Analysis of Self-$\star$ and P2P Systems Using Refinement

Manamiary Bruno Andriamiarina[1], Dominique Méry[1,*], and Neeraj Kumar Singh[2]

[1] Université de Lorraine, LORIA, BP 239, 54506 Vandœuvre-lès-Nancy, France
{Manamiary.Andriamiarina,Dominique.Mery}@loria.fr
[2] McMaster Centre for Software Certification,
McMaster University, Hamilton, Ontario, Canada
singhn10@mcmaster.ca, Neerajkumar.Singh@loria.fr

Abstract. Distributed systems and applications are becoming increasingly complex, due to factors such as dynamic topology, heterogeneity of components, failure detection. Therefore, they require effective techniques for guaranteeing safety, security and *convergence*. The self-$\star$ systems are based on the idea of managing efficiently complex systems and architectures without user interaction. This paper presents a methodology for verifying distributed systems and ensuring safety and *convergence* requirements: *Correct-by-construction* and *service-as-event* paradigms are used for formalizing the system requirements using incremental refinement in EVENT B. Moreover, this paper describes a mechanized proof of correctness of the self-$\star$ systems along with a case study related to the P2P-based self-healing protocol.

Keywords: Distributed systems, self-$\star$, self-healing, self-stabilization, P2P, EVENT B, liveness, *service-as-event.*

1 Introduction

Nowadays, our daily lives are affected by technologies such as computers, chips, smartphones. These technologies are integrated into large distributed systems that are widely used, which provide required functionalities, (*emergent* [11]) behaviors and properties from interactions between components. Self-$\star$ systems and their autonomous properties (e.g, self-stabilizing systems autonomically recovering from faults [5]) tend to take a growing importance in the development of distributed systems. In this study, we use the *correct by construction* approach [7] for modelling the distributed self-$\star$ systems. Moreover, we emphasize on the *service-as-event* [2] paradigm, that identifies the phases of *self-stabilization* mechanism.

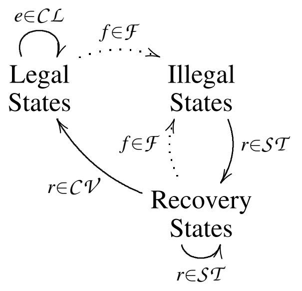

Fig. 1. Diagram for a Self-Stabilizing System $\mathcal{S}$

We consider that a system is characterized by events modifying the states of a system, and modelling abstract phases/procedures or basic actions according to the abstraction level. We define a self-stabilizing system $\mathcal{S}$

---

[*] This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project `http://impex.loria.fr`) from the Agence Nationale de la Recherche (ANR).

Y. Ait Ameur and K.-D. Schewe (Eds.): ABZ 2014, LNCS 8477, pp. 117–123, 2014.
© Springer-Verlag Berlin Heidelberg 2014

with three states (see in Fig.1): *legal states* (*correct states* satisfying a safety property P), *illegal states* (violating the property P) and *recovery states* (states leading from *illegal* to *legal states*). The system $\mathcal{S}$ is represented by a set of events $\mathcal{M} = \mathcal{CL} \cup \mathcal{ST} \cup \mathcal{F}$. The subset $\mathcal{CL}$ models the computation steps of the system and introduces the notion of *closure* [4] : any computation starting from a *legal state* leads to another *legal state*. The occurence of a fault, modelled by an event $f \in \mathcal{F}$ (dotted transition in Fig.1), leads the system $\mathcal{S}$ into an *illegal state*. When a fault occurs, we assume that some procedures identify the current *illegal states* and simulate the stabilization (recovery ($r \in \mathcal{ST}$) and convergence ($r \in \mathcal{CV}$, with $\mathcal{CV} \subseteq \mathcal{ST}$)) procedure to legal states.

This paper is organised as follows. Section 2 introduces the formal verification approach including *service-as-event* paradigm and illustrates the proposed methodology with the study of the self-healing P2P-based protocol [8]. Section 3 finally concludes the paper along with future work.

2 Stepwise Design of the Self-healing Approach

In this section, we propose a formal methodology for self-$\star$ systems that integrates the EVENT B method, the related toolbox RODIN platform and elements of temporal logics, such as traces properties (liveness). Using refinement, we gradually build models of self-$\star$ systems in the EVENT B framework [1]. Moreover, we use the *service-as-event* paradigm to describe the *stabilization* and *convergence* from *illegal* states to *legal* ones. The concept of *refinement diagrams* [2,9] intends to capture the intuition of the designer for deriving progressively the target self-$\star$ system.

2.1 Introduction to the Self-healing P2P-Based Approach

The development of *self-healing P2P-based approach* is proposed by Marquezan et al. [8], where the *reliability* of a P2P-system is the main concern. The self-healing process ensures that if a management service (a *task* executed by peers) of the system enters a *faulty/failed* state, then a self-healing/recovery procedure guarantees that the service switches back to a *legal* state. The self-healing is as follows: **(1) Self-detection** identifies failed instances (peers) of a management service. **(2) Self-activation** is started, whenever a management service is detected as failed. A failed service does not trigger recovery if there are still enough instances for running the service; otherwise, **(3) Self-configuration** repairs the service: new peers running the service are instantiated, and the service is returned into a *legal* state. We illustrate the use of *service-as-event* paradigm and *refinement diagrams* with the formal design of *self-healing approach*.

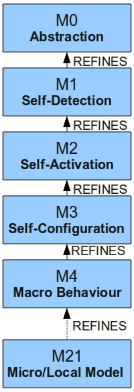

Fig. 2. Architecture

2.2 The Formal Design

Figure 2 depicts the formal design of *self-healing P2P-based approach*. The model M0 abstracts the approach. The refinements M1, M2, M3 introduce the *self-detection*, *self-activation* and *self-configuration*. Models from M4 to M20 are used for localising the self-healing algorithm. The last refinement M21 presents a local model that describes procedures for recovering process of P2P system.

Abstracting the Self-healing Approach (M0). We use the *service-as-event* paradigm to describe the main *functionality (i.e. recovery) offered* by the self-healing protocol. Each service (s) is described by two states: *RUN* (*legal/running* state) and *FAIL* (*illegal/faulty* state). A property $\mathsf{P} \mathrel{\widehat{=}} (s \mapsto RUN \in serviceState)$ expresses that a service (s) is in a *legal running* (*RUN*) state. An event FAILURE leads service (s) into a faulty state (*FAIL*), satisfying $\neg\mathsf{P}$. The *self-healing* of service (s) is expressed by a liveness (*leads to*) property as follows : $(\neg\mathsf{P}) \leadsto \mathsf{P}$, meaning that each faulty state will *eventually* be followed by a legal one. The procedure is stated by an abstract event HEAL, where service (s) recovers from a *faulty* state to a *legal running* one. The refinement diagram[1] (see Fig.3) and events sum up the abstraction of a *recovery* procedure.

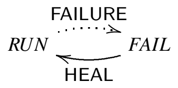

Fig. 3. Abstraction

EVENT FAILURE
ANY
s
WHERE
$grd1 \;: s \in SERVICES$
THEN
$act1 \;: serviceState :=$
$(\{s\} ⩤ serviceState) \cup \{s \mapsto FAIL\}$

EVENT HEAL
ANY
s
WHERE
$grd1 \;: s \in SERVICES$
$grd2 \;: s \mapsto FAIL \in serviceState$
THEN
$act1 \;: serviceState :=$
$(serviceState \setminus \{s \mapsto FAIL\})$
$\cup \{s \mapsto RUN\}$

This *macro/abstract view* of the *self-healing* is detailed by refinement[2], using intermediate steps guided by the three phases : *Self-detection*, *Self-activation* and *Self-configuration*. New variables denoted by $NAME\_\{Refinement\ Level\}$ are introduced.

Introducing the Self-detection (M1). A new state (FL_DT_1) defines the *detection of failures* : a service (s) can ***suspect*** and ***identify*** a failure ($FAIL\_1$) before triggering recovery (HEAL). We introduce a new property $\mathsf{R}_0 \mathrel{\widehat{=}} (s \mapsto FL\_DT\_1 \in serviceState\_1)$ and a new event FAIL_DETECT. The steps of self-detection are introduced, using the inference rules [6] related to the operator *leads to* ($\leadsto$), as illustrated by refinement diagram 4 and proof tree. The event FAIL_DETECT expresses the *self-detection*: the failed state ($FAIL\_1$) of a service (s) is detected (state FL_DT_1). The property $(\neg\mathsf{P}) \leadsto \mathsf{R}_0$ is expressed by the event FAIL_DETECT. $\mathsf{R}_0 \leadsto \mathsf{P}$ is defined by the event HEAL, where the service (s) is restored to a *legal running* state after failure detection. The same method is applied to identify all the phases of *self-healing* algorithm. Due to limited space, we focus on the interesting parts of models and liveness properties. The complete development can be downloaded from web[3] and details can be found in the companion paper [3].

$$\frac{(\neg\mathsf{P}) \leadsto \mathsf{R}_0 \qquad \mathsf{R}_0 \leadsto \mathsf{P}}{(\neg\mathsf{P}) \leadsto \mathsf{P}}\ trans$$

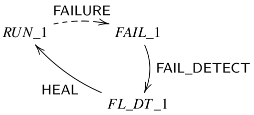

Fig. 4. Self-Detection

---

[1] The assertions ($s \mapsto st \in serviceState$), describing the state (st) of a service (s), are shorten into (st), in the nodes of the refinement diagrams, for practical purposes.

[2] $\oplus$: to add elements to a model, $\ominus$: to remove elements from a model.

[3] `http://eb2all.loria.fr/html_files/files/selfhealing/self-healing.zip`

EVENT FAILURE
REFINES FAILURE
...
WHERE
$\oplus\ s \mapsto RUN\_1 \in serviceState\_1$
THEN
$\ominus\ act1\ :\ \dots$
$\oplus\ serviceState\_1 :=$
$(serviceState\_1 \setminus \{s \mapsto RUN\_1\})$
$\cup\{s \mapsto FAIL\_1\}$

EVENT FAIL_DETECT
ANY
s
WHERE
$grd1\ : s \in SERVICES$
$grd2\ : s \mapsto FAIL\_1 \in serviceState\_1$
THEN
$act1\ : serviceState\_1 :=$
$(serviceState\_1 \setminus \{s \mapsto FAIL\_1\})$
$\cup\{s \mapsto FL\_DT\_1\}$

EVENT HEAL REFINES HEAL
...
WHERE
$\ominus\ grd2 \dots$
$\oplus\ s \mapsto FL\_DT\_1 \in serviceState\_1$
THEN
$\ominus\ act1 \dots$
$\oplus\ serviceState\_1 :=$
$(serviceState\_1 \setminus \{s \mapsto FL\_DT\_1\})$
$\cup\{s \mapsto RUN\_1\}$

Introducing the Self-activation (M2) and Self-configuration (M3). The *self-activation* is introduced in M2 (see Fig. 5), where a failure of a service (s) is evaluated as critical or non-critical using a new state FL_ACT_2 and an event FAIL_ACTIV. The *self-configuration* step is introduced in M3 (see Fig.6): if the failure of service (s) is critical, then *self-configuration* for a service (s) is triggered (state FL_CONF_3), otherwise, the failure is ignored (state FL_IGN_3).

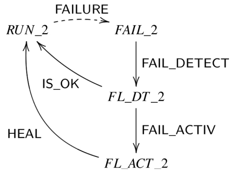

Fig. 5. Self-Activation

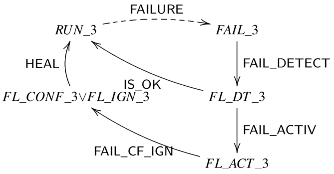

Fig. 6. Self-Configuration

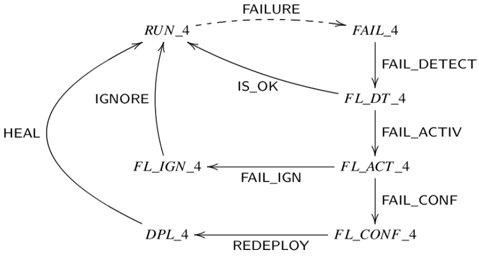

Fig. 7. Self-Healing steps

The Global Behaviour (M4). The models are refined and decomposed into several steps (see Fig.7) [8]. **(1)** ***Self-Detection*** phase is used to detect any failure in the autonomous system. The event FAIL_DETECT models the failure detection; and the event IS_OK states that if a detected failure of a service (s) is a *false alarm*, then the service (s) returns to a *legal* state (RUN_4). **(2)** ***Self-Activation*** evaluates detected failures which are actual. The events FAIL_IGN and IGNORE are used to ignore the failure of service (s) when it is not critical (FL_IGN_4). The event FAIL_CONF triggers the reconfiguration of service (s) when failure is critical (FL_CONF_4). **(3)** ***Self-Configuration*** presents the healing procedure of a *failed* service using an event REDEPLOY.

The refinements M5, M6, M7 introduce gradually the running ($run\_peers(s)$), faulty ($fail\_peers[\{s\}]$), suspicious ($susp\_peers(s)$) and deployed instances ($dep\_inst[\{s\}]$) for a service (s). Each service (s) is associated with the minimal number of instances required for running service (s): during the *self-activation* phase, if the number of running instances of service (s) is below than minimum, failure is critical. Models from M8 to M10 detail the *self-detection* and *self-configuration* phases to introduce the *token owners* for the services. Models from M11 to M20 localise gradually the events (we switch from a *service* point of view to the point of view of peers). Due to limited space[3], in the next section, we present only M21.

```
MACHINE 21 ...
  EVENT SUSPECT_INST
    ANY
      s, susp
    WHERE
      grd1 : s ∈ SERVICES
      grd2 : susp ⊆ PEERS
      grd3 : susp = run_inst(token_owner(s) ↦ s) ∩ unav_peers
      grd4 : suspc_inst(token_owner(s) ↦ s) = ∅
      grd5 : inst_state(token_owner(s) ↦ s) = RUN_4
      grd6 : susp ≠ ∅
    THEN
      act1 : suspc_inst(token_owner(s) ↦ s) := susp
    END
  EVENT FAILURE ...
  EVENT RECONTACT_INST_OK ...
  EVENT RECONTACT_INST_KO ...
  EVENT FAIL_DETECT ...
  EVENT IS_OK ...
  EVENT FAIL_ACTIV ...
  EVENT FAIL_IGNORE ...
  EVENT IGNORE ...
  EVENT FAIL_CONFIGURE ...
  EVENT REDEPLOY_INSTC ...
  EVENT REDEPLOY_INSTS ...
  EVENT REDEPLOY ...
  EVENT HEAL ...
  EVENT MAKE_PEER_UNAVAIL ...
  EVENT UNFAIL_PEER ...
  EVENT MAKE_PEER_AVAIL ...
```

The Local Model (M21). This model details locally the *self-healing* procedure of a service (s). The notion of *token owner* is more detailed: the *token owner* is a peer instance of service (s) that is marked as a *token owner* for the Management Peer Group (MPG), i.e. the set of peers instantiating service (s). It controls *self-healing* by applying *self-detection*, *self-activation*, and *self-configuration* steps. **(1) Self-Detection** introduces an event SUSPECT_INST that states that the *token owner* is able to *suspect* a set ($susp$) of unavailable instances of service (s). Events RECONTACT_INST_OK and RECONTACT_INST_KO are used to specify the successful and failed recontact, respectively, of the unavailable instances for ensuring failures. Moreover, the *token owner* is able to monitor the status of service (s) using two events FAIL_DETECT, and IS_OK. If instances remain unavailable after the recontacting procedure, the *token owner* informs the safe members of MPG of failed instances (FAIL_DETECT); otherwise, the *token owner* indicates that service (s) is running properly (IS_OK). **(2) Self-Activation** introduces an event FAIL_ACTIV where the *token owner* evaluates if a failure is critical. Event FAIL_IGNORE specifies that the failure is not critical. It is ignored (event IGNORE), if several instances (more than minimum) are running correctly. Otherwise, the failure will be declared critical, and *self-configuration* will be triggered using an event FAIL_CONFIGURE. **(3) Self-Configuration** introduces three events REDEPLOY_INSTC, REDEPLOY_INSTS and REDEPLOY that specify that new instances of running service (s) are deployed until the minimal number of instances is reached. And after, the event HEAL can be triggered, corresponding to the *convergence* of the self-healing process.

Moreover, in this model, we have formulated *hypotheses* for ensuring the correct functioning of the self-healing process: *(1) If the token owner of a service (s) becomes unavailable, at least one peer, with the same characteristics as the disabled token owner (state, local informations about running, failed peers, etc.) can become the new token owner; (2) There is always a sufficient number of available peers that can be deployed to reach the legal running state of a service (s).* In a nutshell, we say that our methodology allows users to understand the self-⋆ mechanisms, to gain insight into their architectures (components, coordination, etc.); and gives evidences of their correctness under some assumptions/hypotheses.

3 Discussion, Conclusion and Future Work

We present a methodology based on liveness properties and *refinement diagrams* for modelling the self-⋆ systems using Event B. The key ideas are to characterize the self-stabilizing systems by modes : 1) *legal (correct)* state, 2) *illegal (faulty)* state, and 3) *recovery* state (see Fig.1); then identify the required abstract steps between modes, for ensuring *convergence*; and enrich abstract models using refinement. We have illustrated our methodology with the *self-healing approach* [8]. The complexity of the development is measured by the number of proof obligations (POs) which are automatically/manually discharged (see Table 1). A large majority ($\sim 70\%$) of the 1177 manual proofs is solved by simply running the provers from the Atelier B. The actual summary of POs is given by Table 2. Manually discharged POs require analysis and skills, whereas *quasi*-automatically discharged POs would only need a *tuning* of RODIN (e.g. provers run automatically).

Table 1. Summary of Proof Obligations

Model	Total	Auto		Interactive	
CONTEXTS	30	26	86.67%	4	13.33%
M0	3	3	100%	0	0%
M1	21	15	71.4%	6	28.6%
M2	46	39	84.8%	7	15.2%
M3	68	0	0%	68	100%
M4	142	16	11.27%	126	88.75%
OTHER MACHINES	1111	158	14.22%	953	85.78%
M21	13	0	0%	13	100%
TOTAL	1434	257	17.9%	1177	82.1%

Table 2. Synthesis of POs

Total	Auto		*Quasi*-Auto		Manual	
1434	257	17.9%	850	59.3%	327	22.8%

Furthermore, our refinement-based formalization produces local models close to the *source code*. Our future works include the generation of applications from the resulting model extending tools like EB2ALL [10]. Moreover, further case studies will help us to discover new patterns that could be implemented in the RODIN platform. Finally, another point would be to take into account dependability properties and concurrency.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Andriamiarina, M.B., Méry, D., Singh, N.K.: Integrating Proved State-Based Models for Constructing Correct Distributed Algorithms. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 268–284. Springer, Heidelberg (2013)
3. Andriamiarina, M.B., Méry, D., Singh, N.K.: Analysis of Self-$\star$ and P2P Systems using Refinement (Full Report). Technical Report, LORIA, Nancy, France (2014)
4. Berns, A., Ghosh, S.: Dissecting self-* properties. In: Proceedings of the 2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009, pp. 10–19. EEE Computer Society, Washington, DC (2009)
5. Dolev, S.: Self-Stabilization. MIT Press (2000)
6. Lamport, L.: A temporal logic of actions. ACM Trans. Prog. Lang. Syst. 16(3), 872–923 (1994)
7. Leavens, G.T., Abrial, J.-R., Batory, D.S., Butler, M.J., Coglio, A., Fisler, K., Hehner, E.C.R., Jones, C.B., Miller, D., Jones, S.L.P., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) GPCE, pp. 221–236. ACM (2006)

8. Marquezan, C.C., Granville, L.Z.: Self-* and P2P for Network Management - Design Principles and Case Studies. Springer Briefs in Computer Science. Springer (2012)
9. Méry, D.: Refinement-based guidelines for algorithmic systems. International Journal of Software and Informatics 3(2-3), 197–239 (2009)
10. Méry, D., Singh, N.K.: Automatic code generation from event-b models. In: Proceedings of the Second Symposium on Information and Communication Technology, SoICT 2011, pp. 179–188. ACM, New York (2011)
11. Smith, G., Sanders, J.W.: Formal development of self-organising systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 90–104. Springer, Heidelberg (2009)