A compositional framework for fault tolerance by specification transformation

Doron Peled
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Mathai Joseph*
Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

Abstract


The incorporation of a recovery algorithm into a program can be viewed as a program transformation, converting the basic program into a fault-tolerant version. We present a framework in which such program transformations are accompanied by a corresponding specification transformation which obtains properties of the fault tolerant versions of the programs from properties of the basic programs. Compositionality is achieved when every property of the fault tolerant version can be obtained from a transformed property of the basic program.

A verification method for proving the correctness of specification transformations is presented. This makes it possible to prove just once that a specification transformation corresponds to a program transformation, removing the need to prove separately the correctness of each transformed program.

1. Introduction

A fault tolerant program can be viewed as the result of superposing a recovery algorithm on a basic, non-fault tolerant program. Thus, adding a recovery algorithm can be seen as a program transformation, producing a program that can recover from faults in its execution environment. The combined execution of the basic program and the recovery algorithm ensures that the fault tolerant program achieves its goal.
Despite the occurrence of specified faults. Many different recovery algorithms have been devised [4,13] and these are often generic in the sense that they can be superposed over (or combined with) many different basic programs.

Recovery algorithms may differ from one another in both the method used to achieve fault tolerance and in the kind of immunity they guarantee: for example, in their ability to handle multiple faults, the type of faults for which recovery is possible [26], and the amount of computation that is lost or required to be repeated during recovery [4]. A recovery algorithm can therefore be characterized by the kind of recovery that will take place after the occurrence of a fault and the kind of faults for which such immunity can be guaranteed. But it is important also to define the exact relation between the properties of a basic program and the properties of the fault tolerant program obtained after the superposition of a recovery algorithm.

In this paper, we describe a framework for reasoning about distributed fault tolerant programs based on a transformational approach to representing faults and recovery algorithms. We show how the properties of recovery algorithms can be specified and how to obtain the relation between the behavior of a basic program and the version with recovery superposed. We define a criterion for correctness based on the use of a specification transformation, which transforms any property of the basic program into a property of its fault tolerant version (i.e. after superposing the recovery algorithm). This approach is especially beneficial when the behavior of the basic program is closely related to that of its fault tolerant version. For example, this is the case in fail-stop recovery [26], where the normal execution of the program temporarily stops when a failure occurs and recovery actions restore the program state using values stored in some nondestructive storage.

This framework has an important practical advantage, as a relatively complete verification method is available to verify that a specification transformation corresponds to a program transformation. The specification transformation needs to be verified only once and can then be used to obtain properties of the transformed program from the properties of the basic program.

The method is compositional: the specification transformation can be embedded in a compositional proof rule so that proving the properties of a fault-tolerant version of a basic program can be performed by proving properties of the basic program and then applying the specification transformation. For example, it allows deducing which classes of properties (e.g. mutual exclusion, eventual grant of a service) are preserved after the superposition.

Specification transformations provide also a means of generically specifying fault tolerance in programs, rather than having to specify the behavior of each such transformed program separately. It may be observed that specification transformations corresponding to program transformations can be used for other applications of program superposition, such as for deadlock detection [2].

In Section 2, some preliminary notions concerning programs, temporal logic and interleaving semantics are reviewed. Specification transformations are introduced in Section 3. Examples of the specification of generic fault tolerance program
transformations are given in Section 4. In Section 5, a verification method for specification transformations is presented. Some particular difficulties of distributed fault tolerance are treated in Section 6, and Section 7 concludes with a discussion and suggestions for further research.

2. Representing programs and their properties

A program $P$ is a triple $\langle \tilde{y}, T, \Theta \rangle$, where $\tilde{y}$ is a finite, ordered set of variables $y_1, y_2, \ldots, y_m$, $T$ is a finite set of atomic operations, and $\Theta$ is a (first-order) initial condition.

Each variable $y_i \in \tilde{y}$ is defined over some domain. The Cartesian product $S$ of these domains is called the state space of the program. A state $s$ of a program is an assignment function from the set of variables to their appropriate domains. Let $s \models \varphi$ denote that the (first-order) formula $\varphi$ is satisfied by the state $s \in S$. In the sequel, first-order formulas will have free variables from $\tilde{y}$ (e.g. $\varphi$ is actually $\varphi(\tilde{y})$), unless denoted otherwise.

An operation $\tau \in T$ is defined using transition formulas [17, 21] that specify the relation between the old values $\tilde{y}$ of the set of variables before executing the operation and the set of new values $\tilde{y}'$ after its execution. For an operation $\alpha$ we write $[\mu_\alpha(\tilde{y}, \tilde{y}')]_s$, where $\mu_\alpha(\tilde{y}, \tilde{y}')$ is a formula that is interpreted over pairs of states $\langle s, s' \rangle \models \mu_\alpha(\tilde{y}, \tilde{y}')$ such that the variables $\tilde{y}$ are assigned values according to the state $s$, and $\tilde{y}'$ according to the state $s'$. The variables $z_\alpha$ are the only variables that may be changed by $\alpha$ (i.e. if $x \notin z_\alpha$, then $s' (x) = s(x)$). It is assumed that there is no contradiction between $\mu_\alpha$ and $\varphi$ in the sense that all the variables that may change according to $\mu_\alpha$ also appear in $\varphi$. Then, for any pair of states $s, s'$ such that $s'$ is obtained from $s$ by executing $\alpha$, it holds that $\langle s, s' \rangle \models \mu_\alpha(\tilde{y}, \tilde{y}') \land \bigwedge_{y_i \in \tilde{y} \setminus z_\alpha} y'_i = y_i$.

The initial condition $\Theta$ characterizes the initial states of the program, i.e. $s \models \Theta$ iff the program can begin from the state $s$. The enabling condition of an operation is a first order formula over free variables from $\tilde{y}$ that holds for the states in which that operation can be executed, i.e. $en_\alpha = \exists \tilde{y}' \mu_\alpha(\tilde{y}, \tilde{y}')$ where $\exists \tilde{y}'$ is a shorthand for $\exists y_1 \exists y_2 \ldots$

Representing a concurrent program by a set of operations is convenient both for verification and for defining different kinds of program transformations. "Actual" programs are written in a programming language which incorporates various specific features, such as the use of shared variables or synchronous or asynchronous communication. To give a uniform treatment of fault tolerance, we shall assume that such programs are translated into sets of operations and ignore the original language used. Examples of the translation of concurrent programs written in languages with shared variables and inter-process communication can be found in [20].

Definition 2.1. An execution sequence (or an interleaving sequence) of a program is an infinite sequence of states $\sigma = s_0, s_1, s_2, \ldots$, satisfying the following conditions:
(1) \( s_0 \models \Theta \), where \( \Theta \) is the initial condition,
(2) for each \( i \geq 0 \), either \( \sigma_i = \sigma_{i+1} \), or there exists some \( \tau \in T \) such that \( \langle s_i, s_{i+1} \rangle \models \mu_{\tau}(\bar{y}, \bar{y}') \), and for each \( x \in \mathcal{Z} \), \( s_i(x) = s_{i+1}(x) \), and
(3) \( s_n \models \bigwedge_{\tau \in T - \mathcal{T}} \text{en} \), iff for each \( m > n \), \( s_m = s_n \) ("basic liveness assumption").

The semantics of a program \( P \) is denoted by the function \( \mathcal{M}[P] \) and is the set of interleaving sequences of \( P \).

Sometimes, an additional condition called "fairness" [8] further restricts the set of interleaving sequences that are considered to be executions of the program. Assume initially that there are no fairness restrictions; we consider later how fairness can affect fault tolerance. We begin by using linear temporal logic (LTL) [20] as the specification language over the execution sequences of the program. We shall later give an alternative interpretation of LTL over equivalence classes of interleaving sequences.

In LTL, \( \varphi \) means that \( \varphi \) holds in every future state, and \( \varphi U \psi \) that \( \varphi \) will continue to hold unless and up to the moment where \( \psi \) holds in the future. \( \varphi W \psi \) is stronger than \( \varphi U \psi \) and requires also that \( \psi \) will actually hold in the future. We allow quantification (i.e. \( \exists z \varphi \), \( \forall z \varphi \)) over rigid variables, whose value is fixed over all states, and over flexible variables, whose value can change from state to state. The formal definition of LTL can be found in the Appendix.

A property can be seen either as a temporal formula \( \varphi \), or as the set of sequences \( [\varphi] \) that satisfy \( \varphi \). Thus, a program \( \varphi \) has a property \( \varphi \) iff \( \mathcal{M}[P] \subseteq [\varphi] \). This is also denoted by \( P \models \varphi \).

**Definition 2.2** (Manna and Pnueli [20]). An exact specification \( \Phi_P \) of a program \( P \) is a formula that is satisfied exactly by the set of interleaving sequences of \( P \), i.e. \( \mathcal{M}[P] = [\Phi_P] \).

It is easy to see that for every formula, \( \varphi \), \( P \models \varphi \) iff \( \Phi_P \rightarrow \varphi \).

Exact specifications can be used instead of the code of the program. In fact, in Lamport's Temporal Logic of Actions (TLA) [17] the code of the program is described as its exact specification in a canonical form. Using the same notation for both programs and specification is appealing in the context of specification transformation. However, we will still make the more widely used traditional distinction between programs and specification.

3. Formalizing recovery algorithms with specification transformations

Consider a basic program that is designed to perform a computational task in a fault-free system. A generic recovery program is one which can be combined with

1 "Stuttering" [15] is allowed in order to achieve compositionality.
various basic programs to give programs which execute "correctly" when certain faults occur. Each recovery algorithm is intended to overcome some specific kinds of faults. Obviously, when faults do occur, the combined program will not behave exactly as the original basic program behaves when no faults occur. This section formalizes what it means for the combined program to be executed correctly under the faulty system.

The executions of the combined program (in a fault-prone system) are related to the executions of the basic program (in a fault-free system) in a particular way that is guaranteed by the recovery algorithm. The relation can be viewed in terms of the transformation of the properties of the basic program to the properties of the combined program, when each program is executed under its specified environment. This relation will be formalized as a specification transformation, which takes a specification of the basic program and produces a specification of the combined program.

3.1. Program transformations and specification transformations

The incorporation of a recovery algorithm into a basic program is a superposition which takes the union of the operations of the two programs and strengthens the enabledness conditions of some selected operations of the basic program. Various other forms of superpositions (also called super-imposition) have been discussed, e.g., in [3,6]. In this way, operations that come from the superposed program are executable in the combined program, and further the superposed program controls the execution of the operations that come from the basic program. The superposition transformation enlarges the original state space $S$ to a related space $\tilde{S}$ which is the Cartesian product of more domains than are included in $S$.

Superposition is an example of a program transformation. A program transformation $\mathcal{F}$ is a function $\mathcal{F} : P \mapsto P$, where $P$ is a domain of programs (e.g. programs written in a certain language, or programs given as sets of operations that satisfy certain syntactical conditions).

Definition 3.1. A program transformation $\mathcal{F}$ is monotonic if for each pair of programs $P_1, P_2 \in P$, if $\mathcal{M}[P_1] \subseteq \mathcal{M}[P_2]$ then $\mathcal{M}[\mathcal{F}(P_1)] \subseteq \mathcal{M}[\mathcal{F}(P_2)]$.

Not every program transformation is monotonic. For example, consider the following program transformation $\mathcal{F}(\langle \{x, y\}, T, \Theta \rangle) = \langle \{x, y\}, T \cup \{y : \langle y' = y + 1 \rangle_{[x, y]}\}, \Theta \rangle$. Given $P_1 = \langle \{x, y\}, \{x : \langle y = 0 \land x' = x + 1 \rangle_{[x]}\}, x = 0 \land y = 0\rangle$, and $P_2 = \langle \{x, y\}, \{\beta : \langle x' = x + 1 \rangle_{[x]}\}, x = 0 \land y = 0\rangle$; it is evident that $\mathcal{M}[P_1] = \mathcal{M}[P_2]$ and thus also that $\mathcal{M}[\mathcal{F}(P_1)] \subseteq \mathcal{M}[\mathcal{F}(P_2)]$. However, $\mathcal{M}[\mathcal{F}(P_1)] \subset \mathcal{M}[\mathcal{F}(P_2)]$. The reason is that $en_x = y = 0$, while $en_x = T$. This is not reflected in the semantics of $P_1$ and $P_2$ as $y$ is unchanged during any execution of these programs. However, when the operation $\gamma$ is added, the value of $y$ can change at some point in the execution of $P_1$, preventing $x$ from being further executed, while in $P_2, \beta$ can still occur.
A specification transformation $\mathcal{F}$ is a function $\mathcal{F} : \mathcal{L} \rightarrow \mathcal{L}$, where $\mathcal{L}$ is the set of formulas in some specification language such as LTL.

**Definition 3.2.** A specification transformation $\mathcal{F}$ corresponds to a program transformation $\mathcal{T}$ if it satisfies the condition that for each program $P$, if $P \models \psi$, then $\mathcal{T}(P) \models \mathcal{F}(\psi)$ (see Fig. 1).

Specification transformations can be used to verify properties of the transformed program $\mathcal{F}(P)$, from verified properties of the program $P$. Proving $\mathcal{T}(P) \models \psi$ is then reduced to proving a property of $P$, applying the specification transformation and doing some pure LTL verification. There is no need to use the actual code of $\mathcal{F}(P)$ in the verification. The proof uses the following proof rule:

\[
\begin{align*}
1 & \quad P \models \varphi \\
2 & \quad \mathcal{F}(\varphi) \rightarrow \psi \\
\hline
\mathcal{T}(P) \models \psi
\end{align*}
\]

(1)

Ideally, the transformation should make it possible to derive logically every property of the transformed program from the basic program (cf. Zwiers' compositional completeness [27]). That is, the proof rule (1) needs to be applicable to every formula $\psi$ that holds in $\mathcal{T}(P)$. The following (relative) completeness definition applies to transformations in which every property of the transformed program can be proved from the basic program and the transformation. This completeness is relative to the ability of the temporal language used here to formulate such an appropriate formula $\varphi$ and proving temporal properties of programs (as needed in premise 1, $P \models \varphi$), and the ability to prove LTL tautologies, (as required in premise 2, $\mathcal{F}(\varphi) \rightarrow \psi$).

**Definition 3.3.** A specification transformation $\mathcal{F}$ is said to be compositional complete with respect to a program transformation $\mathcal{T}$ if for every program $P$ and every formula $\psi$ such that $\mathcal{T}(P) \models \psi$ there exists a formula $\varphi$ such that $P \models \varphi$ and $\mathcal{F}(\varphi) \rightarrow \psi$. 
The following two lemmas show that compositional completeness is highly dependent upon the monotonicity of the program transformation.

**Lemma 3.4.** If a specification transformation $\mathcal{F}$ that corresponds to a program transformation $\mathcal{F}$ is nonmonotonic, then $\mathcal{F}$ cannot be compositional complete w.r.t. $\mathcal{F}$.

**Proof.** Let $\mathcal{F}$ be a nonmonotonic program transformation. Then, there exists a pair of programs $P_1$ and $P_2$ with $\mathcal{M}[P_1] \subseteq \mathcal{M}[P_2]$ such that $\mathcal{M}[\mathcal{F}(P_1)] \not\subseteq \mathcal{M}[\mathcal{F}(P_2)]$. Let $\mathcal{F}$ be a specification transformation that corresponds to $\mathcal{F}$. Let $\psi_2$ be an exact specification of $\mathcal{F}(P_2)$. Assume compositional completeness of $\mathcal{F}$ w.r.t. $\mathcal{F}$. Then, there exists a property $\varphi$ of $P_2$ such that $\mathcal{F}(\varphi) \rightarrow \psi_2$. Since $\mathcal{M}[P_1] \subseteq \mathcal{M}[P_2]$, it is also the case that $P_1 \models \varphi$.

Since $\mathcal{F}$ corresponds to $\mathcal{F}$, $\mathcal{F}(P_2) = \mathcal{F}(\varphi)$. From the fact that $\psi_2$ is an exact specification of $\mathcal{F}(P_2)$, $\psi_2 \rightarrow \mathcal{F}(\varphi)$ and, therefore, $\mathcal{F}(\varphi) \leftrightarrow \psi_2$, i.e. $\mathcal{F}(\varphi)$ is also an exact specification of $\mathcal{F}(P_2)$. However, since $\mathcal{M}[\mathcal{F}(P_1)] \not\subseteq \mathcal{M}[\mathcal{F}(P_2)] = \{\psi_2\}$, then $\mathcal{F}(P_1) \models \psi_2$ and, thus, $\mathcal{F}(P_1) \models \mathcal{F}(\varphi)$, which contradicts the fact that $\mathcal{F}$ corresponds to $\mathcal{F}$. □

**Definition 3.5.** A specification transformation $\mathcal{E}$ is exact w.r.t. a program transformation $\mathcal{F}$ if for each property $\varphi$, $\mathcal{E}(\varphi)$ is a formula that is satisfied exactly by the execution sequences $\bigcup \{\mathcal{M}[\mathcal{F}(P)] \mid P \models \varphi \}$. The converse of Lemma 3.4 depends on the expressiveness of the specification language used as stated concisely in the following lemma.

**Lemma 3.6.** Compositional completeness can be achieved for monotonic program transformations, given a specification language that is expressive enough to formulate exact specification transformations.

**Proof.** Let $\varphi$ be an exact specification of a given program $P_1$. Then $\mathcal{E}(\varphi)$ is satisfied exactly by the sequences $E = \bigcup \{\mathcal{M}[\mathcal{F}(P)] \mid P \models \varphi \}$. Since $\{\varphi\} = \mathcal{M}[P_1]$, it holds that $E = \bigcup \{\mathcal{M}[\mathcal{F}(P)] \mid P \models \varphi \}$. Because of monotonicity, $E = \mathcal{M}[\mathcal{F}(P_1)]$, which are exactly the sequences of the transformed version $\mathcal{F}(P_1)$ of the program $P_1$. Thus, $\mathcal{E}$ transfers an exact specification of a program $P_1$ to an exact specification of $\mathcal{F}(P_1)$. For proving that $\mathcal{F}(P_1) \models \mathcal{E}(\varphi)$, it is always possible to formulate an exact specification $\eta$ of $P_1$, and apply $\mathcal{E}$ on $\eta$ to obtain a formula that is an exact specification of $\mathcal{F}(P_1)$ and, this implies $\psi$, as required in Definition 3.3. □

### 3.2. Specification transformation for fault tolerance

To represent failure and recovery, superposition can be done in two stages [19] (see Fig. 2).

1. **Superposing the recovery algorithm.** First the recovery algorithm, represented by a set of operations, is superposed on the basic program.
(2) **Superposing the failure mechanism.** As the second step, a "fault program", which represents faults by a set of operations, is superposed on the program obtained in the first step.

This may seem surprising at first: a recovery algorithm is normally seen as a procedure which is intended to be superposed on a program, but faults are not represented as pieces of code. However, if faults are considered as events that "interfere" with the execution of a program [7] they can be represented by a set of operations and this is a convenient way to avoid the need to deal with faults at the semantic level (see [19] for a detailed discussion). By encoding faults as occurrences of specially constructed operations, we have the flexibility of using a single semantics for various different types of faults, as well as for fault-free executions, rather than dealing with a specialized semantics for each kind of fault.

Let the basic program be denoted by $P$. Let the superposition of a recovery algorithm on a basic program $P$ give $R = F_1(P)$ and let the superposition of the failure mechanism on $R$ give $F_2(R)$. Then the entire transformation is $F(P) = F_2(F_1(P))$. In the rest of this paper, we apply both transformations, namely, the recovery algorithm and the failure mechanism, in this order, on basic programs.

The first step in formalizing a recovery algorithm is to specify those of its properties that are independent of the basic program: the recovery operations will have these properties regardless of the basic program to which they are superposed. Such properties may include, for example, the guarantee that recovery will be eventually completed (perhaps conditional on the type and frequency of the faults), the non-blocking of the basic program by the addition of the recovery actions, the ability to overcome multiple faults, and other such fixed properties. It may also be asserted that recovery will occur within a bounded time (e.g. using a real time temporal logic such as [10]). We call such properties the **fixed part** of the specification. The fixed part defines the responsiveness of the recovery algorithm to the occurrence of failures. It does not describe the effect of recovery on the program, but rather the behavior of the
recovery algorithm itself when superposed on a basic program. This includes recovery actions (such as maintaining consistent checkpoints by taking periodic snapshots) that are executed when no failure occurs in order to achieve fault tolerance.

The second part of the formalization is a transformation $\mathcal{F}$ which converts properties of the basic program into properties of the fault-tolerant version. Given a basic program $P$ which satisfies a certain property, applying the transformation $\mathcal{F}$ to $P$ results in a program which satisfies a new specification. The transformation $\mathcal{F}$ is applied to specifications, so that for each $\phi$ satisfied by $P$, $\mathcal{F}(\phi)$ is a specification of $\mathcal{F}(P)$ (i.e. $\mathcal{F}$ corresponds to $\mathcal{F}$). The fixed part of the specification may be embedded as a constant part of every formula obtained from the specification transformation.

Different behaviors can be guaranteed for different fault situations. For example, superposition of a particular recovery algorithm on a basic program might preserve some properties when assumptions are made to limit the repetitive occurrence of faults. If, however, faults do occur during recovery, some weaker properties of the basic algorithm are preserved. In general, the effect of the transformation can be conditioned by the nature and the frequency of occurrence of faults. For example, some liveness properties may be maintained whenever only a finite number of faults occur, but not otherwise.

3.3. Concealment

In many cases, the internal details of the recovery algorithm are of less interest than the effect of the recovery on the basic program, so it would be useful to suppress these details. Hiding operations and program variables are essential to specification methods such as process algebras or refinement mappings. In temporal logic, this can be defined using a state function $\mathcal{C}$ which filters out irrelevant parts of the state, e.g. by removing references to variables that are not of interest. Then a specification can be given with respect to the filtered state space, rather than the original state space. But for more complex kinds of program transformations, the filtering may need to be more complicated, e.g. filtering out the sending and receiving of control messages that are needed for the recovery but not for the basic program.

The state function $\mathcal{C}$ can be generalized in the obvious way to a function on sequences of states and on sets of sequences. If $\mathcal{M}[P]$ is the semantic representation of program $P$ as a set of interleaving sequences, then $\mathcal{C}(\mathcal{M}[P])$ represents the result of applying the mapping $\mathcal{C}$ to conceal unwanted detail. The function $\mathcal{C}$ can be chosen appropriately to reveal all the information about failure and recovery, or some or none of this information. Note that it is possible that a program transformation is not monotonic but becomes monotonic under a given concealment.

The exact internal details of the recovery part of the combined program are also not of great interest and can thus be ignored in favor of observing only the effect of superposition on the basic program. Thus, compositional completeness can instead be weakened, and defined with respect to the concealment of the superposed program, i.e. by replacing $\mathcal{F}(P)$ with $\mathcal{C}((\mathcal{F}(P))$ in Definition 3.3.
4. Examples of specification transformations

A typical recovery algorithm restores a program to a consistent state after a fault has occurred so that its execution can be resumed. A common way to achieve this is to preserve a checkpoint or a snapshot, which is a global state of the basic program $P$ to which execution can be restored upon recovery. However, different recovery algorithms will have different ways of establishing the relation between this global state and the state in which the failure occurred. Some possible schemes are to permit recovery to one of the following states:

1. a state in the past of the current interleaving sequence,
2. a state in a possible future of the last global state of the system before the most recent failure occurred.

As will be shown in Section 6, in distributed programs it is sometimes necessary to weaken the conditions for a snapshot so that it can be taken efficiently. This will require special treatment, and will not correspond directly to either of the two cases mentioned above.

4.1. Backward recovery

Backward recovery allows a program suffering a failure to resume execution from a previously saved checkpoint. After recovery, a nondeterministic program will not necessarily repeat its previous execution in the interval between the time the checkpoint was established and the time the failure occurred. We shall illustrate the use of specification transformations for backward recovery using the example of a program subject to fail-stop faults during its execution.

First, the operations of the basic program are augmented by superposing an interrupt mechanism: the enabling conditions of both sets of operations (of the basic program and the interrupt mechanism) are strengthened by testing the value of the boolean variable $main$. Each enabling condition $en_i$ of a basic program operation $r$ is strengthened to $en_i \land main$ which requires now $main$ also to be $T$, corresponding to the case where there is no interrupt. On the occurrence of an interrupt, $main$ is set to $F$ to disable the basic program operations and enable the interrupt operations. When interrupt processing is over, $main$ is reset to $T$. Thus the addition of such an interrupt mechanism fits in well with the superposition framework.

Assume that during the normal execution of the program a timer interrupt periodically initiates the saving of consistent states of the program in a nonvolatile, nondestructive stable memory. On the occurrence of a fail-stop fault [26], the contents of the main memory are assumed to be destroyed, and an interrupt triggers the execution of a failure interrupt handler.

\[^{2}\text{In a distributed system this may be expensive to implement and thus an alternative model of computation is defined in Section 6.}\]
Consider the following recovery algorithm. The stable memory is partitioned into two segments, each having a dedicated area for every program variable. When a new snapshot is to be taken, each variable \( x \) is saved in either area 1 or area 2. A snapshot is taken when an interrupt handler is triggered by a periodic timer. The handler stores the current contents of the entire program memory alternatively in area 1 or area 2, according to the value of a special variable \( \text{var\_set} \), which also resides in stable memory. The value in this variable is altered only after the entire contents of memory have been saved, so that a failure which occurs while the memory is being saved does not affect the last complete saved copy. Executing the transformed program begins after initializing Area 1 to hold a copy of main memory's initial values and setting \( \text{var\_set} \) to 1.

<table>
<thead>
<tr>
<th>Timer interrupt handler</th>
<th>Failure interrupt handler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>on timer_interrupt do</strong></td>
<td><strong>on failure_interrupt do</strong></td>
</tr>
<tr>
<td>disable timer_interrupt;</td>
<td>disable timer_interrupt;</td>
</tr>
<tr>
<td>if ( \text{var_set} = 2 ) then</td>
<td>if ( \text{var_set} = 1 ) then</td>
</tr>
<tr>
<td>for each variable ( x ) do</td>
<td>for each variable ( x ) do</td>
</tr>
<tr>
<td>write ( x ) to area 1;</td>
<td>read ( x ) from area 1;</td>
</tr>
<tr>
<td>( \text{var_set} := 1; )</td>
<td>else</td>
</tr>
<tr>
<td>for each variable ( x ) do</td>
<td>read ( x ) from area 2;</td>
</tr>
<tr>
<td>write ( x ) to area 2;</td>
<td>enable timer_interrupt;</td>
</tr>
<tr>
<td>( \text{var_set} := 2; )</td>
<td>end handler;</td>
</tr>
<tr>
<td>enable timer_interrupt;</td>
<td></td>
</tr>
<tr>
<td><strong>end handler:</strong></td>
<td><strong>end handler:</strong></td>
</tr>
</tbody>
</table>

The interrupt handler for a failure disallows timer interrupts (so there is no saving of memory when a failure occurs) and copies back the contents of the last correctly saved segment of stable memory to the main memory. Note that a failure can occur during the execution of the timer interrupt handler, or while recovering from a previous failure. The transformation that combines the interrupt handlers with basic program, and takes care of an appropriate initialization, can be shown to be monotonic.

We define the following special state predicates. The specification (transformation) of the fault-tolerant version of the program will conceal the internal variables of the interrupt handlers and the interrupt timer, revealing only the values of these predicates and the basic program variables.

- \( f (\text{fault}) \): A fault occurs in this state (a single fault may hold through multiple states).
- \( r (\text{recovery}) \): Recovery has begun by the occurrence of a fault, but has not yet been completed.
- \( s (\text{snapshot}) \): States in which a snapshot is taken.
- \( c (\text{checkpoint}) \): A new checkpoint is established just after a snapshot was successfully taken. This predicate holds when a new snapshot is available and the execution of the basic computation is just about to resume.
w (wasted): A fault will occur before the next checkpoint is reached and hence the present state of the computation will be wasted. Suffixes of an execution sequence from some w-state satisfy the temporal property $\neg r \land (r \Rightarrow \exists r)$ and thus w can be seen as a shorthand for this formula.

Let a state in which the boolean state function $x$ holds be called an $x$-state and a finite sequence of $x$-states be called an $x$-interval.

The transitions between states with different values of the state functions are shown in Fig. 3. In each state, only state predicates with value T are listed; e.g. the state "j; r" represents $\neg w \land f \land \neg c \land r \land \neg s$. The symbol $\epsilon$ represents a state in which all the above predicates are F. Each node has a self-loop (hence, it is not possible to distinguish stuttering). In addition, no infinite sequence of states in any execution corresponds to an infinite path of dashed arrows (e.g. it is not possible that from some point in the computation 's, w' will hold forever). Figure 3 distinguishes the following combinations of state functions, abbreviated as $x_1, \ldots, x_7$:

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr.</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress_State</td>
<td>$x_1$</td>
<td>$\neg w \land f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>Snapshot_Taking</td>
<td>$x_2$</td>
<td>$\neg w \land f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>Checkpoint_Established</td>
<td>$x_3$</td>
<td>$\neg w \land f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>No_Effective_Progress</td>
<td>$x_4$</td>
<td>$w \land \neg f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>Fault_Occurs</td>
<td>$x_5$</td>
<td>$\neg w \land f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>Recovery_In_Progress</td>
<td>$x_6$</td>
<td>$\neg w \land f \land \neg c \land r \land \neg s$</td>
</tr>
<tr>
<td>Incompleted_Snapshot</td>
<td>$x_7$</td>
<td>$w \land \neg f \land \neg c \land r \land \neg s$</td>
</tr>
</tbody>
</table>

To present the fixed part of the specification of the recovery algorithm, i.e. the conditions of Fig. 3, let $x, y, \ldots$ represent a boolean combination of all the above state predicates (e.g. $x_3 = \neg w \land \neg f \land c \land \neg r \land \neg s$).

A typical execution of the original program and of the fault-tolerant transformed program are depicted in Fig. 4. This figure illustrates the correspondence between execution sequences of both programs. For each sequence in the basic program, there is a sequence in the transformed program that simulates its $\epsilon$-intervals (corresponding intervals are shown with the same particular shading). Conversely, for every sequence...
of the transformed program there exists an execution sequence of the basic program in which everything but the ε-intervals are removed. Consider the following formulas:

- A state in which x holds can be changed into a state in which one of y₁, y₂, ..., yₙ, hold: \( \Box (x \rightarrow (x \cup \bigvee_{i=1}^{n} y_i)) \).
- From no point in the computation will all states satisfy one of y₁, y₂, ..., yₙ: \( \Box \neg \bigvee_{i=1}^{n} y_i \).

Then, the conditions imposed on the state predicates in Fig. 3 can be written as

\[
\text{FIXED} = \Box \neg x_1 \land \Box \neg x_2 \land \Box \neg x_3 \land \Box \neg x_4 \land \Box \neg x_5 \land \Box \neg x_6 \land \Box \neg x_7 \land \\
\Box (x_1 \rightarrow (x_1 \cup x_2)) \land \Box (x_2 \rightarrow (x_2 \cup x_3)) \land \\
\Box (x_3 \rightarrow (x_3 \cup (x_1 \lor x_4))) \land \Box (x_4 \rightarrow (x_4 \cup (x_5 \lor x_7))) \land \\
\Box (x_5 \rightarrow (x_5 \cup x_6)) \land \Box (x_6 \rightarrow (x_6 \cup (x_1 \lor x_4))) \land \\
\Box (x_7 \rightarrow (x_7 \cup x_5)).
\]

The second part of the transformation, defined below recursively as \( l(\varphi, \neg r \land \neg w) \), reflects the fact that after taking away the intervals in which r or w hold (as implied from \( \text{FIXED} \), an f-state is also an r-state, and thus \( \neg f \) is not mentioned explicitly) the part of the execution that remains behaves like the basic program with the addition of checkpointing actions (the snapshot states, namely s \lor c-states, behave like stuttering states when their internal variables are concealed). The effect of the internal computation in taking and restoring checkpoints is easily hidden by concealing the additional variables in the transformed specification.

The transformation is therefore

\[
F(\varphi) = \text{FIXED} \land l(\varphi, \neg r \land \neg w),
\]

where \( l(\varphi, \eta) \) is defined inductively as

- \( l(\varphi, \eta) = (\neg \eta \cup (\eta \land \varphi)) \), where \( \varphi \) is a classical (nontemporal) formula,
- \( l(\Box \varphi R, \eta) = \Box l(\varphi, \eta) \), \( l(\varphi \forall \psi, \eta) = l(\varphi, \eta) \forall l(\psi, \eta) \),
negations are pushed inwards to stand adjacent to classical subformulas.

This transformation first pushes the negation symbols inside the formula as much as possible. This can be done using known LTL equivalences (e.g. \( \neg \propto \varphi = \Box \neg \varphi \)) and results in a formula in which negation symbols do not precede any temporal subformula. Then each classical (nontemporal) formula \( \varphi \) is replaced by \( (\neg r) U (q \land \varphi) \), which means that a state satisfying \( \varphi \) can be delayed by the “bad” (no progress) states \( (\neg \eta) = r \lor w \)-states.

It is possible to give a stronger transformation \( \tilde{\varphi} \) (i.e. \( \tilde{\varphi} \) returns a stronger property than \( \varphi \) when given the same property of the original program) at the expense of some elegance. Recall that more information is known about the \( w \)-intervals than is revealed in the transformation: any \( w \)-interval, together with all the previous \( \neg r \land \neg w \)-intervals, forms a prefix of a legal interleaving sequence of the basic program. Thus, each \( w \)-interval should be separately distinguished while the preceding \( w \)-intervals are ignored.

Let \( d \) be a new boolean state function which changes at most once in each execution sequence (i.e. \( \Box (d \rightarrow \Box d) \)) with the change being synchronized with a change of \( r \) from \( F \) to \( T \). This is described using the predicate \( \text{synch}(d, r) = \Box ((\neg d \land \varphi) \rightarrow (\neg d \land (\neg r \land \Box (d \land r))) \). The \( w \)-interval which comes immediately before the change of \( r \), is the distinguished interval and quantifying universally over \( d \) selects each such possible interval. Thus, the resultant transformation is

\[
\tilde{\varphi} \varphi = \forall d (FIxED \land \text{synch}(d, r) \land \Box (d \rightarrow \Box d) \land \bar{\tilde{\varphi}}(w, r, d)),
\]

where the transformation \( \bar{\tilde{\varphi}}(\varphi, \eta_1, \eta_2, \eta_3) \) is similar to \( \bar{\tilde{\varphi}}(\varphi, \eta) \), except for the basic case (where \( \varphi \) is a classical formula) which is now \( \bar{\tilde{\varphi}}(\varphi, \eta_1, \eta_2, \eta_3) = \eta_3 \lor (\eta_1 \land \neg (\neg \eta_2 \land \varphi)) \). Note that when \( \eta_1, \eta_2 \) and \( \eta_3 \) are instantiated to \( w, r \) and \( d \), respectively, the expression \( \eta_1 \land \neg (\neg \eta_2 \land \varphi) \) holds in any \( w \)-interval that is not the distinguished \( w \)-interval. For the distinguished interval, this expression does not hold and, thus, \( \varphi \) must hold immediately. For yet a stronger transformation, \( \varphi \) can be replaced in the above formula for \( \bar{\tilde{\varphi}}(\varphi, \eta_1, \eta_2, \eta_3) \) by the expression \( \varphi \land (\eta_1 \rightarrow (\neg \eta_2 \land \varphi)) \), which holds in \( \varphi \) states that are either not in a \( \neg w \)-interval, or in the distinguished \( w \)-interval.

Now, consider some properties that the original program \( P \) might be required to satisfy: the first is responsiveness to requests, i.e. that when a request for some service is issued, the appropriate service is eventually granted. This can be written in LTL, using appropriate predicates \( \text{request} \) and \( \text{service} \) as

\( \Box (\text{request} \rightarrow \Diamond \text{service}) \).

The second property is mutual exclusion, i.e. that the program does not enter two critical sections \( CS_1 \) and \( CS_2 \) simultaneously:

\( \Box (\neg (\text{in}(CS_1) \land \text{in}(CS_2))) \).
Then, under the above program transformation \( \hat{\phi} \), these properties cannot be guaranteed. However, some weaker properties, related to them can be achieved: in the presence of a finite number of failures, any request is serviced.

\[ \Box((\text{request} \land \Box \Box \text{if}) \rightarrow \Diamond \text{service}). \]

When a failure or recovery are not active (since failures are detected immediately here, we may refer to recovery states, which include the failure states), mutual exclusion between \( CS_1 \) and \( CS_2 \) is achieved (however, because of failure, the program counters can be corrupted so that mutual exclusion is violated).

\[ \Box(\neg r \rightarrow \neg (\text{in}(CS_1) \land \text{in}(CS_2))). \]

These properties can be formally proved by using the formula transformation \( \hat{\phi} \) and the proof rule (1) in Section 3.1.

In some other cases, the environment may impose restrictions over the kind of recovery that is possible. Backward recovery assumes that the program is a closed system [18] and that the recovery algorithm has the ability to control all parts of the system. But for program recovery in an open system, special care must be taken since the environment now includes other processes that might not be affected by the faults or the recovery algorithm (of course, these processes may be affected by their own faults). In this case, inputs from the environment must be stored in nonvolatile memory to be used upon recovery and outputs to the environment that have been sent once must be blocked after recovery until real progress is being made [11].

Thus, in open systems the inputs and outputs (channels) must appear explicitly in the abstract representation of the basic program and, after recovery, the computation must not return immediately to its original mode, but pass through a temporary "silent" mode in which no actual input and output occur. This can be represented in the transformation using another boolean state function which identifies when such an interval occurs, and a state mapping which relates the states during this period to the original state when normal execution is in progress (noting the difference in the values of the input and output channels).

4.2. Forward recovery

Forward recovery, in which the system recovers from a fault by restarting the execution from a possible future global state, is sometimes essential to overcome the effects of faults, particularly in real-time systems. Here, a part of the computation is not repeated or redone, as in backward recovery, but is skipped when a fault occurs, i.e. the computation reaches a point at which a failure occurs and then, skipping up to some future state, it resumes an execution which is consistent with its most recent initial execution segment.

Let \( p \) be a boolean variable which is \( T \) exactly when recovery from the last fault has not yet been completed (including states in which a fault has occurred and recovery
has not yet started) and F otherwise. For the moment, ignore the details of the recovery method, i.e. what happens between the occurrence of a fault and the resumption of the basic program from a future state. Then the forward recovery transformation $\mathcal{F}'$ is defined as $FIXED' \land l'(q, \neg p)$ where $FIXED'$ is the fixed property of the recovery algorithm and $l'$ is defined inductively as:

- $l'(q, q) = \neg q \rightarrow q$, where $q$ is a classical (nontemporal) formula,
- $l'(\Box q, q) = \Box l'(q, q)$,
- $l'(q \lor q, q) = l'(q, q) \lor l'(q, q)$,
- $l'(q \land q, q) = l'(q, q) \land l'(q, q)$,
- negations are pushed inwards to stand adjacent to classical subformulas.

That is, an execution $\sigma$ of the transformed program $\mathcal{T}(P)$ resembles an execution $\sigma'$ of $P$, except that arbitrary intervals of $\sigma'$ are replaced by $p$-intervals (not necessarily of the same length). The $p$-intervals correspond to the states in which a fault occurs and the program then recovers. Notice that this transformation is very similar to the previous one in the sense that when negations are pushed inside it preserves the structure (e.g. the boolean operators, quantifiers and modals) of the transformed formula.

5. Proving the correctness of specification transformations

Verifying that a given program transformation corresponds to some specification transformation means that a whole class of properties is being verified at the same time, and with respect to a class of programs. Thus, a higher order logic verification is needed. The verification of program transformations that preserve program semantics was studied by Huet and Lang [9]. They used a second order language to verify program transformations that are used for code optimization of sequential programs. Our notion of correctness is more general as the transformations under discussion are not restricted to preserve the semantics of the transformed programs.

Since a recovery algorithm can be applied to an entire class of programs, we use a suitable representation of an uninterpreted basic program $B$ which can later be replaced by a concrete program. The uninterpreted representation is a program written using some uninterpreted relation and function symbols. A concrete program is obtained from the uninterpreted representation by instantiating the uninterpreted symbols.

An uninterpreted program may use a single variable $x$ to represent its entire state by an encoding of its variables. But many other representations of uninterpreted programs are possible. (Recovery algorithms that are more involved than the simple examples used here may need more complicated representation; see [24] for another way to represent uninterpreted programs.) Each atomic program operation $\mathcal{A}$ can then be written using the transition formula $\mu_{\mathcal{A}}(x, x')$ (with $\mathcal{A} = \{x\}$) and, because of the
uniformity of the operations, we use a single operation to represent the entire program\(^3\)

\[
\left( (x=x') \land \bigwedge_{s \in T} \neg en_s \right) \lor \bigvee_{s \in T} \mu_s(x, x') .
\]

Thus, a noninterpreted program can be represented by a single operation \(\mu(x, x')\). Let \(\Theta(x)\) be a predicate that is true over the \(x\)'s that are initial values of the basic program. Then an exact specification of the program (with rigid variable \(y\)) is

\[
\Phi_B = \Theta(x) \land \Box y (x = y \land \mu(y, x)).
\] (3)

This formula is only partially interpreted, as \(\mu\) and \(\Theta\) are not given. When a concrete program \(P\) is given, the uninterpreted relation and function symbols of \(B\) are instantiated to the given relation and functions of \(P\). At the same time, the occurrences of symbols in \(\Phi_B\) can be instantiated similarly to form an exact specification \(\Phi_P\) of \(P\).

The representation of uninterpreted programs must be chosen appropriately to facilitate the application of the program transformation. Let \(B\) be an uninterpreted program. Apply \(Y\) on \(B\) such that \(Y(B)\) is an uninterpreted fault tolerant program (or not fully interpreted, as at least the relation and function symbols concerned with the recovery algorithm and fault mechanism representation are given). Let \(\Psi_B\) be an exact specification of \(Y(B)\). When instantiating \(B\) to a concrete \(P\), the same instantiation to the occurrences of these relation and function symbols in \(\Psi_B\) will form an exact specification \(\Psi_P\) of \(P\).

The relation between formulas, transformed programs and specification transformations is illustrated in Fig. 5. As we have pointed out, a specification transformation is a formula of a higher-order logic since treating temporal formulas as objects requires a higher-order logic. With this in mind, the following lemma can be proved directly from Definitions 3.2 and 3.5.

**Lemma 5.1.** If \(\delta\) is an exact specification transformation w.r.t. a program transformation \(\mathcal{F}\), and \(\mathcal{F}\) is another specification transformation, then \(\mathcal{F}\) corresponds to \(\mathcal{F}\) iff for each LTL formula \(\varphi\), \(\delta(\varphi) \rightarrow \mathcal{F}(\varphi)\).

We now describe a relatively complete proof method for specification transformations. In order to verify that \(\mathcal{F}\) corresponds to \(\mathcal{F}\),

1. formulate \(\delta\), an exact specification transformation w.r.t. \(\mathcal{F}\), and
2. prove that for each formula \(\varphi\), \(\delta(\varphi) \rightarrow \mathcal{F}(\varphi)\).

The completeness of this method (this is a different completeness concept from the one defined in 3.3) is w.r.t. the expressiveness of the language used to formulate \(\delta(\varphi)\) in 1, and the ability to prove the implication in 2.

\(^3\)This single operation allows further simplification of the exact specification of the program by representing program termination as infinite stuttering.
Under the above (example of) abstract representation, there are only two relation symbols $\mu$ and $\Theta$. Then, an exact specification transformation w.r.t. $\mathcal{F}$ can be formulated by

$$\mathcal{E}(\varphi) = \exists \mu \exists \Theta((\llbracket \Phi_B \rrbracket \leq \llbracket \varphi \rrbracket) \land \Psi_B),$$

where, as before, $\Phi_B$ and $\Psi_B$ are the exact specifications of the uninterpreted programs $B$ and of $\mathcal{F}(B)$, respectively (the generality of the uninterpreted relations and functions can be limited by adding some appropriate conjunct). The assertion $\llbracket \Phi_B \rrbracket \leq \llbracket \varphi \rrbracket$ is a high-order formula that asserts that any sequence satisfying $\Phi_B$ also satisfies $\varphi$, i.e. that $\varphi$ is a consequence of $\Phi_B$. Formula (4) should be read as follows: an interleaving sequence $\sigma$ satisfies $\mathcal{E}(\varphi)$ iff there exists a program $P$ (i.e. an instantiation of the relation symbols $\mu$ and $\Theta$) that satisfies $\varphi$, such that $\sigma$ is an execution of $\mathcal{F}(\varphi)$. That is, the formula $\mathcal{E}(\varphi)$ is satisfied exactly by execution sequences of programs obtained by the transformation $\mathcal{F}$ from programs satisfying $\varphi$.

5.1. An example

Consider a simple recovery algorithm which is a simplified version of the algorithm in Section 4.1. This algorithm occasionally stops the basic computation, records the global state, and then allows the basic computation to continue. On the occurrence of a failure, the program is rolled back to the last saved state. It is assumed that taking snapshots and recovering are instantaneous.

We use the previous boolean state functions $r$, $s$ and $w$ (of states of the transformed program) when expressing the transformations:
r (affected): A fault has occurred and recovery from it has not yet completed. Thus, part or all of the global state may be corrupted.

s (snapshot): A snapshot is taken of the state space. The basic program state is not changed during the recording of its state space.

w (wasted): The present state of the computation does not correspond to effective progress of the basic computation, because the next fault will occur before the next checkpoint is reached (and rolling back to the last saved checkpoint will undo this part of the computation). Again, w can be seen as a shorthand for the formula \( \neg r \land ((\neg s) \lor r) \).

The possible transitions between these states are described in Fig. 6. One can easily formulate the fixed part of the specification transformation, as in Section 4.1.

Similar to the example in Section 4.1, the second part of the transformation, defined recursively as \( I(q, \neg w \land \neg r) \), must reflect the fact that after taking away the intervals in which \( w \) holds, the part of the execution that remains behaves like the basic program (with the addition of checkpointing operations, which uses additional variables that are concealed). With this simplified transformation, no claim is made about the basic program's behavior during the \( w \lor r \)-intervals. This corresponds to the assumption that the variables of the original program are arbitrarily corrupted. As in Section 4.1, it is possible to distinguish between \( w \)-states, in which the system performs as intended in the basic program and \( r \)-states, in which the predicted fault has occurred. We will show here only how to verify the simple case where all the \( w \)-states are treated as “bad” states, about which the only interesting information is that they do not contribute anything to the computation.

Let \( B \) be an uninterpreted representation of the basic program using a single variable (as in the previous subsection), a single operation \( \tau_{op}: [\mu(x, x')]_x \), and let the initial value of \( x \) satisfy \( \Theta(x) \). Consider a transformation of \( B \) that adds to the program the following operations to represent taking snapshots, faults and recovery:

\[ \tau_{op}: [\neg r \land \mu(x, x') \land \neg s']_x, s. \]

The modified original operation is not enabled when recovery from the most recent fault has not yet occurred. The snapshot flag is reset for the case where a snapshot has just been taken.

\[ \tau_{snap}: [\neg s \land \neg r \land save' = x \land s']_{save, s}. \] The current value of \( x \) is copied into the variable \( save \) in a nondestructive memory.

\[ \tau_{fault}: [r' \land \neg s']_{r, x, s}. \] The variable \( r \) is \( T \) exactly when a fault has occurred. In addition, the variable \( x \) can take any value (i.e. be arbitrarily corrupted) when a fault occurs.
The value of $x$ is restored using the last value stored in $\text{save}$.

The initial condition is $\neg r \land (\text{save} = x) \land \Theta(x)$.

The atomic operation $\tau_{\text{snap}}$ takes a global snapshot of the entire program state (in practice, it should be replaced by several operations that perform this task).

We introduce a new temporal modal $\langle \eta \rangle$ (actually, a family of modals) for each temporal formula $\eta$, such that $\langle \eta \rangle \phi$ is true iff the sequence obtained by keeping only the states that satisfy $\eta$ also satisfies $\phi$ (or if it is finite, is a prefix of a sequence that satisfies $\phi$). A formal definition of this modal appears in the Appendix.

It can be proved using temporal logic verification methods [20] that for this program, the following formula $\kappa$ (with $y$ a rigid variable) holds for $\mathcal{F}(B)$:

$$
\kappa = w \cup (\neg w \land \Theta(x)) \land \exists y((w \cup (\neg w \land x = y)) \cup (w \cup (\neg w \land \mu(y,x))).
$$

Hence, $\Psi_B \rightarrow \kappa$ holds. Using the axioms of $\langle \eta \rangle$ below and other linear temporal logic axioms, it is possible to deduce that

$$
\Psi_B \rightarrow \langle \neg w \rangle (\Theta(x) \land \exists y(x = y \cup \mu(y,x))).
$$

(5)

Now consider the following higher-order deduction for the correctness of the specification transformation $\mathcal{F}(\phi) = \langle \neg w \rangle \phi$:

1. $\Psi_B \rightarrow \langle \neg w \rangle \Phi_B$ follows directly from (5).
2. $\varepsilon(\phi) = \exists \mu \exists \Theta(([\Phi_B] \subseteq \lbrack \phi \rbrack) \land \Psi_B)$ formulated in (4)
3. $\varepsilon(\phi) \rightarrow \exists \mu \exists \Theta(([\Phi_B] \subseteq \lbrack \phi \rbrack) \land \langle \neg w \rangle \Phi_B)$ from 1 and 2.
4. $\varepsilon(\phi) \rightarrow \exists \mu \exists \Theta(([\Phi_B] \subseteq \lbrack \phi \rbrack) \land \langle \neg w \rangle \phi)$ from 3.
5. $\varepsilon(\phi) \rightarrow ((\exists \mu \exists \Theta([\Phi_B] \subseteq \lbrack \phi \rbrack) \land \langle \neg w \rangle \phi)$ from 4 ($\mu$ and $\Theta$ do not occur free in $\phi$).
6. $\varepsilon(\phi) \rightarrow \langle \neg w \rangle \phi$ from 5.
The next step is to get rid of the new modal $\langle \eta \rangle$ so that $\mathcal{F}(\phi)$ can be written in an equivalent form using only the traditional modal operators. The following axioms of $\langle \eta \rangle$ enable this translation:

- $(\langle \eta \rangle\phi) \iff (\neg\eta U (\eta \land \phi))$, when $\phi$ is a classical (nontemporal) formula.
- $(\langle \eta \rangle \Box \phi) \iff (\Box \langle \eta \rangle \phi)$, $(\langle \eta \rangle (\phi U \psi)) \iff (\langle \eta \rangle \phi U \langle \eta \rangle \psi)$.
- $(\langle \eta \rangle \forall z \phi) \iff (\forall z \langle \eta \rangle \phi)$, $(\langle \eta \rangle \exists z \phi) \iff (\exists z \langle \eta \rangle \phi)$.
- $(\langle \eta \rangle (\phi \lor \psi)) \iff (\langle \eta \rangle \phi \lor \langle \eta \rangle \psi)$.

These axioms reflect the fact that $\langle \eta \rangle$ distributes with every other modal and connective except negation. The transformation $I(\phi, \eta)$, given in Section 4.1 is easily obtained from these axioms.

Note that the exact transformation $\delta(\phi)$ is useful only for proving the formula transformation $\mathcal{F}(\phi)$ but not as a formula transformation in itself. Besides being a higher-order temporal formula, its component subformula $\Psi_B$ is an exact specification of the transformed program, where the details of the recovery mechanism are explicitly expressed (although the details of the basic program are kept abstract). This results in a rather complicated and detailed transformation. Furthermore, $\delta(\phi)$ does not provide a clear correspondence between the transformed formula and the resulted formula.

Returning to the resultant specification transformation, the new boolean state predicates (e.g. $w$ and $r$ that appear in the transformations in Section 4.1) correspond to auxiliary variables in the program. Although for simplicity they were used in this example directly as boolean program variables, they are not in general part of the recovery program. Some (like $r$) are simple history variables, while others (in particular $w$) are (generalized) prophecy variables [1]. Some variables can have complicated rôles, like the variable $d$ of Section 4.1, which should be added to the program as an auxiliary variable to satisfy particular conditions.

6. Generalizing distributed recovery

6.1. Extending the semantic and syntactic formalism

Representing executions of distributed programs as interleaving sequences does not always lend itself to a simple specification transformation: consider an interleaving sequence $\sigma$ of the transformed program in which a snapshot is taken. It is often the case that the snapshot does not represent any state that occurs in $\sigma$, but a state that could have occurred in a sequence which is related to $\sigma$ in some sense. This property of snapshot algorithms stems from the fact that maintaining a snapshot that indeed occurs in the same sequence is costly, as it requires all the processes of the program to be stopped before the snapshot is taken.

Recovery schemes are often based on establishing recovery points that do not necessarily belong to the same interleaving sequence, but merely to some equivalent sequence or, equivalently, a global state which is a “slice” of a partial order execution.
For this reason, we extend the semantics by defining equivalence classes of interleaving sequences. Partial order executions and their alternative representation by sets of linearizations or interleaving sequences are described in [16]. Figure 8 depicts the relation between a partial order execution, interleaving semantics and global snapshots. A partial order execution consists of partially ordered set of occurrences of operations. The occurrences of operations of the same process are totally ordered (there are processes $P_1, \ldots, P_4$ in Fig. 8), and the event of sending a message precedes the event of receiving the same message. To each such execution there exists a set of linearizations, or completion of the partial order to total orders that form interleaving sequences of this execution. A snapshot of a distributed program is a collection of local states, one for each process. This collection satisfies that there are no messages that are sent from one process $P_i$ after (according to the partial order) its local state was recorded into the snapshot, and received by a process $P_j$ before the local state of $P_j$ is recorded. Among the set of linearizations that correspond to each such partial order execution, there exists at least one that has a global state that agrees (i.e. has the same value assigned to all the variables) with the snapshot recorded [12].

As an alternative semantics for partial order execution, we construct equivalence classes of interleaving sequences that correspond directly to the linearizations of such partial orders. This semantics is based on infinite traces [14], a generalization of finite
traces [22]. Each equivalence class corresponds to one partial order execution. A partial order execution is then an intersection of the total orders between occurrence of operations in such an equivalence class.

The operations are now assumed to be deterministic in the sense that for any state \( s \in S \) and operation \( x \in T \), \( s \) can have at most a single successor generated by executing \( x \). An interleaving sequence will also be represented as a pair \( \sigma = \langle s_0, v \rangle \), where \( s_0 \) is the first (initial) state of the sequence, and \( v \) is a sequence of operations \( x_0x_1x_2 \ldots \) such that \( x_i \) is the operation that transforms \( s_i \) to \( s_{i+1} \). We add a special no_op symbol that generates stuttering, which is independent of any other operation.

Let \( I \subseteq T \times T \) be a symmetric and irreflexive relation between operations called an independence relation [22], satisfying the properties: (1) each pair of operations \( (\tau_1, \tau_2) \in I \) is commutative, when treated as global state transformers, and (2) the execution of \( \tau_1 \) cannot enable, or disable \( \tau_2 \), and vice versa [14]. If \( (\tau_1, \tau_2) \in I \), we say that \( \tau_1 \) and \( \tau_2 \) are independent, otherwise they are dependent. The independence relation \( I \) identifies when it is possible to commute adjacent occurrences of operations in an execution sequence. It is used to define an equivalence relation on interleaving sequences. First, equivalence between finite strings of operations is defined [22]:

**Definition 6.1.** Two strings \( v, w \in T^* \) are equivalent, denoted \( v \equiv w \), iff there exists a sequence of strings \( u_0, u_1, \ldots, u_n \), where \( u_0 = v, u_n = w \), and for each \( 0 \leq i < n \), \( u_i = \alpha \beta u' \) and \( u_{i+1} = \beta \alpha u' \) for some \( u, u' \in T^* \), \( (\alpha, \beta) \in I \).

That is, \( w \) is equivalent to \( v \) if it can be obtained from it by repeatedly commuting independent operations. This definition of an equivalence relation among finite strings is now extended to interleaving sequences. Let \( \text{Pref}(w) \) be the set of finite prefixes. In order to allow stuttering as before, we define \( \text{Pref}(w) \) to be closed under inserting or removing any finite number of no_op symbols. Thus, e.g.

\[ \alpha \text{ no_op no_op } \beta \in \text{Pref}(\alpha \beta \gamma) \]

of the (finite or infinite) string \( w \). Let \( \sqsubseteq \) be a relation between pairs of interleaving sequences: \( \langle s, v \rangle \sqsubseteq \langle s', v' \rangle \) iff

- \( s = s' \), and
- \( \forall u \in \text{Pref}(v) \exists w \in \text{Pref}(v') \exists \gamma \in T^* (w \equiv z \land u \in \text{Pref}(z)) \).

That is, the two interleaving sequences have the same initial state, and each finite prefix of \( v \) is a prefix of some permutation of a prefix of \( v' \). Let \( \sigma \approx \sigma' \) iff \( \sigma \sqsubseteq \sigma' \) and \( \sigma' \sqsubseteq \sigma \) [14]. Then clearly \( \approx \) is an equivalence relation between interleaving sequences.

The following definition gives an alternative interpretation of LTL formulas over equivalence classes of interleaving sequences, rather than over all the execution sequences of the program.

**Definition 6.2.** An LTL formula \( \varphi \) is existentially satisfied [23] by a class of sequences \( \Sigma \) if there exists a sequence in this class that satisfies it (according to the usual definition of the satisfaction of temporal formula w.r.t. an interleaving sequence). This is denoted as \( \Sigma \models \varphi \).
The existential satisfaction relation is weaker than the ordinary satisfaction relation of temporal logic formulas over sequences, because it demands only that representatives of the interleaving sequences satisfy one formula rather than all of them.

6.2. Behavior of distributed recovery algorithm

Algorithms for recovering to a previous checkpoint (snapshot), such as [13], or those based on the snapshot algorithm of Chandy and Lamport [5], have the following properties:

- The snapshot or checkpoint to which the program recovers does not necessarily occur in the same interleaving sequence in which the fault occurred. However, if one considers the set of equivalent execution sequences (up to permuting adjacent independent operations), the state to which the algorithm recovers appears in one of the equivalent sequences.
- Recovery does not have to be done simultaneously by all the processes. A process may start execution from its saved local state before another process senses the fault. However, in each equivalence class of execution sequences, there is a sequence in which the processes start recovering simultaneously, i.e. from the same state.

It is easy to describe the behavior of the combined fault-tolerant program by means of representative sequences. For each equivalence class of sequences, there is a sequence which always recovers to the most recent global snapshot saved, and this global state is a state that occurred in this sequence. This can be specified using existential satisfaction of LTL formulas. In fact, an appropriate transformation could be similar or even identical to $F(\varphi)$ (or $\bar{F}(\varphi)$) in Section 4.1. However, $F(\varphi)$ should now be interpreted existentially, i.e. satisfied by at least one sequence out of each equivalence classes of sequences, rather than universally as before. That is, if $P \models \varphi$, then $F(P) \models \exists \varphi$.

This means that we obtain a transformation from a universally quantified formula into the weaker form of an existentially quantified formula. However, such a formula $\varphi$ is also a specification of all the possible interleaving sequences in the sense that it requires each of the program's interleaving sequences to be equivalent to a sequence that satisfies $\varphi$. Thus, we need to supply the means to deduce universally satisfied properties from the existential formula. That is, a verification method is needed, that allows to deduce from $P \models \exists \varphi$ that $P \models \psi$, for appropriate existential and universal properties $\varphi$ and $\psi$, respectively. This ability has been demonstrated elsewhere [23], where it is shown that by using appropriate definitions, enough information about all the equivalent sequences can be obtained from the representative sequences. This guarantees that compositional completeness is also achievable in this generalized case as well.

7. Discussion and further research

A specification method was presented to handle the superposition of generic recovery algorithms on basic programs designed to be executed on fault-free systems.
This produces a program that is fault tolerant with respect to the failures that can be handled by the recovery algorithm. Rather than specifying the recovery algorithm separately, we have suggested a method by which it is specified as a specification transformation from the properties of basic programs to the properties of the superposed fault tolerant version. The properties of the recovery algorithm itself appear only as a fixed part of any application of this transformation on a formula.

This method is compositional, as verification is done with respect to the basic program and using the transformation. There is no further need to reason about, or even to denote, the version of the program after the superposition. A relatively complete verification method was given to prove that a transformation correctly corresponds to a superposition. Hence, a specification transformation is proved once for all possible basic programs to which the recovery algorithms can be applied.

Using the framework described in this paper, it is possible to achieve other benefits. One of these is the classification and comparison of different schemes for distributed recovery: various schemes have advantages and disadvantages with respect to the properties that hold after superposition of the recovery algorithm.

Another comparison is of the amount of work needed to be done for achieving recovery. An extreme case could be that recovery is achieved by rolling back to the very first state. Other methods can vary in the complexity of the recovery algorithm and the amount of computation lost/redone.

The effect of fairness on program and specification transformations is a tantalizing issue which demands a careful study. For example, an operation which is enabled continuously in an execution of the basic program and thus needs to be executed eventually, might not retain these properties in the transformed program: it can happen that the enabledness is interrupted by repeated occurrence of faults that will disable it infinitely often for short periods. Thus, it is crucial to consider carefully the fairness assumptions under which a formula corresponds to a given program transformation.

Probabilistic specification can be used to allow using algorithms that achieve fault tolerance “almost always”. Another possibility is to give a precise probabilistic characterization of the computation as a function of the probability of faults. This is a generalization of the transformation of the basic program’s properties. Temporal logics that incorporate probabilistic reasoning can be used [26].

Appendix: linear temporal logic

**Definition A.1** An LTL structure $\mathcal{A}$ is a triple $\langle Q, \hat{q}, \xi \rangle$ where $Q$ is a set of states, $\hat{q}$ is some distinguished state of $Q$ and $\xi$ is a sequence of states from $Q$, $\xi_0, \xi_1, \xi_2, \ldots$, starting with $\xi_0 = \hat{q}$. The labeling function $L$ assigns to each state $q \in Q$ an interpretation over the flexible variables, and for each rigid variable an interpretation that is fixed over all states. Let $L(q)$ be the interpretation of both the flexible and the rigid variables in the state $q$. 
Definition A.2 (Syntax of LTL). If $\Gamma$ is the set of formulas in LTL then
- for each classical formula $\phi, \psi \in \Gamma$
- if $\phi \in \Gamma$ then $\square \phi, \Diamond \phi, \neg \phi \in \Gamma$
- if $\phi, \psi \in \Gamma$ then $\phi \land \psi, \phi \lor \psi, \phi \lor \psi \in \Gamma$
- if $\phi \in \Gamma$ and $z$ is a variable, then $\exists z \phi, \forall z \phi \in \Gamma$

Definition A.3 (Semantics of LTL). The satisfaction of an LTL formula over a sequence $\xi$, position $0 \leq i < |\xi|$ and labeling function $L$ is defined inductively:
- $(L, \xi, i) \models \top$
- $(L, \xi, i) \models \phi \iff L(\xi_i) \models \phi$, if $\phi$ is a state formula.
- $(L, \xi, i) \models \phi \lor \psi \iff (L, \xi, i) \models \phi$ or $(L, \xi, i) \models \psi$.
- $(L, \xi, i) \models \neg \phi \iff \neg (L, \xi, i) \models \phi$.
- $(L, \xi, i) \models (\phi \lor \psi) \iff$ there exists $i \leq j < |\xi|$ such that $(L, \xi, j) \models \psi$, and for every $k$, such that $i < k < j$, $(L, \xi, k) \models \phi$.
- $(L, \xi, i) \models \exists z \phi \iff$ there exists a labeling function $\hat{L}$, which differs from $L$ by at most the assignment given to $z$, and $(\hat{L}, \xi, i) \models \phi$.

Each interleaving sequence $\xi$ of a program $P$ can be seen as an LTL structure, with a set of states $S$ (the state space of the program) and a labeling function $L$ that agrees with the assignment of variables to their domain in each state of $\xi$. Thus, we define $\xi \models \phi$ iff $(L, \xi, 0) \models \phi$. The satisfaction of a formula is extended to a set of interleaving sequences $\Xi$ such that $\Xi \models \phi$ iff for each $\xi \in \Xi$, $\xi \models \phi$.

The other operations such as $\square \phi, \forall z \phi$ can be defined similarly. They satisfy the following relations: $F = \neg \top$, $\phi \land \psi = \neg ((\neg \phi) \lor (\neg \psi))$, $\phi \lor \psi = (\neg \phi) \lor (\neg \psi)$, $\phi \land \psi = (\phi \lor \psi) \land (\psi \lor \phi)$, $\Diamond \phi = \text{true}(\phi)$, $\neg \phi = \neg (\top \lor (\neg \phi))$, $\phi \lor \psi = (\phi \lor \psi) \lor (\phi \lor \psi)$. $\forall \phi = \neg \exists \neg \phi$.

The modal $\langle \eta \rangle$ from Section 5.1 is now defined formally: $(L, \xi, i) \models \langle \eta \rangle \phi$ iff the sequence $\xi'$, which is obtained from $\xi$ by omitting the states $\xi(j)$ in which either $j < i$ or $(L, \xi, j) \models \eta$ holds, satisfies either
- (1) $\xi'$ is infinite and $(L, \xi', 0) \models \phi$, or
- (2) $\xi'$ is finite, and there exists an infinite sequence $\hat{\xi}$, and an appropriate extension $L'$ of $L$ to the new states in $\hat{\xi}$, such that $(L', \xi', 0) \models \phi$ (where $\xi', \hat{\xi}$ denotes the concatenation of the sequences $\xi'$ and $\hat{\xi}$).

Acknowledgment

We would like to express our gratitude to Liu Zhiming for his careful reading of this manuscript and helpful comments, and to anonymous TCS referees for many valuable suggestions.

---

$^4$ i.e., there exists a partial monotonic surjective mapping $\Delta$ from the positive integers $Z^+$ onto $Z^+$, or some prefix of $Z^+$, in which $i$ is in the domain of $\Delta$ if $i > j$ and $(L, \xi, i) \models \eta$, and the relation between $\xi$ and $\xi'$ is $\xi'(\Delta(i)) = \xi(i)$. 

A compositional framework for fault tolerance

References