ENSURING VALUE LIVENESS of DISTRIBUTED SOFTWARE THROUGH CHANGETING†

Grace Tsai, Matt Insall and Bruce McMillin

CSC-93-03

March 15, 1993

† This work was supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546.

Grace Tsai and Bruce McMillin are with the Department of Computer Science at the University of Missouri-Rolla, Rolla, MO 65401.

Matt Insall is with the Department of Mathematics and Statistics at the University of Missouri-Rolla, Rolla, MO 65401.
ABSTRACT

This paper extends the Changeling methodology, which ensures safety of concurrent and distributed software, to provide responsiveness properties. Changeling employs formal methods to generate executable assertions which ensure that the physical state, in the actual run time environment, is consistent with the logical state specified in the assertion. The focus, here, is on the generation of executable assertions from value liveness properties established in the temporal specifications. Specifically, a translation process was developed to generate run-time-assured programs. The verification tool adopted by Changeling is Interleaving Set Temporal Logic (ISTL*) since the logic is based on partial order semantics and capable of expressing the intermediate assertions (behavior) of distributed programs. The translation is shown to be sound and relatively complete; an example of ensuring value liveness properties is presented for distributed Branch and Bound type problems.

1. INTRODUCTION

A system is responsive [Male90] if it responds to internal programs or external inputs in a timely, dependable and predictable manner. It is a necessity for a responsive system to manage initiation and termination of activities to meet the specified timing constraints. Also a responsive system is dependable and predictable, i.e. the system behaves as anticipated, and the occurrence of undesired actions (e.g. faults) does not necessarily lead to a failure.

The incorporation of real-time and fault tolerance into distributed parallel environments is a challenging task, while the specifications of the distributed system must be met within the deadlines in spite of the presence of failures. However, this integrated system or responsive computer system [Male90] can greatly benefit from the application of formal methods. Without formal techniques, life-critical distributed computer control programs cannot be relied on to produce correct results, in time, and, in the presence of failures.

Changeling provides a formal strategy for the integration of fault tolerance and distributed computing through executable assertions [LuMe91b, LuSM92, LuSM92a]. Within this framework, axiomatic proof system is chosen as the mathematical model and the assertions generated from the proof are translated into executable assertions which can detect errors in the presence of failures. However, these executable assertions are mainly derived from the safety properties because of the choice of axiomatic proof system. This paper focuses on fault detection using executable assertions generated from value liveness properties which is a subclass of liveness properties.

The value liveness assertion \((p \rightarrow \text{EF}q)\) ensures what values the program variables must possess eventually at a state with assertion \(q\), starting from a state with assertion \(p\). These assertions are derived from the temporal proof system of Interleaving Set Temporal Logic [PePn90], and are used to reason about progress property from one communication point to another. This paper extends the work of
Changeling to create executable assertions that are, essentially, operational evaluation of value liveness assertions. The derived executable assertions are really safety properties evaluated on a time-indexed computation history collected by the processes of the program. If these executable assertions are satisfied, at run time, then it is ensured that the program satisfies its value liveness properties.

Since the operational evaluation of any assertion is, essentially, a check of a safety property, it might be questionable whether, given \((p \rightarrow \text{EF}q)\), it is really necessary to evaluate the assertions \(p\) and \(q\) in time order. To see why, consider the following example. In a poker game, it is possible, during an evening, to be $10 in debt and $10 ahead many times. However, from the point of view of a player it is much better to be ahead $10 at the end of the night, rather than $10 in debt! In our logic, we would like to express \(p(= \text{$10 behind}) \rightarrow \text{EF}q(= \text{$10 ahead})\) and ensure, during this poker game, that this temporal assertion is met. If we don’t evaluate this assertion in time order, then \(p\) and \(q\) may be met, but \(p\) may occur last which means that we leave the game $10 poorer, a sorry proposition. In this paper, we take great pains to ensure that we can evaluate value liveness assertions in the correct order, in a faulty distributed system using only message passing for communications and no tightly synchronized clocks.

This paper is organized as follows. An overview of the temporal proof system of Interleaving Set Temporal Logic (ISTL*) is provided in Section 2. Section 3 presents Changeling and its translation scheme for the generation of error-detecting algorithms. Section 4 describes formal generation of executable assertions and examines faulty clock problem. In Section 5, the application of Changeling is demonstrated on the Branch and Bound N-puzzle problem. Section 6 concludes this paper.

### 2. INTERLEAVING SET TEMPORAL LOGIC

The approach of Changeling, which supports run time assurance through the application of formal methods, is that a mathematical model is chosen to generate assertions which are then converted into executable assertions to monitor the run time behavior of the program. Among many mathematical models, we choose Interleaving Set Temporal Logic (ISTL*) [KaPe88,PePn90], since it is capable of representing intermediate behavior of distributed programs.

Traditionally, concurrent and distributed programs are verified using variants of temporal logics with interleaving semantics [ClEm83, ClEm86, MaPn82, MaPn89, HeMP90]. However, verification using sets of state sequences which represent the executions of a program is tedious and unnatural since all the possible interleavings of a program must be checked. Thus, many attempts have been made to design a temporal logic with appropriate formalism for distributed programs. That is the partial order approach [KaPe87, PiWo84, Reis88, PePn90], where the order is defined by the local events (events are executions of atomic operations) within a process and the events of sending and later receiving a message. All other events that are not related are arbitrary.

Interleaving Set Temporal Logic (ISTL*) is based on a partial order approach; thus, the assertions derived are capable of describing the distributed nature of a program. These assertions can also serve as
expected behavior to monitor the run time behavior of a program.

In the application to programs, an ISTL* [KaPe88, PePn90] structure corresponds to a computation (run) of a program and each (branching) structure denotes the global states of a single partial order as well as causal relations among these states. An ISTL* structure $M$ is a quadruple $\langle \Sigma, \theta, E, L \rangle$ where

- $\Sigma$ is a set of states,
- $\theta$ denotes the initial state of $\Sigma$,
- $E$ is a set of sequences of states starting at $\theta$,
- $L$ is a function which assigns to each state $s \in \Sigma$ an interpretation.

A value liveness assertion of the form $(p \rightarrow EFq)$ states that every computation contains some state sequence (path) eventually satisfying the assertion $q$ when starting from a state satisfying the assertion $p$.

### 3. THE TRANSLATION SCHEME

[Mili81] adopted formal methods to obtain fault tolerance through software specified executable assertions, which shows that program verification is a viable starting point for generating assertions. Since assertions developed from verification proofs characterize a program’s behavior, we developed a translation scheme for Changeling to generate error-detecting algorithms in distributed parallel environments. The following steps outline the translation scheme.

1. Obtain value liveness assertions from temporal specifications: value liveness properties are derived from the proof system of Interleaving Set Temporal Logic (ISTL*) [PePn90] which adopts partial order semantics, hence, providing a more suitable representation of concurrency than interleaving semantics does.

2. Translate value liveness assertions into executable assertions: executable assertions demonstrate a program’s expected behavior, since they are developed based on the definitions of the problems and the behavior they exhibit in the verification proofs.

3. Derive error-detecting algorithms: a translation procedure is developed for the generation of executable assertions from value liveness assertions. Then executable assertions are embedded into the code and the satisfiability of the run time behavior is measured by the evaluation of executable assertions.

The value liveness assertions $(p \rightarrow EFq)$ are intermediate assertions, which allow us to check when the execution reaches some (intermediate) location in the program, the postassertion $q$ is satisfied. These intermediate or value liveness assertions are translated into executable assertions to catch faulty behavior of a program, in that a program is faulty if it violates some of these assertions. This forms the basis of our translation procedure.
3.1 Translation Procedure

The purpose of the translation process is to generate an error-detecting algorithm from the embedding of executable assertions into the code. Thus, value liveness assertions need to be translated into executable assertions that can be directly applied into the operational environment, i.e.,

\[
\text{Value Liveliness Assertions} \xrightarrow{T} \text{Executable Assertions}
\]

where T is the translation process. In other words, a transformation that takes into account the distributed operational environment is necessary to convert value liveness assertions into executable assertions or run-time assertions.

In the translation, we adopt a notion similar to that of HAA proof system[LuSM92a] to closely match the distributed operational environment. Each process maintains its own view of a system by a set of auxiliary variables that record process communications. Also, every process keeps track of its updates of auxiliary variables with respect to other processes. When communication occurs, processes exchange their sets of auxiliary variables, i.e., processes exchange their views of the system. In summary, the translation process involves the following steps:

- update auxiliary variables before the communications.
- exchange auxiliary variables following the communications and update auxiliary variables with the new values received.

Auxiliary variables are used to communicate variables in the assertions, which allows processes to evaluate the satisfiability of the behaviors of other processes. Hence, we can avoid having a process test itself or do self-checking.

Before describing the translation, the following definitions[PePn90] are needed. These definitions describe the history and the equivalence class of histories under partial order semantics.

**Definition 3.1:** Let a transition \( \tau \) be defined as \(< enabled(\tau) \rightarrow y := f_x(y) >\), which is interpreted as follows: when the transition \( \tau \) is enabled (enabled(\( \tau \))), the execution of \( \tau \) transforms a state \( s_i \) into another state \( s_j \) where some of the program variables are changed (\( y := f_x(y) \)).

**Definition 3.2:** A **history** of a program \( P \) is a pair \( h = < J, v > \) where \( J \) is the initial state and \( v = \alpha_1, \alpha_2, \ldots, \alpha_n \) is a sequence of operations in the program.

**Definition 3.3:** Two histories \( h = < J, v > \) and \( h' = < J, w > \) with a common initial state \( J \) are **equivalent** (\( v, w \in A^*; A \) is the alphabet), if there exist histories \( < J, v_1 >, < J, v_2 >, \ldots, < J, v_n > \) with \( v_1 = v \) and \( v_n = w \) and for each \( 1 \leq i < n, (\alpha, \beta) \in D \) and \( x, y \in A^* \), such that \( v_i = x\alpha\beta y, v_{i+1} = x\beta\alpha y \). In other words, \( v_i \) and \( v_{i+1} \) only differ by the order of adjacent symbols which are independent according to a dependency relation \( D \), where two operations \( \alpha \) and \( \beta \) are said to be dependent, if there exists at least one variable which is explicitly referenced by both \( \alpha \) and \( \tau \).
**Definition 3.4:** A trace is an equivalence class of histories, denoted by \([J, w]\) where \(J\) is the common initial state and \(<J, w>\) is some member of the equivalence class.

Next, we formally describe the operations for the translation. The following definition describes the actions with respect to auxiliary variables in the translation.

**Definition 3.5:** Let \(T_j\) denote the local counter of a process \(P_j\). The actions performed in the translation include updates of auxiliary variables, sending and receiving messages, which are described below.

- **send** \((P_j, t, v)\): \((P_j!v, t)\) in CSP [Hoar69] notion, which denotes that a message with content \(v\) is sent to process \(P_j\) at time \(T_j = t\) with respect to the local counter \(T_j\) of process \(P_j\).

- **receive** \((P_j, t, v)\): \((P_j?v, t)\) in CSP notion, which denotes that a message with content \(v\) is received from process \(P\) at time \(T_j = t\) with respect to the local counter \(T_j\) of process \(P_j\).

- **(P_j, t, var, val)**: at time \(T_j = t\) the variable \(var\) has value \(val\) in process \(P_j\).

The tuple \((P, t, var, val)\) denotes an update of some auxiliary variable, while **send** and **receive** are used to communicate auxiliary variables. The counter \(T_i\) of a process \(P_i\) is incremented by one after every execution of an operation and is updated after the receipt of a message. The incorporation of logical clocks into the translation is to obtain a total ordering of all causally related events of the system, which is based on the concept of “happened before” relation [Lamp78].

To keep track of operations, each process must maintain a history that records all the operations performed and observed so far, which is defined below.

**Definition 3.6:** Let \(V_{h_j}\) be the collection of operations observed by process \(P_j\), where the operations are described in Definition 3.5. \(V_{h_j}\) is used to keep track of the operations including updates of auxiliary variables, communications of non-auxiliary variables, and communications of auxiliary variables.

Each process keeps a collection of sets of auxiliary variables with respect to the other processes in the system, so that every process has state information of other processes and can evaluate whether an assertion about the behavior of other processes is satisfied. The following definition describes the auxiliary variables maintained by process \(P_j\) with respect to \(n\) processes in the system.

**Definition 3.7:** Let \(G_j\) be a collection of subsets \(g_{j0}, g_{j1}, \ldots, g_{j(n-1)}\), where each subset is a set of operations from Definition 3.5. The set \(g_{ji}(j \neq i)\) represents the changes made to the auxiliary variables in \(P_j\) since the last communication with \(P_i\); \(g_{jj}\) describes the auxiliary variables updated by process \(P_j\) since the last communication with any process.
The set $g_{jj}$ is updated whenever $P_j$ makes an assignment to an auxiliary variable and resets to the empty set at communication time. The following definition describes that before the communication with process $P_i$, process $P_j$ updates $(G_j \setminus \{g_{jj}\})$ as well as the history $V_{h_j}$ with respect to the operations in $g_{jj}$.

**Definition 3.8:** The function $update_\sigma(g_{ji}, g_{jj}, V_{h_j}, T_j)$ describes the operations performed by $P_j$ before the communication with $P_i$.

\[
update_\sigma(g_{ji}, g_{jj}, V_{h_j}, T_j)
\]

(1) apply each operation in $g_{jj}$ to $(G_j \setminus \{g_{jj}\})$;
(2) $V_{h_j} \leftarrow V_{h_j} \| g_{jj}$;
(3) $g_{jj} \leftarrow \emptyset$;
(4) $T_j := T_j + 1$;

The above equations are explained below.

(1) update $(G_j \setminus \{g_{jj}\})$ with respect to $g_{jj}$, where "\" denotes set difference.
(2) record operations in $g_{jj}$, where "\|" denotes concatenation.
(3) $g_{jj}$ is set to empty, since $(G_j \setminus \{g_{jj}\})$ are updated.
(4) increment the counter $T_j$.

Before the communication, processes $P_i$ and $P_j$ perform their respective updates of $update_\sigma(g_{ij}, g_{ii}, V_{h_i}, T_i)$ and $update_\sigma(g_{ji}, g_{jj}, V_{h_j}, T_j)$; $g_{ij}$ and $g_{ji}$ are exchanged when $P_i$ and $P_j$ communicate. The operations in $update_\sigma$ are performed before the communication. For example, let $v = x\alpha\beta z$ be a sequence of operations, where $\alpha$ is the send operation. Then the sequence of operations to be executed after the translation is $v' = xy\alpha\beta z$ where $y$ denotes the operations in $update_\sigma$. Later, it will be shown that if an assertion holds in history $[J, v]$ then that assertion holds in history $[J, v']$, where $J$ is a given initial state.

The following definition formally defines the operations following the communication of non-auxiliary variables.

**Definition 3.9:** The function $update_\rho$ describes the communications of auxiliary variables and the updates following the exchanges of auxiliary variables. Let $V_{h_i}$ be the collection of operations observed by $P_i$ and let $g^i_{recv}$ denote the auxiliary variables received by $P_i$ from $P_j$. The following function describes the operations performed by $P_i$.
The above equations are explained as follows.

1. receive auxiliary variables $g_{j\text{recv}}$ from process $P_j$ and increment the counter $T_i$.  
2. record the operation receive in $V_{h_i}$.  
3. send $g_{ij}$ to process $P_j$ and increment the clock $T_i$.  
4. record the operation send in $V_{h_i}$.  
5. update $g_{ij}$ with the new values received.  
6. record the operations of $g_{j\text{recv}}$ in history $V_{h_i}$.  
7. increment the counter $T_i$.

In other words, after the exchange of auxiliary variables between $P_i$ and $P_j$, the old values in $g_{ji}(g_{ij})$ common to the received new values from communication will be replaced by the new values in $g_{j\text{recv}}(g_{i\text{recv}})$. Note that the function $update_\rho(g_{ji}, g_{j\text{recv}}, V_{h_i}, T_j)$, which describes the operations performed by $P_j(j<i)$, has the same operations as in $update_\rho(g_{ij}, g_{i\text{recv}}, V_{h_i}, T_i)$ except that the send operation is executed before the receive operation. Notice that $(j<i)$ is used to introduce an arbitrary order to the communications of auxiliary variables, which avoids the occurrence of deadlock. The interchange of auxiliary variables is described in Figure 3.1 for one matching communication pair between $P_i$ and $P_j$.

### 3.2 Soundness and Completeness of the Translation

This section shows that the assertions derived from the verification proof can be preserved after the transformation. Figure 3.3 describes the translation process for one matching communication pair of send* and receive* of Figure 3.2, where send* and receive* denote the operations of $(P!VAR,t)$ and $(Q?VAR,t)$, respectively. In Figure 3.3, the updates before the matching(primary) communication $(C_{i,t}(C_{j,t}))$ represent the updates of auxiliary variables before the communication, and the two pairs of
For process \( P_i \):

/* execute arbitrary set of statements excluding communication */
/* but including assignments to auxiliary variables */
\( S_{i1}; T_i := 1 \)
\( S_{i2}; T_i := T_i + 1 \)
\[ \ldots \]
\( S_{ik}; T_i := T_i + 1 \)
/* update the auxiliary variables */
update\( _p (g_{ij}, g_{ii}, V_h, T_i) \);
/* perform communications with process \( P_j \) */
\( (P_j ? \text{VAR}, t); T_i := \max(T_i, t) + 1 \)
/* and update the auxiliary variables */
update\( _p (g_{ij}, g_{jrecv}, V_h, T_i) \);

For process \( P_j \):

/* execute arbitrary set of statements excluding communication */
/* but including assignments to auxiliary variables */
\( S_{j1}; T_j := 1 \)
\( S_{j2}; T_j := T_j + 1 \)
\[ \ldots \]
\( S_{jk}; T_j := T_j + 1 \)
/* update the auxiliary variables */
update\( _p (g_{ji}, g_{jj}, V_h, T_j) \);
/* perform communications with process \( P_i \) */
\( (P_i ! \text{VAR}, t); T_j := T_j + 1 \)
/* and update the auxiliary variables */
update\( _p (g_{ij}, g_{irecv}, V_h, T_j) \);

**Figure 3.1.** communications of auxiliary variables for one matching communication pair.

\( send \) and \( receive \) denote the message exchanges of sets of auxiliary variables maintained in \( P \) and \( Q \).

In a verification proof of an underlying algorithm, we start from initial computations which satisfies the property \( p \), establish assertions for local and sequential operations within each process, and according to the fairness requirement, processes will come to some synchronized state and thus conclude \( (p \rightarrow \text{EF} q) \). Before the communication, sequential operations within each process proceed independently, i.e., a transition in one process cannot enable or disable transitions in another process. Hence, after the transformation the same assertions can be established before the communication, since the operations of update are local computations and only involve auxiliary variables instead of program variables. However, if the receipts of messages are not in order after transformation, then that could
result in testing assertions different from the assertions expected. In other words, we may conclude the same assertions only if the communication events occur in the same order after the transformation.

The arrival of messages in order after the transformation is guaranteed by the introduction of “happened before” relation. In the translation, we incorporate this notion from [Lamp78], which defines a total ordering of all causally related events of distributed systems.

For simplicity, within the proofs each transition is referenced by its name denoted below the arrow instead of its operation denoted above the arrow, as shown in Figure 3.2 and Figure 3.3.

**Theorem 3.10: (Soundness)** If $p$ is a property of an underlying algorithm, then after the translation we can derive $p$ from its corresponding error-detecting algorithm.

**Proof:** Let $< J, x >$ be the history before the matching communication ($C_{a}C_{b}$) in Figure 3.2 and let $< J, y >$ be the histories before the matching communication ($C_{i+1}C_{j+1}$) in Figure 3.3. Let $[J, xC_{a}C_{b}]$ and $[J, yC_{i}C_{j}C_{i+1}C_{j+1}w]$ denote the computations of Figure 3.2 and Figure 3.3, respectively. Since the translation only affects the communication transitions, $x$ and $y$ have the same operations and may be different by the order of independent operations. Thus, the histories $< J, x >$ and $< J, y >$ are equivalent, which follows from partial order semantics. Thus, $< J, x > \models p \iff < J, y > \models p$.

By assumption we know that $p$ is a property of the underlying algorithm, i.e., $[J, xC_{a}C_{b}] \models p$, which implies there exists a proof of $p$ from weakest predicate $wp(\tau, p)$ where $\tau$ is the synchronized communication ($C_{a}C_{b}$) and $wp(\tau, p) = enabled(\tau) \land p[f/(\overline{f}/\overline{f})]$. Since $\tau$ is enabled after the execution of $x$
and $p[f_{i}(\bar{y})/\bar{y}]$ holds because on error-free environments the communication event $\tau$ has the correct effect on substituting $i \in 1, \ldots , |\bar{y}|$, the $i^{th}$ element of $f_{\tau}(\bar{y})$ for each occurrence of $\bar{y}$, $wp(\tau, p)$ is true over the interpretation of $[J, x]$. Extend the model of $[J, x]$ by adding the assignments of $(C, C_{j})$, $wp(\tau, p)$ holds over the new interpretation $[J, xC_{i}C_{j}]$, because the operations of update($C_{i}C_{j}$) affect neither program variables nor the control of flow. Then we can conclude $p$ from $wp(\tau, p)$ by the execution of synchronized communication $(C_{i+1}C_{j+1})$, i.e. $[J, xC_{i}C_{j}C_{i+1}C_{j+1}]$ holds and $p$ is a property of the corresponding error-detecting algorithm.□

**Theorem 3.11:** (Relative Completeness) If $p$ is a property of the error-detecting algorithm then $p$ can be derived from its corresponding underlying algorithm.

**Proof:** Let $[J, xC_{i}C_{j}]$ and $[J, yC_{i}C_{j+1}C_{j+1}]$ denote the computations of Figure 3.2 and Figure 3.3, respectively.

By assumption we know that $p$ is a property of the error-detecting algorithm, which is to say $[J, yC_{i}C_{j+1}C_{j+1}] \models p$. Let $\tau$ be the synchronized communication $(C_{i+1}C_{j+1})$. Since $\tau$ is enabled after the execution of $(yC_{i}C_{j})$ and $p[f_{\tau}(\bar{y})/\bar{y}]$ holds because in error-free environments the communication event $\tau$ has the correct effect on substituting $i \in 1, \ldots , |\bar{y}|$, the $i^{th}$ element of $f_{\tau}(\bar{y})$ for each occurrence of $\bar{y}$, $wp(\tau, p) = enabled(\tau) \land p[f_{\tau}(\bar{y})/\bar{y}]$ is true over the interpretation of $[J, yC_{i}C_{j}]$. Thus, we obtain $[J, yC_{i}C_{j}] \models wp(\tau, p)$.

Since the updates $(C_{i}C_{j})$ affect neither program variables nor the control of flow, we may still have $wp(\tau, p)$ after removing the interpretation of $(C_{i}C_{j})$ from the model of $[J, yC_{i}C_{j}]$, i.e., $[J, y] \models wp(\tau, p)$. From the previous proof, we know that the histories of $< J, x >$ and $< J, y >$ belong to the same equivalent class of histories. That is, $[J, x] \models wp(p)$.

Because the synchronized communication is the only operation immediately following the execution of $x$ on the underlying algorithm and this communication event transforms the assertion $wp(\tau, p)$ to $p$, $[J, xC_{i}C_{j}] \models p$ follows, i.e., $p$ holds over the underlying algorithm.□

Suppose $p$ is derived from the error-detecting algorithm which has two message exchanges. Clearly $p$ is derived from the second communication event $\tau_{2}$ and $q_{1} = wp(\tau_{2}, p)$ holds since we have shown Theorem 3.11 holds for one message exchange. The weakest predicate $wp(\tau_{1}, q_{1})$ holds, since the communication event $\tau_{1}$ is enabled after the execution of its previous operations and $\tau_{1}$ has the correct effect on the assertion $q_{1}$ under error-free environments. Thus, $wp(\tau_{1}, q_{1})$ holds before the communications. Then we can conclude that $p$ holds on the underlying algorithm by successively applying concatenations(compositions) according to the operations of the underlying algorithm.

Likewise we can inductively show that $p$ can be derived from the underlying algorithm when the number of message exchanges is $n$. 
4. Formal Generation of Executable Assertions

The result of Section 3.2, soundness and completeness of the translation, shows that the assertions derived from the verification proof are preserved after the translation. However, to guarantee that the physical state, in the actual run time environment, is consistent with the logical state specified in the assertion, run-time assertions or executable assertions are needed. This section shows that executable assertions can be obtained by the evaluation of value liveness assertions against the computation history of a program.

In the translation, a process \( P_i \) maintains a history \( V_{ih} \) to keep track of the operations observed so far. This history \( V_{ih} \) records updates of auxiliary variables and process communications including the communications of auxiliary variables. Thus, \( V_{ih} \) includes all the observable operations in \( v \) of the history \([J,v]\). Since value liveness assertions are derived from the observable operations, specifically the synchronized communications, testing a value liveness assertion is equivalent to checking this assertion against the history \( V_{ih} \). The following definitions formally describe how a process can utilize its history to evaluate value liveness assertions.

**Definition 4.1:** Let \( \Pi \) be a mapping, which maps a value liveness assertion into \( \emptyset \) or a pair of tuples with respect to the history \( V_{ih} \) of a process \( P_i \), i.e., \( \Pi(p \rightarrow EF q) = \emptyset \) denotes that the assertion \((p \rightarrow EF q)\) is not satisfied with respect to the history \( V_{ih} \), while \( \Pi(p \rightarrow EF q) = ((P, t_1, \text{var}, \text{val}), (Q, t_2, \text{var}', \text{val}')) \) denotes that the tuple \((P, t_1, \text{var}, \text{val})\) satisfies the assertion \( p \), \((Q, t_2, \text{var}', \text{val}')\) satisfies the assertion \( q \), and \((t_1 < t_2)\).

In other words, the assertion \((p \rightarrow EF q)\) is satisfied over the execution of a program can be regarded as that in the history \( V_{ih} \) of \( P_i \), there exist two tuples \( \text{tuple}_r \) and \( \text{tuple}_s \), satisfying the respective assertions of \( p \) and \( q \), and \( \text{tuple}_r \) happened before \( \text{tuple}_s \). Figure 4.1 shows the pseudocode \( \Xi \) of executable assertions which examines the assertion \((p \rightarrow EF q)\).

\[
\text{If } \neg \left[ \Pi(p \rightarrow EF q) = ((P, t_1, \text{var}, \text{val}), (Q, t_2, \text{var}', \text{val}')) \right] \\
\text{then ERROR.}
\]

**Figure 4.1:** \( \Xi \) examines value liveness assertions.

Executable assertions denoted by \( \Xi(p \rightarrow EF q) \) measure the validity of assertions at run time, hence, we embed them into the program to monitor the run time behavior of the program.

4.1 Faulty Clock Problem

In the translation, each process maintains a logical clock which is updated based on the notion of “happened before” relation [Lamp78]. The incorporation of “happened before” relation into the
The N-puzzle problem [Quin88, LuSM92b] is described below. A search strategy is assumed and an optimization function is applied to reduce exponential search space. For our model problem of branch and bound N-puzzle, the best-first approach is employed to demonstrate the application of Changeling to obtain error-detecting algorithms. These algorithms have been applied to many well-known problems such as the Traveling Salesman problem and the N-puzzle problem. These problems typically have exponential search space and heuristics such as breadth-first, depth-first or best-first search are applied to find optimal solutions. For our model problem of branch and bound N-puzzle, the best-first search strategy is assumed and an optimization function is applied to reduce exponential search space. The N-puzzle problem [Quin88, LuSM92b] is described below.

From partial order semantics, we know that whether \( p \) holds over the history (computation) \([J, \tau_{n-1} \land y \tau_n \land z \tau_{n+1}]\) depends on whether the sequence of communication events are performed in order in spite of the occurrence of faulty clock value \( t \). Let \( p \) be a formula of the form \( (r \rightarrow \text{EF}s) \), the formula \( (r \rightarrow \text{EF}s) \) holds over the interpretation \([J, \tau_{n-1} \land y \tau_n \land z \tau_{n+1}]\), only if the sequence of observable or communication operations in \([J, \tau_{n-1} \land y \tau_n \land z \tau_{n+1}]\) is maintained despite of the presence of faults.

5. APPLICATION

This section employs branch and bound algorithms to demonstrate the application of Changeling to obtain error-detecting algorithms. These algorithms have been applied to many well-known problems such as the Traveling Salesman problem and the N-puzzle problem. These problems typically have exponential search space and heuristics such as breadth-first, depth-first or best-first search are applied to find optimal solutions. For our model problem of branch and bound N-puzzle, the best-first search strategy is assumed and an optimization function is applied to reduce exponential search space. The N-puzzle problem [Quin88, LuSM92b] is described below.
Figure 5.1: Objective of the N-puzzle problem (N=8)

Figure 5.1 shows an example of initial and final board configurations for N-puzzle problem. Each board configuration has N+1 tile positions with N tiles distinctly numbered from 1 to N and one blank space denoted by 0. A tree structure is used to represent the possible paths of obtaining the final configuration; each node in the tree denotes a board configuration and is referred to as a state \( s \). The goal is to move the tiles to a known final board configuration from a given initial configuration with minimal moves, which is shown in Figure 5.2.

Figure 5.2: An abstraction of the N-puzzle problem (N=8)

In the N-puzzle problem, the problem is solved when the final configuration is reached by a minimal number of moves. To reduce the search space, an optimization function is introduced, which is defined below.
Definition 5.1: The cost of a board configuration or a state $s_i$ at level $k$ is determined by a function $f$, where $f(s_i) = md_i + k$ and $md_i$ is the manhattan distance of the state $s_i$.

The manhattan distance is the sum of distances that each tile must move to its appropriate position, while the level $k$ denotes the number of legal moves taken so far. Thus, the optimization function $f$ is the sum of manhattan distance of tiles plus the level of the tree. The following two theorems describe the properties of function $f$.

Theorem 5.2: [KoSt74] Let $s$ represent a minimal cost node according to the function $f$, where $f$ is monotonically nondecreasing. Then $s$ is an optimal node in the search space.

Theorem 5.3: [LuSM92b] The optimization function $f$ is monotonically nondecreasing as the level $k$ increases.

By Theorems 5.2 and 5.3, a solution is found when the manhattan distance decreases to zero, i.e. $f(s_i) = 0 + k$, where $s_i$ is a solution node and $k$ is the number of moves taken to solve the puzzle. This value $k$ then acts as an upper bound to the problem such that all the nodes with higher bounds than $k$ can be pruned.

Figure A.1 shows the distributed branch and bound N-puzzle algorithm. Initially process $i$ distributes tasks to process $j$. Then each process works independently on lowest cost paths, and broadcasts when a solution is found or some process becomes idle. The algorithm terminates when the task set is empty, i.e. all of the nodes have been either expanded or discarded.

It is assumed that there exists a solution. For simplicity, the steps of choosing a candidate node (board configuration) for exploration are denoted by “expansion”. Thus, during the expansion a process will continuously branch a node of lowest cost and calculate the costs of its child nodes until a solution is found. Once a solution is found, processes exchange their solutions. Hence, those nodes that have bounds higher than current best solution can be pruned. This avoids exhaustive explorations and reduces the search space. To achieve load balancing, communication is needed, which allows idle processes to accept tasks from other processes.

Figure B.1 shows the Branch and Bound N-puzzle algorithm. The formal generation of executable assertions in Section 4.1 allows us to derive the corresponding error-detecting algorithm, which is presented in Figure B.7.

6. CONCLUSION

This paper extends the Changeling methodology to include responsiveness properties. Changeling utilizes executable assertions from the temporal specifications of Interleaving Set Temporal Logic (ISTL*), where the assertions are value liveness assertions($p \rightarrow EFq$) which ensure what values the program variables must possess at a state satisfying assertion $q$, starting from a state satisfying assertion $p$. A translation process which takes into account faulty distributed operational environments was
developed to generate executable assertions from value liveness assertions. Then these executable assertions are embedded into the code to create run-time-assured programs. Moreover, the translation is shown to be sound and relatively complete, and an example of ensuring value liveness assertions is presented for distributed Branch and Bound type problems.

Future research will examine executable assertions derived from responsiveness properties for the run time analysis of responsive computing systems.

Appendix A.

This appendix presents the proof rules of ISTL∗; for details the reader may refer to [KaPe88, PePn90].

SS-TRANS: This rule handles the case where progress is made by one single operation and derives the temporal formula of the form EXq from premises which are all state formulas.

\[
\begin{align*}
(1) & \quad p \rightarrow \phi \\
(2) & \quad \phi \rightarrow enabled(\tau) \\
(3) & \quad \{ \phi \} T - \tau \{ \phi \lor q \} \\
(4) & \quad \{ \phi \} \tau \{ q \} \\
\hline
p \rightarrow EXq
\end{align*}
\]

To justify this rule, suppose that the premises are all valid, consider a computation with a p-state at \(i\). By premise (1), \(p\) implies \(\phi\) and from premises (2) and (3), \(\phi\) holds at \(i\) and all subsequent states until a q-state is reached. Hence a q-state is reached or the transition \(\tau\) is taken, which results in a q-state by premise (4). The desired conclusion follows.

TRANS: This transitivity rule combines a finite number of successive constraints into a more complicated property. The state formula \(r\) is called the link of the rule.

\[
\begin{align*}
& \quad r \rightarrow EFq \\
\hline
& \quad p \rightarrow EFq
\end{align*}
\]

CONF: This confluent rule allows us to prove eventual properties that result from combining a number of parallel eventual assertions.

\[
\begin{align*}
(1) & \quad p \rightarrow EF(r \lor s) \\
(2) & \quad r \rightarrow EFq \\
(3) & \quad s \rightarrow EFq \\
\hline
p \rightarrow EFq
\end{align*}
\]

To justify this rule, assume that the premises are all valid, and consider an arbitrary computation containing a p-state. By premises (1), a p-state is followed by an r-state or an s-state, both of which are followed by a q-state. The conclusion follows.

Rule WIND: This rule applies induction over well-founded sets, which is commonly used in liveness proofs. Assume that the variable \(\alpha\) ranges over the natural numbers \(\mathbb{N}\) and \(n \in \mathbb{N}\).

\[
\begin{align*}
(1) & \quad p \rightarrow \phi(n) \\
(2) & \quad (\forall i)(\phi(\alpha) \land \alpha = i \land i > 0) \\
& \quad \quad \quad \quad EF(\phi(\alpha) \land (\exists j)(\alpha = j \land i > j)) \\
(3) & \quad \phi(0) \rightarrow q \\
\hline
p \rightarrow EFq
\end{align*}
\]
\( \phi(\alpha) \land \alpha = i \) denotes that the state formula which results from the invariant \( \phi(\alpha) \) by replacing all occurrences of the variable \( \alpha \) with the value \( i \). To justify this rule, suppose that the premises are all valid, and consider a computation containing a \( p \)-state \( \sigma_{k_n} \). By premise (1), \( \phi(n) \) holds at \( \sigma_{k_n} \). From premise (2), it follows that there is a sequence of positions \( k_n \geq k_{n-1} \geq \ldots \geq k_0 \), such that each \( \sigma_{k_i} \) (\( 0 \leq i \leq n \)) is a \( \phi(i) \)-state and the index \( \alpha \) is decreasing in that sequence. In other words, there exists a sequence of assertions, \( \sigma_{k_n}, \sigma_{k_{n-1}}, \ldots, \sigma_{k_0} \), where the index is decreasing.

**Appendix B.**

This appendix shows the termination of branch and bound N-puzzle algorithm of Figure B.1. Since, in this example, all the processes are doing the same computations, we only show the verification involving \( P_i \) and \( P_j \). The rest of the proofs are symmetric and can be derived similarly. Note that for the synchronized communications between \( P_i \) and \( P_j \) where \( i < j \), \( P_i \) will send its message to \( P_j \) first and wait for the message from \( P_j \). On the contrary, \( P_j \) will not send its message to \( P_i \) until it receives the message from \( P_i \). For clarity, superscript \( i \) is used to identify a variable being a local variable of process \( i \). The following variables are used in Figure B.1:

- \( \text{task}_i \): task set of process \( i \)
- \( S_{\text{current}} \): current optimal solution
- \( S_{\text{temp}} \): tasks migrated to the other process
- \( S_i \): current task set in process \( i \), empty or not

\( S_{\text{recv}} \): solution received from the other process

\( P_j ::
\]
/* distribute task to process \( j */
\[ l_0: < P_j!\text{task}_0 > \]
\[ l_1: \text{while } \neg \text{end' } >
\]
/* work on lowest cost paths */
\[ l_2: \text{expansion}; \]
/* broadcast current solution */
\[ l_3: [ < P_j!(S_i, S_i) > \rightarrow \text{skip} ]; \]
\[ l_4: [ < P_j?(S_{\text{recv}}, S_j) > \rightarrow
\]
\[ [ l_5: < S_{\text{recv}} < S_{\text{current}} > \rightarrow < S'_{\text{current}} = S_{\text{recv}} > ]; \]
/* tasks migration */
\[ l_6: [ \neg S_i \neq \varnothing \land S_j = \varnothing > \rightarrow < P_j!S_{\text{temp}} \rightarrow \text{skip} ]; \]
\[ < S_i = \varnothing \land S_j \neq \varnothing > \rightarrow < P_j?S_{\text{temp}} \rightarrow \text{skip} ]; \]
\[ < S_i = \varnothing \land S_j = \varnothing > \rightarrow < \text{end' } = \text{true } ]; \]
\[ l_7: \text{end}; \]

**Figure B.1**: Branch and Bound N-puzzle Algorithm
Theorem B.1: $\phi_1 \rightarrow \text{EF}\phi_2$, where $\phi_1 = \text{at}(l_i^0) \land \text{at}(l_j^0)$, $\phi_2 = (\text{at}(l_i^1) \land \text{at}(l_j^1) \land (\text{task}_j = \text{task}_0))$, (task$_j = \text{task}_0$) denotes that $P_j$ receives its task set from $P_i$. This theorem shows that $P_i$ retains task$_i$ and distributes task$_0$ to $P_j$.

Proof: Let $\theta$ be the initial condition for the Branch and Bound algorithm. Thus, $\theta$ implies $\phi_1$. First, to prove $\theta \rightarrow \text{EF}(\text{at}(l_i^0) \land \text{at}(l_j^0))$, by Rule CONF it suffices to show the premises

1. $\theta \rightarrow \text{EF}(\text{at}(l_i^0) \lor \text{at}(l_j^0))$
2. $\text{at}(l_i^0) \rightarrow \text{EF}(\text{at}(l_i^0) \land \text{at}(l_j^0))$
3. $\text{at}(l_j^0) \rightarrow \text{EF}(\text{at}(l_i^0) \land \text{at}(l_j^0))$

The above three premises are valid since initially the control of $P_i$ and $P_j$ resides in $l_i^0$ and $l_j^0$, respectively. Then the matching synchronous communication transitions establish the desired conclusion.
Theorem B.2: $\phi_2 \rightarrow \text{EF}\phi_3$, where

$\phi_2 = \text{at}(l'_1) \land \text{at}(l_1)$,

$\phi_3 = (\text{at}(l'_2) \land \text{at}(l'_3) \land (S_{\text{recv}}^{l_1} = S_{\text{new}}^{l_1}) \land (S_i = S_j) \land (\alpha' < \alpha)).$

$(S_i = S_j)$ denotes that process $P_j$ is informed of the current task set in process $P_i$, $(S_{\text{recv}}^{l_1} = S_{\text{new}}^{l_1})$ asserts that $P_j$ receives the current best bound of $P_i$, and $(\alpha' < \alpha)$ represents the total number of unexplored states is decreasing. This theorem shows that each process works independently on its lowest cost path (expansion), and broadcasts when a solution is found. (See Figure B.3)

Proof: Let the assertion $\text{start}$ hold initially before the computation. Thus, $\text{start}$ implies $\phi_2$. To establish $\text{start} \rightarrow \text{EF(} \text{at}(l'_1) \land \text{at}(l'_3)\text{)}$ by Rule CONF it suffices to show the following premises

1. $\text{start} \rightarrow \text{EF(} \text{at}(l'_1) \lor \text{at}(l'_3)\text{)}$
2. $\text{at}(l'_1) \rightarrow \text{EF(} \text{at}(l'_1) \land \text{at}(l'_3)\text{)}$
3. $\text{at}(l'_3) \rightarrow \text{EF(} \text{at}(l'_3) \land \text{at}(l'_1)\text{)}$

Premise (1) is obviously valid and premise (2) says that $P_i$ will progress to $\text{at}(l'_1)$. This premise can be derived by the transitive rule (TRANS). It suffices to show the premises: (a) $\text{start} \rightarrow \text{EXat}(l'_2)$ and (b) $\text{at}(l'_3) \rightarrow \text{EFat}(l'_2)$. Premise (a) is derived according to the following premises via single step progress rule (SS-TRANS).

$\text{start} \rightarrow \text{at}(l'_1)$

$\text{at}(l'_1) \rightarrow \text{enabled}(\tau)$

$\{ \text{at}(l'_0) \} \tau \rightarrow \{ \text{at}(l'_1) \land \text{at}(l'_3) \}$

$\{ \text{at}(l'_0) \} \tau \rightarrow \{ \text{at}(l'_1) \}$

All of them are trivially valid with respect to one single transition $= l'_1 \rightarrow l'_2$. Notice that the transitions between $l'_2$ and $l'_3$ are not detailed, thus, we omit the proof of $(\alpha' < \alpha)$ from these sequential and non-communication transitions. However, since the set of all nodes to be examined is $N^N$ and each expansion either examines or prunes some nodes, the set of nodes that have not been examined is decreasing.

Premise (3) is similar to premise (2). Thus, we establish $\text{start} \rightarrow \text{EF(} \text{at}(l'_1) \land \text{at}(l'_3) \land (\alpha' < \alpha)).$ By synchronous communication transitions we derive $\text{at}(l'_1) \land \text{at}(l'_3) \rightarrow \text{EF(} \text{at}(l'_1) \land \text{at}(l'_3) \land (S_{\text{recv}}^{l_1} = S_{\text{new}}^{l_1}) \land (S_i = S_j) \land (\alpha' < \alpha)).$ The conclusion follows.
**Theorem B.3:** \( \phi_1 \rightarrow E\!F \phi_2 \), where \( \phi_1 = \text{at}(l_2') \land \text{at}(l_2') \), 
\( \phi_2 = (\text{at}(l_2') \land \text{at}(l_2') \land (S_{\text{recv}} = S_{\text{current}}) \land (S_i = S_j)) \).

\( S_{\text{recv}} = S_{\text{current}} \) asserts that \( P_i \) receives the current best bound of \( P_j \), and \( (S_j = S_i) \) denotes that \( P_i \) is informed of the current task set in \( P_j \). This example shows that \( P_i \) receives the current best bound and the current task set of \( P_j \). (See Figure B.4)

**Proof:** Let the assertion \( \text{start} \) hold initially before the execution. Thus, \( \text{start} \) implies \( \Phi_i \).

To prove \( \text{start} \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \), we first apply confluent rule (CONF). It suffices to show the premises

1. \( \text{start} \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
2. \( \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
3. \( \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)

Premise (1) and (2) are valid since initially the control of \( P_i \) and \( P_j \) are at \( l_2' \) and \( l_2' \) respectively.

Premise (3) requires a case analysis on \( (S_{\text{recv}} < S_{\text{current}}) \) that determines which path is taken. The assertion \( \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2')) \) is implied by the following two formulas:

\[
\text{at}(l_2') \land (S_{\text{recv}} < S_{\text{current}}) \rightarrow \text{EF}(\text{at}(l_2') \land (S_{\text{current}} = S_{\text{recv}}))
\]

\[
\text{at}(l_2') \land (S_{\text{recv}} \geq S_{\text{current}}) \rightarrow \text{EF}(\text{at}(l_2'))
\]

Each of them can be derived via single step progress rule (SS-TRANS) with respect to the transition \( \Rightarrow l_2' \rightarrow l_2' \). Thus, we establish \( \text{start} \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \). Then synchronous communication transition establishes

\[
\text{at}(l_2') \land \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2'))
\]

\[
\text{at}(l_2') \land (S_{\text{recv}} = S_{\text{current}}) \land (S_j = S_i), \text{ and, thus, derives the conclusion.}
\]

**Theorem B.4:** \( \phi_4 \rightarrow \text{EF} \phi_5 \), where \( \phi_4 = \text{at}(l_2') \land \text{at}(l_2') \), 
\( \phi_5 = ((\text{at}(l_2') \land \text{at}(l_2')) \lor (\text{at}(l_2') \land \text{at}(l_2'))). \) This theorem shows task migration. (See Figure B.5)

**Proof:** Let the assertion \( \text{start} \) hold initially before the computation. Thus, \( \text{start} \) implies \( \Phi_i \).

First to prove \( \text{start} \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \) by rule CONF, it suffices to show the following premises

1. \( \text{start} \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
2. \( \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
3. \( \text{at}(l_2') \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)

Premise (1) is obviously valid and premise (3) is valid due to the synchronization communication requirement. Premise (2) is implied by the following formulas

\[
\text{at}(l_2') \land (S_{\text{recv}} < S_{\text{current}}) \rightarrow \text{EF}(\text{at}(l_2') \land (S_{\text{current}} = S_{\text{recv}}))
\]

\[
\text{at}(l_2') \land (S_{\text{recv}} \geq S_{\text{current}}) \rightarrow \text{EF}(\text{at}(l_2'))
\]

Each of them can be derived by single step transitive reasoning (SS-TRANS). To conclude \( \Phi_5 \) requires a case analysis to determine which path is taken.

Three cases to consider:

1. \( (\text{at}(l_2') \land \text{at}(l_2')) \land S_i \neq \emptyset \land S_j = \emptyset \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
2. \( (\text{at}(l_2') \land \text{at}(l_2')) \land S_i = \emptyset \land S_j \neq \emptyset \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)
3. \( (\text{at}(l_2') \land \text{at}(l_2')) \land S_i = \emptyset \land S_j = \emptyset \rightarrow \text{EF}(\text{at}(l_2') \land \text{at}(l_2')) \)

which may be concluded by SS-TRANS for synchronized communication transitions.
This diagram shows that either the processes continuously execute the loops or the exit transitions \( l'_1 \rightarrow l'_2 \) and \( l'_1 \rightarrow l'_2 \) are taken. Define \( \text{enter}(l) = \text{at}(l) \land \text{completed}(\tau) \).

**Theorem B.5:** Show termination of the Branch and Bound program, i.e., \( \phi_2 \Rightarrow \text{EF} \phi_3 \):

\[
(\text{at}(l'_1) \land \text{at}(l'_1)) \Rightarrow \text{EF}(\text{at}(l'_2) \land \text{at}(l'_2))
\]

**Proof:** Let \( \text{start} \) hold initially before the computation. Applying the WIND rule, it suffices to show

1. \( \text{start} \rightarrow \text{enter}(l'_1) \land \text{enter}(l'_1) \land (\exists n, \alpha = n) \)
2. \( (\forall i)(\text{enter}(l'_1) \land \text{enter}(l'_1) \land (\alpha = i \land \alpha > 0) \Rightarrow \text{EF}(\text{enter}(l'_2) \land \text{enter}(l'_2)) \land (\exists j)(\alpha = j \land j < i)) \)
3. \( \text{enter}(l'_1) \land \text{enter}(l'_1) \land (\alpha = 0) \Rightarrow \text{EF}(\text{enter}(l'_2) \land \text{enter}(l'_2)) \)

Premise (1) is trivially valid. Premise (3) can be concluded by SS-TRANS rule with respect to the transitions \( \tau = l'_1 \rightarrow l'_2 \) and \( \tau = l'_1 \rightarrow l'_2 \), which says that the control resides at the beginning of the loop and the loop condition is false, thus, the exit transitions are taken. Premise (2) which asserts that the total number of unexplored states is decreasing at each iteration can be established by applying transitivity rule over a finite number of successive properties of \( \phi_2 \Rightarrow \text{EX} \phi_3, \phi_3 \Rightarrow \text{EX} \phi_4, \) and \( \phi_4 \Rightarrow \text{EX} \phi_5 \).

---

**Figure B.6**

\[
\begin{align*}
P_1 & \rightarrow l'_1 & \text{end}^j & \rightarrow l'_2 \\
& \downarrow & \downarrow & \\
\hat{l'}_1 & \rightarrow & \hat{l'}_2
\end{align*}
\]

\[
\begin{align*}
P_j & \rightarrow l'_1 & \text{end}^j & \rightarrow l'_2 \\
& \downarrow & \downarrow & \\
\hat{l'}_1 & \rightarrow & \hat{l'}_2
\end{align*}
\]

---

**Figure B.7:** Error-Detecting Algorithm

---

\[
P_j::
\]

\[
/* \text{distribute task to process j} */
\]

\[
\begin{align*}
\text{I}_0: & < P_j ! \text{task}_0 > \\
\text{Call } \Xi(\phi_1 \Rightarrow \text{EF} \phi_2); \\
\text{I}_1: & \text{while } < \neg \text{end}^i > \\
\text{/* \text{work on lowest cost paths} */}
\end{align*}
\]

\[
\begin{align*}
\text{I}_2: & \text{expansion}; \\
\text{/* \text{broadcast current solution} */}
\end{align*}
\]

\[
\begin{align*}
\text{I}_3: & [ < P_j \text{?} \text{S}_{\text{task}} \cdot S_j > \rightarrow \text{skip} ]; \\
\text{I}_4: & [ < P_j \text{?} \text{S}_{\text{recv}} \cdot S_j > \rightarrow \\
& \text{Call } \Xi(\phi_3 \Rightarrow \text{EF} \phi_4); \\
& [ \text{I}_1: < S_{\text{recv}} \prec S_j \rightarrow \text{skip} ]; \\
& < S_{\text{current}} = S_{\text{recv}} > ]; \\
& \text{Call } \Xi(\phi_4 \Rightarrow \text{EF} \phi_5); \\
\text{/* \text{tasks migration} */}
\end{align*}
\]

\[
\begin{align*}
\text{I}_5: & [ < S_i \notin \emptyset \land S_j = \emptyset \rightarrow < P_j \text{?} \text{S}_{\text{temp}} > \rightarrow \text{skip} ]; \\
& < S_i = \emptyset \land S_j \notin \emptyset \rightarrow < P_j \text{?} \text{S}_{\text{temp}} > \rightarrow \\
& \text{Call } \Xi(\phi_5 \Rightarrow \text{EF} \phi_6); \\
& < S_i = \emptyset \land S_j = \emptyset \rightarrow \text{skip} ]; \\
& < \text{end}^i = \text{true} > ; \\
& \text{Call } \Xi(\phi_6 \Rightarrow \text{EF} \phi_7); \\
\text{I}_6: & \text{end};
\end{align*}
\]
REFERENCES


