Contents

SECTION I  This is a Part

CHAPTER 6 • Formalising Goal-Oriented Development of Resilient CPS 3

Inna Pereverzeva and Elena Troubitsyna

6.1 INTRODUCTION 4
6.2 GOALS AND ARCHITECTURE OF A RESILIENT CPS 5
  6.2.1 Goal-Oriented Development and Reconfiguration 6
  6.2.2 Generic Requirements for Goal-Oriented Resilient Systems 6
  6.2.3 General Approach of Developing Distributed Systems in Event-B 8
6.3 MODELLING AND REFINEMENT IN EVENT-B 9
6.4 DERIVING RESILIENT ARCHITECTURE 12
  6.4.1 Functional Decomposition by Refinement 13
  6.4.2 Second Refinement 14
  6.4.3 Representing Manager’s Local Knowledge 20
  6.4.4 Explicit Model of Component Failures 22
6.5 MODELLING OF COMMUNICATION 23
  6.5.1 Communication Between Managers 23
  6.5.2 Communication Between Managers and Workers 25
6.6 TOWARDS MODEL DECOMPOSITION 25
6.7 RELATED WORK AND CONCLUSIONS 27
  6.7.1 Related Work 27
  6.7.2 Conclusions 29
List of Figures

6.1 Base station interface component 26
6.2 Worker interface component 27
I

This is a Part
CHAPTER 6

Formalising Goal-Oriented Development of Resilient CPS

Inna Pereverzeva
Aabo Akademi University, Turku, Finland

Elena Troubitsyna
Aabo Akademi University, Turku, Finland

CONTENTS

6.1 Introduction ................................................ 4
6.2 Goals and Architecture of a Resilient CPS ............ 5
   6.2.1 Goal-Oriented Development and Reconfiguration .. 6
   6.2.2 Generic Requirements for Goal-Oriented Resilient
       Systems ............................................. 6
   6.2.3 General Approach of Developing Distributed
       Systems in Event-B ............................... 8
6.3 Modelling and Refinement in Event-B .................. 9
6.4 Deriving Resilient Architecture ........................ 12
   6.4.1 Functional Decomposition by Refinement .......... 13
   6.4.2 Second Refinement ................................ 14
   6.4.3 Representing Manager’s Local Knowledge .......... 20
   6.4.4 Explicit Model of Component Failures ............ 22
6.5 Modelling of Communication .............................. 23
   6.5.1 Communication Between Managers ................. 23
   6.5.2 Communication Between Managers and Workers .. 25
6.6 Towards Model Decomposition .......................... 25
6.7 Related Work and Conclusions .......................... 27
   6.7.1 Related Work ...................................... 27
   6.7.2 Conclusions ....................................... 29
DEVELOPMENT of resilient CPS, i.e., the systems that can deliver trustworthy services despite changes, is a complex engineering task. Resilience can be considered as an ability of a system to achieve its goals despite negative changes, such as failures of components, or as a capability to achieve the desired goals more efficiently, for instance by increasing component utilisation. In this work, we present a formal goal-oriented approach to development of resilient CPS in Event-B. We define the main abstractions required for reasoning about system goals and introduce the specification patterns explicitly defining architectural reconfiguration mechanisms allowing the system to achieve resilience. We demonstrate how formal goal-oriented development in Event-B facilitates structuring of component interdependencies and derivation of the overall distributed architecture of CPS.

6.1 INTRODUCTION

Cyber-Physical Systems (CPS) are examples of complex distributed systems that carry on their functions in constant interaction with the physical world [20]. Due to continuously changing operating conditions, achieving resilience – an ability of a system to provide trustworthy services despite changes [19], becomes one of the main objectives of CPS development.

Essentially, resilience means that the system can absorb the effect of changes, e.g., tolerate a failure of a component or scale to handle an increased load. Resilience is typically implemented via reconfiguration. To cope with a component failure the system should reconfigure, i.e., reallocate responsibilities for providing different functionality from failed components to the healthy ones. To cope with an increased load, either caused by increased rate of external service requests or due to degradation of internal conditions, the system should optimise resource allocation, e.g., by employing idle components.

In this work, we adopt a goal-oriented style of resilient CPS development. Goals [39, 40] are the functional and non-functional objectives of a system. In software engineering, goals have been recognised as useful primitives for capturing system requirements [39, 40]. In particular, resilience can be seen as a property that allows the system to progress towards achieving its goals [26]. The reasoning in terms of goals promotes structuring the top-down system design.

We formalise goal-oriented development of CPS in Event-B. Event-
B\textsuperscript{1} is a state-based framework that relies on abstract modelling, refinement, and theorem proving to create and verify specifications of complex systems. Development in Event-B starts with an abstract model that captures only the most essential properties and behaviour of the system. In a number of correctness-preserving steps – refinements – the abstract model is transformed into a detailed specification of the overall system. The resultant specification is decomposed into the specifications of the independent subsystems that through their interactions are guaranteed to preserve the system-level properties. The Rodin platform\textsuperscript{29} supports development and verification in Event-B.

In this work, we demonstrate how formal Event-B development of CPS in a goal-oriented style facilitates reasoning about resilience. In our development, we start by decomposing system-level goals into subgoals (tasks) and establish a correspondence between tasks and components’ functional capabilities. We introduce a hierarchical structure of the components that essentially corresponds to the distributed supervisory control over the lower level service-provisioning components. Then we explicitly define the communication between the system components. Our modelling allows us to systematically derive complex reconfiguration mechanisms ensuring that the system correctly adapts to changing operating conditions.

We argue that the goal-oriented style employed in our formal development facilitates modelling of component interactions, structuring of inter-component communication and derivation of complex reconfiguration mechanisms.

The chapter is structured as follows. In Section 6.2 we introduce the main concepts of our goal-oriented development style, discuss the link between goal reachability and resilience, as well as define the basic architectural and functional properties of a resilient CPS. In Section 6.3 we describe our formal modelling framework – Event-B. In Section 6.4 and Section 6.5 we present our formal development in a goal-oriented style. We define generic specification patterns addressing goal decomposition, modelling reconfiguration and component interdependencies. In Section 6.6 we describe the last refinement step that allows us to arrive at a decentralised model by decomposition. Finally, in Section 6.7 we review the related work and discuss the achieved results.

6.2 GOALS AND ARCHITECTURE OF A RESILIENT CPS
6.2.1 Goal-Oriented Development and Reconfiguration

In this paper, we use the notion of a goal as a basis for reasoning about resilient CPS. Goals are the functional and non-functional objectives that the system should achieve [39, 40]. Reasoning in terms of goals facilitates structuring requirements and supports the top-down system design. In particular, resilience can be seen as a property that allows the system to progress towards achieving its goals [26].

Usually, a system has different types of interdependent goals. Thus, goals can be structured, e.g., to form a hierarchy. Generally, they can be formulated at different levels of abstraction. The process of goal detailisation (i.e., decomposition into subgoals) is performed until a certain level of granularity is reached, i.e., when a subgoal can be assigned to and consecutively realised by the system components [39, 40].

In order to achieve their individual or common goals, system components interact with each other. Component interactions might be simple ones, e.g., information exchange, or complex, e.g., involving requests for service provisioning from one component to another. The system components interact with other and behave cooperatively to implement system-level goals. Component interactions are achieved by communication. Communication allows the individual components to share their local information with other components to facilitate goal achievement.

Since in this work we focus on studying the functional aspect of resilience of CPS behaviour, we should explicitly represent off-nominal situations such as component or communication failures and design mechanisms allowing the system to cope with them. As a result of failure, components usually lose an ability to perform their predefined tasks. Therefore, to ensure resilience, the system needs to reconfigure and guarantee that the goals remain reachable. The reconfiguration is based on reallocation of responsibilities between components either to ensure that the healthy components can substitute the failed ones or to enable more efficient utilisation of components. In both cases, the system components should collaborate to ensure system resilience.

6.2.2 Generic Requirements for Goal-Oriented Resilient Systems

The goal-oriented framework enables reasoning about the system behaviour at different levels of abstraction. At the same time, the goal decomposition process facilitates incremental unfolding of the system architecture. It also helps us to build a hierarchy of components ac-
cording to their responsibilities in achieving goals. Moreover, the goal-oriented framework allows us to formulate reconfigurability as an ability of components to redistribute their responsibilities to ensure goal reachability.

In our formal goal-oriented development, we rely on the following assumptions about system architecture and behaviour:

(PR1) There is the main system goal that the system aims at accomplishing during its execution.

(PR2) The main system goal can be decomposed into a subset of corresponding subgoals (tasks) that can be executed by the system components.

(PR3) The system consists of a number of autonomic software components that are organised hierarchically, i.e., one component may be a manager of one or a group of other components called workers;

(PR4) The manager components coordinate the activities related to task achievement; they can be viewed as a distributed implementation of the supervisory control;

(PR5) The worker components perform activities to achieve the task.

(PR6) Every task is associated with a specific manager that it supervises its execution;

(PR7) A manager component can also be associated with a number of worker components that it is supervising at the moment.

(PR8) The manager components interact with the corresponding associated workers in order to assign tasks to them and receive the confirmation about task completion.

(PR9) A worker can not perform more than one task simultaneously.

(PR10) The manager components interact with other managers in order to distribute their local information about the statuses of the tasks assigned to them;

(PR11) The system components are unreliable, i.e., they might fail during system execution;
(PR12) In the case of failures of components, the system should, if possible, dynamically reconfigure itself to achieve the main system goal;

(PR12.1) In case of a worker failure during a task execution, its associated manager reassigns the task of the failed worker component to another available worker.

(PR12.2) In case of a manager failure, other available manager component becomes responsible for the tasks and the workers of the failed one.

(PR12.3) The system can also reconfigure to achieve some of its goals more efficiently e.g., by means of deploying the idle components of both types.

The aim of our modelling is to derive the distributed architecture of a CPS described above. We demonstrate how to formally define the system goal and, in a stepwise manner, derive a detailed specification of the system architecture. While refining the system specification, we gradually introduce a representation of the main elements of the architecture – managers and workers – as well as their communication. Moreover, we explicitly model failures and define the reconfiguration mechanisms for coping with them. We rely on the properties (PR1)–(PR12) given above to derive a distributed architecture of a resilient CPS.

6.2.3 General Approach of Developing Distributed Systems in Event-B

To facilitate the development of a distributed architecture in Event-B, we define a set of specification and refinement patterns that reflect the main concepts of the goal-oriented development. Namely, we start by abstractly defining system goals, then perform goal decomposition by refinement. We define and prove the relevant gluing invariants establishing a formal relationship between goals and the corresponding subgoals. Next we introduce a representation of system components and the required reconfiguration mechanisms to ensure that the system progresses towards achieving its goals.

We consider dynamic reconfiguration as a powerful technique for achieving system resilience because it allows the system to adapt to changes by modifying its structure, inter-component relationships and
dependencies. However, ensuring correctness of the incorporated reconfiguration mechanisms is a complex task. To address this issue, we formalise the possible interdependencies between goals and components as well as formulate the conditions for ensuring correctness of the design of the reconfiguration mechanism. In the refinement process, we also explicitly introduce component failures. To introduce a possibility of reconfiguration, we define the conditions on the information exchange between the components, i.e., determine which part of the local state the components should communicate to provide a sufficient basis for reconfiguration. Finally, we decompose the obtained specification into a number of independent components, i.e., arrive at the distributed architecture.

The proposed formalisation facilitates a systematic disciplined development of CPS and can be seen as an example of generic guidelines for designing reconfigurable systems.

6.3 MODELLING AND REFINEMENT IN EVENT-B

In this section, we give an overview of our formal development framework – Event-B. Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm 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 important system properties to be preserved during the execution are defined as model invariants. A machine usually has the accompanying component, called context. A context may include user-defined carrier sets and constants. Their properties are formulated as model axioms.

The dynamic behaviour of the system is defined by a collection of atomic events. An event is essentially a guarded command that, in the most general form, can be defined as follows:

\[ e \equiv \text{any } a \text{ where } G_e \text{ then } R_e \text{ end}, \]

where \( e \) is the event’s name, \( a \) is the list of local variables, and (the event guard) \( G_e \) is a predicate over the model state. The body of an event is defined by a multiple (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically
defined as the next-state relation $R_e$. The event 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 none of the events is enabled then the system deadlocks. The occurrence of events represents the observable behaviour of the system.

**Event-B Refinement.** Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional system behaviour. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. A machine can be refine in two possible ways either using *data refinement* or *superposition refinement*. In particular, we can replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. In this case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables. Via such a *gluing* invariant – “refinement relation” – we mathematically establish a correspondence between the state spaces of the refined and the abstract machines.

During *superposition refinement*, new implementation details are introduced into the system specification by means of *new* events and *new* variables. The new events can not affect the variables of the abstract specification and only define computations on the newly introduced variables.

The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit *skip* (the null action). To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events *converge*. To prove it, we have to define a *variant* – an expression over a finite subset of natural numbers – and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. In that case, the event obtains the status *anticipated*. This obliges the designer to prove, at some later refinement step, that the event indeed converges.

The correctness and consistency of Event-B models, i.e., verification of the model well-formedness, invariant preservation, deadlock-freeness and correctness of the refinement steps, is demonstrated by proving
the relevant verification theorems – *proof obligations*. Proof obligations are expressed as logical sequences, ensuring that the transformation is performed in a correctness-preserving way [1].

Modelling, refinement and verification of Event-B models is supported by an automated tool – the Rodin platform [29]. The platform provides the designers with an integrated modelling environment as well as supports automatic generation and proving of the necessary proof obligations by means of wide range of automated provers. Moreover, various plug-ins created for the Rodin platform allow a modeller to transform models from one representation to another, e.g., from UML to Event-B language [36, 31], or from Event-B specification to programming languages C/C++ [25, 24], ADA [9, 8], etc.

The Event-B refinement process allows us to gradually introduce implementation details and ensure adherence to the abstract specification. Such an approach seamlessly weaves verification with model development and allows us to construct detailed models of complex systems in a highly automated incremental manner. By providing an immediate feedback on the correctness of the model transformations, it helps to cope with the complexity of system development. Another important mechanism for handling complexity of formal development is *decomposition*.

Model decomposition helps the designers to separate component development from the overall system model but ensure that the components can be recomposed into the overall system in a correctness-preserving way [15]. Event-B is equipped with three forms of decomposition: shared-variable [2, 14, 5], shared-event [5] and modularisation [16], all of which are supported by the corresponding Rodin plug-ins [35, 30]. In this work we rely on a modularisation extension of Event-B [16].

**Modularisation.** Modularisation extension allows the designers to decompose a system into *modules*. Modules are components containing groups of callable atomic operations [16, 30]. Modules can have their own (external and internal) state and invariant properties. An important characteristic of modules is that they can be developed separately and, when needed, composed with the main system. Since decomposition is a special kind of refinement, such a model transformation is also a correctness-preserving step that has to be verified. Hence, a number of corresponding proof obligations are generated and should be proved.
A module description consists of two parts – the module interface and the module body. A module interface is a separate Event-B component that consists of the external module variables, the module invariants, and a collection of module operations, characterised by their pre- and postconditions. In addition, a module interface may contain a group of standard Event-B events. These events model autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, they describe how the module external variables may change between operation calls. Development of a module starts with deciding on its interface. Once the interface is defined, it cannot be changed in any manner during development. This ensures that a module body may be constructed independently from a system model that relies on the module interface. A module body is an Event-B machine. It implements the interface by providing a concrete behaviour for each of the interface operations. To guarantee that each interface operation has a suitable implementation, a set of additional proof obligations is generated.

The modularisation extension of Event-B facilitates formal development of complex systems by allowing the designers to decompose large specifications into separate components and verify system-level properties at the architectural level. As a result, proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of complex distributed systems.

6.4 DERIVING RESILIENT ARCHITECTURE

We start our development in Event-B by creating a high-level abstract specification. In the abstract model, we focus on specifying the overall system behaviour. Here we aim at specifying property (PR1). The dynamic behaviour of the system is modelled by the abs behavioural machine. We define a variable $goal \in STATES$ that models the current state of the system goal, where $STATES = \{incompl, compl\}$. The variable $goal$ obtains the value $compl$ when the main goal is achieved. Otherwise $goal$ has the value $incompl$. To abstractly model the process of achieving the goal – specified in the event process – the variable $goal$ may change its value from $incompl$ to $compl$. The system continues its execution until the goal is reached:
6.4.1 Functional Decomposition by Refinement

The objective of our first refinement step is to elaborate on the structure and the behaviour of the abstract model. The overall system execution consists of a number smaller steps that we call tasks (as defined by (PR2) property). Essentially, a task represents a single functional step that the system has to execute to progress towards the goal achievement. In this work, for simplicity, we consider only a simple forward execution scenario. In general, our model permits the definition of any type of a complex execution scenario, e.g., combining sequential and parallel task execution, branching, rollbacking, etc.). In Event-B, we define the execution scenario via formulating a set of axioms in the model’s context.

In our first refinement step, we focus on representing the overall system execution as an iterative execution of a set of tasks. To represent all tasks that constitute system execution, we introduce a new abstract type (set) TASKS into the model’s context. Naturally, we require that the set TASKS is finite and non-empty.

In the machine part of the first refinement, we define the new variable tasks_state that stores the current execution status of each task:

\[
\text{tasks_state} \in \text{TASKS} \rightarrow \text{STATES}.
\]

Initially, none of the task is completed, i.e., the status of each task is incompl. After successful execution, the task’s status changes to compl. The event process is now refined to represent the progress in task executions. Note that this event is parametrised – the parameter \( t \) designates the id of task being processed:
To establish the relationship between the main goal and tasks, i.e., to model the property (PR2) in our specification, we formulate and prove the following gluing invariant:

\[ \text{goal=} \text{compl} \iff (\forall t. t \in \text{TASKS} \Rightarrow \text{tasks}._{\text{state}}(t) = \text{compl}) \].

The invariant postulates that the main system goal is achieved if and only if all the involved tasks are successfully completed.

Let us note, that the proposed refinement step can be repeatedly used to refine tasks into subtasks of finer granularity until the desired level of details is reached.

### 6.4.2 Second Refinement

**Introducing System Components.** In our previous refinement step, we focused on functional decomposition of the system behaviour into execution of atomic tasks. We have deliberately abstracted away from associating these tasks with the specific system components that perform the tasks and orchestrate their execution. The goal of our second refinement step is to (i) introduce into the model representation of the system components; (ii) model their possible activation and deactivation (including both normal and abnormal situations), and (iii) abstractly define system reconfiguration mechanisms.

Our system consists of two types of components – managers and workers. The manager components orchestrate the tasks execution, while the worker components actually perform the tasks. A manager monitors its associated tasks, assigns these tasks to workers, and receives/processes the results from the workers. Moreover, it interacts with the other manager components to share the information on task status. By integrating into the system design information sharing functionality, we ensure that in case of failure of a manager, some other available manager component has sufficient information to take over the responsibilities for the tasks and workers of the failed one.

We assume that both types of system components are unreliable, i.e., any component might fail during the system execution. To ensure
that all system tasks will be accomplished (and, consequently, the main system goal will be eventually achieved), we should incorporate into the system design some mechanisms that would allow the system to complete its execution despite the component failures.

To model system components in Event-B, we first introduce several new data structures into our model’s context. In particular, we introduce two new abstract types (sets) – $\text{MANAGERS}$ and $\text{WORKERS}$ – that store all available components of each type. Moreover, we define the constants, $\text{Init}\_\text{Managers}$ and $\text{Init}\_\text{Workers}$, which are subsets of the sets $\text{MANAGERS}$ and $\text{WORKERS}$ correspondingly. These constants define the sets of active components of the corresponding type at the beginning of the system operation.

The set $\text{TASKS}$ contains all tasks that the system should accomplish to reach its main goal. We assume that each worker is capable of performing a certain subset of tasks. Hence, before giving an assignment to a worker, we have to choose a worker of a suitable type, i.e., the one that is able to perform the assigned task. In order to associate the worker components with computational tasks they are able to perform, we define the following functions as axioms in the model’s context:

\[
\begin{align*}
\text{wtype} & \in \text{WORKERS} \rightarrow \text{WTYPE}, \\
\text{WT}\_\text{Rel} & \in \text{WTYPE} \rightarrow \mathcal{P}(\text{TASKS}).
\end{align*}
\]

Here $\text{WTYPE}$ is the set that contains all possible types of workers. Essentially, $\text{wtype}$ associates each worker with its respective type. In its turn, $\text{WT}\_\text{Rel}$ associates each worker type with a subset of specific tasks that the workers of this type are capable of executing. Such mappings allow us to check in a very straightforward way whether a worker is able to accomplish a specific task.

In the machine part of the Event-B specification, we define a set of active manager components as the variable $\text{managers}$, where $\text{managers} \subseteq \text{MANAGERS}$. In the similar way, we introduce the set of currently active worker components $\text{workers} \subseteq \text{WORKERS}$.

To model activation and deactivation of system components, we introduce a number of new model events: $\text{m}\_\text{activation}$, $\text{m}\_\text{deactivation}$, $\text{w}\_\text{activation}$, $\text{w}\_\text{deactivation}$. For instance, the event $\text{w}\_\text{activation}$ presented below models activation of the worker component:
Modelling Dependencies Between System Elements. A manager component can be associated with a number of tasks to be supervised. Moreover, each task should always be associated with some manager. However, during the system execution, a system task can be dynamically reallocated from one manager to another. The association of the tasks to the managers is defined by the function \( \text{Responsible} \):

\[
\text{Responsible} \in \text{TASKS} \rightarrow \text{managers}.
\]

The function assigns each task to some manager. The mapping might dynamically change during the system operation. Note that \( \text{Responsible} \) is a total function because every task should be associated with some manager. Such an approach allows us to represent the property (PR6).

Furthermore, each manager can be associated with a number of workers to supervise. Again, the manager-worker relationship is dynamic since any worker can be reassigned to another manager during the system execution. To introduce this dynamic characteristic, and formalise property (PR7), we define a new variable \( \text{Attached} \) that maps managers to workers:

\[
\text{Attached} \in \text{workers} \mapsto \text{managers}.
\]

Here \( \mapsto \) denotes a partial function, which reflects the fact that some workers may be not attached to any manager.

Finally, the new variable \( \text{Assigned} \) associates the currently executed tasks with the corresponding workers:

\[
\text{Assigned} \in \text{TASKS} \mapsto \text{workers}.
\]

Here \( \mapsto \) denotes a partial injection. The function is injective because a worker can not perform more than one task simultaneously. In such a way, we formulate the property (PR9). Naturally, a manager can assign only incomplete task to a worker:

\[
\forall t. t \in \text{dom}(\text{Assigned}) \Rightarrow \text{tasks\_state}(t) = \text{incompl}.
\]
Moreover, as we have discussed earlier, only workers of a correct type can be assigned for a task execution. We formulate this property by the following model invariant:

\[ \forall t, w. (t \mapsto w) \in Assigned \Rightarrow t \in WT_{Rel}(wtype(w)). \]

Essentially, all components and tasks defined here are connected to each other via the introduced dependencies. For instance, if some task \( t \) has been assigned to the worker \( w \), and \( m \) is the manager responsible for this task then \( m \) also supervises \( w \):

\[ \forall m, t, w \cdot (t \mapsto w) \in Assigned \land (t \mapsto m) \in Responsible \Rightarrow (w \mapsto m) \in Attached. \]

Similarly, if task \( t \) is assigned to some worker \( w \), and \( w \) is supervised by the manager \( m \), then \( m \) becomes also responsible for the task \( t \):

\[ \forall m, t, w \cdot (t \mapsto w) \in Assigned \land (w \mapsto m) \in Attached \Rightarrow (t \mapsto m) \in Responsible. \]

The new event assign (given below) models assignment of a task \( t \) to a worker component \( w \) by its manager \( m \). To make such an assignment, a number of conditions should be satisfied. Firstly, we have to be sure that \( t \) is not yet accomplished, and is not being currently executed by any other worker. Secondly, the type of the worker should be suitable for performing the task \( t \). Moreover, the worker \( w \) should be free, i.e., it is not involved into an execution of an already assigned task. Finally, we have to check that both \( t \) and \( w \) “belong” to the manager \( m \).

```plaintext
assign \in
any w, t, m
where tasks_state(t) = incompl // task t is not yet accomplished
t \notin dom(Assigned) // the task t is not assigned to any other worker
t \in WT_{Rel}(wtype(w)) // the worker w is able to execute the task t
w \notin ran(Assigned) // the worker w has not already an assigned task
(w \mapsto m) \in Attached // the worker w is attached to the manager m
Responsible(t) = m // the manager m is responsible for the task t
then Assigned(t) := w
end
```
Modelling Task Execution. Upon receiving the assignment from its manager, the worker starts to perform the assigned task. After successfully completing the task, the worker reports this to the manager. However, while performing a given task, a worker might fail, which subsequently leads to the failure to perform the assigned task. In case of a worker failure, the manager should choose another available worker to execute the failed task.

To reflect this behaviour in our model, we refine the abstract event process by two events `task_compl` and `task_incompl`, which model respectively successful and unsuccessful execution of the task. If the task has been successfully performed by the assigned worker `w`, its supervising manager `m` changes the status of the tasks `t` to completed:

\[
\text{task_compl} \equiv \text{refines process}
\]

\[
\begin{align*}
\text{any } w, t, m \\
\text{where } t \rightarrow w \in \text{Assigned} \\
\text{then } \text{tasks.state}(t) := \text{compl} \\
\text{Assigned} := \text{Assigned} \setminus \{t \rightarrow w\}
\end{align*}
\]

The event `task_incompl` abstractly models the opposite situation when the worker fails to complete the assigned task due to a failure. In this case, the task `t` can be reassigned to another available worker.

In our model, we assume that in our system, there exists a special component – the middleware – that is responsible for detecting the component failures at low level. For example, failure detection can be based on the following heartbeat mechanism. The system components periodically send simple messages to middleware to inform it about their status. These messages are called heartbeats. If an expected heartbeat is not received during a specified time period, it is assumed that the corresponding component has failed. In case of a worker failure, the middleware will notify the supervising manager about the failure of its worker. In case of a manager failure, the middleware will notify the other managers about that. It would allow them to initiate a system reconfiguration.

Towards Modelling System Reconfiguration. As we described earlier, the responsibilities for executing a task and supervising the workers can be reallocated from one manager to another as a part of the system reconfiguration. To model redistribution of tasks and workers between the responsible/supervising managers, we define the
new event **redistribute**. It models the situation when the manager $m_2$ takes over the tasks $ts$ and workers $ws$ of the manager $m_1$:

```
redistribute ≡
any ws, ts, m_1, m_2
where m_1 ∈ managers
    ts = dom(Responsible ▷ {m_1}) // all the tasks of the manager m_1
    ws = dom(Attached ▷ {m_1}) // all the workers of the manager m_1
    ts ≠ ∅ ∧ ws ≠ ∅ ∧ m_1 ≠ m_2
    ws ⊈ ran(Assigned)
then
    Responsible := Responsible − (ts × {m_2})
    Attached := Attached − (ws × {m_2})
end
```

Here $\Leftrightarrow$ denotes the overriding relation, i.e., $q \Leftrightarrow r = r ∪ (\text{dom}(r) \Leftrightarrow q)$, where $\Leftrightarrow$ denotes the domain substraction, i.e., $S \Leftrightarrow r = \{x \mapsto \rightarrow y | x \mapsto y ∈ r \land x /∈ S\}$.

We also allow the system to reassign the workers from one manager to another as a part of component cooperation aiming at improving resource utilisation. In that case, the workers of the manager that completed all its tasks can be sent to some other manager that still has some unfinished tasks. Additionally, we reserve a possibility to cancel all the current assignments for a group of workers. This functionality will be needed later on, e.g., to describe the effect of manager failures$^1$.

Let us note that in this refinement step we have abstracted away from the reasons behind redistribution of responsibilities between manager components. However, later on, when we introduce the failures of manager and worker components, we will refine this behaviour by adding the specific conditions imposed on the system reconfiguration.

Since the currently available resources (the active manager and worker components) might be insufficient to accomplish the main goal, the system has a possibility to activate idle (spare) components. This behaviour has been modelled by the abstract events **w_activation** and **m_activation**. Note that the system should be able to integrate the new components into its execution flow. We model this by the new event **attach_worker**. Finally, to release the excessive system resources, some components might be deactivated. However, while deactivating system components, we should guarantee that the associated inter-components relationships are preserved.

$^1$We do not show here the corresponding events to avoid overloading the chapter details of the Event-B specifications
6.4.3 Representing Manager’s Local Knowledge

The variable tasks_state represents the global knowledge about statuses of all tasks. In our abstract specification, to orchestrate the system operation (assigning uncompleted task to workers, etc.) the active managers access (read) this global knowledge. However, in the decentralised systems the components typically lack knowledge about the global system state. Thus, in our system, the managers should rely on their local state – local knowledge – while coordinating the workers activities. The local knowledge of each manager should contain accurate information about the statuses of its own tasks, and also the most recent (up to specified delay) information about the statuses of the tasks of all other managers in the system.

To model the local knowledge of the active managers, we introduce the new variable local_tasks:

\[ \text{local_tasks} \in \text{managers} \rightarrow (\text{TASKS} \rightarrow \text{STATES}). \]

Essentially, the local knowledge is defined for all active managers that have any tasks to coordinate:

\[ \text{ran(Responsible)} \subseteq \text{dom(local_tasks)}. \]

Our abstract variable tasks_state is now refined by the newly introduced local_tasks variable. To prove data refinement and establish the relationships between the values of concrete and abstract variables, we formulate and prove the following gluing invariant:

\[ \forall m, t \cdot t \in \text{TASKS} \land m \in \text{managers} \Rightarrow (\text{local_tasks}(m)(t) = \text{compl} \Rightarrow \text{tasks_state}(t) = \text{compl}). \]

The invariant defines the restrictions of the consistency of information between global and local state to be maintained by the system. Namely, it states that for any active manager, its local knowledge is consistent with the global one, i.e, if a particular task is marked as completed in the local knowledge of this manager, then this task is considered as completed in the global knowledge as well. Moreover, for each manager component, the local information about its own tasks always coincides with the global knowledge about these tasks:

\[ \forall m, t \cdot t \in \text{TASKS} \land \text{Responsible}(t) = m \Rightarrow (\text{local_tasks}(m)(t) = \text{compl} \Leftrightarrow \text{tasks_state}(t) = \text{compl}). \]
A manager component keeps track of the completed and non-completed tasks, and also periodically receives the information from the other managers about their completed tasks. However, the knowledge is inaccurate for the time span when the information is sent but not yet received. In this refinement step, we abstractly model receiving the information by a manager by introducing the new event $\text{update}_{\text{local}}$:

\[
\text{update}_{\text{local}} \triangleq \\
\text{any } m, t, mm \\
\text{where } m \in \text{managers} \\
\quad \text{local\_tasks}(m)(t) \neq \text{compl} \\
\quad \text{Responsible}(t) = mm \\
\quad \text{local\_tasks}(mm)(t) = \text{compl} \\
\text{then } \text{local\_tasks}(m) := \text{local\_tasks}(m) \cup \{t \mapsto \text{compl}\}
\]

The event $\text{update}_{\text{local}}$ models updating the local knowledge of a manager $m$. Here we have to ensure that the obtained information is always consistent with the information stored in the knowledge of the responsible manager. Specifically, the manager $m$ marks the task $t$ as completed only if it has completed status in the knowledge of the manager $mm$ that is responsible for executing this task.

Moreover, in order to accept responsibilities for new tasks, the newly activated manager should also have information about the current situation of the task statuses. Thus, its local knowledge should be initialised according to the current situation. We model this behaviour by the new event $\text{initialise}_{\text{local}}$ (omitted here).

In our previous refinement step, in the event $\text{redistribute}$ we abstractly modelled the possibility to redistribute responsibilities for tasks and workers between two managers. However, a manager can take a responsibility for the new tasks only if it has an accurate knowledge about statuses of these tasks. We add this as an additional condition in the event guards, where we check that the local knowledge about reallocated tasks coincides for both involved managers:

\[
\text{redistribute} \triangleq \text{refines } \text{redistribute} \\
\text{any } ws, ts, m_1, m_2 \\
\text{where } ... \\
\quad ts = \text{dom}(\text{Responsible} \supset \{m_1\}) \\
\quad \forall t. t \in ts \Rightarrow \text{local\_tasks}(m_1)(t) = \text{local\_tasks}(m_2)(t) \\
\quad ... 
\]

Let us note that at this stage we still rely on the ability of the manager components to “read” the knowledge of the other managers, i.e., we
are not yet ready to decompose the system into a distributed model. However, in the fifth refinement step the managers will rely solely on the inter-component communication to learn about the global system state.

**Local knowledge about attached workers and responsible tasks.** During reconfiguration, the manager receiving new responsibilities should have information about the tasks and workers it will accept. There are two ways to model this. The first one is similar to what we used for modelling the local knowledge about the task statuses – in the local knowledge of each manager we can also store the data about the attached workers of all managers in the system. In that case, the managers should periodically exchange this information amongst themselves.

The second mechanism that can be employed is based on reliance on middleware services. Specifically, middleware might have a responsibility to inform the managers which tasks and workers they should accept. In our modelling, we have chosen the second strategy.

### 6.4.4 Explicit Model of Component Failures

In our fourth refinement step, we aim at modelling possible component failures. To achieve this, we partition the active components into operational and failed ones. For example, the current set of all operational managers is defined by a new variable `operational_managers`. Initially all active managers are operational, i.e., `operational_managers=Init_Managers`. To model possible failure of an active manager, we define the `managerFailure` event:

```latex
\text{managerFailure} \equiv \\
\text{any } m \\
\text{where } m \in \text{operational}\_\text{managers} \\
\text{then } \\
\text{operational}\_\text{managers} := \text{operational}\_\text{managers} \setminus \{m\}
```

The events `assign`, `redistribute`, `reattach` and `attach_worker` are now refined to reflect that only the operational managers can give task assignments and participate in the system reconfiguration. Moreover, in the event `redistribute` we add an additional condition – only if a manager is classified as failed (i.e., $m \in \text{managers} \land m \notin \text{operational}\_\text{managers}$), another manager can take over its tasks and workers. In the similar
way, we introduce worker failures and specify their effects on the task execution and redistribution.

6.5 MODELLING OF COMMUNICATION

6.5.1 Communication Between Managers

The aim of the fifth refinement step is to define the communication model between the system components. After receiving a notification from a worker about a successful task completion, a manager updates its local knowledge and broadcasts the message about the completed tasks to the other managers. In its turn, upon receiving such a message, each manager component correspondingly updates its own local knowledge. Here we assume that the communication between managers is reliable, i.e., no sent message is lost and every base station will eventually receive it. We could also consider communication failures, as we did it in, e.g., [37]. However, since communication failures and recovery are not the main focus of this work, for the sake of simplicity we decided not to include them into the model.

The middleware is responsible for enabling the inter-component communication. In particular, the middleware implements a simple point-to-point communication protocol. The middleware observes the outgoing buffer of the manager components. When a manager component puts a message into its outgoing buffer, the middleware delivers the message to all other managers, i.e., the middleware puts this message into all corresponding incoming buffers. As soon as a new message is delivered to the manager, the manager updates its local knowledge.

To model communication between manager components, we introduce the following communication structures:

- $\text{taskOutgoingM}$: manager’s outgoing (one-place) buffer for its accomplished task id;
- $\text{taskIncomingM}$: manager’s incoming buffer for accomplished task ids.

The middleware constantly observes the managers’ outgoing buffers and reacts on the appearance of a new message awaiting delivery. Here we assume that a manager can place only one task at a time to its outgoing buffer.

In the Event-B machine created at this refinement step, we model
the corresponding buffers as the variables \(\text{taskOutgoingM}\) and \(\text{taskIncomingM}\) with the following properties:

\[
\text{taskOutgoingM} \in \text{managers} \rightarrow \mathcal{P}(\text{TASKS}),
\]

\[
\text{ran(Responsible)} \subseteq \text{dom(}\text{taskOutgoingM}),
\]

\[
\forall m. \text{card}(\text{taskOutgoingM}(m)) \leq 1.
\]

and

\[
\text{taskIncomingM} \in \text{managers} \rightarrow \mathcal{P}(\text{TASKS}),
\]

\[
\text{ran(Responsible)} \subseteq \text{dom(}\text{taskIncomingM}),
\]

If for any manager \(m\), a task \(t\) belongs to its outgoing buffer, i.e., \(t \in \text{taskOutgoingM}(m)\), then this task \(t\) has been completed. This property can be formulated as the following model invariant:

\[
\forall m, t. \ m \in \text{managers} \land t \in \text{taskOutgoingM}(m) \Rightarrow \text{local}\_\text{tasks}(m)(t) = \text{compl}.
\]

Moreover, if for any manager \(m\), a task \(t\) belongs to its incoming buffer, i.e., \(t \in \text{taskIncomingM}(m)\), then this task \(t\) has been completed before:

\[
\forall m, t, mm. \ m \in \text{managers} \land t \in \text{taskIncomingM}(m) \land \text{Responsible}(t) = mm \Rightarrow \text{local}\_\text{tasks}(mm)(t) = \text{compl}.
\]

Additionally, if for any manager \(m\), a task \(t\) belongs to its incoming buffer, i.e., \(t \in \text{taskIncomingM}(m)\), then the task \(t\) is still considered as unfinished in the local knowledge of this manager:

\[
\forall m, t. \ m \in \text{managers} \land t \in \text{taskIncomingM}(m) \Rightarrow \text{local}\_\text{tasks}(m)(t) = \text{incompl}.
\]

A manager component keeps track of the completed and non-completed tasks and periodically receives the information from the other managers about their completed tasks. However, the knowledge is inaccurate for the time span when the information is sent but not yet received.
6.5.2 Communication Between Managers and Workers

Similarly to communication between managers, we introduce the communication between a manager and its supervised workers. The core manager functionality is to control the tasks execution and to assign tasks to its workers. To assign a task, a manager sends a corresponding message to a worker, which is delivered by middleware. In its turn, upon completion the task, the worker sends a report message to its manager. Once the manager receives the message, it marks the corresponding task as completed. To model this behaviour, we introduce the following worker’s communication structures:

- $task\text{Incoming}W$: worker’s incoming buffer for the assigned task from the manager;
- $task\text{Outgoing}W$: worker’s outgoing buffer for the completed task.

The manager’s structures for communication with a worker are:

- $task\text{Assigned}M$: manager’s outgoing buffer for an assigned task;
- $task\text{Completed}M$: manager’s incoming buffer for the accomplished task;

Middleware constantly observes the changes made to the output buffers of components and reacts on the appearance of a new message awaiting the delivery.

6.6 TOWARDS MODEL DECOMPOSITION

Now we arrive at a centralised specification of the resilient CPS. Our next goal is to derive its distributed implementation by refinement. We employ modularisation facilities of Event-B to achieve this.

To model a distributed architecture of the system, we decompose by refinement the centralised model into a machine defining middleware functionality and two separate indexed modules representing the managers and workers. Moreover, we explicitly model communication between them. We assume that all managers and workers are identical in terms of communication. Since indexed module instantiation allows for the creation of an arbitrary number of copies of the same module, we define one interface for a manager and one interface for a worker. These modules will be imported to the refined machine instantiated
by the finite set of managers $Init\_Managers$ and workers $Init\_Workers$, respectively.

The generic module interfaces reflecting the described functionality for a manager component presented in Fig.6.1. The interface contains processes that model the described behaviour of a manager (see Process clause). Moreover, to model component communication, the callable operations for each component are defined as well (see Operations clause). These operations will be invoked by the middleware every time a message is delivered or received.

The generic module interface for a worker is presented in Fig.6.2. The refined machine Middleware models all types of communication and invokes these operations in the bodies of its events. As a result of the decomposition we arrive at a specification of distributed system. The manager and worker components are represented by the corresponding modules. The middleware serves as a communication infrastructure.

<table>
<thead>
<tr>
<th>Interface Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables ...</td>
</tr>
<tr>
<td>Invariants ...</td>
</tr>
<tr>
<td>Process ...</td>
</tr>
<tr>
<td>AssignTask       // a manager sends a task to a worker</td>
</tr>
<tr>
<td>TaskCompleted    // a manager records a task as completed and broadcast to others</td>
</tr>
<tr>
<td>TaskFailed       // a manager records a task as failed</td>
</tr>
<tr>
<td>UpdateKnowledge  // a manager updates its local knowledge</td>
</tr>
<tr>
<td>NewResponsibility // reassigning workers and tasks from other manager</td>
</tr>
<tr>
<td>GiveResponsibility // reassigning workers from other manager</td>
</tr>
<tr>
<td>ManagerFailure   // a manager fails</td>
</tr>
<tr>
<td>Operations</td>
</tr>
<tr>
<td>FromWorker       // incoming message from a worker about accomplished task</td>
</tr>
<tr>
<td>FromManager      // incoming message from other manager</td>
</tr>
<tr>
<td>WorkerFailureNotif // a worker has failed</td>
</tr>
<tr>
<td>ManagerFailureNotif // a manager has failed</td>
</tr>
<tr>
<td>NewWorkers       // a manager receives new worker(s)</td>
</tr>
</tbody>
</table>

Figure 6.1  Base station interface component
Formalising Goal-Oriented Development of Resilient CPS

6.7 RELATED WORK AND CONCLUSIONS

6.7.1 Related Work

Significant work has been done on goal-oriented requirement engineering approaches. The foundational work on goal-oriented development belongs to van Lamsweerde [6, 39, 40]. The proposed KAOS framework [6] introduces a goal-oriented approach for requirements modelling, specification, and analysis as well as addresses both functional and non-functional system requirements. Based on the KAOS framework, Lamsweerde [38] has proposed a method for deriving the software architecture from its requirements. Specifically, according to the method, a software specification is developed from the given system requirements and then used to build the architectural system design. The design is developed by consecutive refinements, which take into account the system constraints and non-functional goals. The KAOS approach is supported by the GRAIL tool [6].

Over the last decade, the goal-oriented approach has also received several extensions that allow the designers to link it with formal modelling [18, 27]. In particular, the work [18] presents a translation technique of KAOS operational models into event-based tabular specifications, which can be then analysed by the SCR* toolset [13]. The technique consists of a number of transformation steps, each of which solves semantic, structural or syntactic differences between the KAOS and SCR (Software Cost Reduction) languages.

A significant body of research has been also devoted for translating formal specifications built according to the KAOS goal-oriented
method into event-based transition systems. For example, the work [21] presents an approach to use the formal analysis capabilities of LTSA (Labelled Transition System Analyser) to analyse and animate KAOS operational models. The mapping allows the designers to translate goal-oriented operational requirements into a black-box event-based model of the software behaviour, expressed in a formalism appropriate to reason about system behaviours at the architectural level.

One of the first attempts to bridge the KAOS goal-oriented framework with the B formalism was presented in [28]. More recently, the study to formalise KAOS requirements in Event-B was attempted in [4]. The paper proposes a constructive approach that allows linking of high-level system requirements expressed as linear temporal logic formulae to the corresponding Event-B elements. The notion of a triggered event is used to translate time operators that are used in KAOS models. Similar, Matoussi et al. [22, 23] present works on coupling requirements engineering methods with formal methods. In contrast, in our work we have relied on goals to facilitate structuring of the system behaviour, while connecting them with the required agent collaboration and system reconfiguration mechanisms.

The goal behind our research is to both formally model and verify systems with intricate relationships between system components and goal structures, that are able to dynamically reconfigure themselves in order to tolerate various failures or changes.

The problem of formal verification and validation of CPS has been largely studied in the software engineering research community. For instance, in the ADVANCE project [3], 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 Event-B development and co-simulation with tool independent physical components via the FMI interface [32].

The DESTECS project [7] also studies the problem of engineering of resilient CPS by integration of model-based formal methods with discrete-event models. 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 also be augmented with descriptions of potential failures and fault tolerance
mechanisms [12, 10, 11]. 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 yet.

The similar idea of modelling a dynamic system architecture with the integrated reconfiguration mechanisms and assessing system resilience characteristics is presented in paper [17]. Using Event-B and its refinement technique, the authors derive a complex architecture of data processing capabilities of CPS. To assess the resilience of the obtained data processing architecture, the authors rely on the statistical model checking in UPPAAL.

The problem of system reconfiguration and its connection to predictable dynamic resilience is presented in works [34, 33]. The proposed architectural framework – MetaSelf – is suitable to 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 work [41] the authors propose a goal-oriented approach for self-reconfiguration by employing the idea of a monitoring feedback loop. The proposed framework uses goal models as software requirements models, and employs SAT solvers to check the current execution records against the models to diagnose task failures. The monitoring component monitors requirements and records data, while the diagnostic component analyses the recorded data and identifies failures of the system behaviour. In our work, we focus on modelling the system behaviour and dependencies between components, while not distinguishing the system components to be the monitoring or diagnostic ones. However, this idea can be incorporated into our system design.

6.7.2 Conclusions

In this work we have proposed a formal goal-oriented approach to development of resilient CPS by refinement in Event-B. We considered resilience as an ability of the system to achieve its goals despite changes.
We believe that a goal-oriented approach offers an suitable basis for reasoning about functional aspects of system resilience.

In our modelling, we have formalised the main concepts of goal-oriented development and formally defined connections between system goals, functional capabilities of the components and component interactions. The derived hierarchical distributed architecture can be seen as a generic pattern for defining supervisory control of reconfigurable CPS.

We formalised the main functional properties of the functional aspect of resilience in the goal-oriented style and demonstrated how refinement can help to structure complex requirements and scale proof-based verification. The formal development has allowed us to rigorously define and verify the intricate relationships between system components and goal structures (tasks). The presented development is generic. It defines a number of specification patterns that allow the designers to explicitly define architectural reconfiguration mechanisms to build the resilient system.

In this work, we focused on reasoning about functional aspect of resilience. In the future work, we are planning to address non-functional properties of resilience, such as reliability and performance as well as investigate how to formally model learning – an intrinsic part of future CPS.
Bibliography


34 Bibliography


