Synthesis of Fault-Tolerant Embedded Systems Using Games: From Theory to Practice

Chih-Hong Cheng[1], Harald Rueß[2], Alois Knoll[1], and Christian Buckl[2]

[1] Department of Informatics, Technische Universität München
Boltzmann Str. 3, Garching D-85748, Germany
[2] Fortiss GmbH, Guerickestr. 25, D-80805 München, Germany
{chengch,knoll}@in.tum.de, {ruess,buckl}@fortiss.org

Abstract. In this paper, we present an approach for fault-tolerant synthesis by combining predefined patterns for fault-tolerance with algorithmic game solving. A non-fault-tolerant system, together with the relevant fault hypothesis and fault-tolerant mechanism templates in a pool are translated into a distributed game, and we perform an incomplete search of strategies to cope with undecidability. The result of the game is translated back to executable code concretizing fault-tolerant mechanisms using constraint solving. The overall approach is implemented to a prototype tool chain and is illustrated using examples.

1 Introduction

Given a distributed straight-line program with hard real-time constraints together with a fault model we are considering the problem of synthesizing a corresponding program that tolerates the specified faults. Solving this problem is challenging as it involves complexities along several dimensions arising from interleaving, timing constraints, and non-deterministic fault appearance. In fact, already the synthesis of (untimed) distributed systems is undecidable in general.

In a first step we augment the problem description with pre-defined fault tolerance patterns such as fail-resend or voting mechanisms in order to guide synthesis. Thus our synthesis method emphasizes automated selection and instantiation of predefined FT patterns, and it includes synthesis of tedious and error-prone implementation details such as timing constraints. Given such a problem statement including a distributed program, a fault hypothesis, and a finite set of FT patterns, we translate the problem into a corresponding distributed game [10].

Solving distributed games is undecidable in general [10]. As we are mainly interested in synthesizing embedded programs with bounded resources, it is natural to restrict ourselves to the problem of searching for, say, positional strategies. It turns out that the problem of finding a positional strategy of a distributed game (for control) is NP-Complete. This result motivates our approach of translating the problem of finding positional strategies of distributed games into a corresponding SAT problem.

The final step in our synthesis approach is to transform these strategies such obtained to a executable problem. The main problem here is that these strategies only incorporate restrictions on the partial order of executions but they may not obey the given timing requirements. Based on our modeling framework, this problem is translated to a linear constraint system.

Due to lack of space we do not include complete algorithms and proofs; these can be found in [4].

2 Motivating Scenario

2.1 Adding FT Mechanisms to Resist Message Loss

We give a motivating scenario in embedded systems to facilitate our mathematical definitions. The simple system described in Figure 1 contains two *processes* $\mathcal{A}$, $\mathcal{B}$ and one bidirectional *network* $\mathcal{N}$. Processes $\mathcal{A}$ and $\mathcal{B}$ start executing sequential actions together with a looping period of $100ms$. In each period, $\mathcal{A}$ first reads an input using a sensor to variable m, followed by sending the result to the network $\mathcal{N}$ using the action `MsgSend(m)`, and outputing the value (e.g., to a log).

In process $\mathcal{A}$, for the action `MsgSend(m)`, a message containing value of m is forwarded to $\mathcal{N}$, and $\mathcal{N}$ broadcasts the value to all other processes which contain a variable named m, and set the variable m_v in $\mathcal{B}$ as $\top$ (indicating that the content is valid). However, $\mathcal{A}$ is unaware whether the message has been sent successfully: the network component $\mathcal{N}$ is unreliable, which has a faulty behavior of *message loss*. The fault type and the frequency of the faulty behavior are specified in the *fault model*: in this example for every complete period ($100ms$), at most one message loss can occur.

In $\mathcal{B}$, its first action `RecvMsg(m)` has a property describing an interval $[60, 100)$, which specifies the *release time* and *deadline* of this action to be $60ms$ and $100ms$, respectively. By posing the release time and the deadline, in this example, $\mathcal{B}$ can finalize its decision whether it has received the message m successfully using the equality constraint $(m_v = \bot)$, provided that the time interval $[40, 60)$ between (a) deadline of `MsgSend(m)` and (b) release time of `RecvMsg(m)` overestimates the *worst case transmission time* for a message to travel from $\mathcal{A}$ to $\mathcal{B}$. After `RecvMsg(m)`, it outputs the received value (e.g., to an actuator).

Due to the unreliable network, it is easy to observe that two output values may not be the same. Thus the *fault-tolerant synthesis* problem in this example is to perform suitable modification on $\mathcal{A}$ and $\mathcal{B}$, such that two output values from $\mathcal{A}$ and $\mathcal{B}$ are the same at the end of the period, regardless of the disturbance from the network.

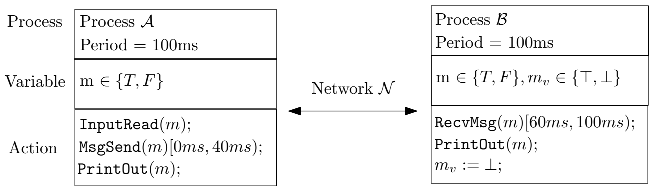

Fig. 1. An example for two processes communicating over an unreliable network

2.2 Solving Fault-Tolerant Synthesis by Instrumenting Primitives

To perform FT synthesis in the example above, our method is to introduce several slots (the size of slots are fixed by the designer) between actions originally specified in the system. For each slot, an atomic operation can be instrumented, and these actions are among the pool of predefined *fault-tolerant primitives*, consisting of message sending, message receiving, local variable modifications, or `null-ops`. Under this setting we have created a game, as the original transitions in the fault-intolerant system combined with all FT primitives available constitute the controller (player-0) moves, and the triggering of faults and the networking can be modeled as environment (player-1) moves.

3 System Modeling

3.1 Platform Independent System Execution Model

We first define the execution model where timing information is included; it is used for specifying embedded systems and is linked to our code-generation framework. In the definition, for ease of understanding we also give each term intuitive explanations.

Definition 1. *Define the syntax of the* ***Platform-Independent System Execution Model (PISEM)*** *be* $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$.

- $\mathcal{T} \in \mathbb{Q}$ *is the replication period of the system.*
- $\mathcal{A} = \bigcup_{i=1\dots n_A} \mathcal{A}_i$ *is the set of processes, where in* $\mathcal{A}_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i *is the set of variables, and* V_{env_i} *is the set of environment variables. For simplicity assume that* V_i *and* V_{env_i} *are of integer domain.*
  - $\overline{\sigma_i} := \sigma_1[\alpha_1, \beta_1); \dots; \sigma_j[\alpha_j, \beta_j); \dots; \sigma_{k_i}[\alpha_{k_i}, \beta_{k_i})$ *is a sequence of actions.*
    - $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c) \mid a \leftarrow \texttt{e} \mid \texttt{receive}(pre, c)$ *is an atomic action (action pattern), where* $a, c \in V_i$, $\texttt{e}$ *is function from* $V_{env_x} \cup V_i$ *to* V_i *(this includes* `null-op`*),* pre *is a conjunction of over equalities/inequalities of variables,* $s, d \in \{1, \dots, n_A\}$ *represents the source and destination,* $v \in V_d$ *is the variable which is expected to be updated in process* d, $n \in \{1, \dots, n_N\}$ *is the network used for sending, and* $index \in \{1, \dots, size_n\}$ *is the index of the message used in the network.*
    - $[\alpha_j, \beta_j)$ *is the execution interval, where* $\alpha_j \in \mathbb{Q}$ *is the release time and* $\beta_j \in \mathbb{Q}$ *is the deadline.*
- $\mathcal{N} = \bigcup_{i=1\dots n_N} \mathcal{N}_i$, $\mathcal{N}_i = (\mathcal{T}_i, size_i)$ *is the set of network.*
  - $\mathcal{T}_i : \mathbb{N} \rightarrow \mathbb{Q}$ *is a function which maps the index (or priority) of a message to the worst case transmission time.*
  - $size_i$ *is the number of messages used in* $\mathcal{N}_i$.

[Example] Based on the above definitions, the system under execution in section 2.1 can be easily modeled by PISEM: let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{N}$ in section 2.1 be renamed in a PISEM as $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{N}_1$. For simplicity, we use $\mathcal{A}.j$ to represent the variable j in process $\mathcal{A}$, assume that the network transmission time is 0, and let v_{env} contain only one variable v in $\mathcal{A}_1$. Then in the modeled PISEM, we have $\mathcal{N}_1 = (f : \mathbb{N} \rightarrow 0, 1)$, $\mathcal{T} = 100$, and the action sequence of process $\mathcal{A}_1$ is

$$m \leftarrow \texttt{InputRead}(v)[0, 40); \texttt{send}(true, 1, 1, 1, 2, m, \mathcal{A}_1.m)[0, 40); v \leftarrow \texttt{PrintOut}(m)[40, 100);$$

For convenience, we use $|\overline{\sigma_i}|$ to represent the length of the action sequence $\overline{\sigma_i}$, $\sigma_j.deadline$ to represent the deadline of σ_j, and $iSet(\overline{\sigma_i})$ to represent a set containing (a) the set of subscript numbers in $\overline{\sigma_i}$ and (b) $|\overline{\sigma_i}| + 1$, i.e., $\{1, \ldots, k_i, k_i + 1\}$.

Definition 2. *The configuration of $\mathcal{S}$ is $(\bigwedge_{i=1\ldots n_A}(v_i, v_{env_i}, \Delta_{next_i}), \bigwedge_{j=1\ldots n_N}(occu_j, s_j, d_j, var_j, c_j, t_j, ind_j), t)$, where v_i is the set of the current values for the variable set V_i, v_{env_i} is the set of the current values for the variable set V_{env_i}, $\Delta_{next_i} \in [1, |\overline{\sigma_i}|+1]$ is the next atomic action index taken in $\overline{\sigma_i}$*[1]*, $occu_j \in \{\texttt{false}, \texttt{true}\}$ is for indicating whether the network is busy, $s_j, d_j \in \{1, \ldots, n_A\}$, $var_j \in \bigcup_{i=1,\ldots,n_A}(V_i \cup V_{env_i})$, $c_j \in \mathbb{Z}$ is the content of the message, $ind_j \in \{1, \ldots, size_j\}$ is the index of the message occupied in the network, t_j is the reading of the clock used to estimate the time required for transmission, t is the current reading of the global clock.*

The change of configuration is caused by the following operations.

1. (*Execute local action*) For machine i, let s and j be the current configuration for var and Δ_{next_i}, and v_i, v_{env_i} are current values of V_i and V_{env_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing (all actions in $\overline{\sigma_i}$ have been executed in this cycle); else the action $\sigma_j := var \leftarrow \texttt{e}[\alpha_j, \beta_j)$ updates var from s to $\texttt{e}(v_i, v_{env_i})$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$. This action should be executed between the time interval $t \in [\alpha_j, \beta_j)$.
2. (*Send to network*) For machine i, let s and j be the current configuration for var and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else the action $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c)[\alpha_j, \beta_j)$ should be processed between the time interval $t \in [\alpha_j, \beta_j)$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$.
   - When pre is evaluated to true (it can be viewed as an `if` statement), it then checks the condition $occu_n = false$: if the condition holds, it updates network n with value $(occu_n, s_n, d_n, var_n, c_n, t_n, ind_n) := (true, i, d, v, c, 0, index)$. Otherwise it blocks until the condition holds.
   - When pre is evaluated to false, it skips the sending.
3. (*Process message*) For network j, for configuration $(occu_j, s_j, d_j, var, c_j, t_j, ind_j)$ if $occu_j = true$, then during $t_j < \mathcal{T}_j(ind_j)$, a transmission occurs, which updates $occu_j$ to $\texttt{false}$, $A_{d_j}.var$ to c_j, and $A_{d_j}.var_v$ to $\texttt{true}$.
4. (*Receive*) For machine i, let s and j be the current configuration for c and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else for $\texttt{receive}(pre, c)[\alpha_j, \beta_j)$ in machine i, it is processed between the time interval $t \in [\alpha_j, \beta_j)$ and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$[2].
5. (*Repeat Cycle*) When $t = \mathcal{T}$, t is reset to 0, and for all $x \in \{1, \ldots, n_A\}$, Δ_{next_x} are reset to 1.

Notice that by using this model to represent the embedded system under analysis, we make the following assumptions:

[1] Here an interval $[1, |\overline{\sigma_i}| + 1]$ is used for the introduction of FT mechanisms described later.

[2] In our formulation, the $\texttt{receive}(pre, c)$ action can be viewed as a syntactic sugar of `null-op`; its purpose is to facilitate the matching of send-receive pair with variable c.

– *All processes and networks in $\mathcal{S}$ share a globally synchronized clock.*
– For all actions σ, $\sigma.deadline < \mathcal{T}$; for all send actions $\sigma := \texttt{send}(pre, index, n, s, d, v, c)$, $\sigma.deadline + \mathcal{T}_n(index) < \mathcal{T}$, i.e., all processes and networks should finish its work within one complete cycle.

3.2 Interleaving Model (IM)

Next, we establish the idea of interleaving model (IM) which is used to offer an intermediate representation to bridge PISEM and game solving, such that (a) it captures the execution semantics of PISEM without explicit statements of timing, and (b) by using this model it is easier to connect to the standard representation of games.

Definition 3. *Define the syntax of the* ***Interleaving Model (IM)*** *be $S_{IM} = (A, N)$.*

- *$A = \bigcup_{i=1\dots n_A} A_i$ is the set of processes, where in $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,*
  – *V_i is the set of variables, and V_{env_i} is the set of environment variables.*
  – *$\overline{\sigma_i} := \sigma_1[\wedge_{m=1\dots n_A}[pc_{1,m_{low}}, pc_{1,m_{up}})]; \dots ; \sigma_j[\wedge_{m=1\dots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]; \dots ; \sigma_{k_i}[\wedge_{m=1\dots n_A}[pc_{k_i,m_{low}}, pc_{k_i,m_{up}})]$ is a fixed sequence of actions.*
    - *$\sigma_j := \texttt{send}(pre, index, n, s, d, v, c) \mid \texttt{receive}(pre, c) \mid a \leftarrow \texttt{e}$ is an action, where a, c, e, pre, v, n, s, d are defined similarly as in PISEM.*
    - *For σ_j, $\forall m \in \{1, \dots, n_A\}$, $pc_{j,m_{low}}, pc_{j,m_{up}} \in \{1, \dots, |\overline{\sigma_m}| + 2\}$ is the lower and the upper bound (PC-precondition interval) concerning*
      1. *precondition of program counter in machine k, when $m \neq i$.*
      2. *precondition of program counter for itself, when $m = i$.*
- *$N = \bigcup_{i=1\dots n_N} N_i$, $N_i = (T_i, size_i)$ is the set of network.*
  – *$T_i : \mathbb{N} \to \bigwedge_{m=1\dots n_A}(\{1, \dots, |\overline{\sigma_m}|+2\}, \{1, \dots, |\overline{\sigma_m}|+2\})$ is a function which maps the index (or priority) of a message to the PC-precondition interval of other processes.*
  – *$size_i$ is the number of messages used in $\mathcal{N}_i$.*

Definition 4. *The configuration of S_{IM} is $(\bigwedge_i(v_i, v_{env_i}, \Delta_{next_i}), \bigwedge_j(occu_j, s_j, d_j, c_j))$, where $v_i, v_{env_i}, \Delta_{next_i}, occu_j, s_j, d_j, c_j$ are defined similarly as in PISEM.*

The change of configurations in IM can be interpreted analogously to PISEM; we omit details here but mention three differences:

- For an action σ_j having the precondition $[\wedge_{m=1\dots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]$, it should be executed between $pc_{j,m_{low}} \leq \Delta_{next_m} < pc_{j,m_{up}}$, for all m.
- For processing a message, constraints concerning the timing of transmission in PISEM are replaced by referencing the PC-precondition interval of other processes in IM, similar to 1.
- The system repeats the cycle when $\forall x \in \{1, \dots, n_A\}$, $\Delta_{next_x} = |\overline{\sigma_x}| + 1$ and $\forall x \in \{1, \dots, n_N\}$, $occu_x = \texttt{false}$.

3.3 Games

For the proof of complexity results, we use similar notations in [10] to define a distributed game, which are games formulating multiple processes with no interactions among themselves but only with the environment. For details we refer readers to [10,4].

A *local game graph* is a directed graph $G = (V_0 \uplus V_1, E)$ whose nodes are partitioned into two classes V_0 (player-0 or control) and V_1 (player-1 or environment), and E is the set of edges. A *distributed game graph* $\mathcal{G} := (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$ can be viewed as a combination of n local games $G_1, \ldots, G_n$: during the execution player-1 can execute a global move (the translation is explicitly specified but does not need to respect the local game graph), while player-0 executes a move for all local games i which is in the player-0 vertex using his strategy f_i from his strategy set $\langle f_1, \ldots, f_n \rangle$. Notice that f_i is local, i.e., it is insensitive of contents in other subgames G_j, for all $j \neq i$.

4 Step A: Front-End Translation from Models to Games

4.1 Step A.1: From PISEM to IM

To translate from PISEM to IM, the key is to generate abstractions from the release time and the deadline information specified in PISEM. As in our formulation, the system is equipped with a globally synchronized clock, the execution of actions respecting the release time and the deadline can be translated into a partial order. Algorithm 1 concretizes this idea[3].

Starting from the initialization where no PC is constrained, the algorithm performs a restriction process using four if-statements $\{$**(1)**, **(2)**, **(3)**, **(4)**$\}$ listed.

- In (1), if $\sigma_m.releaseTime > \sigma_n.deadline$, then before σ_m is executed, σ_n should have been executed.
- In (2), if $\sigma_m.deadline < \sigma_n.releaseTime$, then σ_n should not be executed before executing σ_m.
- Similar analysis is done with (3) and (4). However, we need to consider the combined effect together with the network transmission time: we use 0 to represent the best case, and $\mathcal{T}_n(ind)$ for the worst case.

[Example] For the example in sec. 2, consider the action $\sigma_1 := m \leftarrow \texttt{InputRead}(v)[0, 40)$ in $\mathcal{A}_1$ of a PISEM. Algorithm 1 returns $mapLB(\sigma)$ and $mapUB(\sigma)$ with two arrays $[1, 1]$ and $[2, 2]$, indicated in Figure 2a. Based on the definition of IM, σ_1 should be executed with the temporal precondition that no action in $\mathcal{A}_2$ is executed, satisfying the semantics originally specified in PISEM. For the analysis of message sending time, two cases are listed in Figure 2b and Figure 2c, where the WCMTT is estimated as 15ms and 30ms, respectively.

4.2 Step A.2: From IM to Distributed Game

Here we give main concepts how a game is created after step A.1 is executed. To create a distributed game from a given interleaving model $S_{IM} = (A, N)$, we need to proceed with the following three steps:

[3] Here we assume that in each period, for all $\mathcal{N}_j$, each message of type $ind \in \{1, \ldots, size_j\}$ is sent at most once. In this way, the algorithm can assign an unique PC-precondition interval for every message type.

Algorithm 1. GeneratePreconditionPC

```
Data: PISEM model $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$
Result: Two maps $mapLB$, $mapUB$ which map from an action $\sigma$ (or a msg processing
        by network) to two integer arrays $lower[1 \dots n_A]$, $upper[1 \dots n_A]$
begin
    /* Initial the map for recording the lower and upper bound for action */
    for action $\sigma_k$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        $mapLB.put(\sigma_k$, new int[1... $n_A$](1)) /* Initialize to 1 */
        $mapUB.put(\sigma_k$, new int[1... $n_A$])
        for $\mathcal{A}_j \in \mathcal{A}$ do $mapUB.get(\sigma_k)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
        $mapLB.get(\sigma_k)$[i] = k; $mapUB.get(\sigma)$[i] = k+1; /* self PC */
    for action $\sigma_m$ in $\mathcal{A}_i$ of $\mathcal{A}$, $m = 1, \dots, |\overline{\sigma_i}|$ do
        for action $\sigma_n$ in $\mathcal{A}_j$ of $\mathcal{A}$, $n = 1, \dots, |\overline{\sigma_j}|$, $j \neq i$ do
1           if $\sigma_m.releaseTime > \sigma_n.deadline$ then
                $mapLB.get(\sigma_m)$[j] := max{$mapLB.get(\sigma_m)$[j], $n + 1$}
2           if $\sigma_m.deadline < \sigma_n.releaseTime$ then
                $mapUB.get(\sigma_m)$[j] := min{$mapUB.get(\sigma_m)$[j], $n + 1$};
    /* Initialize the map for recording the lower and upper bound for msg transmission */
    for action $\sigma_k = \mathtt{send}(pre, ind, n, s, d, v, c)$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        $mapLB.put(n.ind$, new int[1... $n_A$](1)) /* Initialize to 1 */
        $mapLB.get(n.ind)$[i] := k+1 /* Strictly later than executing send() */
        $mapUB.put(n.ind$, new int[1... $n_A$])
        for $\mathcal{A}_j \in \mathcal{A}$ do $mapUB.get(n.ind)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
    for action $\sigma_k = \mathtt{send}(pre, ind, n, s, d, v, c)$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        for action $\sigma_m$ in $\mathcal{A}_j$ of $\mathcal{A}$, $n = 1, \dots, |\overline{\sigma_j}|$ do
3           if $\sigma_k.releaseTime + 0 > \sigma_m.deadline$ then
                $mapLB.get(n.ind)$[j] := max{$mapLB.get(n.ind)$[j], $m + 1$}
4           if $\sigma_k.deadline + \mathcal{T}_n(ind) < \sigma_m.releaseTime$ then
                $mapUB.get(n.ind)$[j] := min{$mapUB.get(n.ind)$[j], $m + 1$};
end
```

Step A.2.1: Creating non-deterministic timing choices for existing actions. During the translation from a PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$ to its corresponding IM $S_{IM} = (A, N)$, for all process $\mathcal{A}_i$ in $\mathcal{A}$, for every action $\sigma[\alpha, \beta)$ where $\sigma[\alpha, \beta) \in \overline{\sigma_i}$, algorithm 1 creates the PC-precondition interval $[\wedge_{m=1 \dots n_A}[pc_{m_{low}}, pc_{m_{up}})]$ of other processes. Thus in the corresponding game, for $\sigma[\wedge_{m=1 \dots n_A}[pc_{m_{low}}, pc_{m_{up}})]$, each element $\sigma[\wedge_{m=1 \dots n_A}(pc_m)]$, where $pc_{m_{low}} \leq pc_m < pc_{m_{up}}$, is a nondeterministic transition choice which can be selected separately by the game engine.

Step A.2.2: Introducing fault-tolerant choices as $\sigma_{\frac{a}{b}}$. In our framework, fault-tolerant mechanisms are similar to actions, which consist of two parts: *action pattern* σ and *timing precondition* $[\wedge_{m=1 \dots n_A}[pc_{m_{low}}, pc_{m_{up}})]$. Compared to existing actions where nondeterminism comes from timing choices, for fault-tolerance transition choices include all combinations from (1) timing precondition and (2) action patterns available from a predefined pool.

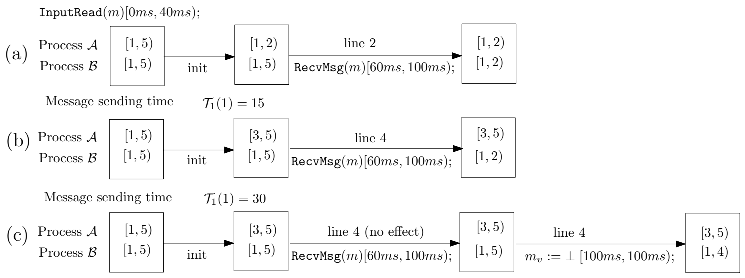

Fig. 2. An illustration for Algorithm 1

Algorithm 2. DecideInsertedFTTemplateTiming

```
Data: $\sigma_c[\wedge_{m=1\ldots n_A}[pc_{c,m_{low}}, pc_{c,m_{up}})]$, $\sigma_d[\wedge_{m=1\ldots n_A}[pc_{d,m_{low}}, pc_{d,m_{up}})]$, which are
      consecutive actions in $\overline{\sigma_i}$ of $A_i$ of $S_{IM} = (A, N)$, and one newly added action
      pattern $\sigma_{\frac{a}{b}}$ to be inserted between
Result: Temporal preconditions for action pattern $\sigma_{\frac{a}{b}}$: $[\wedge_{m=1\ldots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$
begin
    for $m = 1, \ldots, n_A$ do
        if $m \neq i$ then
            $pc_{\frac{a}{b},m_{low}} := pc_{c,m_{low}}$ /* Use the lower bound of $c$ for its lower bound */
            $pc_{\frac{a}{b},m_{up}} := pc_{d,m_{up}}$ /* Use the upper bound of $d$ for its upper bound */
        else
            $pc_{\frac{a}{b},m_{low}} := \frac{a}{b}$; $pc_{\frac{a}{b},m_{up}} := d$
end
```

We use the notation $\sigma_{\frac{a}{b}}$, where $\frac{a}{b} \in \mathbb{Q}\backslash\mathbb{N}$, to represent an inserted action pattern between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\lceil \frac{a}{b} \rceil}$. With this formulation, multiple FT mechanisms can be inserted within two consecutive actions σ_i, σ_{i+1} originally in the system, and the execution semantic follows what has been defined previously: as executing an action updates Δ_{next_i} to $min\{x|x \in iSet(\overline{\sigma_i}), x > j\}$, updating to a rational value is possible. Note that as $\sigma_{\frac{a}{b}}$ is only a fragment without temporal preconditions, we use algorithm 2 to generate all possible temporal preconditions satisfying the semantics of the original interleaving model: after the synthesis only temporal conditions satisfying the acceptance condition will be chosen.

Step A.2.3: Game Creation by Introducing Faults. In our implementation, we do not generate the primitive form of distributed games (DG), as the definition of DG is too primitive to manipulate. Instead, algorithms in our implementations are based on the variant called ***symbolic distributed games*** *(SDG)*:

Definition 5. *Define a symbolic distributed game* $\mathcal{G}_{ABS} = (V_f \uplus V_{CTR} \uplus V_{ENV}, A, N, \sigma_f, pred)$.

- V_f, V_{CTR}, V_{ENV} *are disjoint sets of (fault, control, environment) variables.*
- $pred : V_f \times V_{CTR} \times V_{ENV} \rightarrow \{\texttt{true}, \texttt{false}\}$ *is the partition condition.*

	DG	SDG
State space	product of all vertices in local games	product of all variables (including variables used in local games)
Vertex partition (V_0 and V_1)	explicit partition	use $pred$ to perform partition
Player-0 transitions	defined in local games	defined in $\overline{\sigma_i}$ of A_i, for all $i \in \{1, \ldots, n_A\}$
Player-1 transitions	explicitly specified in the global game	defined in N and σ_f

Fig. 3. Comparison between DG and SDG

- $A = \bigcup_{i=1 \ldots n_A} A_i$ *is the set of* ***symbolic local games (processes)*** *, where in* $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$*,*
  - V_i *is the set of variables, and* $V_{env_i} \subseteq V_{ENV}$*.*
  - $\overline{\sigma_i} := \bigcup \sigma_{i_1} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{1_m}} \rangle; \ldots; \bigcup \sigma_{i_k} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{k_m}} \rangle$ *is a sequence, where* $\forall j = 1, \ldots, k$*,* $\bigcup \sigma_{i_j} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{j_m}} \rangle$ *is a set of choice actions for player-0 in* A_i*.*
    - σ_{i_j} *is defined similarly as in IM.*
    - $\forall m = \{1, \ldots, n_A\}$, $pc_{i_{j_m}} \in [pc_{i_j, m_{low}}, pc_{i_j, m_{up}})$, $pc_{i_j, m_{low}}, pc_{i_j, m_{up}} \in iSet(\overline{\sigma_m})$.
  - $V_{CTR} = \bigcup_{i=1 \ldots n_A} V_i$.
- $N = \bigcup_{i=1 \ldots n_N} N_i$*,* $N_i = (T_i, size_i, tran_i)$ *is the set of network processes.*
  - T_i *and* $size_i$ *are defined similarly as in IM.*
  - $tran_i : V_f \times \{\mathtt{true}, \mathtt{false}\} \times \{1, \ldots, n_A\}^2 \times \bigcup_{i=1,\ldots,n_A} (V_i \cup V_{env_i}) \times \mathbb{Z} \times \{1, \ldots, size_i\} \rightarrow V_f \times \{\mathtt{true}, \mathtt{false}\} \times \{1, \ldots, n_A\}^2 \times \bigcup_{i=1,\ldots,n_A} (V_i \cup V_{env_i}) \times \mathbb{Z} \times \{1, \ldots, size_i\}$ *is the network transition relation for processing messages (see sec. 3.1 for meaning), but can be influenced by variables in* V_f*.*
- $\sigma_f : V_f \times V_{CTR} \times V_{ENV} \times \bigwedge_{i=1 \ldots n_A} iSet(\overline{\sigma_i}) \rightarrow V_{ENV} \times V_f \times \bigwedge_{i=1 \ldots n_A} iSet(\overline{\sigma_i})$ *is the environment update relation.*

We establish an analogy between SDG and DG using Figure 3.

1. The configuration v of a SDG is defined as the product of all variables used.
2. A play for a SDG starting from state v_0 is a maximal path $\pi = v_0 v_1 \ldots$, where
   - In v_k, player-1 determines the move $(v_k, v_{k+1}) \in E$ when $pred(v_k)$ is evaluated to `true` (`false` for player-0); the partition of vertices V_0 and V_1 in SDG is implicitly defined based on this, rather than specified explicitly as in DG.
   - A move (v_k, v_{k+1}) is a selection of executable transitions defined in N, σ_f, or A; in our formulation, transitions in N and σ_f are all environment moves[4], while transitions in A are control moves[5].
3. Lastly, a distributed positional strategy for player-0 in a SDG can be defined analogously as to uniquely select an action from the set $\bigcup \sigma_{\alpha_j} \langle \wedge_{m=1,\ldots,n_A}, pc_{\alpha_{j_m}} \rangle$, for all A_i and for all program counter j defined in $\overline{\sigma_i}$. Each strategy should be insensitive of contents in other symbolic local games.

[4] As the definition of distributed games features multiple processes having no interactions among themselves but only with the environment, a SDG is also a distributed game. In the following section, our proof of results and algorithms are all based on DG.

[5] This constraint can be released such that transitions in A can either be control (normal) or environment (induced by faults) moves; here we leave the formulation as future work.

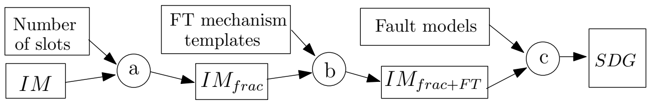

Fig. 4. Creating the SDG from IM, FT mechanisms, and faults

We now summarize the logical flow of game creation using Figure 4.

- (a) Based on the fixed number of slots (for FT mechanisms) specified by the user, extend IM to IM_{frac} to contain fractional PC-values induced by the slot.
- (b) Create $IM_{frac+FT}$, including the sequence of choice actions (for the SDG) by
  - Extracting action sequences defined in IM_{frac} to choices (step A.2.1).
  - Inserting FT choices (step A.2.2).
- (c) Introduce faults and partition player-0 and player-1 vertices: In engineering, a *fault model* specifies potential undesired behavior of a piece of equipment, such that engineers can predict the consequences of system behavior. Thus, a *fault* can be formulated with three tuples: (1) the fault type (an unique identifier, e.g., `MsgLoss`, `SensorError`), (2) the maximum number of occurrences in each period, and (3) additional transitions not included in the original specification of the system (*fault effects*). We perform the translation into a game using the following steps.
  - For (1), introduce variables to control the triggering of faults.
  - For (2), introduce counters to constrain the maximum number of fault occurrences in each period.
  - For (3), for each transition used in the component influenced by the fault, create a corresponding fault transition which is triggered by the variable and the counter; similarly create a transition with normal behavior (also triggered by the variable and the counter). Notice that our framework is able to model faults actuating on the FT mechanisms, for instance, the behavior of network loss on the newly introduced FT messages.

[Example] We outline how a game (focusing on fault modeling) is created with the example in sec. 2; similar approaches can be applied for input errors or message corruption; here the modeling of input (for `InputRead(m)`) is skipped.

- Create the predicate $pred$: $pred$ is evaluated to `false` in all cases except (a) when the boolean variable $occu$ (representing the network occupance) is evaluated to `true` and (b) when for all $i \in \{1, \ldots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$ (end of period); the predicate partitions player-0 and player-1 vertices.
- For all process i and program counter j, the set of choice actions $\bigcup \sigma_{\alpha_j}$ $\langle \wedge_{m=1,\ldots,n_A} pc_{\alpha_{j_m}} \rangle$ are generated based on the approach described previously.
- Create variable $v_f \in V_f$, which is used to indicate whether the fault has been activated in this period.
- For each message sending transition t in the network, create two normal transitions $(v_f = \texttt{true} \wedge v'_f = \texttt{true}) \wedge t$ and $(v_f = \texttt{false} \wedge v'_f = \texttt{false}) \wedge t$ in the game.
- For each message sending transition t in the network, generate a transition t' where the message is sent, but the value is not updated in the destination. Create a fault transition $(v_f = \texttt{false} \wedge v'_f = \texttt{true}) \wedge t'$ in the game.

- Define σ_f to control v_f: if for all $i \in \{1, \ldots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$, then update v_f to `false` as Δ_{next_i} updates to 1 (reset the fault counter at the end of the period).

5 Step B: Solving Distributed Games

We summarize the result from [10] as a general property of distributed games.

Theorem 1. *There exists distributed games with global winning strategy but (a) without distributed memoryless strategies, or (b) all distributed strategies require memory. In general, for a finite distributed game, it is undecidable to check whether a distributed strategy exists from a given position [10].*

As the problem is undecidable in general, we restrict our interest in finding a distributed positional strategy for player 0, if there exists one. We also focus on games with reachability winning conditions. By posing the restriction, the problem is NP-Complete.

Theorem 2. *[PositionalDG$_0$] Given a distributed game $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, an initial state $x = (x_1, \ldots, x_n)$ and a target state $t = (t_1, \ldots, t_n)$, deciding whether there exists a positional (memoryless) distributed strategy for player-0 from x to t is NP-Complete.*

Proof. The proof can be found in the extended version [4].

With the NP-completeness proof, finding a distributed reachability strategy amounts to the process of searching. For searching, we consider (a) bounded forward searching which combines the nodes in the search tree with BDDs, and (b) distributed version of the witness algorithm using SAT unrolling[6].

6 Conversion from Strategies to Concrete Implementations

Once when the distributed game has returned a positive result, and assume that the result is represented as an IM, the remaining problem is to check whether the synthesized result can be translated to PISEM and thus further to concrete implementation. If for each existing action or newly generated FT mechanism, the worst case execution time is known (with available WCET tools), then we can always answer whether the system is implementable by a full system rescheduling, which can be complicated. Nevertheless, based on our system modeling (assumption with a globally synchronized clock), perform modification on the release time and the deadline from the synthesized IM can be translated to a linear constraint system, as in IM every action contains a timing precondition based on program counters. Here we give a simplified algorithm which performs *local timing modification (LTM)*. Intuitively, LTM means to perform partitions on either

1. the interval d between the deadline of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and release time of $\sigma_{\lceil \frac{a}{b} \rceil}$, if (a) $\sigma_{\frac{a}{b}}$ exists and (b) $d \neq 0$, or
2. the execution interval of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$, if $\sigma_{\frac{a}{b}}$ exists.

[6] A sketch of the algorithm can be found in the appendix.

In the algorithm, we assume that for every action σ_d, $d \in \mathbb{N}$ where FT mechanisms are not introduced between σ_d and σ_{d+1} during synthesis, its release-time and deadline should not change; this assumption can be checked later or added explicitly to the constraint system under solving (but it is not listed here for simplicity reasons). Then we solve a constraint system to derive the release time and deadline of all FT actions introduced. Algorithm 3 performs such execution[7]: for simplicity at most one FT action exists between two actions σ_i, σ_{i+1}; in implementation this assumption is released:

- Item (1) performs a interval split between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\frac{a}{b}}$.
- Item (3) assigns the deadline of $\sigma_{\lfloor \frac{a}{b} \rfloor}$ to be the original deadline of $\sigma_{\frac{a}{b}}$.
- Item (4), (5) ensure that the reserved time interval is greater than the WCET.
- Item (6) to (11) introduce constraints from other processes:
  - Item (6) (7) (8) consider existing actions which do not change the deadline and release time; for these fetch the timing information from PISEM.
  - Item (9) (10) (11) consider newly introduced actions or existing actions which change their deadline and release time; for these actions use variables to construct the constraint.
- Item (12) is a conservative dependency constraint between $\sigma_{\frac{a}{b}}$ and a send σ_d.

7 Implementation and Case Studies

For implementation, we have created our prototype software as an Eclipse-plugin, called GECKO, targeting to offer an open-platform based on the model-based approach for the design, synthesis, and code generation for fault-tolerant embedded systems.

To evaluate our approach, here we reuse the example in sec. 2 and perform automatic tuning synthesis for the selected FT mechanisms[8]. The user now selects a set of FT mechanism templates with the intention to implement a *fail-then-resend* operation, which is shown in Figure 5a. The selected patterns introduce two additional messages in the system, and the goal is to orchestrate multiple synchronization points introduced by the FT mechanisms between $\mathcal{A}$ and $\mathcal{B}$ (the timing in FT mechanisms is unknown). The fault model assumes that in each period at most one message loss occurs.

Once when GECKO receives the system description (including the fault model) and the reachability specification, it translates the system into a distributed game. In Figure 5b, the set of possible control transitions are listed[9]; the solver generates an appropriate PC-precondition for each action to satisfy the specification. In Figure 5b, bold numbers (e.g., $\langle$**0000**$\rangle$) indicate the synthesized result. The time line of the execution (the synthesized result) is explained as follows:

- Process $\mathcal{A}$ reads the input, sends `MsgSend`(m), and waits.
- Process $\mathcal{B}$ first waits until it is allowed to execute (`RecvMsg`(m)). Then it performs a conditional send `MsgSend`(req) and waits.

[7] Here we list case 2 only; for case 1 similar analysis can be applied.

[8] The complete of the case study (including the implementability analysis via constraint solving and the screenshots) can be found in the extended version.

[9] In our implementation, the PC starts from 0 rather than 1; which is different from the formulation in IM and PISEM.

Algorithm 3. LocalTimingModification

```
Data: Original PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$, synthesized IM $S = (A, N)$
Result: For each $\sigma_{\frac{a}{b}}$ and $\sigma_{\lfloor\frac{a}{b}\rfloor}$, their execution interval $[\alpha_{\frac{a}{b}}, \beta_{\frac{a}{b}})$, $[\alpha_{\lfloor\frac{a}{b}\rfloor}, \beta_{\lfloor\frac{a}{b}\rfloor})$
For convenience, use $(X\ in\ \mathcal{S})$ to represent the retrieved value $X$ from PISEM $\mathcal{S}$.
begin
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\ldots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        let $\alpha_{\frac{a}{b}}, \beta_{\frac{a}{b}}, \alpha_{\lfloor\frac{a}{b}\rfloor}, \beta_{\lfloor\frac{a}{b}\rfloor}$ // Create a new variable for the constraint system
        /* Type A constraint: causalities within the process */
1       $constraints$.add($\alpha_{\frac{a}{b}} = \beta_{\lfloor\frac{a}{b}\rfloor}$)
2       $constraints$.add($\alpha_{\lfloor\frac{a}{b}\rfloor} = (\alpha_{\lfloor\frac{a}{b}\rfloor} in\ \mathcal{S})$)
3       $constraints$.add($\beta_{\frac{a}{b}} = (\beta_{\lfloor\frac{a}{b}\rfloor} in\ \mathcal{S})$)
4       $constraints$.add($\beta_{\frac{a}{b}} - \alpha_{\frac{a}{b}} > WCET(\sigma_{\frac{a}{b}})$)
5       $constraints$.add($\beta_{\lfloor\frac{a}{b}\rfloor} - \alpha_{\lfloor\frac{a}{b}\rfloor} > WCET(\sigma_{\lfloor\frac{a}{b}\rfloor})$)
    /* Type B constraint: causalities crossing different processes */
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\ldots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        for $\sigma_d[\wedge_{m=1\ldots n_A}[pc_{d,m_{low}}, pc_{d,m_{up}})]$ in $\overline{\sigma_j}$ of $A_j$ do
            if $d \in \mathbb{N}$ and not exists $\sigma_{\frac{x}{y}} \in \overline{\sigma_j}$ where $\lfloor\frac{x}{y}\rfloor = d$ then
6               if $pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}}$ then $constraints$.add($(\beta_d\ in\ \mathcal{S}) < \alpha_{\frac{a}{b}}$)
7               if $pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then $constraints$.add($(\alpha_d\ in\ \mathcal{S}) > \beta_{\frac{a}{b}}$)
                if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then
8                   $constraints$.add($(\alpha_d\ in\ \mathcal{S}) > \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
            else
9               if $pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}}$ then $constraints$.add($\beta_d < \alpha_{\frac{a}{b}}$)
10              if $pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then $constraints$.add($\alpha_d > \beta_{\frac{a}{b}}$)
                if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then
11                  $constraints$.add($\alpha_d > \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
    /* Type C constraint: conservative data dependency constraints */
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\ldots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        for $\sigma_d[\wedge_{m=1\ldots n_A}(pc_{d,m_{low}}, pc_{d,m_{up}})]$ in $\overline{\sigma_j}$ of $A_j$ do
            if $\sigma_d := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}} \wedge \sigma_{\frac{a}{b}}$ reads variable $c$ then
12              $constraints$.add($(\beta_d\ in\ \mathcal{S}) + WCMTT(n, ind) < \alpha_{\frac{a}{b}}$)
    solve $constraints$ using (linear) constraint solvers.
end
```

- Process $\mathcal{A}$ performs $\mathtt{RecvMsg}(req)$, following a conditional send $\mathtt{MsgSend}(rsp)$.
- Process $\mathcal{B}$ performs conditional assignment, which assigns the value of rsp to m, if m_v is empty.

Concerning the running time of the above example, the engine (based on forward searching + BDD for intermediate image storing) is able to report the result in 3 seconds, while constraint solving is also relatively fast (within 1 second). Our engine offers a translation scheme to dump the BDD to mechanisms in textual form; this process occupies most of the execution time. Note that the NP-completeness result does not bring

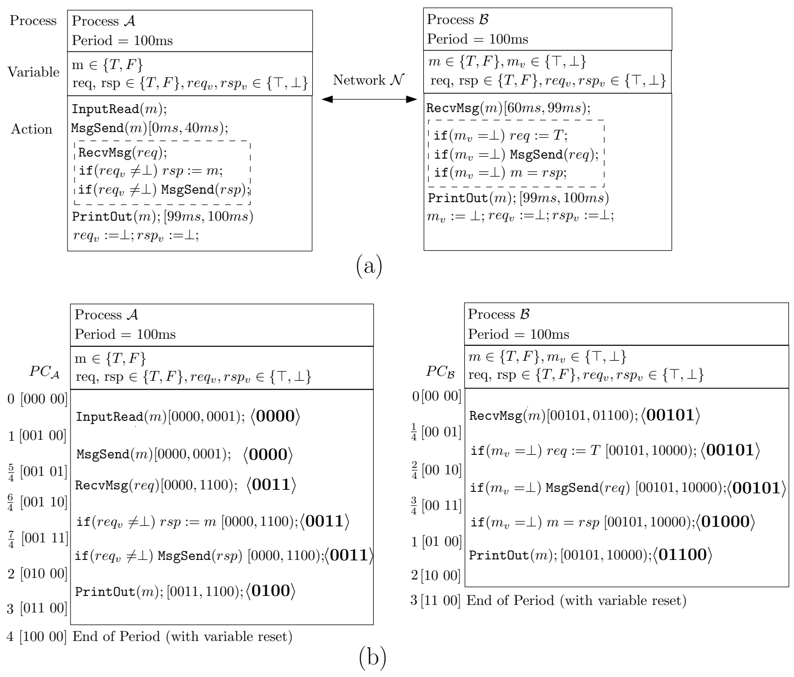
Process
Variable
Action
Process $\mathcal{A}$
Period = 100ms
m ∈ $\{T, F\}$
req, rsp ∈ $\{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
InputRead(m);
MsgSend(m)[$0ms, 40ms$);
RecvMsg(req);
if($req_v \neq \bot$) $rsp := m$;
if($req_v \neq \bot$) MsgSend(rsp);
PrintOut(m); [$99ms, 100ms$)
$req_v := \bot; rsp_v := \bot$;
Network $\mathcal{N}$
Process $\mathcal{B}$
Period = 100ms
$m \in \{T, F\}, m_v \in \{\top, \bot\}$
req, rsp ∈ $\{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
RecvMsg(m)[$60ms, 99ms$);
if($m_v = \bot$) $req := T$;
if($m_v = \bot$) MsgSend(req);
if($m_v = \bot$) $m = rsp$;
PrintOut(m); [$99ms, 100ms$)
$m_v := \bot; req_v := \bot; rsp_v := \bot$;
(a)
$PC_{\mathcal{A}}$
0 [000 00]
1 [001 00]
$\frac{5}{4}$ [001 01]
$\frac{6}{4}$ [001 10]
$\frac{7}{4}$ [001 11]
2 [010 00]
3 [011 00]
4 [100 00] End of Period (with variable reset)
InputRead(m)[0000, 0001); ⟨0000⟩
MsgSend(m)[0000, 0001); ⟨0000⟩
RecvMsg(req)[0000, 1100); ⟨0011⟩
if($req_v \neq \bot$) $rsp := m$ [0000, 1100);⟨0011⟩
if($req_v \neq \bot$) MsgSend(rsp) [0000, 1100);⟨0011⟩
PrintOut(m); [0011, 1100);⟨0100⟩
$PC_{\mathcal{B}}$
0[00 00]
$\frac{1}{4}$ [00 01]
$\frac{2}{4}$ [00 10]
$\frac{3}{4}$ [00 11]
1 [01 00]
2[10 00]
3 [11 00] End of Period (with variable reset)
RecvMsg(m)[00101, 01100);⟨00101⟩
if($m_v = \bot$) $req := T$ [00101, 10000);⟨00101⟩
if($m_v = \bot$) MsgSend(req) [00101, 10000);⟨00101⟩
if($m_v = \bot$) $m = rsp$ [00101, 10000);⟨01000⟩
PrintOut(m); [00101, 10000);⟨01100⟩
(b)

Fig. 5. An example where FT primitives are introduced for synthesis (a), and concept illustration for the control choices in the generated game (b)

huge benefits, as another exponential blow-up caused by the translation from variables to states is also unavoidable.

8 Related Work

Recently there has been an increasing interest in applying games for synthesis, including work by Bloem and Jobstmann et al. (the program repair framework [7]), Henzinger and Chatterjee et al. (Alpaga and the interface synthesis [3,5]), and David and Larson et al. (Uppaal TIGA [2]); these approaches are usually based on non-distributed settings. Our starting model of distributed and timed straight-line programs is based on commonly used paradigms from the real-time community [8].

The work that is closest to ours is by Kulkarni et.al. [9] and Girault et al. [6]. Their work is on protocol synthesis based on synthesizing finite state machines (FSMs) from non-fault tolerant FSMs and pre-specified fault models. In contrast to the work mentioned above we do not address the complete synthesis of FT mechanisms such as voting; instead we are synthesizing the FT mechanisms based on the application of pre-defined FT patterns, thereby considerably restricting the space of possible solutions. Our approach is based on standard notions of distributed games and game-theoretic algorithms for solving these games. As we are mainly interested in synthesizing distributed embedded programs, we naturally restrict ourselves to finding positional strategies; this problem turns out to be NP-complete. The synthesis algorithm we describe in this paper works in several steps from distributed embedded timed programs to interleaving semantics and the formulation of distributed games. Solutions for these game-theoretic problems are obtained by translation to a corresponding Boolean SAT problem (although our implementation, Gecko, is currently still based on specialized search strategies). This formulation of solving distributed games based on SAT closely follows the main steps as provided by [1].

9 Concluding Remarks

This paper presents a comprehensive approach (see Figure 6 for concept illustration) for augmenting fault-tolerance in real-time distributed systems under a game-theoretic framework. These mechanisms may have interesting applications in distributed process control and robotics. We plan to further validate our approach using our prototype implementation Gecko. To handle the complexity in practice it will also be necessary to partition systems into subsystems in a compositional way.

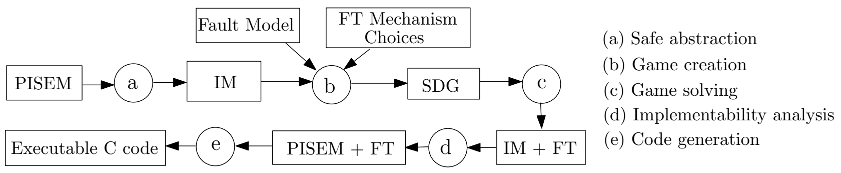

Fig. 6. Concept illustration of the overall approach for fault-tolerant synthesis; IM+FT means that an IM model is equipped with FT mechanisms

References

1. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. International Journal on Software Tools for Technology Transfer (STTT) 7(2), 118–128 (2005)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
3. Berwanger, D., Chatterjee, K., De Wulf, M., Doyen, L., Henzinger, T.: Alpaga: A tool for solving parity games with imperfect information. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 58–61. Springer, Heidelberg (2010)
4. Cheng, C.-H., Ruess, H., Knoll, A., Buckl, C.: A game-theoretic approach for synthesizing fault-tolerant embedded systems (extended version). In: arXiv:1011.0268 [cs.GT] (2010)
5. Doyen, L., Henzinger, T., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT 2008, pp. 79–88. ACM, New York (2008)
6. Girault, A., Rutten, É.: Automating the addition of fault folerance with discrete controller synthesis. Formal Methods in System Design 35(2), 190–225 (2009)
7. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)

8. Kshemkalyani, A., Singhal, M.: Dirstributed computing: principles, algorithms, and systems. Cambridge University Press, Cambridge (2008)
9. Kulkarni, S., Arora, A.: Automating the addition of fault-tolerance. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 82–359. Springer, Heidelberg (2000)
10. Mohalik, S., Walukiewicz, I.: Distributed games. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 338–351. Springer, Heidelberg (2003)

Appendix: Sketch of the SAT Witness Algorithm for Distributed Games

Madhusudan, Nam, and Alur [1] designed the *bounded witness algorithm* for solving reachability games using SAT. Although based on their experiments, the witness algorithm is not as efficient as the BDD-based approach in centralized games, with some modifications, we find it potentially useful for distributed games.

We first paraphrase the concept of *witness* defined in [1], a set of states which witnesses the fact that player 0 wins. In [1], consider the generated SAT problem from a local game $G = (V_0 \uplus V_1, E)$ trying to reach from V_{init} to V_{goal}: for $i = 1, \dots, d$ and vertex $v \in V_0 \uplus V_1$, variable $\langle v \rangle_i = \texttt{true}$ when one of the following holds:

- $v \in V_{init}$ and $i = 1$ (if $v \notin V_{init} \wedge i = 1$ then $\langle v \rangle_i = \texttt{false}$).
- $v \in V_{goal}$ (if $v \notin V_{goal} \wedge i = d$ then $\langle v \rangle_i = \texttt{false}$).
- $v \in V_0 \setminus V_{goal}$ and $\exists v' \in V_0 \uplus V_1.\ \exists e \in E.\ \exists j > i.\ (e = (v, v') \wedge \langle v' \rangle_j = \texttt{true})$
- $v \in V_1 \setminus V_{goal}$ and $\forall e = (v, v') \in E.\ \exists j > i.\ \langle v' \rangle_j = \texttt{true}$

This recursive definition implies that if $v \in V_0$ (resp. $v \in V_1$) is not the goal but in the witness set, then its successor (resp. all of its successors) v' should either be (i) in a goal state or (ii) also in the witness: note that for (ii), the number of allowable steps to reach the goal is decreased by one.

In general, our algorithm creates SAT problems based on the above concept but contains modifications to ensure (1) the unique selection of local edges and (2) the progress of a global move is a combination of local moves (see Algorithm 4 for fragments)[10].

Algorithm 4. PositionalDistributedStrategy_BoundedSAT_0 (fragment only)

```
...
1 for local control transition $e = (x_i, x'_i) \in E_i, x_i \in V_{0_i}$ do
      for local transition $e_1 = (x_i, x'_{i_1}), \dots, e_k = (x_i, x'_{i_k}) \in E_i, e_1 \dots e_k \neq e$ do
          clauses.add([$\langle e \rangle \Rightarrow (\neg\langle e_1 \rangle \wedge \dots \wedge \neg\langle e_k \rangle)$])
2 for $v = (v_1, \dots, v_m) \in \mathcal{V}_0$ do
      for $(e_1, \dots, e_m)$: $e_i = (v_i, v'_i) \in E_i$ if $v_i \in V_{0_i}$ or $e_i = (v_i, v_i)$ if $x_i \in V_{1_i}$ do
          // $e_i = (v_i, v_i)$ when $x_i \in V_{1_i}$ are dummy edges for ease of formulation
          for $j = 1, \dots, d-1$ do
              clauses.add([$\langle v_1, \dots, v_m \rangle_j \Rightarrow ((\bigwedge_{\{i | v_i \in V_{0_i}\}} \langle e_i \rangle) \Rightarrow (\langle v'_1, \dots, v'_m \rangle_{j+1}]$)
```

[10] The complete algorithm can be found in the full version [4].