Recovery Oriented Programming ⋆

(Extended Abstract)

Olga Brukman, Shlomi Dolev

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel brukman,dolev@cs.bgu.ac.il

Abstract. Writing a perfectly correct code is a challenging and a nearly impossible task. In this work we suggest the *recovery oriented programming* paradigm in order to cope with eventual Byzantine programs. The program specification composer enforces the program specifications (both the safety and the liveness properties) in run time using predicates over input and output variables. The component programmer will use these variables in the program implementation. We suggest using the "sand-box" approach in which every instruction of the program that changes a specification variable, is executed first with temporary variables and that is in order to avoid execution of an instruction that violates the specifications. In addition, external monitoring is used for coping with transient faults and for ensuring convergence to a legal state. The implementation of these ideas includes the definition of new instructions in the programming language with the purpose of allowing addition of predicates and recovery actions. We suggest a design for a tool that extends the Java programming language. In addition to that, we provide a correctness proof scheme for proving that the code combined with the predicates and the recovery actions is self-stabilizing and, under the restartability assumption, eventually fulfills its specifications.
Keywords: self-stabilization, autonomic computing.

1 Introduction

Writing a perfectly correct code is a challenging and a nearly impossible task. Paradigms, tools and programming environments, including structured programming, object oriented programming, design patterns and others, were created to assist the programmer in writing a manageable and correct code. Tools that ensure testing during the programming phase complement the above effort [2, 5, 13]. Still, in many cases the program specifications are not fulfilled [19] – a situation that can cause a great deal of damage. In our previous work on self-stabilizing autonomic recoverer [4], we suggested a formal framework for the recovery oriented paradigm [20]. The suggested approach fitted existing (black

⋆ Partially supported by the Lynn and William Frankel Center for Computer Sciences, by Deutsche Telecom grant, by IBM faculty award, the Israeli Ministry of Science, and the Rita Altura Trust Chair in Computer Sciences.

box) software packages, which resulted in high overhead, as each IO action had to be intercepted to detect a faulty state.

Fault tolerance paradigms and eventual Byzantine software. Self-stabilization [9] is a strong fault tolerance property for systems that ensures automatic recovery once faults stop occurring. A self-stabilizing system is able to start from any possible configuration in which processors, processes, communication links, communication buffers and any other process-related components are in an arbitrary state (e.g., arbitrary variable values, arbitrary program counter). The designer can only assume that the system's programs are executed. Based on that assumption, he or she proves that the system converges to a legal state, i.e., to a state in which the system satisfies its specifications. If the system is started from a legal initial state, the execution will ensure that the system remains in a legal state. This is called "closure property". In case the system is started in an illegal state (possibly after encountering transient faults), the execution of the self-stabilizing program will ensure that eventually (within a finite number of steps) a legal state will be reached. This is called "convergence property". Again, once the program reaches a legal state, it will continue running and will remain in a legal state until a (transient) fault reoccurs. A self-stabilizing algorithm never terminates. The algorithm does not necessarily need to "identify" the failure occurrence and to recover, but rather it continues to be executed and brings the system into a legal state. The time complexity of a self-stabilizing algorithm is the number of steps required for an algorithm started in an arbitrary state to converge to a legal state. Note that when all the processors execute incorrect programs (programs with bugs), they may exhibit any kind of behavior and, therefore, there is no guarantee for convergence.

The Byzantine fault model [16, 10] is used for modeling arbitrary (in fact, malicious) behavior of a program that contains bugs and, therefore, does not obey the specifications. Systems that tolerate a bounded number of Byzantine processors (typically, less than one third of the processors can be Byzantine) were designed and proved to be correct.

Research on self-stabilizing systems and systems that model faults through Byzantine behavior has not yet provided solutions for systems in which software packages contain bugs with a very high probability. We observe that software packages usually function as required for a long period of time after being started from an initial state. The initial correct behavior can be attributed to the testing done by the software manufacturer. Therefore, programs started from an initial state run correctly for bounded length executions. System administrators and users occasionally restart such software in order to cope with failures.

Our contribution. Our goal is to incorporate the program predicates and the recovery actions provided by the program specification composer with the program code so that the predicates violation would be detected and avoided during run-time. Both predicates and recovery actions will be an integral part of the program specification. In this scope we are interested in a new programming paradigm that complements the case of black box software packages (addressed in [4]).

This approach alters the responsibility of the program specification composer (e.g., project manager). In addition to providing complete formal specifications, the composer has to state critical safety and liveness properties, and provide recovery actions for each property (there can be a few recovery actions for each property). The recovery actions will be executed if the predicate is violated. The programmer will make the best effort to write a program that satisfies these specifications. Still, the program may encounter some unpredicted states due to bugs or transient faults. Our framework automatically generates additional code for the program. This code will enforce the program satisfactory behavior (assuming the restartability property of programs [20]) by checking the predicates supplied by the program specification composer during runtime and by executing recovery actions, e.g., restart, in case of predicates failures.

Execution of an instruction that violates the specifications should be avoided. We suggest using the "sand-box": every instruction of the program that changes a specification variable will be first executed on temporary variables. In case the predicates are not violated with regard to the temporary variables, the instruction is executed. Otherwise, the execution of this instruction is stalled and a recovery action takes place.

A program may be in an illegal state (possibly, infinite loop), executing portions in which no predicate variable is updated. Therefore, the specifications may not be checked, possibly ever. In such cases external monitoring is required in order to cope with transient faults and to ensure convergence to a legal state. The external monitor will check the specifications periodically and enforce a state in which the specifications hold.

We suggest a design for a generic tool that extends an object oriented language, e.g., the Java programming language. The pre-compiler is designed to support new primitives for recovery oriented programming. Moreover, we provide a correctness proof scheme for proving that under the assumption of *rsf-execution* (Definition 1) the code combined with the predicates and the recovery actions fulfills its specifications.

Our framework is the first, to the best of our knowledge, to ensure the eventual validity of the specifications starting in any initial state. This is achieved by relying on self-stabilizing software platform, by using sandbox and by using external monitoring. We address full monitoring of liveness properties, while other works consider safety properties only.

While full specifications define in fact a program [22], we consider abstract task specifications, that leaves freedom to the programmer to choose the (efficient) way to write the program, including the specific data structures and algorithms. The abstract task specifications reflect the minimal desired functionality of the system.

The suggested framework is able to cope with transient faults. In addition, we assume that the software is either correct or *eventually Byzantine*. The software is called *eventually Byzantine* if, after being restarted, it can be trusted to perform correctly throughout the execution for a significant portion of the execution. The correct execution after restart is attributed to the testing and debugging process

the software undergoes when being released. Our framework is not intended for dealing with totally incorrect (Byzantine) programs, such as empty programs (programs with no code).

The suggested approach is more efficient than the approach proposed in our previous work [4] as we now assume that we have access to the program. In our previous work we had to intercept all IO actions in order to detect a faulty state – a feature that implied a substantial overhead. In this work we consider the case in which the code is given and we are able to avoid this overhead by monitoring the variables, mainly when their values are changed.

Related work. There are tools that monitor safety by augmenting the programs with monitoring code for safety problems [13, 7] and making some kind of recovery, e.g., throwing exception or executing predefined recovery action.

The well known and widely used exception mechanism [14] is a technique for handling illegal input or underlying system failures. In [23] transactions are used as a tool for achieving atomic actions and use exceptions as a recovery tool in case a transaction fails. However, practice shows that exceptions are not practical for programming an alternative flow of the program in case of a failure [12].

The *recovery block* concept [21] suggested to use component redundancy (e.g., N-programming) for dealing with failures in critical parts of the system. The recovery block concept does not support full monitoring of liveness properties and does not provide guarantees for stability of a monitoring mechanism.

There are several well known languages, Nurpl [8], ASM [15] and IO Automata [18], that provide a formal language for writing program specifications and framework for gradually and manually translating them into a fully verified program. Still, since the process is not fully automated, there is no guarantee that the resulting code is correct.

Writing a program as a collection of SRC (Software Cost Reduction) specifications (detailed specification describing in full the program automata) and then automatically transforming them into the code is suggested in [22]. However the produced program may have the same problems as the same program written from scratch by a programmer due to mistake in the detailed SRC specifications, which is, in fact, the program.

The work in [1, 17] attempts to model a monitoring and correcting middleware layer for arbitrary faulty software. The correcting actions are arbitrary. Thus, the system original software can be completely ignored or its private state can be altered by the correcting layer. Therefore, the programmer of the component correcting actions is, in fact, the component programmer. In our work we limit ourselves only to non-intruding recovery actions, such as restarting.

The rest of the paper is organized as follows. The system architecture appears in the Section 2. The design and implementation details of our framework are presented in Section 3. Section 4 presents a study case that uses our framework. In this study case we investigate the producer-consumer classical problem. Conclusions appear in Section 5.

2 The System Architecture

A *processor* is a multitasking entity that may execute several *processes*. Each *process* is modeled by a state machine that executes *atomic steps* of a *program* that might be faulty. An *atomic step* $a = \langle j, s, s', io\rangle$ of a *process* is a transition from state s to state s' by a process p_j. The transition consists of internal calculations and of a single interaction of p_j with other processes by an input/output operation (io). The communication capabilities of the processes are defined by a directed communication graph $G(V, E)$. An edge (i, j) in $G(V, E)$ denotes the ability of a process p_j to receive information from a process p_i by means of messages or shared memory. The *system configuration* consists of a vector $\langle s_1, s_2, \ldots, s_n\rangle$, where s_i is a state of a process p_i in the system, and of the contents of the communication devices. The contents of the communication devices are either the contents of the messages queues $\langle m_{1,2}, m_{1,3}, \ldots, m_{i,j}, \ldots\rangle$, where $m_{i,j}$ is a queue for messages sent by a process p_i to a process p_j, or the shared communication registers $\langle r_{1,2}, r_{1,3}, \ldots r_{i,j}, \ldots\rangle$, where $r_{i,j}$ is a register shared by processes p_i and p_j. An *execution* is a sequence $E = c_1, a_1, c_2, a_2, \ldots$ of configurations c_i and atomic steps a_i so that c_{i+1} is reached from c_i by the execution of a_i. An execution E is *fair* if every process executes a step infinitely often in E.

A *subsystem* is a set of dependent processes that may include one or more processes. Subsystems can be nested according to a *directed acyclic graph* (DAG) defined by the system designer. The composition of subsystems is required to ensure that both the state of each subsystem component and the combined state of the subsystem components are legal. The DAG hierarchy implies conclusive recovery scenario, where a cyclic dependencies graph may cause infinite recovery loop. Further discussions concerning processes and subsystems will be in terms of subsystems.

The *software/task specification function* is a function *sf(I)=IO*, where $I \in \mathcal{I}$ is a particular sequence of inputs in the set $\mathcal{I}$ of all possible (finite and infinite) sequences of inputs, and $IO \in \mathcal{IO}$ is a particular sequence $\langle i_1, o_1, i_2, o_2, \ldots\rangle$ of alternating inputs and outputs in the set $\mathcal{IO}$. The set $\mathcal{IO}$ defines the desired behavior of the software. A (sub)system sub_i *respects its specification function* sf_i in an execution E with input/output sequence IO if $IO \in \mathcal{IO}$.

A *legal state* of a process/subsystem is a state in which process/subsystem does not violate any safety properties and in which any fair execution that starts in this state does not violate any safety or liveness properties.

For the sake of a correctness proof we assume that a recovery action of a process/subsystem results in a process/subsystem that respects its specification function sf forever, i.e., after executing the recovery action the process/subsystem will be in a legal state. Once a process/subsystem reaches a legal state, the process will continue and stay in a legal state.

We suggested modeling the behavior of software as eventually Byzantine, and to use restarts as recovery actions that bring the system to a legal state.

Definition 1 (Rsf-execution). *An execution E is a* recovery supporting fair execution *(rsf-execution) iff E is a fair execution in which every subsystem sub_i that executes a recovery action during E, respects its specification function sf_i.*

We are now ready to state the system requirement.

Requirement 1 *Every rsf-execution E has a suffix in which the system respects its specification function sf.*

We will prove that any process or subsystem that starts from an arbitrary state will satisfy Requirement 1 in every sufficiently long execution. This proof technique is frequently used for proving self-stabilization [9].

3 Recovery Oriented Programming

Subsystems configuration file. A program specification composer provides a file with the subsystems dependencies graph. A process is the name of a thread or of an object (phantom process) instance. Each process forms a (minimal) subsystem. For each subsystem *sub* the following information is provided: the subsystem name and the list of all the subsystems names that constitute *sub*.
Recovery tuple. We suggest that the code contract between the specification composer and the programmer will be in the form of recovery tuples for subsystems. There might be more than one recovery tuple for a subsystem. Each recovery tuple is a list consisting of a *triggering event*, a *snapshot instruction*, a *predicate*, a list of *recovery actions*, and a *rule for trimming the history log*. The recovery tuples are used for augmenting the program with monitoring and recovery code. Next, we elaborate on each field of the recovery tuples.
• *Triggering event.* A triggering event is defined by the name of a specification (input/output) variable and a method used for modifying the variable value. The augmented code produced for this tuple is activated whenever there is a modification of the variable value using the method.
• *Snapshot instruction.* The snapshot instruction is a tuple $\langle sub_{tag}, \{var_1, \ldots var_k\}, \{var_{k+1}, \ldots, var_l\}\rangle$, where sub_{tag} is the name of the subsystem the recovery tuple is for, and var_i is a variable that its value should be recorded during the snapshot. The variables in the first clause ($\{var_1, \ldots var_k\}$) are recorded before the triggering event is executed. The variables from the second clause ($\{var_{k+1}, \ldots var_l\}$) are recorded immediately after the triggering event execution. After the snapshot is completed a new entry is added to the history log of the subsystem sub_{tag} and to the history log of each subsystem sub_j, such that $sub_{tag} \subset sub_j$. That is, $history_{tag}$ is an ordered list consisting of snapshots made for subsystem sub_{tag} and for all subsystems that sub_{tag} consists of. This is done to ensure that the recovery tuple predicate for some subsystem sub_i would use history log of sub_i only and would not need access to the history log of some sub_j, where $sub_j \subset sub_i$.
• *Predicate.* The predicate is a linear temporal logic (LTL) expression specifying the required program behavior using the input and output variables of the

program. In addition, the linear temporal logic expression may have the history log as one of its variables.

For the sake of simplicity, we assume that a predicate that contains the LTL operator *eventually* is a *liveness predicate*. Otherwise, it is a *safety predicate*. The predicate can be either a process predicate or a subsystem predicate. A process predicate is a logical expression on process variables and the process history only. The subsystem predicate is a logical expression on variables from several different processes and on the subsystem history log entries. A recovery tuple with a safety predicate can be either for an event-driven check or for external monitoring. The recovery tuple with safety predicate for external monitoring will have an empty triggering event field.

The scope of the recovery tuple is the scope of the tuple snapshot instruction and of the predicate.

Recovery tuples are classified according to their predicate to be either a liveness recovery tuple or a safety recovery tuple.

• *Recovery actions.* The recovery actions field of a recovery tuple is a list of several procedure calls or actual code segments. Whenever the activated augmented monitoring code discovers that a predicate of a recovery tuple does not hold, some recovery action is invoked. Typical recovery action procedures use non-intrusive actions such as rolling back to a safe state, waiting, rescheduling or restarting.

The recovery actions of a recovery tuple are listed in the severity order. Each time the predicate of the recovery tuple does not hold, the next more severe recovery action from the list is invoked. The last recovery action in the list of recovery actions is always the restart of the whole subsystem and the initialization of the subsystem history.

The programmer must implement the *Restartable* interface for each process that might be restarted. The *Restartable* interface provides the structure for implementing several recovery action functions. A recovery action for a process is a call for one of its recovery action function.

• *History trimming rule.* A history trimming rule is a (simple) function on the history log of a subsystem sub_i, where the scope of the recovery tuple, for which the history trimming field belongs to, is sub_i as defined by the tag in the snapshot field.

Roughly speaking, history trimming is used for efficiency reasons and as a way for supporting a liveness indication. A *liveness event* (such as entrance to the critical section) associated with a recovery tuple is identified during run time when (1) the triggering event of the recovery tuple occurs and (2) the liveness predicate of the tuple holds. Whenever the liveness event occurs, the history log is trimmed according to the function defined in the history trimming field. Lack of liveness is detected when the subsystem is in the same state twice, executing steps in between, without making progress. This implies that the system can repeat this behavior forever.

Theoretically, since non-terminating computation can infinitely increase a variables, an unbounded history log might be required. In reality, the possible

values of variables are bounded by the type of the variable. Therefore, the history log can be considered to be bounded. Moreover, a program specifications composer that would like to have an efficient liveness detection, may choose variables with very limited possible values. The bounded history log implies that if there is a failure of a liveness property it will be detected.

Next we will elaborate on the way recovery tuples are used in creation of the monitoring augmented code.

Monitoring of a subsystem. The augmented monitoring code has two main components: a code for event driven monitoring and a code for the external monitoring. The external monitoring of a subsystem is required for two reasons. The first reason is to ensure that the system would be able to recover from transient faults. That is, even if the subsystem does not reach a triggering event that activates augmented monitoring code (possibly due to its state corruption), the predicates will be checked, and the specifications will be enforced by invoking a recovery action. The second reason is the fact that the detection of livelock is sometimes impossible from within the subsystem.

Each subsystem has an external monitor process (thread). The existence of the external monitor threads and their scheduling is ensured by a self-stabilizing OS [11] and by the framework of the self-stabilizing autonomic recoverer [4]. The external monitor of the subsystem repeatedly checks the subsystem recovery tuples.

An external monitor will have an additional responsibility, namely, checking the syntax and the length bounds (that may be related to the number of possible states of the subsystem) of the history log entries.

Next we describe the augmented code for each type of the recovery tuple.

Liveness recovery tuple. Every liveness recovery tuple has a triggering event, a snapshot instruction, a predicate, and a history trimming rule. Event driven monitoring for liveness recovery tuples only trims the subsystem history log upon the predicate satisfaction.

The external monitor uses only the snapshots recorded in the history and the recovery actions. Namely, in case the value of the variables of this set appear twice in the history, while the subsystem has been scheduled to execute steps in between, the external monitor invokes a recovery action.

For each recovery tuple with a liveness predicate event the pre-compiler (Figure 1) inserts a code for checking the predicate each time the triggering event takes place.

Event driven safety recovery tuple. If a safety recovery tuple has a triggering event, then such an event driven safety recovery tuple yields augmented code for monitoring that uses temporary variables for checking the predicate before the actual modification. For each such recovery tuple the pre-compiler (Figure 1) inserts code for checking the predicate in a "sand-box" and only if the predicate is satisfied the actual assignment takes place. Otherwise, a recovery action is invoked.

Externally checked safety recovery tuple. A non event driven safety recovery tuple yields a code for the external monitor only. The generated code implies repeated snapshots, a predicate check and recovery action when needed.

Pre-compiler. The pseudocode for the pre-compiler is presented in Figure 1. The pre-compiler receives as an input the program file F and the file G with a definition of the subsystems hierarchy. The pre-compiler output is the transformed program file F' and files with code for external monitors, $em_1, em_2, \ldots, em_N$, where N is the number of subsystems in the system as stated in G. We denote the augmented code inserted by the pre-compiler during the program file transformation by $\{\}$ brackets.

Pre-Compiler
input: F, G
output: $F', EM_1, EM_2, \ldots, EM_N$
1 analyze G and form subsystems $sub_1, \ldots, sub_N$
Transforming code of sub$_i$ processes
2 $\forall sub_i$
3 $\{$ *new* $history_i$ $\}$
(* Add new parameter, $history_i$,
to the constructor of each process in sub_i *)
4 $\forall p_j \subseteq sub_i$
5 $\{$ $linkHistory(constructor_j, history_i)$ $\}$
6 $\forall\ rt = \langle$ $event = \{var, method\}$;
$\langle$**tag**$, \{var_1, \ldots, var_k\}, \{var_{k+1}, \ldots, var_l\}\rangle$;
$pred$;
$actions = \{ra_1, ra_2, \ldots, ra_m\}$;
$trimmingRule$ $\rangle$
(* Event Driven Recovery Tuple *)
7 if $rt.event \neq \emptyset$
8 $\{$ *new global int* $rai_{rt} := 0$ $\}$
(* Safety Recovery Tuple *)
9 if $evetually\ \notin rt.pred$
10 replace $rt.event$ in F with
11 $\{$ $snapshot(var_1, ..., var_k)$
12 $temp$=$rt.event.var$
13 *if (!pred(rt.event.method(temp)))*
14 $ra_{rai_{rt}}$
15 $rai_{rt} = (rai_{rt} + 1)\%m$
16 *else*
17 $snapshot(var_{k+1}, ..., var_l)$
18 $\forall sub_k : sub_{rt.tag} \subseteq sub_k$
19 $history_k = history_k \circ \langle rt.tag, snapshot(var_1, ..., var_l)\rangle$
20 $trimmingRule()$
21 $rt.event.method(rt.event.var)$ $\}$
(* Liveness Recovery Tuple *)
22 else
23 replace $rt.event$ in F with
24 $\{$ $history_{tag} = history_{tag} \circ \langle tag, snapshot(var_1, ..., var_l)\rangle$
25 *if (rt.pred)*
26 $trimmingRule()$ $\}$
Creating external monitors
27$\forall$ sub_i
28 create an instance of external monitor for the subsystem, EM_i(code in Figure 2)

Fig. 1. Pre-compiler pseudocode.

In line 1 the pre-compiler analyzes G and forms data structures for subsystems according to the information stated in G. For each subsystem sub_i, the pre-compiler declares a new variable for the subsystem history log – $history_i$ (line 3). Then, the pre-compiler adds $history_i$ as an additional constructor parameter for each process in sub_i (lines 4-5).

Next, the pre-compiler iterates over each recovery tuple rt (line 6). If the recovery tuple has a non-empty event trigger field, the pre-compiler declares a new global variable – recovery action index rai_{rt} (lines 7-8). If the recovery tuple predicate is a safety predicate, the pre-compiler replaces the triggering event execution according to the code in lines 11-21, i.e., by taking a snapshot of some variable before the event

execution, by creating a variable *temp* and by assigning it with the current value of the variable from the triggering event and by checking if the predicate still holds with regards to the *temp* variable after the execution of the triggering event method on *temp* (lines 12-13). If the predicate does hold, the snapshot of the rest of the variables from the snapshot field is made (line 17). If the predicate does not hold, then the current recovery action is invoked and the recovery action index is updated (lines 14-15). The new entry is added to the history log of $sub_{rt.tag}$ and to all history logs of subsystems sub_k, that contain the $sub_{rt.tag}$ subsystem (line 19). Next, the trimming rule is executed (line 20). Finally, the triggering event is executed on the triggering event variable (line 21).

If the recovery tuple predicate is a liveness predicate, the pre-compiler adds code according to lines 24-26 after the execution of the triggering event. The augmented code checks the predicate; if the predicate holds, the history trimming rule is executed.

In lines 27-28, the pre-compiler creates a file called EM_i with code for the external monitor for each subsystem sub_i.

The pseudocode for the external monitor for a subsystem is presented in Figure 2. The parameter of the external monitor is the subsystem to monitor. The monitor declares a new variable rai_{rt} for each recovery tuple rt in the subsystem (lines 1-2). Next, the monitor repeatedly executes the loop in lines 3-17. For each recovery tuple in the subsystem, the monitor creates a snapshot according to the snapshot instruction field in the recovery tuple and adds the snapshot to the history log of the $sub_{rt.tag}$ subsystem and to all history logs of subsystems sub_k that contain $sub_{rt.tag}$ (lines 6-7).

```
External monitor thread
input: sub_i
(* Declaring recovery action index
        variable for each tuple *)
1  ∀ rt = ⟨event;
       ⟨tag, {var_1, ..., var_k}, {var_{k+1}, ..., var_l}⟩;
       pred;
       actions = {ra_1, ra_2, ..., ra_m};
       trimmingRule⟩ : sub_tag ⊆ sub_i
2        new int rai_rt := 0
(* Monitoring loop *)
3  do forever
4     ∀rt = ⟨event; ⟨tag, var_1, ..., var_l⟩; pred;
           actions = {ra_1, ra_2, ..., ra_m};
           trimmingRule⟩ : sub_tag ⊆ sub_i
5        snap := snapshot(var_1, ..., var_l)⟩ ∈ sub_i
6        ∀sub_k : sub_{rt.tag} ⊆ sub_k
7           history_k = history_k ∘ ⟨rt.tag, snap⟩
(* Safety Repeated Recovery Tuple *)
8        if evetually ∉ rt.pred & rt.event = ∅
9           if (!rt.pred)
10             ra_{rai_rt}
11             rai_rt := (rai_rt + 1)%m
12          else
13             trimmingRule()
(* Liveness Recovery Tuple *)
14       if eventually ∈ rt.pred
15          if ∃j, k : j ≠ k:
                history_tag[j] = history_tag[k]
                & steps_{sub_tag}(j, k)
16             ra_{rai_rt}
17             rai_rt := (rai_rt + 1)%m
```

Fig. 2. Pseudocode for an external monitor of a subsystem.

Next, if the recovery tuple predicate is a safety predicate and if the recovery tuple is intended for external monitoring, i.e., the event field is empty (line 8), the monitor checks that the predicate is satisfied (line 9). If the predicate is unsatisfied, the current recovery action is executed and the recovery action index is updated (lines 10-11). If the predicate is satisfied, the history trimming rule from the recovery tuple is applied on the subsystem history log (line 13).

If the recovery tuple predicate is a liveness predicate, the monitor checks the subsystem history log for identical entries. In case there are identical entries and between these entries the subsystem processes were scheduled to make steps, then the subsystem is in a livelock. Thus, the recovery action is executed and the recovery action index is updated (lines 14-17).

The snapshot instruction for a subsystem is equivalent to making a distributed snapshot of (part of) the system. There are several algorithms, e.g., [6], for making a distributed snapshot. Another possible solution is to enable the external monitor to request the operating system scheduler to activate solely the monitor (while not activating the processes that are the snapshot subjects) for a number of steps that suffices for executing the snapshot.

Proof outline of an automatic recovery for a transformed program. Next we prove that a system satisfies the requirements for automatic recovery with relation to the specifications after the program was transformed by the pre-compiler using the recovery tuples. The system is a collection of processes $p_1, \ldots, p_n$ with code in the file F. The processes form subsystems $sub_1, \ldots, sub_N$, where $sub_i = \{sub_{i_1}, sub_{i_2}, \ldots, sub_{i_k}\}$. In order to show that a system eventually satisfies Requirement 1, we need to demonstrate that the super-subsystem containing all other subsystems (there is such super-subsystem as we use a DAG hierarchy) respects Requirement 1. This implies that each subsystem respects Requirement 1 too. A subsystem respects its specification function if it respects the subsystem safety and the liveness requirements, i.e., if there is an execution suffix $E' = \{c_j, a_j, \ldots\}$, such that for each configuration c_k the subsystem safety predicates are satisfied and there are infinitely many configurations $c_k \in E'$ in which the liveness predicates are satisfied. Lemma 1 formalizes the claim that need to be proven for each subsystem.

Lemma 1. *Every rsf-execution has a suffix in which a subsystem sub_i eventually satisfies Requirement 1, i.e., the subsystem satisfies its safety and liveness requirements.*

Note that our framework uses event-driven approach for recording predefined state changes and for trimming the histories log for detecting liveness. The specification composer has to include event-driven snapshot instructions in order for the history log to have enough information for liveness detection. The alternative approach that does not use event driven history trimming is to use a flag variable for each recovery tuple with a liveness predicate. Initially, the flag would be set to *false*. Each time the liveness predicate variables are updated, the liveness predicate is checked. If the predicate holds, the flag is updated to be *true*. Each time an external monitor of a subsystem is scheduled, the monitor checks the flag. If the flag is *true*, the monitor executes the history trimming rule and resets the flag to *false*, so the liveness would be identified further on. If the flag is set to *false*, the history log has two or more identical entries, and the subsystem processes execute several steps, the monitor regards this situation as livelock and initiates recovery.

Lastly, we remark that the predicate verification that occurs while some predicate variables are being updated, may yield a false negative, since the predicate

will be satisfied only after the updates completion. The programmer may use *record assignment* in order to overcome this technicality. The predicate variables are stored in a record data structure *rec*. We accomplish simultaneous update of several predicate variables by creating a copy of the record, *copy*. Then, we execute all the assignments on the *copy* variable. Finally, we assign record *rec* with the updated *copy* variable.

4 Producer-Consumer Example

In this section we describe in detail how the classical producer-consumer task is enhanced by our framework. The code produced by our framework is a recovery oriented code for the producer-consumer task. We present the original code with the recovery tuples. We provide the formal correctness proof for the claim that the system with the transformed program is able to recover from any initial state automatically.

The producer-consumer task consists of two threads and a shared queue object. The producer thread repeatedly produces an item and enqueues the item into the queue. The enqueue attempt can be unsuccessful if the queue is full. The consumer thread repeatedly dequeues an item from the queue (and consumes it). The dequeue attempt can be unsuccessful if the queue is empty. The liveness requirements for the producer and the consumer processes are stated in Requirement 2 and 3 respectively.

Requirement 2 (Producer Liveness) *Every rsf-execution has a suffix in which there are infinitely many enqueue events (either successful or unsuccessful).*

Requirement 3 (Consumer Liveness) *Every rsf-execution has a suffix in which there are infinitely many dequeue events (either successful or unsuccessful).*

The producer-consumer task liveness and safety requirements are stated in Requirements 4 and 5 respectively.

Requirement 4 (Liveness) *Every rsf-execution has a suffix in which there are infinitely many successful enqueue and successful dequeue events.*

Requirement 5 (Safety) *Every rsf-execution E has a suffix $E' = c_i, a_i, ...$ in which every item dequeued by the consumer thread has been in the queue, i.e.,* $a_j = \{dequeue, item\} \Rightarrow \left[\exists k\ i \leq k \leq j : \forall l\ k \leq l < j \quad item \in c_l(queue)\right]$ *and items are dequeued in the same order they were enqueued:* $a_k = \{dequeue, item_1\}, a_l = \{dequeue, item_2\}, k < l \Rightarrow \left[\exists m, n :\ m < n < k < l \wedge\ \forall j\ m \leq j \leq k \quad item_1 \in c_j(queue)\ \wedge \forall j\ n \leq j \leq l \quad item_2 \in c_j(queue)\right]$

In this system there are two processes: the producer and the consumer threads. In addition, we have one "phantom" process: the queue object. The queue object is not a real process, but rather a collection of related variables.

We choose treating the queue object as a process. These three processes form six subsystems. The first three subsystems are the processes themselves: the producer is sub_1, the consumer is sub_2 and the queue is sub_3. The producer thread (sub_1) and the queue object (sub_3) form subsystem sub_4. The consumer thread (sub_2) and the queue object (sub_3) form subsystem sub_5. The subsystem sub_4 and sub_5 form the subsystem sub_6. The subsystems configuration graph G for the system is presented in Figure 3. Each of these subsystems has an external monitor as presented in Figure 2.

sub_1: producer
sub_2: consumer
sub_3: queue
sub_4: sub_1, sub_3
sub_5: sub_2, sub_3
sub_6: sub_4, sub_5

Fig. 3. Subsystem configuration graph for the Producer-Consumer task.

Next we present the code for the task processes. The interface for the *Queue* object is presented in Figure 4. The *Queue* object implements the *Restartable* interface (i.e., implements the function *restart()*) in order to meet the requirements of our framework. The *Queue* is implemented as a limited size cyclic queue based on an array. The *Queue* is a non-blocking queue: the *dequeue* function returns *null* if the queue is empty and *enqueue* function returns *false* if the queue is full. The *Queue* object has a public variable N – the queue capacity.

```
Queue implements Restartable
1  int N;
2  Queue(n);
3  boolean enqueue(item);
4  item dequeue();
5  void restart(){this = new Queue(N);}
```

Fig. 4. Pseudocode for the queue object.

The producer thread is implemented by the *Producer* class (Figure 5). The *Producer* class implements the *Restartable* interface. Therefore, it implements the function *restart* (lines 7-10), in which the thread is suspended and then is started again.

We denote $initHistory$ to be the function that receives a subsystem sub_i as a parameter and initializes the history logs of sub_i and of each sub_j, such that $sub_j \subseteq sub_i$.

Recovery tuple I is a liveness recovery tuple for the producer thread. The tuple predicate checks that eventually the producer makes some enqueue attempts. The event trigger variable is the queue object and the event trigger method is an invocation of the enqueue function of the queue object. Upon execution of the event the snapshot adds a new record with label "sub_1" to the history log of the producer thread. The first recovery action is to restart the producer thread (sub_1) and to initialize the producer process history. The second recovery action is to restart the whole system and to initialize all history logs. If the predicate holds, the history log of sub_1 is initialized.

The predicate of recovery tuple II is a safety predicate for sub_4 (the producer thread and the queue). As in recovery tuple I, the event trigger variable is the queue object and the event trigger method is an invocation of the enqueue function of the queue object. The snapshot records the values of the variable *success* and the hash code value of *item* immediately after the triggering event. The predicate checks whether the number of successful enqueue events in the history log of sub_4 (which equals to the number of currently enqueued items in

the queue) is less than the queue capacity. If so, the executed enqueue event must have been successful. The first recovery action is to restart sub_4 and to initialize all sub_4 history logs. The second recovery action is to restart the whole system and to initialize all history logs. The history trimming rule is empty.

The items in the queue are normally large pieces of data. Having items recorded in the history log would be expensive. Thus, we record a certain key instead of an item, e.g., hash code of the item object.

The consumer thread is similar to the producer thread. The predicate of recovery tuple I is a liveness predicate for the consumer process that checks that eventually the consumer makes some dequeue attempts. The event trigger variable is the queue object and the event trigger method is an invocation of the dequeue function of the queue object. The first recovery action is to restart the consumer process (sub_2) and to initialize the history log of sub_2. The second recovery action is to restart the whole system and to initialize all history logs. If the predicate holds, the history log of sub_2 is initialized.

```
Producer implements Restartable
//Liveness for sub_1
I⟨queue.enqueue;
   ⟨"sub_1", {}, {}⟩;
   eventually ⟨"sub_1"; {}; {}⟩ ∈ history_sub_1;
   {{this.restart(); initHistory(sub_1);},
   {queue.restart(); this.restart();
      consumer.restart(); initHistory(sub_6);}};
   {initHistory(sub_1);}⟩
//Safety for sub_4
II⟨queue.enqueue;
   ⟨"sub_4", {}, {success, item.hashCode()}⟩
   |⟨"sub_4"; {}; {true, ¬null}⟩ ∈ history_sub_4|
      < queue.N ⇒ success;
   {{this.restart(); queue.restart();
      initHistory(sub_4)};
   {queue.restart(), this.restart(),
      consumer.restart(); initHistory(sub_6);}};
   {}⟩
1  Producer(Queue queue, Consumer consumer);
2  void run() {
3     do forever
4        item = produce_item();
5        success=queue.enqueue(item);
6  }
7  void restart() {
8     this.suspend();
9     this.start();
10 }
```

Fig. 5. Pseudocode for the producer thread.

The predicate of recovery tuple II is a safety predicate for sub_5 (the consumer thread and the queue). The event trigger variable is the queue object and the event trigger method is an invocation of the dequeue function of the queue object. The predicate checks whether the number of the history entries of sub_4 that reflect successful enqueue events is bigger than zero (i.e., the number of currently enqueued items in queue is bigger than zero). If so, the last dequeue event should have been successful. The first recovery action is to restart sub_5 and to initialize the history logs of sub_5. The second recovery action is to restart the whole system and to initialize all of the history logs. The history trimming rule is empty.

The predicate of recovery tuple III is the safety predicate for the whole system. The event trigger variable is the queue object and the event trigger method is an invocation of the dequeue function of the queue object. The predicate checks that each dequeued item has been previously enqueued and that the dequeued item is the first successfully enqueued item from the current queue. The recovery action is a restart of the whole system and an initialization of all

history logs. The trimming rule is to remove the enqueue event entries of the successfully dequeued item from the history log of sub_6.

The correctness proof of the producer-consumer task is based on the guidelines provided in Lemma 1. The proofs for Lemmas 2, 3 and 4 appear in [3].

Lemma 2 (Liveness of Producer Thread). *In any rsf-execution E the producer thread executes a call for the queue.enqueue function infinitely often.*

Lemma 3 (Liveness of Consumer Thread). *In any rsf-execution E, the Consumer thread executes a call for the queue.dequeue function infinitely often.*

Lemma 4 (Producer-Consumer Task Correctness). *The Producer-Consumer task that uses the pseudocode presented in Figures 4, 5, and 6, eventually satisfies Requirements 4 and 5.*

The code produced by our framework for the producer-consumer task is self-stabilizing only if the uniqueness of the object hash codes is guaranteed. Otherwise, the produced code is pseudo self-stabilizing [9] with regards to the safety property as explained in [3].

```
Consumer implements Restartable
//Liveness for sub_2
I⟨queue.dequeue;
   ⟨"sub_2"{};{}⟩
   eventually ⟨"sub_2"{};{}⟩ ∈ history_sub_2;
   {{this.restart(); initHistory(sub_2)},
   {queue.restart(); this.restart();
      consumer.restart(); initHistory(sub_6)}};
   {initHistory(sub_2)}⟩
//Safety for sub_5
II⟨queue.dequeue;
   ⟨⟩
   |⟨"sub_4";{};{true, item.hashCode}⟩ ∈ history_sub_4| > 0 ⇒
      item ≠ null;
   {{this.restart(); queue.restart(); initHistory(sub_5)},
   {queue.restart(); this.restart();
      consumer.restart(); initHistory(sub_6)}};
   {}⟩
Safety for sub_6
III⟨queue.dequeue;
   ⟨⟩
   item ≠ null ⇒
   ∃i : history_sub_6[i] = ⟨"sub_4", {}, {true, hashCode_item}⟩∧
   ∀j < i history_sub_6[j] = ⟨"sub_4", {}, {false, hashCode_item}⟩
   {{queue.restart(); this.restart(); consumer.restart();
         initHistory(sub_6)}}
   {history_sub_6 = history_sub_6 \ history_sub_6 [1, ..., i]}⟩
1  Consumer(Queue queue, Producer producer);
2  void run(){
3     do forever
4        item = queue.dequeue();
5        consume_item(item);
6  }
7  void restart() {
8     this.suspend();
9     this.start();
10}
```

Fig. 6. Pseudocode for the consumer thread.

5 Conclusions

In this work we have combined fault tolerance paradigms such as self-stabilization and (eventual) Byzantine faults with the restartability recovery paradigm into a single framework for writing recovery oriented programs.

We view the new framework as an important infrastructure that allows the specification composer to monitor the specifications on-line and to act upon violation of the safety and the liveness specifications. There is no doubt that such an approach is vitally important for gaining autonomous, robust and fault-tolerant systems.

Acknowledgment: We thank Marcelo Sihman for discussions during the first stage of this research.

References

1. A. Arora and M. Theimer. "On Modeling and Tolerating Incorrect Software". Microsoft Research Technical Report MSR-TR-2003-27, 2003.
2. K. Beck, C. Andres. " Extreme Programming Explained : Embrace Change". Second Edition, Addison-Wesley, 1999.
3. O. Brukman, S. Dolev. "Recovery Oriented Programming". Technical Report #06-06, Department of Computer Science, Ben-Gurion University, Israel, June 2006.
4. O. Brukman, S. Dolev, E. K. Kolodner. "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software". *Proc. of the IEEE SWSTE*, pp. 20-29, 2003.
5. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll. "An overview of JML tools and applications". *International Journal on Software Tools for Technology Transfer*, vol. 7(3), pp. 212-232, June 2005.
6. K. M. Chandy, L. Lamport. "Distributed snapshots: Determining global states of distributed systems". *ACM TOCS*, vol. 3(1), pp. 63-75, February 1985.
7. F. Chen, G. Rosu. "Java-MOP: A Monitoring Oriented Programming Environment for Java". *Proc. of the TACAS* , pp. 546-550, Edinburgh, U.K., April 2005.
8. R. L. Constable, T. B. Knoblock, J. L. Bates . "Writing Programs that Construct Proofs ". *Journal of Automated Reasoning*, vol. 1(3), pp. 285-326, 1984.
9. S. Dolev. *Self-stabilization*. The MIT press, March 2000.
10. S. Dolev, J. L. Welch. "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults". *Journal of the ACM*, vol. 51(5), pp. 780-799, September 2004.
11. S. Dolev, R. Yagel. "Toward Self-Stabilizing Operating Systems". *Proc. of the SAACS* , pp. 684-688, 2004.
12. *Thinking in Java*. Prentice Hall PTR, December 2002.
13. Eiffel. Eiffel Programming Language. http://www.eiffel.com.
14. D. P. Friedman, M. Wand, C. T. Haynes. "Essentials of Programming Languages". The MIT press, 2nd edition, 2001.
15. Y. Gurevich, B. Rossman, W. Schulte. "Semantic Essence of AsmL". Microsoft Research Technical Report MSR-TR-2004-27, March 2004.
16. L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". *ACM Trans. on Programming Languages and Systems*, vol. 4(3), pp. 382-401, 1982.
17. W. Leal, A. Arora. "Scalable self-stabilization via composition". *Proc. of the ICDCS*, Tokyo, Japan, March 2004.
18. N. Lynch *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
19. P. G. Neumann. "Computer-Related Risks". Addison-Wesley/ACM Press, 1995.
20. D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. "Recovery Oriented Computing(ROC): Motivation, Definition, Techniques and Case Studies". UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002.
21. B. Randell, J. Xu. "The Evolution of the Recovery Block Concept". *Software Fault Tolerance*, pp. 1-22, 1994.
22. T. Rothamel, Y. A. Liu, C. L. Heitmeyer, E. I. Leonard. "Generating Optimized Code from SCR Specifications". *Proc. of the LCTES*, pp. 135-144, Ottawa, Ontario, Canada ,June 2006.
23. J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zoro. "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions". *IEEE Transactions on Computers*, vol. 51(2), pp. 164-179, 2002.