A Game-theoretic Approach for Synthesizing Fault-Tolerant Embedded Systems

Chih-Hong Cheng*, Harald Ruess†, Alois Knoll*, Christian Buckl†

* Department of Informatics, Technische Universität München, Germany
† fortiss GmbH, Germany
Email: {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.

I. INTRODUCTION

In this paper, we investigate methods to perform automatic fault-tolerant (FT for short) synthesis under the context of embedded systems, where our goal is to generate executable code which can be deployed on dedicated hardware platforms.

Creating such a tool supporting the fully-automated process is very challenging as the inherent complexity is high: bringing FT synthesis from theory to practice means solving a problem consisting of (a) interleaving semantics, (b) timing, (c) fault-tolerance, (d) dedicated features of concrete hardware, and optionally, (e) the code generation framework. To generate tamable results, we first constrain our problem space to some simple yet reasonable scenarios (sec. II). Based on these scenarios we can start system modeling (sec. III) taking into account all above mentioned aspects.

To proceed further, we find it important to observe the approach nowadays to understand the need: for engineers working on ensuring fault-tolerance of a system, once the corresponding fault model is decided, a common approach is to select some fault-tolerant patterns [14] (e.g., fragments of executable code) from a pattern pool. Then engineers must fine-tune these mechanisms, or fill in unspecified information in the patterns to make them work as expected. With the above scenario in mind, apart from generating complete FT mechanisms from specification, our synthesis technique emphasizes automatic selection of predefined FT patterns and automatic tuning such that details (e.g., timing) can be filled without human intervention. This also reduces a potential problem where unwanted FT mechanisms are synthesized due to under-specification. Following the statement, we translate the system model, the fault hypothesis, and the set of available FT patterns into a distributed game [18] (sec. V), and a strategy generated by the game solver can be interpreted as a selection of FT patterns together with guidelines of tuning.

For games, it is known that solving distributed games is, in most cases, undecidable [18]. To cope with undecidability, we restrict ourselves to the effort of finding positional strategies (mainly for reachability games). We argue that finding positional strategies is still practical, as the selected FT patterns may introduce additional memory during game creation. Hence, a positional strategy (by pattern selection) combined with selected FT patterns generates mechanisms using memory. By posing this restriction, the problem of finding a strategy of the game (for control) is NP-Complete (sec. VI), and searching techniques (e.g., SAT translation or combining forward search with BDD) are thus applied to assist the finding of solutions.

The final step of the automated process is to translate the result of synthesis back to concrete implementation: the main focus is to ensure that the newly synthesized mechanisms do not change the implementability of the original system (i.e., the new system is schedulable). Based on our modeling framework, this problem can be translated to a linear constraint system, which can be solved efficiently by existing tools.

To evaluate our methods, we have created our prototype software, which utilizes the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. We demonstrate two small yet representative examples with our tool for a proof-of-concept (sec. VIII); these examples indicate the applicability of the approach. Lastly, we conclude this paper with an overview of related work (sec. IX) and a brief summary including the flow of our approach (sec. X).

II. MOTIVATING SCENARIO

A. 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 \( A, B \) and one bidirectional network \( N \). Processes \( A \) and \( B \) start executing sequential actions together with a looping period of 100ms. In each period, \( A \) first reads an input using a sensor to
variable $m$, followed by sending the result to the network $\mathcal{N}$ using the action $\text{MsgSend}(m)$, and outputing the value (e.g., to a log).

In process $A$, for the action $\text{MsgSend}(m)$, 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 $B$ as $\top$ (indicating that the content is valid). However, $A$ 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 $B$, its first action $\text{RecvMsg}(m)$ 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, $B$ 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 $\text{MsgSend}(m)$ and (b) release time of $\text{RecvMsg}(m)$ overestimates the worst case transmission time for a message to travel from $A$ to $B$. After $\text{RecvMsg}(m)$, 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 $A$ and $B$, such that two output values from $A$ and $B$ are the same at the end of the period, regardless of the disturbance from the network.

B. 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.

III. SYSTEM MODELING

A. 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.
- $\mathcal{A} = \bigcup_{i=1 \ldots n_{\mathcal{A}}} \mathcal{A}_i$ is the set of processes, where in $\mathcal{A}_i = (V_i \cup V_{env_i}, \sigma_i)$,
  - $V_i$ 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.
  - $\sigma_i := \sigma_1(\alpha_1, \beta_1); \ldots; \sigma_j(\alpha_j, \beta_j); \ldots; \sigma_k(\alpha_k, \beta_k)$ is a sequence of actions.
  - $\sigma := \text{send}(pre, index, n, s, d, v, c)$ is an atomic action (action pattern), where
    - $a, c \in V_i$,
    - $e$ is a function from $V_{env_i} \cup V_i$ to $V_i$ (this includes null-op),
    - $pre$ is a conjunction of over equalities/inequalities of variables,
    - $s, d \in \{1, \ldots, n_{\mathcal{A}}\}$ represents the source and destination,
    - $v \in V_d$ is the variable which is expected to be updated in process $d$,
    - $n \in \{1, \ldots, n_{\mathcal{N}}\}$ is the network used for sending, and
    - $index \in \{1, \ldots, size_e\}$ is the index of the message used in the network.
  - $[\alpha_j, \beta_j]$ is the execution interval, where $\alpha_j \in Q$ is the release time and $\beta_j \in Q$ is the deadline.
- $\mathcal{N} = \bigcup_{i=1 \ldots n_{\mathcal{N}}} \mathcal{N}_i$, $\mathcal{N}_i = (T_i, size_i)$ is the set of network.
Based on the above definitions, the system under execution in section II-A can be easily modeled by PISEM:

\[ \text{Example} \]

A

where

B. Interleaving Model (IM)

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

Definition 2: The configuration of \(S\) is \((\prod_{i=1 \ldots n_A}(v_i, v_{env}, \Delta_{next}), \prod_{j=1 \ldots n}(occu_j, s_j, d_j, var_j, c_j, t_j, ind_j), t)\), where

- \(v_i\) is the set of the current values for the variable set \(V_i\),
- \(v_{env}\) is the set of the current values for the variable set \(V_{env}\),
- \(\Delta_{next}, e \in [1,|\sigma|+1]\) is the next atomic action index in \(\sigma\),
- \(occu_j \in \{false, true\}\) is for indicating whether the network is busy,
- \(s_j, d_j \in \{1, \ldots, n_A\}\),
- \(var_j \in \bigcup_{i=1 \ldots n_A}(V_i \cup V_{env})\),
- \(c_j \in \mathbb{Z}\) 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 \(\Delta_{next}\), and \(v_i, v_{env}\) are current values of \(V_i\) and \(V_{env}\). If \(j = |\sigma|+1\) then do nothing (all actions in \(\sigma\) have been executed in this cycle); else the action \(\sigma_j := var \leftarrow e(\alpha_j, \beta_j)\) updates \(var\) from \(s\) to \(e(v_i, v_{env})\), and changes \(\Delta_{next}\) to \(\min\{x | x \in iSet(\sigma), x > j\}\). 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 \(\Delta_{next}\). If \(j = |\sigma|+1\) then do nothing; else the action \(\sigma_j := \text{send}(\text{pre}, \text{index}, n, s, d, v, c)|\alpha_j, \beta_j\) should be processed between the time interval \(t \in [\alpha_j, \beta_j]\), and changes \(\Delta_{next}\) to \(\min\{x | x \in iSet(\sigma), x > j\}\).

- When \(\text{pre}\) is evaluated to true (it can be viewed as an \(if\) statement), it then checks the condition \(\text{occu}_n = false\):
  - if the condition holds, it updates network \(n\) with value \((\text{occu}_n, s_n, d_n, var_n, c_n, t_n, ind_n) := (true, i, d, v, c, 0, \text{index})\).
  - Otherwise it blocks until the condition holds.
  - When \(\text{pre}\) is evaluated to false, it skips the sending.

3) (Process message) For network \(j\), for configuration \((occu_j, s_j, d_j, var_j, c_j, t_j, ind_j)\) if \(occu_j = true\), then during \(t_j < T_j(ind_j)\), a transmission occurs, which updates \(occu_j\) to \(false\), \(A_d\) to \(var\), and \(A_d, var\) to \(true\).

4) (Receive) For machine \(i\), let \(s\) and \(j\) be the current configuration for \(c\) and \(\Delta_{next}\). If \(j = |\sigma|+1\) then do nothing; else for \(\text{receive}(\text{pre}, c)|\alpha_j, \beta_j\) in machine \(i\), it is processed between the time interval \(t \in [\alpha_j, \beta_j]\) and changes \(\Delta_{next}\) to \(\min\{x | x \in iSet(\sigma), x > j\}\).

5) (Repeat Cycle) When \(t = T\), \(t\) is reset to 0, and for all \(x \in \{1, \ldots, n_A\}\), \(\Delta_{next}\) are reset to 1.

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

- All processes and networks in \(S\) share a globally synchronized clock. Note that this assumption can be fulfilled in many hardware platforms, e.g., components implementing the IEEE 1588 protocol.
- For all actions \(\sigma, \sigma.\text{deadline} < T\); for all send actions \(\sigma := \text{send}(\text{pre}, \text{index}, n, s, d, v, c), \sigma.\text{deadline} + T_n(\text{index}) < T\), i.e., all processes and networks should finish its work within one complete cycle.

B. 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 \(S_{IM} = (A, N)\).

- \(A = \bigcup_{i=1 \ldots n_A} A_i\) is the set of processes, where \(A_i = (V_i \cup V_{env}, \sigma_i)\),
- \(V_i\) is the set of variables, and \(V_{env}\) is the set of environment variables.

\(^1\)Here an interval \([1,|\sigma|+1]\) is used for the introduction of FT mechanisms described later.

\(^2\)In our formulation, the \(\text{receive}(\text{pre}, c)\) action can be viewed as a syntactic sugar of null-op; its purpose is to facilitate the matching of send-receive pair with variable \(c\).
• $\sigma_i := \sigma_1[\Lambda_{m=1\ldots n_A}[pc_{1,m_{low}}, pc_{1,m_{up}}]]; \ldots; \sigma_j[\Lambda_{m=1\ldots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}}]]; \ldots; \sigma_k[\Lambda_{m=1\ldots n_A}[pc_{k,m_{low}}, pc_{k,m_{up}}]]$
  is a fixed sequence of actions.

- $\sigma_j := \text{send}(\text{pre}, \text{index}, n, s, d, v, c) \mid \text{receive}(\text{pre}, c) \mid a \leftarrow e$ is an atomic action, where $a, c, e, \text{pre}, v, n, s, d$ are defined similarly as in PISEM.

- For $\sigma_j, \forall m \in \{1, \ldots, n_A\}, pc_{j,m_{low}}, pc_{j,m_{up}} \in \{1, \ldots, |\sigma_m| + 2\}$ 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 $m = i$.

- $N = \bigcup_{i=1\ldots n_N} N_i$, $N_i = (T_i, \text{size}_i)$ is the set of network.

- $T_i : N \rightarrow \Lambda_{m=1\ldots n_A}(1, \ldots, |\sigma_m| + 2) \{1, \ldots, |\sigma_m| + 2\}$ is a function which maps the index (or priority) of a message to the PC-precondition interval of other processes.

- $\text{size}_i$ is the number of messages used in $N_i$.

**Definition 4:** The configuration of $S_{IM}$ is $(\bigwedge_i(v_i, v_{envi}, \Delta_{next_i}), \bigwedge_i(occu_j, s_j, d_j, c_j))$, where $v_i, v_{envi}, \Delta_{next_i}, occu_j, s_j, d_j, c_j$ 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:

1) For an action $\sigma_j$ having the precondition $\Lambda_{m=1\ldots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}}]$, it should be executed between $pc_{j,m_{low}} \leq \Delta_{next_i} < pc_{j,m_{up}}$, for all $m$.

2) 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.

3) The system repeats the cycle when $\forall x \in \{1, \ldots, n_A\}, \Delta_{next_x} = |\sigma_x| + 1$ and $\forall x \in \{1, \ldots, n_N\}, \text{occu}_x = \text{false}$.

**IV. GAMES**

For the proof of complexity results, we use similar notations in [18] to define a distributed game. Intuitively, distributed games are games formulating multiple processes with no interactions among themselves but only with the environment.

**(Local) Games**

A game graph or arena is a directed graph $G = (V_0 \uplus V_1, E)$ whose nodes are partitioned into two classes $V_0$ and $V_1$. We only consider the case of two players in the following and call them player 0 and player 1 for simplicity. A play starting from node $v_0$ is simply a maximal path $\pi = v_0v_1 \ldots$ in $G$ where we assume that player $i$ determines the move $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$ ($i \in \{0, 1\}$). With $\text{Occ}(\pi)$ we denote the set of nodes visited by a play $\pi$. A winning condition defines when a given play $\pi$ is won by player $0$; if $\pi$ is not won by player $0$, it is won by player $1$. A node $v$ is won by player $i$ if player $i$ can always choose his moves in such a way that he wins any resulting play starting from $v$.

**Distributed Games**

We use notations by Mohalik and Walukiewicz [18] to define a distributed game. From now on we call the a game graph defined in sec. IV a local game graph.

**Definition 5:** For all $i \in \{1, \ldots, n\}$, let $G_i = (V_{0,i} \uplus V_{1,i}, E_i)$ be a local game graph with the restriction that it is bipartite. Define a distributed game to be $G = (V_0 \uplus V_1, E, \text{Acc} \subseteq (V_0 \uplus V_1)\omega)$:

- $V_1 = V_{1,1} \times \ldots \times V_{1, n}$ is the set of player 1 (environment) vertices.

- $V_0 = (V_{0,1} \uplus V_{1,1}) \times \ldots \times (V_{0,n} \uplus V_{1,n}) \setminus V_1$ is the set of player 0 (control) vertices.

- For a vertex $x = (x_1, \ldots, x_n)$, we use the function $\text{proj}(x, i)$ to retrieve the $i$-th component $x_i$, and use $\text{proj}(X, i)$ to retrieve the $i$-th component for a set of vertices $X$.

- Let $(x_1, \ldots, x_n), (x'_1, \ldots, x'_n) \in V_0 \uplus V_1$, then define $E$ as follows:

  - If $(x_1, \ldots, x_n) \in V_0, ((x_{i,1}, \ldots, x_{i,n}), (x'_{i,1}, \ldots, x'_{i,n})) \in E$ if and only if $\forall i, (x_i \in V_0 \Rightarrow (x_i, x'_i) \in E_i) \land \forall j, (x_j \in V_1 \Rightarrow x_j = x'_j).

  - For $(x_1, \ldots, x_n) \in V_1, ((x_{i,1}, \ldots, x_{i,n}), (x'_{i,1}, \ldots, x'_{i,n})) \in E$, then for every $x_i$, either $x_i = x'_i$ or $x'_i \in V_0$, and moreover $(x_1, \ldots, x_n) \neq (x'_1, \ldots, x'_n)$.

- $\text{Acc}$ is the acceptance condition.

In a distributed game $G = (V_0 \uplus V_1, E, \text{Acc})$, a play is defined analogously as defined in local games: a play starting from node $v_0$ is a maximal path $\pi = v_0v_1 \ldots$ in $G$ where player $i$ determines the move $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$ ($i \in \{0, 1\}$).

A distributed strategy of a distributed game for player 0 is a tuple of functions $\xi = \langle f_1, \ldots, f_n \rangle$, where each function $f_i : (V_{0,i} \uplus V_{1,i})^* \rightarrow V_{0,i} \uplus V_{1,i}$ is a local strategy which decides the updated location of the local game $i$ based on (a) its observable history of local game $i$ and (b) current position of local game $i$. Lastly, we call a distributed strategy
Algorithm 1: GeneratePreconditionPC

Data: PISEM model \( S = (A, N, T) \)

Result: Two maps \( mapLB, mapUB \) which map from an action \( \sigma \) (or a msg processing by network) to two integer arrays \( \text{lower}[1 \ldots n_A], \text{upper}[1 \ldots n_A] \)

begin

/* Initial the map for recording the lower and upper bound for action */
for action \( \sigma_k \) in \( A_i \) of \( A \) do
  \[ \text{mapLB.put}(\sigma_k, \text{new int}[1 \ldots n_A]) \] /* Initialize to 1 */
  \[ \text{mapUB.put}(\sigma_k, \text{new int}[1 \ldots n_A]) \]
for \( A_j \in A \) do
  \[ \text{mapUB.get(\sigma_k)[j]} := |\sigma_j| + 2 \] /* Initialize to upperbound */
for action \( \sigma_k \) in \( A_i \) of \( A \) do
  \[ \text{mapLB.get(\sigma_k)[i]} = k; \text{mapUB.get(\sigma)[i]} = k+1; \] /* self PC */

/* Initial the map for recording the lower and upper bound for msg transmission */
for action \( \sigma_k = \text{send}(\text{pre, ind, n, s, d, v, c}) \) in \( A_i \) of \( A \) do
  \[ \text{mapLB.put}(\sigma_k, \text{new int}[1 \ldots n_A]) \] /* Initialize to 1 */
  \[ \text{mapUB.put}(\sigma_k, \text{new int}[1 \ldots n_A]) \]
for \( A_j \in A \) do
  \[ \text{mapUB.get(\sigma_k)[j]} := |\sigma_j| + 2 \] /* Initialize to upperbound */

/* for action \( \sigma_k = \text{send}(\text{pre, ind, n, s, d, v, c}) \) in \( A_i \) of \( A \) do */
for action \( \sigma_m \) in \( A_j \) of \( A \), \( n = 1 \ldots |\sigma_j| \), \( j \neq i \) do
  for action \( \sigma_n \) in \( A_j \) of \( A \), \( n = 1 \ldots |\sigma_j| \) do
    if \( \sigma_n.\text{releaseTime} > \sigma_n.\text{deadline} \) then
      \[ \text{mapLB.get(\sigma_n)[j]} := \max\{\text{mapLB.get(\sigma)(j)[], n + 1}\} \]
    if \( \sigma_n.\text{releaseTime} \leq \sigma_n.\text{deadline} \) then
      \[ \text{mapUB.get(\sigma_n)[j]} := \min\{\text{mapUB.get(\sigma)(j)[], n + 1}\}; \]

positionally, if \( f_i \) is a function mapping from \( V_{0_i} \) to \( V_{0_i} \cup V_{1_i} \), i.e., the update of location depends only on the current position of local game.

Definition 6: A distributed game \( \mathcal{G} = (V_0 \cup V_1, \mathcal{E}, \mathcal{A}) \) is reachability-winning by a distributed strategy \( \xi = (f_1, \ldots, f_n) \) over initial states \( V_{ini} \in V_0 \cup V_1 \) and target states \( V_{goal} \in V_0 \cup V_1 \), when the following conditions hold:

- \( \mathcal{A} = \{v_0v_1 \ldots \in (V_0 \cup V_1)^{\omega} | \text{Occ}(v_0v_1 \ldots) \cap V_{goal} \neq \emptyset \}. \)
- For every play \( \pi = v_0v_1v_2 \ldots \) where \( v_0 \in V_{ini}, \) player 0 wins \( \pi \) when the following constraints hold:
  - \( \pi \in \mathcal{A}. \)
  - \( \forall i \in N_0. (v_i \in V_0 \Rightarrow \forall j \in \{1, \ldots, n\}. (\text{proj}(v_i, j) \in V_{0j} \Rightarrow \text{proj}(v_{i+1}, j) = \text{proj}(f_j(v_i), j))). \)

V. Step A: Front-End Translation from Models to Games

A. 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 by generating PC-intervals in all machines as:

- temporal preconditions for an action to execute, or
- temporal preconditions for a network to finish its message processing, i.e., to update a variable in the destination process with the value in the message.

Here we assume that in each period, for all \( N_j \), 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.
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 \(\sigma_m\) is executed, \(\sigma_n\) should have been executed.
- In (2), if \(\sigma_m\.deadline < \sigma_n\.releaseTime\), then \(\sigma_n\) should not be executed before executing \(\sigma_m\).
- Similar analysis is done with (3) and (4). However, we need to consider the combined effect together with the network transmission time: we use \(m\in\{1,4\}\) for the worst case, and \(T_n(ind)\) for the best case.

**Example** For the example in sec. II, consider the action \(\sigma_1 := m \leftarrow \text{InputRead}(v)[0, 40)\) in \(A_1\) of a PISEM. Algorithm [1] returns \(\text{mapLB}(\sigma)\) and \(\text{mapUB}(\sigma)\) with two arrays \([1, 1]\) and \([2, 2]\), indicated in Figure 2a. Based on the definition of IM, \(\sigma_1\) should be executed with the temporal precondition that no action in \(A_2\) 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.

**B. 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:

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

2) **Step A.2.2: Introducing fault-tolerant choices as \(\sigma_2\)**: In our framework, fault-tolerant mechanisms are similar to actions, which consist of two parts: action pattern \(\sigma\) and timing precondition \([\Lambda_m=1...n_A | pc_{m_{low}}, pc_{m_{up}}]\). 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.

We use the notation \(\sigma_2\), where \(\frac{\sigma}{a} \in \{Q \setminus N\}\), to represent an inserted action pattern between \(\sigma[\frac{\sigma}{a}]\) and \(\sigma[\frac{\sigma}{a}]\). With this formulation, multiple FT mechanisms can be inserted within two consecutive actions \(\sigma_i\). In the system, and the execution semantic follows what has been defined previously: as executing an action updates \(\Delta_{nexti}\), to \(\min\{x| x \in iSet(\sigma_i), x > j\}\), updating to a rational value is possible. Note that as \(\sigma_2\) 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.

We conclude this step with two remarks:
- For all existing actions, the non-deterministic choice generation in step A.2.1 must be modified to contain these rational points introduced by FT mechanisms.
- A problem induced by FT synthesis is whether the system behavior changes due to the introduction of FT mechanisms. We answer the problem by splitting into two subproblems:
### Algorithm 2: DecideInsertedFTTemplateTiming

**Data:** $\sigma_{[m=1...n_A; \{pc_e, m_{iow}, \{pc_c, m_{iow}, \{pc_d, m_{ap}\}\}]}, \sigma_d[\{m=1...n_A; \{pc_d, m_{iow}, \{pc_d, m_{ap}\}\}]]$, which are consecutive actions in $\sigma$ of $A_I$ of $S_{TM} = (A, N)$, and one newly added action pattern $\sigma_{\Phi}$ to be inserted between

**Result:** Temporal preconditions for action pattern $\sigma_{\Phi}$: $[\{m=1...n_A; \{pc_e, m_{iow}, \{pc_d, m_{ap}\}\}]$

begin

for $m = 1, \ldots, n_A$
do

if $m \neq i$ then

$pc_{e, m_{iow}} := pc_{c, m_{iow}}$ /* Use the lower bound of $c$ for its lower bound */

$pc_{d, m_{ap}} := pc_{d, m_{ap}}$ /* Use the upper bound of $d$ for its upper bound */

else

$pc_{e, m_{iow}} := a, pc_{d, m_{ap}} := d$

end

end

---

### Comparison between DG and SDG

| State space | product of all vertices in local games |
|-------------|----------------------------------------|
| Vertex partition ($V_0$ and $V_1$) | explicit partition |
| Player-0 transitions | defined in local games |
| Player-1 transitions | explicitly specified in the global game |

---

- **[Problem 1]** Whether the system is still schedulable due to the introduction of FT actions, as these FT actions also consume time. This can only be answered when the result of synthesis is generated, and we leave this to section VII.

- **[Problem 2]** Whether the networking behavior remains the same. This problem must be handled before game creation, as an FT message may significantly influence the worst case message transmission time (WCMTT) of all existing messages, leading to a completely different networking behavior. The answer of this problem depends on many factors, including the hardware in use, the configuration setting, and the analysis technique used for the estimation of WCMTT. In Appendix A we give a simple analysis for ideal CAN buses [7], which are used most extensively in industrial and automotive embedded systems: in the analysis, we propose conditions where newly added messages do not change the existing networking behavior. Similar analysis can be done with other timing-predictable networks, e.g., FlexRay [20].

3) 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 our created variant called **symbolic distributed games (SDG):**

**Definition 7:** Define a symbolic distributed game $G_{ABS} = (V_f \cup V_{CTR} \cup V_{ENV}, A, N, \sigma_f, pred)$.

- $V_f$, $V_{CTR}$, $V_{ENV}$ are disjoint sets of (fault, control, environment) variables.

- $pred : V_f \times V_{CTR} \times V_{ENV} \rightarrow \{true, false\}$ is the partition condition.

- $A = \bigcup_{i=1...n_A} A_i$ is the set of **symbolic local games (processes)**, where in $A_i = (V_i \cup V_{env_i}, \sigma_i)$,
  - $V_i$ is the set of variables, and $V_{env_i} \subseteq V_{ENV}$.
  - $\sigma_i := \langle m_{iow} = 1, \ldots, n_A, pc_{i1}, \ldots, \sigma_k \langle m_{iow} = 1, \ldots, n_A, pc_{ik}\rangle \rangle$ is a sequence, where
    - $\forall j = 1, \ldots, k, \cup \sigma_k \langle m_{iow} = 1, \ldots, n_A, pc_{i}\rangle$ is a set of choice actions for player-0 in $A_i$.
  - $\sigma_i$ is defined similarly as in IM.
  - $\forall m = 1, \ldots, n_A$, $pc_{im} \in \{pc_{i1, m_{iow}} pc_{i2}, m_{iow}, pc_{i1, m}, m_{ap}\}$

- $V_{CTR} = \bigcup_{i=1...n_A} V_i$.

- $N = \bigcup_{i=1...n_A} N_i$, $N_i = (T_i, size_i, tran_i)$ is the set of network processes.

- $T_i$ and $size_i$ are defined similarly as in IM.

- $\forall tran_i : V_f \times \{(true, false) \times \{1, \ldots, n_A\} \times \bigcup_{i=1...n_A} (V_i \cup V_{env_i}) \times \{1, \ldots, size_i\}) \rightarrow V_f \times \{(true, false) \times \{1, \ldots, n_A\} \times \bigcup_{i=1...n_A} (V_i \cup V_{env_i}) \times \{1, \ldots, size_i\})$ is the network transition relation for processing messages (see sec III-A for meaning), but can be influenced by additional variables in $V_f$.

- $\sigma_f : V_f \times V_{CTR} \times V_{ENV} \times \bigcup_{i=1...n_A} iSet(\sigma_i) \rightarrow V_{ENV} \times V_f \times \bigcup_{i=1...n_A} iSet(\sigma_i)$ 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 $\pi = v_0v_1...$.
In $v_k$, player-1 determines the move $(v_k, v_{k+1}) \in E$ when $\text{pred}(v_k)$ is evaluated to $\text{true}$ ($\text{false}$ for player-0); the partition of vertices $V_0$ and $V_1$ in a SDG is implicitly defined based on this, rather than specified explicitly as in a distributed game.

A move $(v_k, v_{k+1})$ is a selection of executable transitions defined in $N, \sigma_f, \text{or } A$; in our formulation, transitions in $N$ and $\sigma_f$ are all environment moves\(^4\) while transitions in $A$ are control moves\(^4\).

Lastly, a distributed positional strategy for player-0 in a SDG can be defined analogously as to uniquely select an action from the set $\bigcup \sigma_{\alpha}(\langle m = 1, ..., n_A, pc_{\alpha_{jm}} \rangle)$, for all $A_i$ and for all program counter $j$ defined in $\sigma_T$. Each strategy should be insensitive of contents in other symbolic local games.

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 (as specified in the SDG) by

(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 tuple\(^6\):

1) The fault type (an unique identifier, e.g., \text{MsgLoss, SensorError}).
2) The maximum number of occurrences in each period.
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. II; similar approaches can be applied for input errors or message corruption; here the modeling of input (for $\text{InputRead}(m)$) is skipped.

- Create the predicate $\text{pred}$; $\text{pred}$ is evaluated to $\text{false}$ in all cases except (a) when the boolean variable $\text{occu}$ (representing the network occupancy) is evaluated to $\text{true}$ and (b) when for all $i \in \{1, ..., n_A\}$, $\Delta_{\text{next}} = |\sigma_T| + 1$ (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 $\bigcup \sigma_{\alpha}(\langle m = 1, ..., n_A, pc_{\alpha_{jm}} \rangle)$ are generated based on the approach described previously.
- Create variable $v_f \in V_f$, which is used to indicate whether the fault ($\text{MsgLoss}$) has been activated in this period.
- In this example, as the maximum number of fault occurrences in each period is 1, we do not need to create additional counters.
- For each message sending transition $t$ in the network, create two normal transitions $(v_f = \text{true} \land v'_f = \text{true}) \land t$ and $(v_f = \text{false} \land v'_f = \text{false}) \land t$ 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 $(v_f = \text{false} \land v'_f = \text{true}) \land t'$ in the game.
- Define $\sigma_f$ to control $v_f$: if all $i \in \{1, ..., n_A\}$, $\Delta_{\text{next}} = |\sigma_T| + 1$, then update $v_f$ to $\text{false}$ as $\Delta_{\text{next}}$ updates to 1 (reset the fault counter at the end of the period).

VI. STEP B: SOLVING DISTRIBUTED GAMES

We summarize the result from [18] as a general property of distributed 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.

\(^6\)For complete formulation of fault models, we refer readers to our earlier work [6].
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 [13].

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: Given a distributed game $G = (V_0 \cup V_1, E)$, 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: We first start by recalling the definition of attractor, a term which is commonly used in the game and later applied in the proof. Given a game graph $G = (V_0 \cup V_1, E)$, for $i \in \{0, 1\}$ and $X \subseteq V$, the map $\text{attr}_i(X)$ is defined by

$$\text{attr}_i(X) := X \cup \{v \in V_i \mid vE \cap X \neq \emptyset\} \cup \{v \in V_{i-1} \mid \emptyset \neq vE \subseteq X\},$$

i.e., $\text{attr}_i(X)$ extends $X$ by all those nodes from which either player $i$ can move to $X$ within one step or player $1-i$ cannot prevent to move within the next step. ($vE$ denotes the set of successors of $v$.) Then $\text{Attr}_i(X) := \bigcup_{k \in \mathbb{N}} \text{attr}_i^k(X)$ contains all nodes from which player $i$ can force any play to visit the set $X$.

We continue our argument as follows.

[NP] The reachability problem for a distributed game can be solved in NP: a solution instance $\xi = (f_1, \ldots, f_n)$ is a strategy which selects exactly one edge for every control vertex in the local game. As the distributed game graph is known, after the selection we calculate the reachability attractor $\text{Attr}_0(\{1\})$ of the distributed game: during the calculation we overlook transitions which is not selected (in the strategy) in the local game. This means that in the distributed game, to add a control vertex $v \in V_0$ to the attractor using the edge $(v, u)$, we must ensure that $\forall j \in \{1, \ldots, n\}, (\text{proj}(v_j, j) \in V_{0_j} \rightarrow \text{proj}(u, j) = \text{proj}(f_j(v_j), j))$. Lastly, we check if the initial state is contained; the whole calculation and checking process can be done in deterministic P-time.

[NP-C] For completeness proof, we perform a reduction from 3SAT to the finding of positional strategies in a distributed game. Given a set of 3CNF clauses $C_1, \ldots, C_m$ under the set of literals $\{\text{var}_1, \overline{\text{var}_1}, \ldots, \text{var}_n, \overline{\text{var}_n}\}$ and variables $\{\text{var}_1, \ldots, \text{var}_n\}$, the distributed game $G$ is created as follows (see Figure 5 for illustration):

- Create 3 local games $G_1$, $G_2$, and $G_3$, where for $G_i = (V_0 \cup V_1, E_i)$:

Figure 5. Illustrations for the reduction from 3SAT to PositionalDG₀.
V₀ = \{\text{var}_1, \ldots, \text{var}_n\}, V₁ = \{S, T_{\text{var}_1}, F_{\text{var}_1}, \ldots, T_{\text{var}_n}, F_{\text{var}_n}\}.

E₁ = \bigcup_{j=1,\ldots,n}\{(\text{var}_j, T_{\text{var}_j}), (\text{var}_j, F_{\text{var}_j})\}.

Create local game G₄ = (V₀ ⊔ V₁, E₁):

V₀ = \{0K₀, NO₀\} ⊔ \bigcup_{j=1,\ldots,n}\{v_{j0}\}.

V₁ = \{S, OK₁, NO₁\} ⊔ \bigcup_{j=1,\ldots,n}\{v_{jn}\}.

E₁ = \bigcup_{j=1,\ldots,n}\{(v_{j0}, v_{jn})\} ⊔ \{(0K₀, OK₁), (NO₀, NO₁)\}.

Second, create the distributed game G from local games above, and define the set of environment transition to include the following types using the 3SAT problem:

1) (Intention to check SAT) In the 3SAT problem, for clause Cᵢ = (l₁v₁ ⊕ l₂v₂ ⊕ l₃v₃), let the variable for literals l₁v₁, l₂v₂, l₃v₃ be \text{var}_1, \text{var}_2, \text{var}_3. Create a transition in the distributed game from \((S, S, S, S)\) to \((\text{var}_1, \text{var}_2, \text{var}_3, v_{m+6})\).

2) (Intention to check consistency) In the 3SAT problem, for variable \text{var}_i, Create a transition in the distributed game from \((S, S, S, S)\) to \((\text{var}_i, \text{var}_1, \text{var}_2, v_{m+6})\).

3) (Result of clause) In the 3SAT problem, for clause Cᵢ = (l₁v₁ ⊕ l₂v₂ ⊕ l₃v₃), let the variable for the clause be \text{var}_1, \text{var}_2, \text{var}_3. We refer the vertex evaluating \text{var}_j, as true to Tᵢ in the local game G₄; similarly, we use Fᵢ for a variable being evaluated false. For each clause Cᵢ, enumerate over 8 cases for the assignments \text{var}_1, \text{var}_2, \text{var}_3 which make Cᵢ true.

a) For cases which makes the assignment true, create an edge from the assignment to \((\text{var}_1, \text{var}_2, \text{var}_3, OK₀)\).

b) For cases which makes the assignment false, create an edge from the assignment to \((\text{var}_1, \text{var}_2, \text{var}_3, NO₀)\).

4) (Result of variable consistency) For all i ∈ \{1, \ldots, n\}:

a) Create two edges \((Tᵢ, Tᵢ, Tᵢ, v_{m+6}), (\text{var}_1, \text{var}_1, \text{var}_1, OK₀)\) and \((Fᵢ, Fᵢ, Fᵢ, v_{m+6}), (\text{var}_1, \text{var}_1, \text{var}_1, OK₀)\).

b) For other 6 combinations \((Tᵢ, Fᵢ, Fᵢ, v_{m+6}), (Fᵢ, Fᵢ, Tᵢ, v_{m+6}), (Tᵢ, Tᵢ, Fᵢ, v_{m+6}), (Tᵢ, Fᵢ, Tᵢ, v_{m+6}), (Fᵢ, Tᵢ, Tᵢ, v_{m+6}), (Tᵢ, Fᵢ, v_{m+6}),\) create edges to \((\text{var}_1, \text{var}_1, \text{var}_1, NO₀)\).

5) (Continuous execution) For all i ∈ \{1, \ldots, n\}:

a) For all combinations \((Tᵢ, Fᵢ, Fᵢ, OK₁), (Fᵢ, Fᵢ, Tᵢ, OK₁), (Tᵢ, Fᵢ, Fᵢ, OK₁), (Fᵢ, Fᵢ, Fᵢ, OK₁), (Tᵢ, Fᵢ, Tᵢ, OK₁), (Fᵢ, Fᵢ, Tᵢ, OK₁), (Tᵢ, Tᵢ, Fᵢ, OK₁), (Fᵢ, Tᵢ, Tᵢ, OK₁), create edges to \((\text{var}_1, \text{var}_1, \text{var}_1, OK₀)\).

b) For all combinations \((Tᵢ, Fᵢ, Fᵢ, NO₁), (Fᵢ, Fᵢ, Tᵢ, NO₁), (Tᵢ, Fᵢ, Fᵢ, NO₁), (Fᵢ, Fᵢ, Fᵢ, NO₁), (Fᵢ, Tᵢ, Tᵢ, NO₁), (Tᵢ, Fᵢ, Tᵢ, NO₁), (Fᵢ, Tᵢ, Tᵢ, NO₁), (Tᵢ, Tᵢ, Tᵢ, NO₁), create edges to \((\text{var}_1, \text{var}_1, \text{var}_1, NO₀)\).

We claim that \{C₁, \ldots, Cₙ\} is satisfiable if and only if G has a positional distributed strategy to reach \((\text{var}_1, \text{var}_1, \text{var}_1, OK₀)\) from \((S, S, S, S)\).

1) If \{C₁, \ldots, Cₙ\} is satisfiable, let the set of satisfying literals be L', and assume that for all literals, in each pair \((\text{var}_i, \text{var}_j)\) exactly one of them is in L' (this is always possible). For the distributed game G, in local games G₁, G₂, and G₃, the positional strategy for control vertex \text{var}_i move to Tᵢ if \text{var}_i ∈ L', and move to Fᵢ if \text{var}_i ∈ L' (for G₄, simply use the local edge). In a play, as player-1 starts the move, any of his selection leads to a player-0 vertex:

- If player-1 choose edges of type 1 (intension to check the clause of SAT), for G₁, G₂, and G₃, the vertex uses its positional strategy, which corresponds to the assignment in the clause. The combined move then forces player-1 to choose an edge of type 3(a), leading to the target state.

- If player-1 choose edges of type 2 (intension to check the consistency), as the positional strategies for G₁, G₂, and G₃ are all derived from the same satisfying instance of the 3SAT problem, for each strategy, it performs the same move from \text{var}_i to Tᵢ or to Fᵢ; the combined move of player-0 forces player-1 to choose an edge of type 4(a), leading to the target state.

2) Consider a distributed positional strategy \(f₁, f₂, f₃, f₄\) which reaches \((\text{var}_1, \text{var}_1, \text{var}_1, OK₀)\) from \((S, S, S, S)\). In G₄, for each control vertex \text{var}_i, it points to Tᵢ or Fᵢ. The positional strategy of G₄ generates a satisfying instance of the 3SAT problem:

- Assign \text{var}_i in the 3SAT problem to true if the strategy points vertex \text{var}_i in G₁ to Tᵢ.

- Assign \text{var}_i in the 3SAT problem to false if the strategy points vertex \text{var}_i in G₁ to Fᵢ.

We analyze the size of the game and the time required to perform the reduction.

- For \(i = 1, 2, 3\), Gᵢ contains \(3n + 1\) vertices, and G₄ has \(2(m + n + 2) + 1\) vertices. As the total vertices of the distributed game is the product, it is polynomial to the original 3SAT problem instance.

- Consider the time required to perform reduction from 3SAT to PositionalDG₀.
A. Solving Distributed Games using SAT Methods

Apart from the search method above, in this section we give an alternative approach based on a reduction to SAT. Madhusudan, Nam, and Alur \[1\] designed the bounded witness algorithm (based on unrolling) for solving reachability (local) games. Although based on their experiment, the witness algorithm is not as efficient as the BDD based approach in centralized games, we find this concept potentially useful for solving distributed games. For this, we have created a variation (Algorithm 3) for this purpose.

To provide an intuition, first we 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 (Algorithm 3) for this purpose.

$$V_{init} \setminus V_{goal}$$

This recursive definition implies that if \(v\) in \(V_0\) (resp. in \(V_1\)) is not the goal but in the witness set, then exists one (resp. for all) successor \(v'\) which should either be \(i\) in a goal state or \(ii\) also in the witness: note that for \(iii\), the number of allowable steps to reach the goal is decreased by one. This definition ensures that all plays defined in the witness reaches the goal from the initial state within \(d - 1\) steps: If a play (starting from initial state) has proceeded \(d - 1\) steps and reached \(v\) \(\notin V_{goal}\), then based on (2), \(\langle v \rangle_d\) should be \text{false}. However, based on (1), (3), (4) the \(\langle v \rangle_d\) should be set to \text{true} (reachable from initial states using \(d - 1\) steps). Thus the SAT problem should be unsatisfiable.

In general, Algorithm 3 creates constraints based on the above concept, but compared to the bounded local game reachability algorithm in \[1\], it contains slight modifications:

1. When a variable \(\langle v \rangle_j\) is evaluated to \text{true}, it means that vertex \(v\) can reach the target state within \(d - i\) steps, which is the same as what is defined in \[1\]. However, we introduce more variables for edges in local games, which is shown in STEP 1: when a variable \(\langle e \rangle\) is evaluated to \text{true}, the distributed strategy uses the local transition \(e\).
2. To achieve locality, we must include constraints specified in STEP 4: the positional (memoryless) strategy disallows to change the use of local edges from a given vertex.
3. We modify the impact of control edge selection in STEP 6 by adding an additional implication \(\langle e \rangle \Rightarrow \langle v' \rangle\) over the original constraint in the witness algorithm \[1\]. Here as in Mohalik and Walukiwitz’s formulation, all subgames in a control position should proceed a move (the progress of a global move is a combination of local moves), we need to create constraints considering all possible local edge combinations.

In appendix B, we give an alternate algorithm working with different formulation of distributed games where in each control location, only one local game can move: a run of the game may execute multiple local moves until it reaches a state where all local games are in an environment position. We find this alternative formulation closer to the interleaving semantics of distributed systems.

VII. CONVERSION FROM STRATEGIES TO CONCRETE IMPLEMENTATIONS

Once 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, e.g., AbsInt\[3\]), 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 or the deadline on existing actions from the

\[http://www.absint.com/\]
Algorithm 3: PositionalDistributedStrategy_BoundedSAT

Data: Distributed game graph $G = (V_0 \cup V_1, E)$, set of initial states $V_{init}$, set of target states $V_{goal}$, the unrolling depth $d$

Result: Output: whether a distributed positional strategy exists to reach $V_{goal}$ from $v_{init}$

begin

/* STEP 1: Variable creation */
let clauseList := getEmptyList() /* Store all clauses for SAT solvers */
for $v = (v_1, \ldots, v_m) \in V_0 \cup V_1$ do
  create $d$ boolean variables $(v_1, \ldots, v_m)_1, \ldots, (v_1, \ldots, v_m)_d$;
for local control transition $e = (v, v') \in E$, $v \in V_0_0$ do
  create boolean variable $(v)$;

/* STEP 2: Initial state constraints */
for $v = (v_1, \ldots, v_m) \in V_0 \cup V_1$ do
  if $(v_1, \ldots, v_m) \in V_{init}$ then
    clauseList.add($(v_1, \ldots, v_m)_1$)
  else
    clauseList.add($(\neg(v_1, \ldots, v_m)_1)$)

/* STEP 3: Target state constraints */
for $v = (v_1, \ldots, v_m) \in V_0 \cup V_1$ do
  if $(v_1, \ldots, v_m) \in V_{goal}$ then
    clauseList.add($(v_1, \ldots, v_m)_1$ \land \ldots \land $(v_1, \ldots, v_m)_d$)
  else
    clauseList.add($(\neg(v_1, \ldots, v_m)_d)$)

/* STEP 4: Unique selection of local transitions (for distributed positional strategy) */
for local control transition $e = (x_i, x'_i) \in E$, $x_i \in V_0$ do
  for local transition $e_1 = (x_i, x'_i), \ldots, e_k = (x_i, x'_i) \in E$, $e_1, \ldots, e_k \neq e$ do
    clauseList.add($(e) \Rightarrow (\neg(e_1) \land \ldots \land \neg(e_k))$)

/* STEP 5: If a control vertex is in the attractor (winning region) but not a goal, an edge should be selected to reach the goal state */
for $v = (v_1, \ldots, v_m) \in V_0$ do
  for $v_i$, $i = 1, \ldots, m$ do
    if $v_i \in V_0 \setminus V_{goal}$ then
      let $\bigcup_j e_j$ be the set of local transitions starting from $v_i$ in $G_i$
      if $\bigcup_j e_j \neq \phi$ then
        clauseList.add($(V_{i=1\ldots d}(v)_i) \Rightarrow (V_j(e)_j)$)

/* STEP 6: Impact of control edge selection (simultaneous progress) */
for $v = (v_1, \ldots, v_m) \in V_0$ do
  forall the edge combination $(e_1, \ldots, e_m)$: $e_i = (v_i, v'_i) \in E_i$ when $v_i \in V_0$, or $e_i = (v_i, v_i)$ when $x_i \in V_1$ do
    /* $e_i = (v_i, v_i)$ when $x_i \in V_1$ are simply dummy edges for ease of formulation */
    for $j = 1, \ldots, d - 1$ do
      clauseList.add($(v_1, \ldots, v_m)_j \Rightarrow ((\bigwedge_{i} e_i \in V_{0\_i}(e_i)) \Rightarrow ((v'_1, \ldots, v'_m)_{j+1})$)

/* STEP 7: Impact of environment vertex */
for environment vertex $v = (v_1, \ldots, v_m) \in V_1$ do
  let the set of successors be $\bigcup_j v_j$; for $j = 1, \ldots, d - 1$ do
    clauseList.add($(v_j) \Rightarrow (\bigwedge_{i} \bigvee_{j=1} v_{i\_j})$);

/* STEP 8: Invoke the SAT solver: return true when satisfiable */
return invokeSATsolver(clauseList)
synthesized IM can be translated to a linear constraint system, as in the synthesized IM each 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 \( \sigma_{\frac{a}{b}} \) and release time of \( \sigma_{\frac{a}{b}} \), if (a) \( \sigma_{\frac{a}{b}} \) exists and (b) \( d \neq 0 \), or
2) the execution interval of action \( \sigma_{\frac{a}{b}} \), if \( \sigma_{\frac{a}{b}} \) exists.

In the algorithm, we assume that for every action \( \sigma_d, d \in \mathbb{N} \) where FT mechanisms are not introduced between \( \sigma_d \) and \( \sigma_{d+1} \) 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 4 performs such execution\(^8\) for simplicity assume at most one FT action exists between two actions \( \sigma_i, \sigma_{i+1} \); in our implementation this assumption is released:

- Item (1) performs a interval split between \( \sigma_{\frac{a}{b}} \) and \( \sigma_{\frac{a}{b}} \).
- Item (2) assigns the deadline of \( \sigma_{\frac{a}{b}} \) 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 \( \sigma_d \).

**VIII. IMPLEMENTATION AND CASE STUDIES**

For implementation, we have created our prototype software as an Eclipse-plugin, called GECKO\(^9\) which offers an open-platform based on the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. Currently the engine implements the search-based algorithms, and the SAT-based algorithm is experimented independently under GAVS\(^10\) a tool for visualization and synthesis of games.

To evaluate our approach, here we reuse the example in sec. III and perform automatic tuning synthesis for the selected FT mechanisms. The models specified in this section, as well as the GECKO Eclipse-plugin which generates the result, are available in the website.

**A. Example from Section 2**

In this example, the user selects a set of FT mechanism templates with the intention to implement a fail-then-resend
operation, which is shown in Figure 6. 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, similar to sec. III 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 7, the set of possible control transitions are listed\(^11\) the solver

---

\(^8\)Here we list case 2 only; for case 1 similar analysis can be applied.
\(^9\)http://www6.in.tum.de/~chengch/gecko/
\(^10\)http://www6.in.tum.de/~chengch/gavs/
\(^11\)In our implementation, the PC starts from 0 rather than 1; which is different from the formulation in IM and PISEM.
Algorithm 4: LocalTimingModification

**Data:** Original PISEM $S = (A, N, T)$, synthesized IM $S = (A, N)$

**Result:** For each $\sigma_{a_1}$ and $\sigma_{\frac{1}{a_1}}$, their execution interval $[\alpha_{a_1}, \beta_{a_1})$, $[\alpha_{\frac{1}{a_1}}, \beta_{\frac{1}{a_1}})$

For convenience, use $(X \times \mathcal{T})$ to represent the retrieved value $X$ from PISEM $S$.

begin
for $\sigma_{a_1}[\wedge_{m=1...n_1}[p_{c_{\sigma_{a_1}}, m_{low}, \rho_{c_{\sigma_{a_1}}, m_{up}}}]$ in $\mathcal{T}$ of $A_1$ do
    let $\alpha_{a_1}, \beta_{a_1}, \alpha_{\frac{1}{a_1}}, \beta_{\frac{1}{a_1}}$ // Create a new variable for the constraint system
    /* Type A constraint: causalities within the process */
    constraints.add($\alpha_{a_1} = (\alpha_{\frac{1}{a_1}} \times \mathcal{T})$)
    constraints.add($\alpha_{a_1} = (\beta_{\frac{1}{a_1}} \times \mathcal{T})$)
    constraints.add($\beta_{a_1} = (\beta_{\frac{1}{a_1}} \times \mathcal{T})$)
    constraints.add($\beta_{a_1} = (\beta_{\frac{1}{a_1}} \times \mathcal{T})$)
    constraints.add($\alpha_{\frac{1}{a_1}} = (\alpha_{a_1} \times \mathcal{T})$)
    constraints.add($\beta_{\frac{1}{a_1}} = (\beta_{a_1} \times \mathcal{T})$)
     
    /* Type B constraint: causalities crossing different processes */
for $\sigma_{a_1}[\wedge_{m=1...n_2}[p_{c_{\sigma_{a_1}}, m_{low}, \rho_{c_{\sigma_{a_1}}, m_{up}}}]$ in $\mathcal{T}$ of $A_1$ do
    for $\sigma_{a_1}[\wedge_{m=1...n_4}[p_{c_{\sigma_{a_1}}, m_{low}, \rho_{c_{\sigma_{a_1}}, m_{up}}}]$ in $\mathcal{T}$ of $A_2$ do
        if $\alpha_{a_1}$ do
            if $p_{c_{\sigma_{a_1}}, m_{low}} < p_{c_{\sigma_{a_1}}, m_{low}}$ then constraints.add($\beta_{a_1} > \beta_{\sigma_{a_1}}$)
            if $p_{c_{\sigma_{a_1}}, m_{low}} > p_{c_{\sigma_{a_1}}, m_{low}}$ then constraints.add($\beta_{a_1} > \beta_{\sigma_{a_1}}$)
        else
            if $p_{c_{\sigma_{a_1}}, m_{low}} < p_{c_{\sigma_{a_1}}, m_{low}}$ then constraints.add($\beta_{a_1} > \beta_{\sigma_{a_1}}$)
            if $p_{c_{\sigma_{a_1}}, m_{low}} > p_{c_{\sigma_{a_1}, m_{low}}}$ then constraints.add($\beta_{a_1} > \beta_{\sigma_{a_1}}$)
     
    /* Type C constraint: conservative data dependency constraints */
for $\sigma_{a_1}[\wedge_{m=1...n_1}[p_{c_{\sigma_{a_1}}, m_{low}, \rho_{c_{\sigma_{a_1}}, m_{up}}}]$ in $\mathcal{T}$ of $A_1$ do
    for $\sigma_{a_1}[\wedge_{m=1...n_2}[p_{c_{\sigma_{a_1}}, m_{low}, \rho_{c_{\sigma_{a_1}}, m_{up}}}]$ in $\mathcal{T}$ of $A_2$ do
        if $\sigma_{a_1} := send(pre, ind, n, dest, v, c) \wedge p_{c_{\sigma_{a_1}}, m_{low}} < p_{c_{\sigma_{a_1}, m_{low}}}$ then
            constraints.add($\beta_{a_1} > WCMTT(n, ind)$)

solve constraints using (linear) constraint solvers.

generates an appropriate PC-precondition for each action to satisfy the specification. In Figure 7, bold numbers (e.g., (0000)) indicate the synthesized result. The time line of the execution (the synthesized result) is explained as follows:

1. Process $A$ reads the input, sends $MsgSend(m)$, and waits.
2. Process $B$ first waits until it is allowed to execute $(RecvMsg(m))$. Then it performs a conditional send $MsgSend(req)$ and waits.
3. Process $A$ performs $RecvMsg(req)$, following a conditional send $MsgSend(rsp)$.
4. Process $B$ performs conditional assignment, which assigns the value of $rsp$ to $m$, if $m่ว$ is empty.

We continue the case study by stating assumptions over hardware and timing; these can be specified in GECKO as properties of the model.

1. Process $A$ and $B$ are running on two Texas Instrument LM3S8962 development board[13] under FreeRTOS[14] (a real-time operating system), and messages are communicating over a CAN bus.
2. For each existing or FT action, its WCET on the hardware is 1ms.
3. For all messages communicating using the network, the WCMETT is 3ms.

We apply the LTM algorithm, such that we can generate timing constraints on dedicated hardware; these timing constraints will be translated to executable C code (based on FreeRTOS). Figure 8 is used to assist the explanation of LTM, where variables used in the linear constraint solver are specified as follows:

- $a_{\frac{1}{a_1}}$: release time for action "RecvMsg(req)" in process $A$.

1. http://www.luminarmicro.com/products/LM3S8962.html
2. http://www.freertos.org/
Type A

Process $A$

Period = 100ms

$m \in \{T, F\}$

req, rsp $\in \{T, F\}$, req$_A$, rsp$_B$ $\in \{T, \bot\}$

- $\text{InputRead}(m)[0000, 0001); (0000$
- $\text{MsgSend}(m)[0000, 0001); (0000$
- $\text{RecvMsg}(req)[0000, 1100); (0011$
- $\text{InputRead}(m), \text{MsgSend}(rsp)[0000, 1100); (0011$
- $\text{PrintOut}(m); [0011, 1100); (0100$

Process $B$

Period = 100ms

$m \in \{T, F\}$, $m_v \in \{T, \bot\}$

req, rsp $\in \{T, F\}$, req$_A$, rsp$_B$ $\in \{T, \bot\}$

- $\text{RecvMsg}(m)[00101, 01100); (00101$
- $\text{MsgSend}(req)[00101, 10000); (00101$
- $\text{PrintOut}(m); [0011, 10000); (01100$

Figure 7. A concept illustration for the control choices in the generated game.

Figure 8. An illustration for applying LTM for the example in sec. VIII-A, and the corresponding linear constraints.

- $\frac{a_2}{\tau}$: release time for action "if(req$_B \neq \bot$) rsp := m" and similarly, the deadline for "RecvMsg(req)"") in process $A$.
- $\frac{a_2}{\tau}$: release time for action "if(req$_B \neq \bot$) MsgSend(rsp)" in process $A$.
- $\frac{b_2}{\tau}$: release time for action "if(m$_v = \bot$) req := T" in process $B$.
- $\frac{b_2}{\tau}$: release time for action "if(m$_v = \bot$) MsgSend(req)" in process $B$.
- $\frac{b_2}{\tau}$: release time for action "if(m$_v = \bot$) m := rsp" in process $B$.

As in process $A$, there exists a time interval $[40, 99]$ between two existing actions $\text{MsgSend}(m)$ and $\text{PrintOut}(m)$, the LTM algorithm will prefer to utilize this interval than splitting $[0, 40)$, as using $[40, 99]$ generates the least modification on the scheduling. The generated linear constraint system is also shown in Figure 8 An satisfying instance for ($a_2, a_4, a_2, b_2, b_2, b_2$) could be $(72, 77, 82, 62, 67, 87)$; instructions concerning the release time and the deadline for the generated fault-tolerant model can be annotated based on this.

B. Another Example

For the second example, the user selects an inappropriate set of FT mechanisms

- When $B$ does not receive $m$ from $A$, it sends a request command.
- When $A$ receives a request message, it does not send the response; this violates the original intention of the designer.

\[14\text{This is originally a design mistake when we specify our FT mechanism patterns; however interesting results are generated.}\]
Figure 9. Screenshots of GECKO when executing the example in sec. VIII-A.
Surprisingly, GECKO reports a positive result with an interesting sequence! For all FT actions in process $A$, they should be executed with the precondition of $PC_B$ equal to 0000, meaning that FT mechanisms in $A$ are executed before $\text{RecvMsg}(m)$ in $B$ starts. In this way, $A$ always sends the message $\text{MsgSend}(\text{rsp})$ containing the value of $m$, and as at most one message loss exists in one period, the specification is satisfied.

$C$. Discussion

Concerning the running time of the above two examples, the searching 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 huge benefits, as another exponential blow-up caused by the translation from variables to states is also unavoidable: this is the reason why currently we use a forward search algorithm combining with BDDs in the implementation.

Nevertheless, this does not mean that FT synthesis in practice is not possible; our argument is as follows:

1) We have indicated that this method is applicable for small examples (similar to the test case in the paper).
2) To fight with complexity we consider it important to respect the compositional (layered) approach used in the design of embedded systems: once when a system have been refined to several subsystems, it is more likely for our approach to be applicable.

IX. RELATED WORK

Verification and synthesis of fault tolerance is an active field [12], [17], [16], [13], [4], [19], [2], [9]. Among all existing works, we find that the work closest to ours is by Kulkarni et.al. [16]. Here we summarize the differences in three aspects.

1) (Problem) As we are interested in real-time embedded systems, our starting model resembles existing formulations used in the real-time community, where time is explicitly stated in the model. Their work is more closely to protocol synthesis and the starting model is based on (a composition of) FSMs.
2) (Approach) As our original intention is to facilitate the pattern selection and tuning process, our approach does not seek for the synthesis of complete FT mechanisms and can be naturally connected to games (having a set of predefined moves). Contrarily, their results focus on synthesizing complete FT mechanisms, for example voting machines or mechanisms for Byzantine generals’ problem.
3) (Algorithm) To apply game-based approach for embedded systems, our algorithms includes the game translation (timing abstraction) and constraint solving (for implementability). In addition, our game formulation enables us to connect and modify existing and rich results in algorithmic game solving: for instance, we reuse the idea of witness in [1] for distributed games, and it is likely to establish connections between incomplete methods for distributed games and algorithms for games of imperfect information [8].

A recent work by Girault et.al. [13] follows similar methodologies (i.e., on protocol level FSMs) to [16] and performs discrete controller synthesis for fault-tolerance; the difference between our work and theirs follows the argument above.

Lastly, we would like to comment on the application of algorithmic games. Several important work for game analysis or LTL synthesis can be found from Bloem and Jobstmann et.al. (the program repair framework [15]), Henzinger and Chatterjee et.al. (Alpaga and the interface synthesis [5], [10]), or David and Larson et.al. (Uppaal TIGA [3]). One important distinction is that due to our system modeling, we naturally start from a problem of solving distributed games and need to fight with undecidability immediately, while the above works are all based on a non-distributed setting.

X. CONCLUDING REMARKS

This paper presents a comprehensive approach (see Figure 10 for concept illustration) for the augmenting of fault-tolerance for real-time distributed systems under a game-theoretic framework. We use simple yet close-to-reality models (PISEM) as a starting point of FT synthesis, translate PISEM to distributed games with safe abstractions, perform game solving and later implementability analysis. The above flow is experimented in a prototype, enabling us to utilize model-based development framework to perform FT synthesis. These mechanisms may have interesting applications in
distributed process control and robotics. To validate our approach, we plan to increase the maturity of our prototype system and study new algorithms for performance gains.

Acknowledgments: The first author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

REFERENCES

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic computational techniques for solving games. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):118–128, 2005.

[2] A. Arora, P. Attie, and E. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 7th annual ACM symposium on Principles of distributed computing (PODC’98)*, pages 173–182. ACM, 1998.

[3] G. Behrmann, A. Coupard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Proceedings of the 19th international conference on Computer aided verification (CAV’07)*, volume 4590 of LNCS, pages 121–125. Springer-Verlag, 2007.

[4] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

[5] D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. Henzinger. Alpaga: A tool for solving parity games with imperfect information. In *Proceedings on the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’10)*, volume 6015 of LNCS, pages 58–61. Springer-Verlag, 2009.

[6] C.-H. Cheng, C. Buckl, J. Esparza, and A. Knoll. Modeling and verification for timing satisfaction of fault-tolerant systems with finiteness. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT’09)*, pages 208–215. IEEE Computer Society, 2009.

[7] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[8] M. De Wulf, L. Doyen, and J. Raskin. A lattice theory for solving games of imperfect information. In *Proceedings of the 9th International Workshop in Hybrid Systems: Computation and Control*, volume 3927 of LNCS, pages 153–168. Springer-Verlag, 2006.

[9] R. Dimitrova and B. Finkbeiner. Synthesis of fault-tolerant distributed systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA’09)*, volume 5799 of LNCS, pages 321–336. Springer-Verlag, 2009.

[10] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT’08)*, pages 79–88. ACM, 2008.

[11] J. Eidson, M. Fischer, and J. White. Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *34th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 243–254, 2002.

[12] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS’08)*, volume 4963 of LNCS, pages 315–331, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[14] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Software Patterns Series, 2007.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV’05)*, volume 3576 of LNCS, pages 226–238. Springer-Verlag, 2005.

[16] S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT’00)*, volume 1926 of LNCS, pages 339–359. Springer-Verlag, 2000.

[17] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT’94)*, volume 863 of LNCS, pages 41–76. Springer-Verlag, 1994.

[18] S. Mohalik and I. Walukiewicz. Distributed games. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS’03)*, volume 2914 of LNCS, pages 338–351. Springer-Verlag, 2003.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[20] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[21] K. Tindell, A. Burns, and A. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.
A. The Need of Reestimating the WCMTT in CAN Buses when FT Messages are Introduced

To have an understanding whether newly introduced FT messages can change the existing networking behavior is both hardware and configuration dependent. In this section, we only describe the behavior when FT messages are introduced in a Control Area Network (CAN bus), which is widely used in automotive and automation domains. Here we give configuration settings (conditions) such that newly introduced messages do not influence the existing networking behavior. For details concerning the timing analysis of CAN, we refer readers to [21], [7].

Proposition 1: Given an ideal CAN bus with message priority from 1 to $k$, when the three conditions are satisfied:
1) No message with priority $k$ is not used in the existing network.
2) The predefined size of the message for priority $k$ is larger than all messages with priority 1 to $k-1$.
3) All FT messages are having priority smaller or equal to $k$, and the size is less than the message size stated in (2).

When the WCMTT is derived using the analysis in [21], concerning the WCMTT of all messages with priority 1 to $k$, it is indifferent to the newly introduced messages.

Proof: (Outline) Based on the algorithm in [21], for a message with priority $i \in \{1, \ldots, k\}$, its timing behavior only changes with two factors:
- (a) The blocking time caused by a message with lower priority $j > i$ changes: when a message with lower priority changes to a bigger message size, the blocking time increases.
- (b) The interference from messages with higher priority $j < i$.

We proceed the argument as follows.

For timing changes due to (b), as FT messages are all with lower priorities (based on condition 3), they do not create or increase interferences with this type.

For timing changes due to (a), we separate two two cases:
- As the size of all FT messages are smaller or equal than the message size specified in (2), then the timing behavior for messages with priority 1 to $k-1$ do not change.
- Lastly, although the message with priority $k$ can change as it can now be blocked by a lower priority message, such message does not exist based on condition (1).

By the above information, in our framework we may assume that all messages transmitted in a CAN bus are with lowest priority $k+1$, and then perform a simple timing analysis at Step A.2.2 before creating the game; in this way, the problem is Step A.2.2 ([Problem 2]) is safe to neglect.

B. Algorithm Modification for Interleaving of Local Games

Algorithm 5: PositionalDistributedStrategy_ControlInLocalGameInterleaving_BoundedSAT_0

Data: Distributed game graph $\mathcal{G} = (V_0 \uplus V_1, \mathcal{E})$, set of initial states $V_{init}$, set of target states $V_{goal}$, the unrolling depth $d$

Result: Output: whether a distributed positional strategy exists to reach $V_{goal}$ from $v_{init}$

begin

let clauseList := getEmptyList() /* Store all clauses for SAT solvers */
execute STEP 1 to STEP 5 mentioned in PositionalDistributedStrategy_BoundedSAT_0

/* STEP 6: Impact of control edge selection */
for local control transition $e = (x_i, x'_i) \in E_i, x_i \in V_0$, do
for $v = (v_1, \ldots, v_m) \in V_0 \uplus V_1$ where $x_i = v_i$ do
clauseList.add($\{e\} \Rightarrow (\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_m \rangle) \Rightarrow$
$\langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m \rangle)_{d+1} \lor \cdots \lor \langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m \rangle_d)$)

/* STEP 7: Impact of environment vertex */
for environment vertex $v = (v_1, \ldots, v_m) \in V_1$ do
let the set of successors be $\bigcup_j v_j$; for $j = 1, \ldots, d-1$ do
clauseList.add($\langle v_j \rangle \Rightarrow (\langle \bigvee \langle v_{j+1} \lor \cdots \lor v_m \rangle \rangle)$)

/* STEP 8: Invoke the SAT solver: return true when satisfiable */
return invokeSATsolver(clauseList)
(Remark) Compared to Mohalik and Walukiwitz’s formulation, as in this formulation, only one subgame in a control position can proceed a move, we do not need to create constraints considering all possible combinations in STEP 6, which is required in the algorithm PositionalDistributedStrategy_BoundedSAT_0 (sec. VI).

C. BRIEF INSTRUCTIONS ON EXECUTING EXAMPLES IN GECKO

Here we illustrate how FT synthesis is done in our prototype tool-chain using the example in sec. VIII-A: first we perform model transformation and generate a new model which equips FT mechanisms. Then executable code can be generated based on performing code-generation over the specified model (optional). Once when the GECKO Eclipse add-on is installed (see our website for instructions), proceed with the following steps:

- The model (F01_FT_Synthesis_Correct.xmi) for sec. VIII-A contains the fault model, the hardware used in the system, and pre-inserted FT mechanism blocks, but their timing information is unknown.
- Right click on the selected model under synthesis, choose "Verification" -> "Gecko: Model Transformation and Analysis". A pop-up window similar to fig. 9a is available.
- In the General tab, choose Symbolic FT synthesis using algorithmic game theory.
- In the Platform Analysis tab, set up the default actor WCET and network WCMTT to be 1 and 3.
- In the Output tab, select the newly generated output file.
- Press "Finish". Results of intermediate steps are shown in the console, including FT mechanisms as interleaving models (fig. 9b), constraints derived from LTM (fig. 9c), and results of timing (fig. 9d) after executing the constraint solver.
- In fig. 9b, the mechanism dumped from the engine specifies the action "if(req, =⊥) MsgSend(rsp)"; note that this action implicitly implies that when req, =⊥, a null-op which only updates the program counter should be executed; this is captured by our synthesis framework.
- In fig. 9d, the total execution time is roughly 18s because the engine dumps the result back to mechanisms in textual form, which consumes huge amount of time: executing the game and performing constraint solving take only a small portion of the total time.
- When the model is generated, users can again right click on the newly generated model, and select Code Generation in the tab General: the code generator then combines the model description and software templates for dedicated hardware and OS to create executable C code.