The paper first studied LINC and H/BZR. Then, it explained how they are combined and discussed their combination. Then, the paper illustrated the proposed implementation through a case study, in the field of building automation. The case study was discussed to show the advantages of the approach.

An important perspective of this work is to improve the handling of communication errors and hardware failures. Another perspective is to automatise the decomposition of a system into different sub-systems, depending on the objectives to achieve. These sub-systems will then be controlled seperately and coordinated through modular DCS.

Acknowledgements

This work is funded by the ARTEMIS ARROWHEAD (grant 332987) and the H2020 TOPas (grant 676760) projects.

7. REFERENCES

[1] C. Andre, F. Boulanger, et al. Software implementation of synchronous programs. In *Int. Conf. on Application of Concurrency to System Design*, pages 133–142. IEEE, 2001.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[3] J. Cano, G. Delaval, and E. Rutten. Coordination of eca rules by verification and control. In *International Conference on Coordination Languages and Models*, pages 33–48. Springer, 2014.

[4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[5] T. Cooper. *Rule-based programming under OPS5.* Morgan Kaufmann Publishers Inc., 1988.

[6] G. Delaval, S. M. Gueye, et al. Modular coordination of multiple autonomic managers. In *Proceedings of the 17th int. ACM Sigsoft symposium on Component-based software engineering*, pages 3–12. ACM, 2014.

[7] G. Delaval, É. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.

[8] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[9] M. García-Valls, D. Perez-Palacin, and R. Mirandola. Extending the verification capabilities of middleware for reliable distributed self-adaptive systems. In *2014 12th IEEE Int. Conf. on Industrial Informatics (INDIN)*, pages 164–169. IEEE, 2014.

[10] M. Louvel and F. Pacull. Linc: A compact yet powerful coordination environment. In *Coordination Models and Languages*, pages 83–98. Springer, 2014.

[11] C. Maternaghan and K. J. Turner. Policy conflicts in home automation. *Computer Networks*, 57(12):2429–2441, 2013.

[12] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[13] I. Sarray, A. Ressouche, et al. Safe composition in middleware for the internet of things. In *Proceedings of the 2nd Work. on Middleware for Context-Aware Applications in the IoT*, pages 7–12. ACM, 2015.

[14] T. G. Stavropoulos, K. Gottis, et al. awesome: A web service middleware for ambient intelligence. *Expert Systems with Applications*, 40(11):4380–4392, 2013.

[15] A. N. Sylla, M. Louvel, and F. Pacull. Coordination rules generation from coloured Petri net models. In *Proceedings of the Int. Workshop on Petri Nets and Software Engineering*, pages 325–326, 2015.