Designing Run-Time Fault-Tolerance Using Dynamic Updates

ICSE SEAMS 2007

Ali Ebnenasir
Computer Science Department
Michigan Technological University
Michigan, USA

Motivation

- It is difficult to anticipate and prevent all types of faults
- Inevitably, unanticipated faults will be detected after program deployment, and …
- programs may not possess the necessary functionalities to tolerate unanticipated faults!

How do we design systems that obtain necessary fault-tolerance functionalities at run-time?

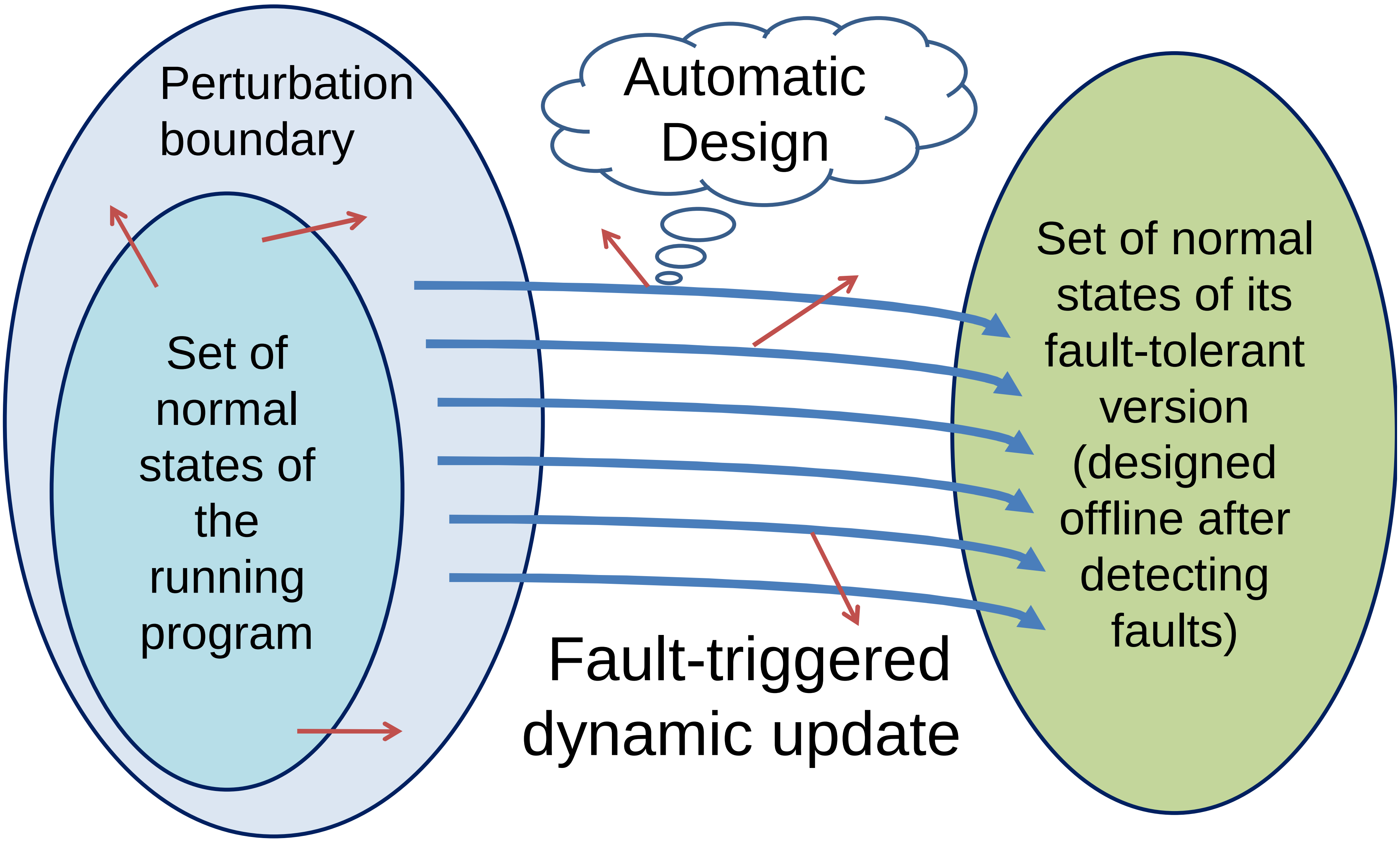
Design Complexity

- A major challenge is to design mechanisms that ensure a system
  - acquires necessary *fault-tolerance* functionalities at run-time under certain correctness requirements, and
  - tolerates faults while obtaining fault-tolerance functionalities.

Contributions

- We present a formal framework for
  1. defining run-time fault-tolerance
     - I.e., dynamic (run-time) *replacement of the controlling software of an autonomous system with its fault-tolerant version in reaction to the occurrence of faults.*
  2. automated design of run-time fault-tolerance

Proposed Solution



Problem Formulation

Programs and Specifications

- Program p: tuple $< V_p , \delta_p>$
  - V_p is a finite set of variables; finite state space (denoted S_p)
  - $\delta_p \subseteq S_p \times S_p$ denotes the set of program transitions (s, s')
- State predicate $X \subseteq S_p$
- Closure of a state predicate X in a program p
  - $\forall(s, s'):(s, s') \in \delta_p : (s \in X \Rightarrow s' \in X)$
- Specification [Alpern and Schneider 1985]:
  - Safety: something bad never happens; modeled as a set of bad transitions $\mathfrak{B} \subseteq S_p \times S_p$
    - E.g., integer variable *t* is not increased by more than 2 units in each transition
  - Liveness: something good will eventually occur; modeled as a set $\mathfrak{L}$ of infinite sequences of states
    - E.g., X *leads-to* Y, if X holds then Y will eventually hold.
- Invariant: a state predicate $\mathfrak{I}$ that captures the set of normal states
  - $\mathfrak{I}$ is closed in p
  - From $\mathfrak{I}$ all computations of p satisfy its specification

B. Alpern and F. B. Schneider. "**Defining liveness**". Information Processing Letters, 21:181–185, 1985.

Union of Programs

- $p_1 = < X_1 , \delta_1 >$ and $p_2 = < Y_2 , \delta_2 >$, where $X_1 \cap Y_2 = \phi$
  - State space S_u: constructed by all possible valuations of variables of $X_1 \cup Y_2$
  - Image of a state $s \in S_{p1}$: a set of states in S_u (denoted $s \uparrow S_u$) such that for each $x_i \in X_1$, $x_i(s) = x_i(s_{img})$, where $s_{img} \in (s \uparrow S_u)$
  - Image of a transition $(s, s') \in \delta_1$: a set of transitions $(s_{img}, s'_{img}) \in S_u \times S_u$, denoted $\delta_1 \uparrow S_u$, such that,
    1. for each $x_i \in X_1$, $x_i(s) = x_i(s_{img})$ and $x_i(s') = x_i(s'_{img})$
    2. for each $y_j \in Y_2$, $y_j(s_{img}) = y_j(s'_{img})$
  - Likewise, we define $(\delta_2 \uparrow S_u)$
- The union of p_1 and p_2:

$$p_u = < X_1 \cup Y_2 , \delta_u >, \text{ where } \delta_u = (\delta_1 \uparrow S_u) \cup (\delta_2 \uparrow S_u)$$

Union of Programs: Example

- Consider $p_1 = < \{x\} , \delta_1 >$ and $p_2 = < \{y\} , \delta_2 >$, where x and y are Boolean variables
- Image of a state <*x* = *false*> includes the following states in S_u

    <x =false, y=false >

    <x =false, y=true >

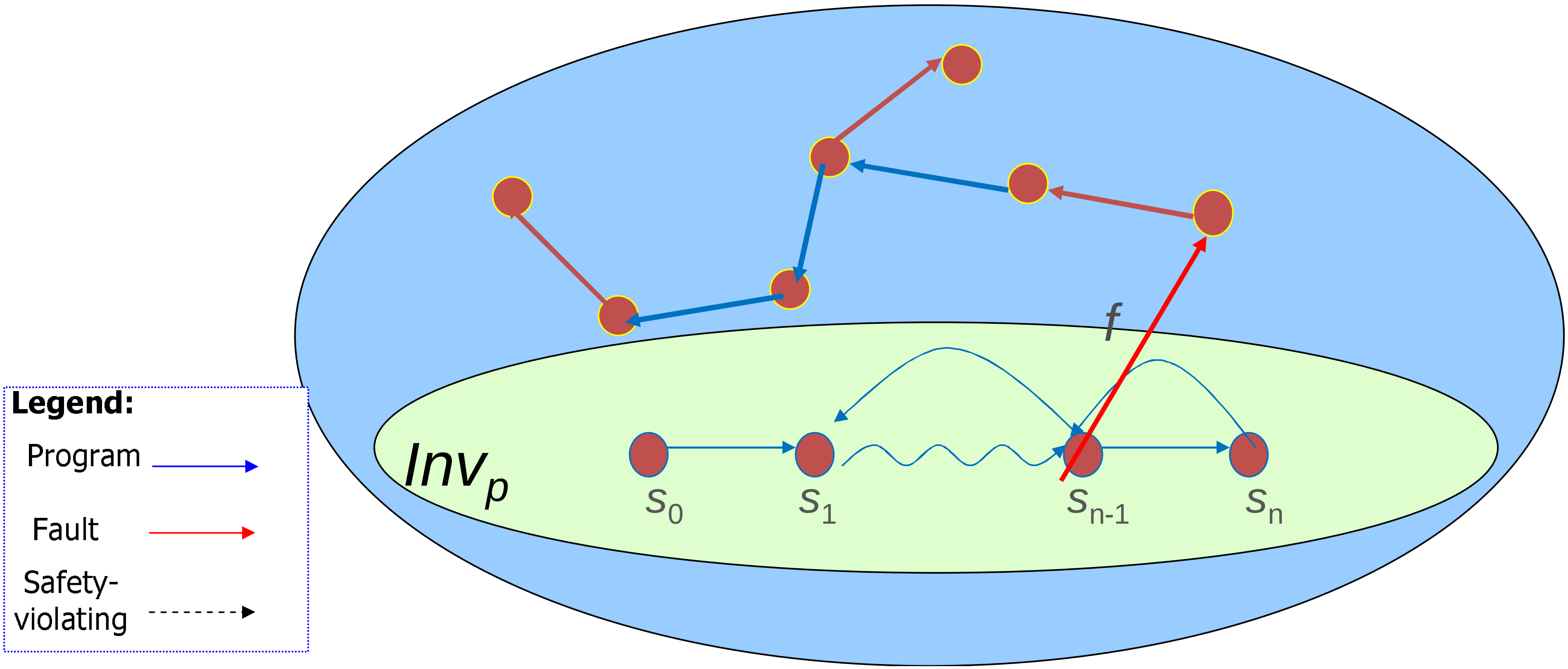
- Image of a transition <*x*=*false*> $\rightarrow$ <*x*=*true*> in δ_1 includes the following transitions in $S_u \times S_u$

    <x=false, y=false > $\rightarrow$ <x=true, y=false >

    <x=false, y=true > $\rightarrow$ <x=true, y=true >

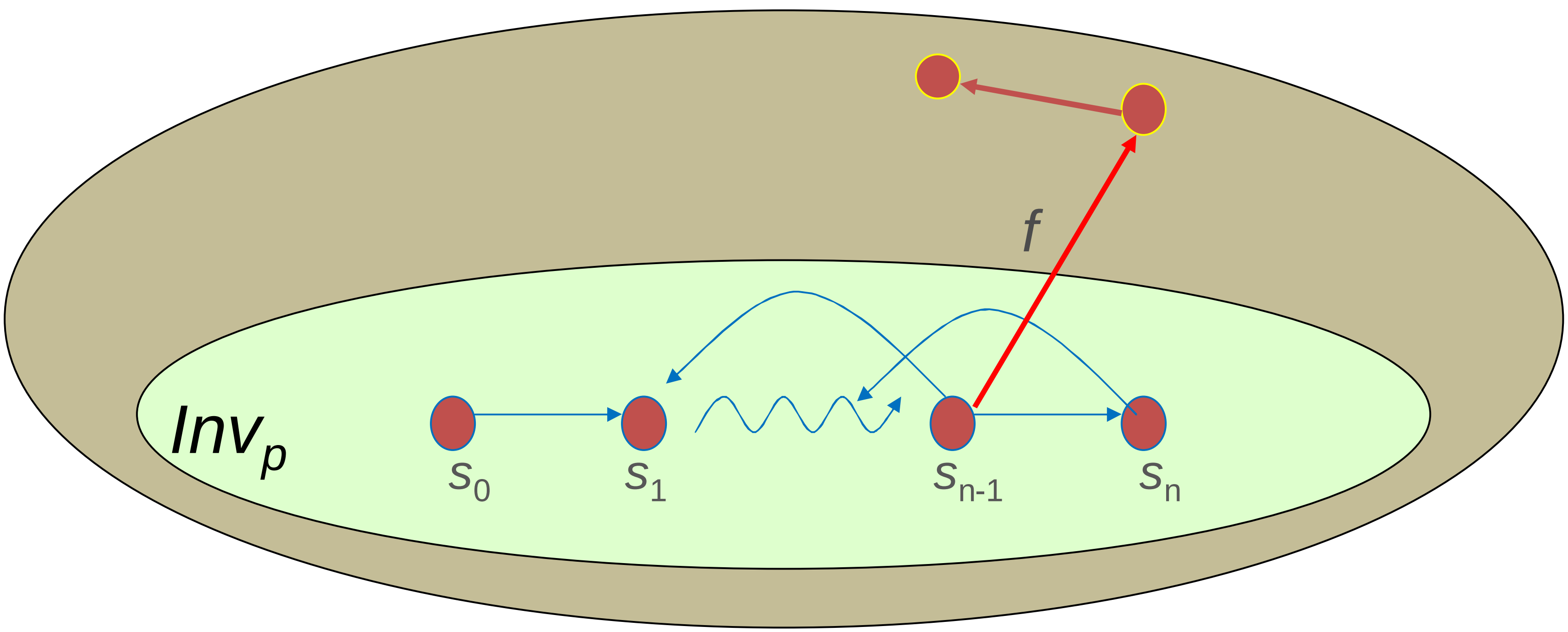
Faults and Fault-Span

- For a program $p = < V_p\ ,\ \delta_p >$, a fault-type *f* is defined as $f \subseteq S_p \times S_p$
- Fault-Span: *f-span* of p is a set of states reachable from the invariant of p by $\delta_p \cup f$



Sub Fault-Span

- Sub *f*-span: set of states outside the invariant that are reachable from the invariant of p by only *f* transitions



Levels of Fault-Tolerance

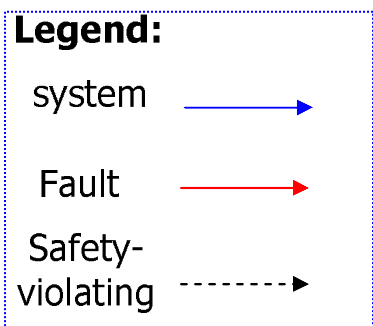
- <u>*Level of fault-tolerance*</u>:
  - the extent to which safety and liveness specifications are satisfied in the presence of faults
- Three levels [Arora and Gouda 1992]:
  1. In the presence of faults,
     - A *failsafe* program satisfies safety of specification
     - A *nonmasking* program eventually recovers to invariant
     - A *masking* program
       - eventually recovers to invariant, and
       - guarantees safety during recovery
  2. In the absence of faults, satisfies safety & liveness *specifications*

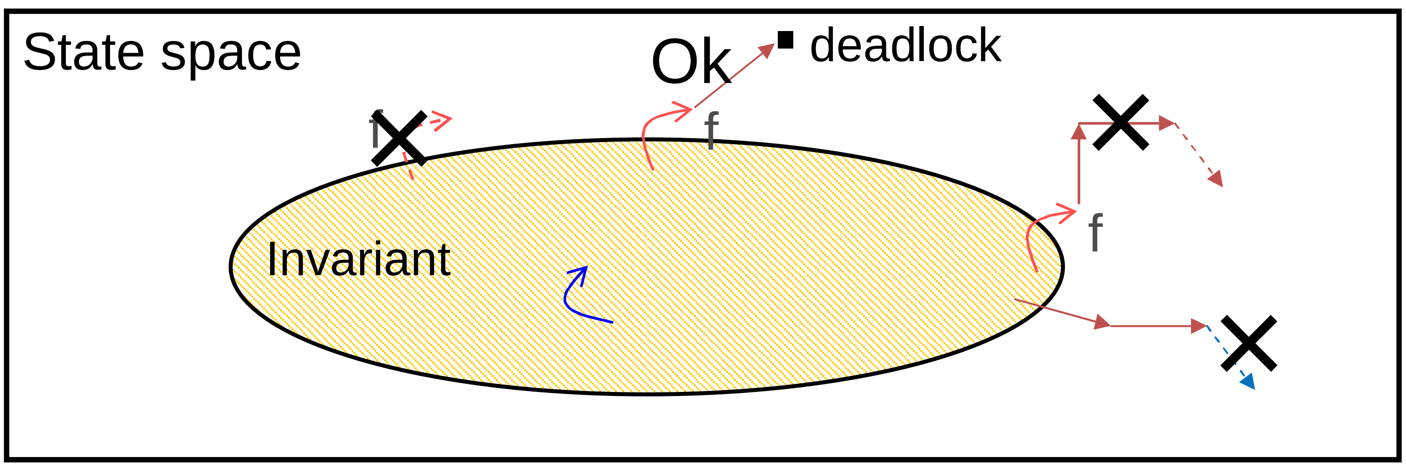
Anish Arora, Mohamed G. Gouda, "**Closure and Convergence: A Foundation of Fault-Tolerant Computing**". IEEE Trans. Software Eng. 19(11): 1015-1027 (1993)

Fault-Tolerance Against Anticipated Faults

Designing Failsafe Fault-Tolerance

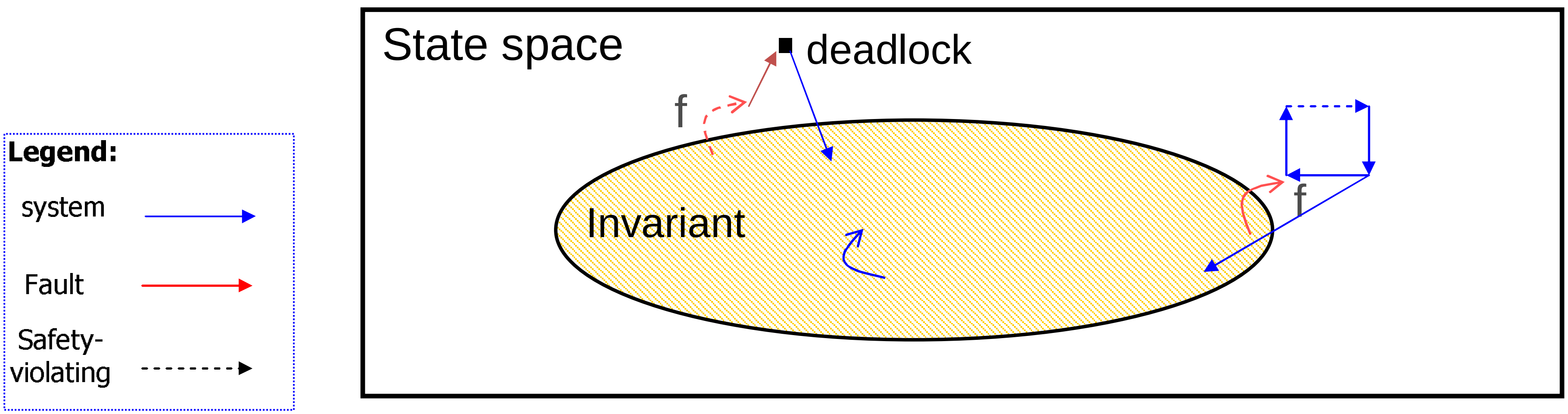
- Failsafe fault-tolerance
  - Safety specification will never be violated
    - E.g., boiler tank will not overflow even if one of the sensors is corrupted.
- Requirements of failsafe fault-tolerance
  - Identify and resolve states
    - from where faults directly violate safety, called *offending states*
    - outside the invariant reachable by faults from where the program itself may violate safety, called *risky states*





Designing Nonmasking Fault-Tolerance

- Nonmasking fault-tolerance
  - Recovers to its invariant when faults stop occurring
    - E.g., if the level of fluid is above MAX then it will eventually go below MAX and above MIN
- Requirements of nonmasking fault-tolerance
  - Identify and resolve
    - Non-progress cycles
    - Deadlocked states



Run-Time Fault-Tolerance Against Unanticipated Faults

Challenges

- If we do not have knowledge about the behavior of faults, then how can we design failsafe/nonmasking/ masking fault-tolerance beforehand so that at run-time we get the desired behaviors?

- Questions:
  - How do we identify offending/risky states?
  - How do we identify reachable non-progress cycles and deadlock states?

Run-Time Fault-Tolerance

- Proposed Design principle:

A running fault-intolerant program is eventually replaced with its fault-tolerant version after the detection of unanticipated faults

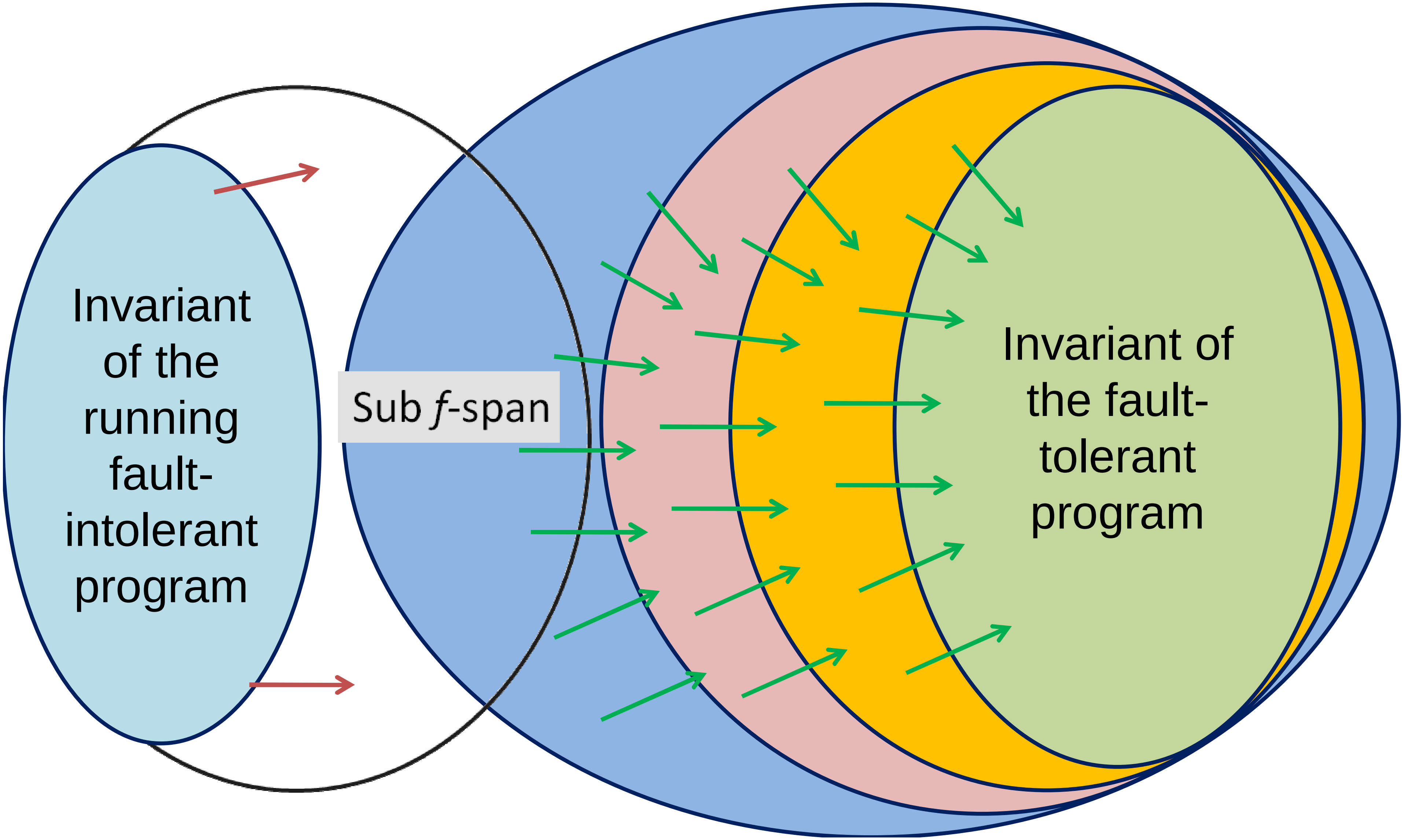
- After the detection of an unanticipated fault-type, a fault-type is formulated based on the effect of faults
- Subsequently, we
  - Automatically synthesize
    1. a fault-tolerant version of the running fault-intolerant program (already solved in our previous work)
    2. a program that is responsible for run-time replacement of the intolerant program with its tolerant version, called the ***fault-triggered update program***

Problem Statement

- Given
  - a fault-intolerant program $p_o = < V_o \, , \delta_o >$,
  - an invariant $\mathfrak{I}_o$ of p_o ,
  - its specification *spec*,
  - a fault-type *f*,
  - sub *f*-span SFS_o of p_o from $\mathfrak{I}_o$,
  - a program $p_n = < V_n \, , \delta_n >$ with an invariant $\mathfrak{I}_n$, that is a failsafe/nonmasking/masking fault-tolerant version of p_o

- Identify a program $p = < V_o \cup V_n \, , \delta >$ in the state space S_u of the union of p_o and p_n , such that
  1. p satisfies the safety of *spec* (***safeness***),
  2. p satisfies "($SFS_o \uparrow S_u$) *leads-to* ($\mathfrak{I}_n \uparrow S_u$)" (***progress***), and
  3. no transition in δ starts in the images of $\mathfrak{I}_o$ and $\mathfrak{I}_n$, and ends in the image of $\mathfrak{I}_o$ (***interference-freedom***).

Synthesizing the Update Program



Soundness and Completeness Results

- Soundness Theorem:
  - The synthesized update program meets the requirements of the problem statement
- Completeness Theorem:
  - If our algorithm fails then there is no solution under the constraints of the problem statement.

Completeness theorem gives us an *impossibility test* for the design of fault-tolerant adaptations

Defining Fault-Tolerant Updates

Fault-Tolerant Dynamic Updates

- Failsafe dynamic update
  - Meets safeness in the presence of faults
  - May violate progress
- Nonmasking dynamic update
  - Meets progress in the presence of faults
  - May violate safeness
- Masking dynamic update
  - Meets both safeness and progress in the presence of faults
- Interference-freedom must be met in failsafe/nonmasking/masking updates

Related Work

1. Dijkstra's predicate transformers
   – Calculate the weakest predicate
2. Dynamic program update
   – Focus on upgrading a program (or part of it) at run-time with less emphasis on dependability issues
3. Model-driven development of adaptation
   – Mostly manual design of adaptation
4. Invariant lattice for adaptive distributed programs
   – A method for verifying adaptations

Our approach has been inspired by all above approaches

Conclusions

- Defined run-time fault-tolerance using dynamic program updates
- Formulated the problem of designing run-time fault-tolerance
- Presented a sound and complete algorithms for automatic synthesis of fault-triggered updates (respectively, adaptations)
- Defined three levels of fault-tolerant dynamic updates (respectively, adaptations)