TCPS Book
Contents

CHAPTER  7 • Formal Reasoning about Resilient CPS 1

Linas Laibinis and Elena Troubitsyna

7.1  INTRODUCTION  2
7.2  CYBER-PHYSICAL SYSTEMS  3
7.3  BACKGROUND: EVENT-B  5
7.4  FORMAL DEVELOPMENT OF A RESILIENT CPS IN EVENT-B  7
    7.4.1  The Initial Model  8
    7.4.2  The First Refinement  11
    7.4.3  The Second Refinement  14
    7.4.4  The Third Refinement  17
    7.4.5  The Fourth Refinement  18
7.5  EXAMPLE: SATELLITE DATA PROCESSING UNIT  20
    7.5.1  Description  20
    7.5.2  Instantiation of Generic Models  22
7.6  CONCLUSIONS  23
List of Figures

7.1 Event-B machine and context 6
7.2 The context Data0 9
7.3 The machine M0 9
7.4 The context Data2 15
7.5 The machine M2 16
7.6 The machine M3 19
CHAPTER 7

Formal Reasoning about Resilient CPS

Linas Laibinis
Aabo Akademi University, Turku, Finland

Elena Troubitsyna
Aabo Akademi University, Turku, Finland

CONTENTS

7.1 Introduction ................................................ 2
7.2 Cyber-Physical Systems ................................... 3
7.3 Background: Event-B ...................................... 5
7.4 Formal Development of a Resilient CPS in Event-B ...... 7
  7.4.1 The Initial Model .................................. 7
  7.4.2 The First Refinement ....................... 11
  7.4.3 The Second Refinement ..................... 14
  7.4.4 The Third Refinement ...................... 17
  7.4.5 The Fourth Refinement .................... 18
7.5 Example: Satellite Data Processing Unit .............. 20
  7.5.1 Description ........................................ 20
  7.5.2 Instantiation of Generic Models ............... 22
7.6 Conclusions ................................................ 23

Cyber-physical systems (CPS) are complex software intensive systems consisting of assemblies of heterogenous components. To guarantee resilience of CPS, i.e., to ensure that a system can adapt to changing operating conditions, CPS architectures should integrate the mechanisms enabling adaptation. In this paper, we employ Event-B to formally derive the reconfiguration mechanisms that allow the system to dynamically interconnect components to cope with failures. The proposed approach formalises relationships between system functionality,
component functional capabilities and failures. It serves as a basis for developing the system health monitoring infrastructure and facilitates rigorous construction of resilient systems.

7.1 INTRODUCTION

Cyber-Physical Systems (CPS) promise great economic and societal benefits in such domains as aerospace, production automation, transportation and healthcare. However, to unlock their full potential, rigorous and powerful methods for ensuring system trustworthiness need to be developed and employed.

CPS operate in continuous interaction with the physical world. They need to constantly monitor external operating conditions and react to them to provide the required services in a correct and safe way. At the same time, CPS should also continuously monitor their internal state and mitigate the effect of component failures on service provisioning. The latter task is usually handled by employing dynamic system reconfiguration.

In this paper, we demonstrate how to reason about dynamically reconfigurable CPS. Our goal is to explicitly define a link between the services that the system should deliver, functional capabilities of the components and their health. We demonstrate how to formalise the proposed approach to reconfiguration, i.e., the mechanisms that allow the components to dynamically interconnect and cooperatively perform the required services even despite the loss of certain functional capabilities by some of the components.

We rely on the Event-B formalism [1] to formally reason about dynamic reconfiguration. Event-B is a state-based approach that promotes correct-by-construction system development. Development starts from creating an abstract system specification that defines only the most essential system properties and behaviour. In a number of correctness-preserving steps, called refinements, the initial specification is transformed into a detailed system model. Each refinement step is accompanied by the proofs that guarantee that the externally observable system behaviour is preserved during such model transformation. The Rodin platform [11] provides an automated support for modelling and verification in Event-B.

The top-down approach to system development promoted by Event-B allows us to gradually unfold the system architecture and formally define the requirements that should be imposed on it to enable dynamic
system reconfiguration. As a result of our development, we derive a number of generic specification and refinement patterns, which represent the essential properties of reconfigurable architectures and define the coordination mechanisms for cooperative service provisioning.

The proposed generic approach explicitly defines the main concepts that should be analysed while developing a resilient CPS. It facilitates a structured derivation of a resilient system architecture by a gradual unfolding of architectural details and revealing the necessary interrelationships between service provisioning and the involved reconfiguration mechanisms. The approach is illustrated by a small example from the aerospace domain.

The chapter is structured as follows. In Section 7.2 we overview the characteristics and properties of systems that we interested in modelling and analysing. Section 7.3 briefly describes our formal framework – Event-B. In Section 7.4 we present our main contribution – formal generic development of a resilient CPS. Section 7.5 illustrates the proposed approach by a small example of a satellite data processing unit. Finally, in Section 7.6 we present some concluding remarks.

7.2 CYBER-PHYSICAL SYSTEMS

In this section we explicitly state our assumptions about the characteristics and properties of the systems that we are interested in modelling and analysing in this chapter. The focus of our investigation is on service provisioning by CPS in the presence of constantly changing operating conditions and fluid system configurations as well as how possible dynamic reconfiguration mechanisms of such systems can improve their resilience.

In general, in this chapter we consider systems that exhibit the following characteristics:

- Architecturally, the system consists of a number of different components, which may communicate and collaborate between each other in order to provide a common service;

- Components are heterogenous, i.e., they may have different functional capabilities. A specific functional capability is provided by a so called component functional block. In other words, we associate the component capabilities with a number of distinct functional blocks it is capable of executing;
• Each component contains its own collection of functional blocks of different types. On the other hand, functional blocks of the same type can reside on different components. Therefore, we assume here a certain degree of system redundancy with respect to provided functionalities, which, in turn, is the main source of system resilience;

• The system accepts a number of external requests to execute specific services and responds with either successful service execution or an error message;

• Each request is split within the system into the necessary functional block types it needs to be completed;

• The execution scenario for a particular service requests then may involve different components with available functional blocks of the required types.

• Additional component functional blocks may provide external or internal communication or other auxiliary services;

• Both component internal state and its environment can affect the availability of a specific component functional block; In other words, the functional block in question can be enabled or disabled depending on both the internal and external component operating states;

• A component functional block can fail. For our system this means that the functional block becomes permanently disabled;

• Each component can be in one of roles (sometimes called modes of operation). The component role is associated with a particular subset of functional blocks it should be currently able to perform and/or with particular duties within the service execution scenario;

• The collective roles of all the system components can be interpreted as the current configuration of the system, while the current role of a component and all the statuses of its functional blocks represent the configuration of this component;

• We assume that there is a pre-defined minimal condition for each role that a component or its current configuration (i.e., collective
statuses of the component functional blocks) should satisfy. If this condition is not satisfied, the component in question should change its role to more restrictive or degraded one;

- We also distinguish a so called primary role (i.e., leading or coordinating) which is responsible for orchestrating the overall service execution and also often serving as a frontend for the outside world, i.e., receiving service requests and sending system responses). It is assumed that the system typically has only one component in such primary role. If the component cannot carry on in the primary role, another component satisfying the minimal condition for the primary role can take over the assigned duties.

In Section 7.4 we will show how we can gradually formalise the concepts and characteristics of a system presented above. Before doing that, we briefly introduce the formal framework we rely on – Event-B.

7.3 BACKGROUND: EVENT-B

Event-B [1] is a state-based framework that promotes correct-by-construction approach to system development and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an abstract state machine [1]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important properties of the systems are defined in the model invariants. Usually, a machine has an accompanying component, called context, which includes user-defined sets, constants and their properties given as a list of model axioms.

A general form for Event-B models is given in Fig. 7.1. The machine is uniquely identified by its name $M$. The state variables, $v$, are declared in the Variables clause and initialised in the Init event. The variables are strongly typed by the constraining predicates $I$ given in the Invariants clause. The invariant clause might also contain other predicates defining properties (e.g., safety invariants) that should be preserved during system execution.

The dynamic behaviour of the system is defined by a set of atomic events. Generally, an event has the following form:

$$e \equiv \text{any } a \text{ where } G_e \text{ then } R_e \text{ end},$$
where \( e \) is the event’s name, \( a \) is the list of local variables, the *guard* \( G_e \) is a predicate over the local variables of the event and the state variables of the system. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. In Event-B, an assignment represents a corresponding next-state relation \( R_e \). Later on, using the concrete syntax in our Event-B models, we will rely on two kinds of assignment statements: deterministic ones, expressed in the standard form \( x := \text{some expression}(x, y) \), and non-deterministic ones, represented as \( x :| \text{some condition}(x, y, x') \). In the latter case, the state variable \( x \) gets non-deterministically updated by the value \( x' \) which may depend on the initial values of the variables \( x \) and \( y \).

The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

If an event does not have local variables, it can be described simply as:

\[
e \equiv \text{when } G_e \text{ then } R_e \text{ end.}
\]

If the event guard \( G_e \) is always true, the event syntax become \( e \equiv \text{begin } R_e \text{ end.} \)

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging a number of verification
conditions – proof obligations. For instance, to verify invariant preservation, we should prove the following logical formula:

\[ A(d, c), \ I(d, c, v), \ G_e(d, c, x, v), \ R_e(d, c, x, v, v') \vdash I(d, c, v'), \quad (INV) \]

where \( A \) are the model axioms, \( I \) are the model invariants, \( d \) and \( c \) are the model constants and sets respectively, \( x \) are the event’s local variables and \( v, v' \) are the variable values before and after event execution.

In turn, each refinement step generates additional proof obligations ensuring that the transformation is performed in a correctness-preserving way. In particular, to formally demonstrate refinement between the corresponding events of the abstract and concrete models, the guard strengthening and event simulation proof obligations must be discharged. The full definitions of all the proof obligations are given in [1].

The Rodin platform [11] provides an automated support for formal modelling and verification in Event-B. In particular, it automatically generates the required proof obligations and attempts to discharge them. The remaining unproven conditions can be dealt with by using the provided interactive provers.

### 7.4 FORMAL DEVELOPMENT OF A RESILIENT CPS IN EVENT-B

In this section we present a formal Event-B development that will gradually incorporate the essential concepts and relationships between them for the systems described above. The development is generic, i.e., the abstract sets, functions and relations are defined via their essential properties and thus can be interpreted as generic parameters of the whole development. As such, models represent a family of suitable systems. In Section 7.5 we show how such a generic development can be instantiated for a concrete system.

We present our development as the initial model of a CPS system and several its subsequent refinements. In each step, we introduce some new concepts and their properties or elaborate on the already introduced ones. We separately discuss both static and dynamic system aspects, represented in the context and machine components respectively. The developed models are rather big, thus in many cases we only highlight the most important changes or introduced features.
7.4.1 The Initial Model

In our initial model we abstract away from many internal details of a CPS, focusing instead on handling external requests for particular services, arriving from outside the system boundaries. The execution of such requests is usually split into smaller tasks, which in turn are handled by specific functional blocks of the system. For the moment, we do not care where such blocks reside physically, i.e., what particular components they belong to. Instead, we concentrate on modelling the connection between the execution of an arrived request and the dynamic availability of the required functional blocks.

In the context component of the initial model (presented in Fig. 7.2), we introduce necessary static data structures. The abstract set $FBLOCK$ models all the functional blocks residing within boundaries of our system. Each functional block belongs to a specific type, an element of the defined abstract set $FB\_TYPE$. In other words, there can be multiple blocks (instances) for the same functional block type, which is the main source for designing various resilience mechanisms for such a system. The defined function $FB\_type$ associates a given functional block with its type.

The system configuration in terms of its functional capabilities (i.e., availability of specific functional blocks) is very fluid. In the context, we model this by introducing an enumerated type $FB\_STATUS^1$, which contains three values: Enabled, signifying that a block is healthy and ready to accept tasks, Disabled, indicating that a block is temporarily unavailable due to some external or internal factors, and Failed, denoting that a block can be considered permanently (at least, with respect to the current request) unavailable.

All possible incoming requests are represented by the abstract set $REQUEST$. We assume that any such request can be decomposed into a set of the required functional blocks or, more precisely, functional block types. The introduced function $Req\_blocks$ associates a given request with such a set. Finally, outgoing system responses are abstractly modelled by the enumerated set $OUTPUT$, containing three elements $NIL$, $Success\_resp$, and $Failure\_resp$.

In the machine component (the structure of which is shown in Fig. 7.3), we describe the dynamics of our abstract system. Essentially, the machine models the arrival of new requests, splitting them

---

1 The pre-defined Event-B operator partition is used to defined an enumerated set with all the necessary axioms for its elements.
Formal Reasoning about Resilient CPS

**CONTEXT** Data0

**SETS** FBLOCK, FB_TYPE, FB_STATUS, REQUEST, OUTPUT

**CONSTANTS** FB_type, Enabled, Disabled, Failed, Req_blocks, NIL, Success_resp, Failure_resp

**AXIOMS**

- **axm1**: \( FB\_type \in FBLOCK \rightarrow FB\_TYPE \)
- **axm2**: partition\((FB\_STATUS, \{\text{Enabled}\}, \{\text{Disabled}\}, \{\text{Failed}\}) \)
- **axm3**: Req_blocks \in REQUEST \rightarrow P_1(FB\_TYPE)
- **axm4**: partition\((OUTPUT, \{\text{NIL}\}, \{\text{Success\_resp}\}, \{\text{Failure\_resp}\}) \)

**END**

Figure 7.2 The context Data0

**MACHINE** M0

**SEES** Data0

**VARIABLES** bstatus, rem_blocks, output

**INVARIANT**

- \( bstatus \in FBLOCK \rightarrow FB\_STATUS \)
- \( rem\_blocks \in P(FB\_TYPE) \)
- \( output \in OUTPUT \)
- \( output = Success\_resp \Rightarrow \text{rem\_blocks} = \emptyset \)
- \( output = Failure\_resp \Rightarrow \text{rem\_blocks} \neq \emptyset \)

**INITIALISATION**

- \( bstatus \in FBLOCK \rightarrow \{\text{Enabled, Disabled}\} \)
- \( \text{rem\_blocks, output := } \emptyset, Success\_resp \)

**EVENTS**

- Start = ...
- Monitor = ...
- Execute = ...
- Finish(success) = ...
- Finish(failure) = ...

**END**

Figure 7.3 The machine M0

into the required functional block types, executing these blocks, if they are available, and finally returning a success or failure response. In parallel, the system regularly monitors the health of its functional blocks and updates their status.

The three machine variables – \( bstatus, \text{rem\_blocks}, \text{output} \) – represent the current status of the system blocks, the remaining block types to be executed for the current request\(^2\), and the outgoing sys-

\(^2\)We assume here that a functional block of a particular type should be executed only once for a request.
tem response respectively. Initially, all the system blocks are non-failed, which modelled as the following non-deterministic initialisation statement:

\[
\text{bstatus} : | \text{bstatus}' \in \text{FBLOCK} \rightarrow \{\text{Enabled, Disabled}\}.
\]

In the model invariants, the non-NIL system response is directly associated with success or failure of request execution, which in turn is formulated as the condition on the remaining blocks, e.g.,

\[
\text{output} = \text{Failure}_\text{resp} \Rightarrow \text{rem blocks} \neq \emptyset.
\]

Next we briefly describe some machine events. The event \textit{Start} models an arrival of a new request and splits it into a set of functional block types to be executed. The context function \textit{Req blocks} does the splitting for the received request.

\begin{verbatim}
EVENT Start
  ANY new_request
  WHERE
    grd1 : new_request \in REQUEST
    grd2 : output \neq NIL
  THEN
    act1 : rem_blocks := Req_blocks(new_request)
    act2 : output := NIL
  END
\end{verbatim}

The event \textit{Monitor} regularly monitors the health status of the system functional blocks and updates the variable \textit{bstatus} accordingly. The event non-deterministic assignment allows pretty much arbitrary change of the block status, with the only restriction that a functional block, once failed, stays failed. At this level, there are no guard conditions for the event so it can happen at any time.

\begin{verbatim}
EVENT Monitor
  BEGIN
    act1 : bstatus | bstatus' \in FBLOCK \rightarrow FB_STATUS ∧
            \forall bb. bstatus(bb) = Failed ⇒ bstatus'(bb) = Failed
  END
\end{verbatim}

The event \textit{Execute} models successful execution of one block type belonging to the variable \textit{rem blocks}. The block should be enabled to perform the task. As a result of event execution, the finished block type is removed from \textit{rem blocks}. 
**EVENT Execute**

```plaintext
ANY block, btype
WHERE
  grd1 : block ∈ FBLOCK
  grd2 : btype ∈ FB_TYPE
  grd3 : btype ∈ rem_blocks
  grd4 : bstatus(block) = Enabled
THEN
  act1 : rem_blocks := rem_blocks\{btype}\}
END
```

The remaining two events respectively model success and failure responses of the system. The success response is returned when `rem_blocks` becomes empty, while the failure one is returned if all the functional blocks are failed for some remaining block type, thus making successful service completion impossible. An alternative or additional way can be used to associate failure responses with some middleware timeouts, e.g., indicating that some required blocks were disabled for too long.

### 7.4.2 The First Refinement

In the first refinement step we introduce the system components and partition the functional blocks among them. Moreover, we explicitly model both (collective) component internal state and component environment (external) state. The monitored states are used to determine the status of specific functional blocks, which in turn affects their availability for execution of the remaining service tasks. We also enforce a specific cyclic behaviour within a component, when the functional block execution step always comes immediately after the component monitoring step.

In the context part of the refined model (presented below), we introduce three additional abstract sets – `COMPONENT`, `CSTATE`, `ESTATE` – standing for all possible components, (collective) component states and environment states respectively. The function `Block_comp` associates each block with one of the system components. Since it is often needed to reason about the functional blocks of a particular component, we also introduce the opposite function `Comp_blocks`, which is required to be the exact inverse of `Block_comp` in the axiom `axm3`. Finally, we require that the updated status of a functional block should depend on the latest monitored internal and external component states by defining the abstract function `Block_monitor`. 
In the machine component (the structure of which is shown below), we introduce model variables, $estate$ and $cstate$, for storing the current values of the internal state and external component states respectively. The array variable $monit\_flag$ is added to enforce the specific execution order within each component, i.e., monitoring, followed by task execution. Finally, the functional block status is now partitioned among the system components, each component storing only statuses of the functional blocks belonging to it. This is achieved by data refinement, where the abstract variable $bstatus$ is now replaced by the array variable $comp\_bstatus$, separately keeping the status information for every component. Since components are heterogeneous and may contain different functional blocks, the resulting status information is constrained by the invariant:

$$\forall cc. \text{dom}(comp\_bstatus(cc)) = \text{Comp\_blocks}(cc)$$

MACHINE M1
SEES Data1
REFINES M0
VARIABLES rem\_blocks, output, estate, cstate, comp\_bstatus, monit\_flag
INVARIANT
$estate \in \text{COMPONENT} \rightarrow \text{ESTATE}$
$cstate \in \text{COMPONENT} \rightarrow \text{CSTATE}$
$comp\_bstatus \in \text{COMPONENT} \rightarrow (\text{FBLOCK} \rightarrow \text{FB\_STATUS})$
$\forall cc. \text{dom}(comp\_bstatus(cc)) = \text{Comp\_blocks}(cc)$
$\forall bb. bstatus(bb) = comp\_bstatus(\text{Block\_comp}(bb))(bb)$
$monit\_flag \in \text{COMPONENT} \rightarrow \text{BOOL}$
INITIALISATION
...
EVENTS
...
END
The introduced changes mostly affect two model events – `Monitor` and `Execute`. Both these events are now distributed – they are defined locally for each system component. The event parameter `cc` defines a specific component the event is executed for.

The event `Monitor` (presented above) relies on the evaluation procedure encoded in the context function `Block_monitor`, which is applied to the observed values of the internal and external state of the component `cc` – `new_cstate` and `new_estate`. The evaluation is done only for the blocks belonging to the component, defined by `Comp_blocks(cc)`. The blocks that were detected as failed earlier remain as such.

Like `Monitor`, the event `Execute` (shown below) now executes a functional block for a particular component, defined by the first parameter `cc`. However, the execution is allowed only after the monitoring step was executed, which indicated by the condition `monit_flag(cc) = TRUE`. As before, the result of event execution is removal of the completed functional block type from the variable `rem_blocks`.

Despite introduction of system component and distribution of functional blocks among them, the information about the progress of service execution (stored in the variable `rem_blocks`) and overall service coordination remains global (centralised). We address this issue in the subsequent refinement steps, where we model component roles and the internal communication between components.
7.4.3 The Second Refinement

In the second refinement step, we introduce the notion of component roles. The component role is associated with a particular subset of functional blocks it should be able to perform currently and/or with particular duties within the service execution scenario. The examples of such roles are Manager, Standby, Master, Spare, Location, Load Balancer, Frontend etc. Sometimes roles can be also to signify the current operational mode of a component, especially when a component is forced to take a degraded role because of failures of its functional blocks.

In the context part of the refined model (see Fig.7.4), we introduce the abstract set \( ROLE \) for all possible component roles. The abstract function \( Role\_blocks \) associates a given role with a number of functional block types, while the function \( Comp\_roles \) relates each component with a number of the roles it can take on. These two functions are constrained together by the axiom \( axm5 \) stating that, for each block of a component, there is a role that the component is able perform, which is associated with the type of that functional block.

As its name suggests, the abstract function \( min\_role\_cond \) defines the minimal condition the component should satisfy while taking on a particular role. For a given role, the function takes the component configuration, i.e., its status function of the type \( FBLOCK \rightarrow FB\_STATUS \), and returns a boolean value indicating whether this configuration is acceptable.

Finally, the function \( Init\_config \) determines the initial system configuration, i.e., all the initial roles for the system components. Ob-
Formal Reasoning about Resilient CPS  ■  15

CONTEXT  Data2
EXTENDS  Data1
SETS  ROLES
CONSTANTS  Role_blocks, Comp_roles, Init_config, min_role_cond
AXIOMS
axm1:  Role_blocks ∈ ROLE → \( \mathbb{P}_1(FB\_TYPE) \)
axm2:  Comp_roles ∈ COMPONENT → \( \mathbb{P}_1(ROLE) \)
axm3:  Init_config ∈ COMPONENT → ROLE
axm4:  min_role_cond ∈  
        ROLE → ((FBLOCK → FB\_STATUS) → BOOL)
axm5:  ∀cc, bb.  bb ∈ Comp_blocks(cc)  ⇒  
        \( {\exists rr. \, rr \in \text{Comp\_roles}(cc) \land FB\_type(bb) \in \text{Role\_blocks}(rr)} \)
axm6:  ∀cc.  cc ∈ COMPONENT  ⇒  
        Init_config(cc) ∈ Comp\_roles(cc)
axm7:  ∀rr, bb.  bb ∈ dom(dom(min\_role\_cond(rr)))  ⇒  
        FB\_type(bb) ∈ Role\_blocks(rr)
END

Figure 7.4  The context Data2

viously, the configuration should be achievable, i.e., the components should be able to take these roles as formulated by the axiom axm6.

In the machine component (see Fig.7.5), we add a new variable, curr_role, to store the current role of each component. We also restrict the status update of component functional blocks to reflect the expectation that, if the component is in a particular role, it can employ only those functional blocks associated with that role. Consequently, all the other functional blocks can be either disabled or failed. This property is added as an additional invariant and enforced by the initialisation and the refined version of the Monitor event.

A component may change its role. Sometimes it is forced to do so after monitoring its internal state. In other cases it can be asked to take a more active role in orchestrating the overall service. In both cases, the component should satisfy the minimal role condition for such a change. The role changing functionality is added as a new event, Change_role, shown below. The guard grd5 of this event checks that the current component configuration (all its block statuses) is acceptable for taking new_role. The component block status is also updated to reflect the necessary changes.
MACHINE M2
SEES Data2
REFINES M1
VARIABLES rem_blocks, output, estate, cstate, comp_bstatus, monit_flag, curr_role

INVARIANT
...
curr_role ∈ COMPONENT → ROLE
∀ cc, cc ∈ COMPONENT ⇒ curr_role(cc) ∈ Comp_roles(cc)
∀ cc, bb.
   bb ∈ Comp_blocks(cc) ∧ FB_type(bb) ∉ Role_blocks(curr_role(cc))
   ⇒ comp_bstatus(cc)(bb) ∈ {Disabled, Failed}

INITIALISATION
...
comp_bstatus : ...
   ∧ (FB_type(bb) ∉ Role_blocks(curr_role(cc)) ⇒
       comp_bstatus'(cc)(bb) ∈ {Disabled, Failed})
curr_role := Init_config

EVENTS
...
Change_role = ...

END

Figure 7.5 The machine M2

EVENT Change_role
ANY cc, new_role, new_bstatus
WHERE
   grd1 : cc ∈ COMPONENT
   grd2 : cc ∈ ROLE
   grd3 : new_role ∈ Comp_roles(cc)
   grd4 : new_role ∉ curr_role(cc)
   grd5 : min_role_cond(new_role)(comp_bstatus(cc)) = TRUE
   grd6 : new_bstatus ∈ FBLOCK → FB_STATUS
   grd7 : dom(new_bstatus) = Comp_blocks(cc)
   grd8 : ∀ bb, bb ∈ dom(new_bstatus) ∧ comp_bstatus(cc)(bb) = Failed ⇒
      new_bstatus(bb) = Failed
   grd9 : ∀ cc, bb, bb ∈ Comp_blocks(cc) ∧ FB_type(bb) ∉ Role_blocks(new_role) ⇒
      new_bstatus(bb) ∈ {Disabled, Failed}
   grd10 : ∀ cc, bb, bb ∈ Comp_blocks(cc) ∧ FB_type(bb) ∈ Role_blocks(new_role) ⇒
      new_bstatus(bb) = comp_bstatus(cc)(bb)
THEN
   act1 : comp_bstatus(cc) := new_bstatus
   act2 : curr_role(cc) := new_role
END
7.4.4 The Third Refinement

In the third refinement step, we elaborate on the concept of component roles. Even if the roles give us a nice abstraction to reason about different subsets of component functionalities or operational modes, we need to designate some role(s) a special status to ensure the overall orchestration and coordination of service execution. We do so by introducing the notion of primary (i.e., leading or coordinating) role. This is the role responsible for orchestrating the overall service execution and also often serving as a frontend for the outside world, i.e., receiving service requests and sending system responses. For simplicity, in this chapter we assume that there is a single component in this role.

Moreover, we also need to introduce the mechanisms for internal communication, when a component in the primary role delegates some remaining tasks to non-primary (e.g., standby, spare) components as well as receives some intermediate results from them.

In the context component of the refined model (presented above), we explicitly designate one role as the primary one by introducing the constant `Primary_role`. We also add a new abstract set, `INT_REQUEST`, for modelling all the internal requests. We explicitly partition this set to contain `INT_NIL`, the nil value indicating the absence of such requests, `FromPrimary`, the subset of requests coming from the primary role component, and `ToPrimary`, a subset of the requests received by the primary component from the non-primary components.

Since the main purpose of the internal communication is to facili-
tate coordinated execution of the requested service, we assume that an internal request contains the remaining tasks (functional block types) that are delegated by the primary role to other components to be executed. The returning requests to the primary role components then contain the tasks still remaining to be completed. To extract the task information from an internal request, we introduce the abstract function \( \text{IntReq} \_\text{blocks} \) of the type \( \text{INT} \_\text{REQUEST} \rightarrow P_1(FB \_\text{TYPE}) \).

In the machine component (shown on Fig. 7.6), we explicitly store the component with the primary role in the variable \( \text{primary} \_\text{comp} \) and require that there should be no more than one such component in the system as an invariant property. Moreover, we introduce the variable \( \text{internal} \_\text{bus} \) to model the bus for internal operations and define four new events for sending and receiving internal requests via this bus, distinguishing between the components in the primary and non-primary roles. Finally, the variable \( \text{execution} \_\text{token} \) is added to ensure that only one component proceeds with the service execution.

After the initialisation, the primary component gets the token. At some point of service execution, it can delegate the remaining task(s), via sending an internal request, to some non-primary component, which then receives the token. When such component can proceed no further, it returns the token by sending an internal request to the primary component. The service execution ends when the primary component sends an external response with either a success or failure message.

7.4.5 The Fourth Refinement

The coordination and communication scheme described in the previous subsection is a very “liberal” one in a sense that the primary component just puts an outgoing internal request, essentially “hoping” that some other component would be able to proceed with the service execution. We assume that if such a request is not taken for processing for a pre-defined period of time (because, e.g., none of the remaining components have the required non-failed functional blocks), the corresponding timeout signal is issued by the system middleware, consequently leading to the respective failure message by the primary component to the outside world.

In the fourth refinement, we introduce some discipline into component coordination, when the primary component explicitly targets the component it delegates the service execution tasks to. To be able to do that, some information about the internal configuration (current
Formal Reasoning about Resilient CPS

Figure 7.6 The machine M3

statuses of the component functional blocks) of the non-primary components should be accessible to the prime component. In this refined model, we add additional communication by which the non-primary components must inform their primary counterpart once they detect that any of their functional components has failed. With such extra information, the primary component can decide which of the components is most suitable to handle the remaining service tasks.

In the case of failures that may force the primary component to relinquish its primary role, the collected information is transferred to a new primary component, if possible.

For brevity, we just sketch the main highlights of this refinement step. As a refinement result, the previous model is elaborated by:

- defining an additional field in the existing messages for service
orchestration, containing the target component (in the context component);

• adding a new type of internal requests for failure messages (in the context component);

• constraining the existing events for receiving internal requests by checking the request target (in the machine component);

• introducing new events for sending the information about the failed blocks to the primary component (in the machine component);

• introducing a new event for the primary component to decide on the most suitable component to delegate the remaining tasks (in the machine component);

• introducing a new event for transferring the collected information to a new primary component (in the machine component).

There are many aspects of a resilient CPS that are still omitted in the obtained formal models. Some of them can be introduced in the subsequent refinement steps. For instance, the actual service execution scenario is very simplistic, and many details about the order, branching of specific service tasks can be introduced. No parallel execution (both within the execution of a single service request and handling multiple service requests at the same time) is considered for the moment. We restrict ourself to modelling a single primary component and therefore just one subordination level between the system components. We hope to address some of these issues in our future work on the subject.

7.5 EXAMPLE: SATELLITE DATA PROCESSING UNIT

7.5.1 Description

Our work is partially inspired by the actual solution to circumvent double failure that occurred in a currently operational on-board satellite system. The architecture of this system is similar to the Data Processing Unit (DPU) – a subsystem of European Space Agency mission BepiColombo [3] that is under development now. The main goal of the mission is to carry various scientific measures to explore the planet Mercury. DPU – an important part of the Mercury Planetary
Orbiter – consists of four independent components (computers) responsible for receiving and processing data from four sensor units: SIXS-X (X-ray spectrometer), SIXS-P (particle spectrometer), MIXS-T (telescope) and MIXS-C (collimator).

The behaviour of DPU is managed by Telecommands (TCs) received from the spacecraft. Processing of each TC results in producing a housekeeping report, switch to some mode, or initiate/continue production of scientific data. Correspondingly, TM would contain either housekeeping data, an acknowledgement of the requested mode transition, or scientific data. Each component acquires fresh scientific data from the corresponding sensor unit (SIXS-X, SIXS-P, MIXS-T and MIXS-C), preprocesses and then makes it available to DPU that eventually forms the entire TM package.

To cope with errors that may occur during the satellite mission, the redundant DPUs are usually used, i.e., in case of failure of any component of the active DPU the entire TM package cannot be formed and the spare DPU is activated. Let us consider a duplicated system that consists of two identical DPUs – \(DPU_A\) and \(DPU_B\). As was explained above, each DPU contains four components responsible for controlling the corresponding sensors.

Traditionally, the satellite systems are designed to implement the following simple redundancy scheme. Initially \(DPU_A\) is active, while \(DPU_B\) is a cold spare. \(DPU_A\) allocates tasks on its components to achieve the system goal – processing of a TC and producing a TM. When some lower-level component of \(DPU_A\) fails, \(DPU_B\) is activated to achieve the goal. Failure of \(DPU_B\) results in failure of the overall system. However, let us observe that even though none of the DPUs can accomplish the overall goal on its own, it might be the case that the components that remained operational can perform the entire set of tasks required to reach the goal. This observation allows us to define the following dynamic reconfiguration strategy.

Initially \(DPU_A\) is active and is assigned to reach the goal. If any of its components fails to execute one of four scientific tasks (let it be \(task_j\), the spare \(DPU_B\) is activated and \(DPU_A\) is deactivated. \(DPU_B\) performs \(task_j\) and the consecutive tasks required to reach the goal. It becomes responsible for achieving the overall goal until some of its component fail. To remain operational, the system then performs dynamic reconfiguration. Namely, it reactivates \(DPU_A\) and tries to assign the failed task to the corresponding component of \(DPU_A\). If such a compo-
nent is operational then $DPU_A$ continues to execute the consequential tasks until it encounters a failed component. Then the control is passed to $DPU_B$ again. Obviously, the overall system stays operational until two identical components of both DPUs have failed.

7.5.2 Instantiation of Generic Models

It is rather easy to demonstrate that the satellite system described above is a special instance of a generic CPS system we modelled in this chapter. To do so, we need to show a valid instantiation of our generic parameters with concrete values pertaining to the described system.

We start by instantiating the abstract set $COMPONENTS$ with two system components $DPU_A$ and $DPU_B$:

$$COMPONENTS = \{DPU_A, DPU_B\}.$$ 

We can distinguish four essential functional blocks – SIXS-X, SIXS-P, MIXS-T and MIXS-C. In addition, we need the functional blocks that responsible for the external and internal communication – EXT-IO and INT-IO. As a result the abstract set $FB_TYPE$ is instantiated as

$$FB_TYPE = \{SIXS_X, SIXSP, MIXST, MIXSC, EXT\_IO, INT\_IO\}.$$ 

The two components are functionally identical, thus we define the data structures $FBLOCKS$, $FB_{type}$, and $Block_{comp}$ as follows:

$$FBLOCKS = \{SIXS_X_A, SIXSP_A, MIXST_A, MIXSC_A, EXT\_IO_A, INT\_IO_A, SIXS_X_B, SIXSP_B, MIXST_B, MIXSC_B, EXT\_IO_B, INT\_IO_B\},$$

$$FB_{type} = \{SIXS_X_A \mapsto SIXS_X, SIXSP_A \mapsto SIXSP, MIXST_A \mapsto MIXST, \ldots\},$$

$$Block_{comp} = \{SIXS_X_A \mapsto DPU_A, SIXS_X_B \mapsto DPU_B, MIXST_A \mapsto DPU_A, \ldots\}.$$ 

We can distinguish three different component roles – Master, Cold_spare, and Hot_spare. The initial system configuration is as follows:

$$Init_{config} = \{DPU_A \mapsto Master, DPU_B \mapsto Cold_spare\}.$$
The defined roles are associated with the following functional blocks:

\[
\text{Role_blocks} = \\
\{ \text{Master} \mapsto \{SIXSX, SIXSP, MIXST, MIXSC, EXT.IO, INT.IO\}, \\
\text{Cold_spare} \mapsto \{INT.IO\}, \\
\text{Hot_spare} \mapsto \{SIXSX, SIXSP, MIXST, MIXSC, EXT.IO, INT.IO\} \}.
\]

While the Master and Hot_spare roles have identical functionalities, the Cold_spare has only the internal communication block, which may be used to activate it and change its current role into either the Master or Hot_spare role.

The described in the previous section dynamic reconfiguration mechanisms can be easily shown as special cases of the formalised ones. The minimal role condition for the Master role is defined by requiring that its EXT.IO block is not failed. If this condition is not satisfied, some other component should take over by switching to the Master role, if possible.

### 7.6 CONCLUSIONS

Our work presented in this chapter combines formal modelling of CPS with enhancing the resilience of such systems by integrating dynamic system reconfiguration mechanisms. The main focus is on gradually describing the involved concepts and their interrelationships as well as unfolding the system architecture with the integrated dynamic reconfiguration mechanisms to enhance system resilience.

There is ongoing very active research in both these areas. DESTECS project [5] studies the problem of engineering of resilient CPS. The research conducted within the project proposes an approach to the model-based design through co-simulation of discrete-event models in the Vienna Development Method (VDM) and continuous-time models in 20-sim. These models are coupled by a co-simulation tool that coordinates execution of the developed models in their respective simulators. The resulting models can be also augmented with descriptions of potential failures and fault tolerance mechanisms [6, 7, 8].

The problem of formal verification and validation of CPS has been also addressed in the ADVANCE project [2]. The main focus of the
proposed methodologies supports the construction of CPS and augmenting formal modelling and verification with simulation and testing. The resulting proposed simulation-based approach combines the Event-B development and co-simulation with tool independent physical components via the FMI interface [12]. The ongoing EU H2020 project Into-CPS [9] aims at further advancing these results by exploring the idea of creating a multi-objective formal modelling framework and an integrated tool chain for model-based design of cyber-physical systems. In contrast, in our work we focused only on modelling the architectural aspects of CPS and did not consider separately the modelling of the physical processes. Essentially, we assume that the system will be able monitor the changes in both environment and internal state and focus on designing most flexible and resilient mechanisms to react on these changes.

In [10], Inverardi et al. investigate system adaptation based on the assume-guarantee concept. In particular, they propose a framework that allows the developers to efficiently define under which conditions adaptation can be performed by preserving the desired system invariant properties. The framework also allows the designers to split the system into parts that can be substituted. In order to guarantee the correctness of adaptation, the special conditions are formulated and have to be proven at run-time. In our approach, the reconfiguration strategies are already defined at development phase and are incorporated into the system architecture. In the case of failures or changes, the system is able to reconfigure by changing interdependencies among components, as well as between the components and the system goals (e.g., providing the required system services).

An extensive body of research investigates the quality of service characteristics of dynamically reconfigurable service-oriented systems. Among the most prominent works in this area is the approach proposed by Calinescu [4]. It aims at defining the optimal configuration with respect to quality of service by assessing the quality of service attributes of various service components that are available at run-time.

The idea of achieving system dependability via reconfiguration is described in [15]. The authors present a method for constructing systems where general properties of reconfiguration can be ensured via formal proofs. The idea of the proposed approach is to introduce a formal definition of reconfiguration as well as a set of hight-level properties. Then a system architecture is introduced which guarantees those reconfiguration properties. In our research, we follow the same idea to
enable the system to be reconfigurable already at a high-level system specification.

The problem of system reconfiguration and its connection to predictable dynamic resilience is presented in [13, 14]. The proposed architectural framework – MetaSelf – is suitable for development of dynamically resilient systems using a specific architectural model. The main idea of the approach is to separate the functional and non-functional descriptions of the system components, formulate necessary resilience policies and reason about system reconfiguration based on these policies. The key feature of the proposed solution is the need for metadata – information about system components – sufficient to enable decision-making about dynamic system reconfiguration. The metadata are used to guide the system reconfiguration with respect to the given reconfiguration policies.

In general, in our work we see reconfigurability as an ability of components to redistribute their responsibilities (roles) to ensure continued system operation and service provision. Therefore, the proposed reconfiguration mechanisms are built by constantly monitoring both internal and external component state and then reacting by changing links and associations between components.


