This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3168672, IEEE Transactions on Software Engineering

Program Synthesis for Cyber-Resilience

Nestor Catano

Abstract—Architectural tactics enable stakeholders to achieve cyber-resilience requirements. They permit systems to react, resist, detect, and recover from cyber incidents. This paper presents an approach to generate source code for architectural tactics typically used in safety and mission-critical systems. Our approach extensively relies on the use of the EVENT-B formal method and the EVENTB2JAVA code generation plugin of the Rodin platform. It leverages the modeling of architectural tactics in the EVENT-B formal language and uses a set of EVENTB2JAVA transformation rules to generate certified code implementations for the said tactics. Since resilience requirements are statements about a system over time, and because of the fact that the EVENT-B language does not provide (native) support for the writing of temporal specifications, we have implemented a novel Linear Temporal Logic (LTL) extension for EVENT-B. We support several architectural tactics for availability, performance, and security. The generated code is certified in the following sense: discharging proof obligations in Rodin - the platform we use for writing the EVENT-B models - attests to the soundness of the architectural tactics modelled in EVENT-B, and the soundness of the translation encoded by the EVENTB2JAVA tool attests to the code correctness. Finally, we demonstrate the usability of our resilience validation approach with the aid of an Autonomous Vehicle System. It further helped us increase our confidence in the soundness of our Event-B LTL extension.

Index Terms—Code Synthesis, Event-B, Formal Methods, Resilience, Security, Testing, Verification.

1 INTRODUCTION

Despite persistent efforts in the cyber-domain to make software applications cyber-resilient, it is clear that the prevention of all possible cyber-attacks is not feasible in the current state of the cyber-infrastructures [17], [16], [5]. This problem is exacerbated in the presence of 0-day attacks in which the software vulnerability is exploited before the software vendor is even aware of it. Therefore, cyber-resilience is an increasing concern for people in the military and people in charge of designing, building, or deploying safety-critical systems. The cyber-resilience of a system is assessed in terms of its ability to anticipate, withstand, recover from, and adapt to adverse conditions or attacks. A contributing factor is that cyber-security is often a secondary concern for people in the software security industry, hence, whereas software security efforts are mainly focused on delivering services and functionalities, security is often enforced as an after-thought, realized through security patches once a security breach is exposed.

• *E-mail: nestor.catano@berkeley.edu.*

Despite the importance of the architecture-first approach for enhancing and ensuring the resilience of safety-critical systems [5], state-of-the-art research and practice lack automated techniques that enable engineers and software architects to *generate certified code for architectural tactics*. Certified code is code that includes a proof of correctness of the code besides the code itself. Existing work focuses on the automated generation of code for *functional requirements* [26], [20] (high-level specifications) rather than for a wide variety of resilience architectural tactics, which are non-functional and are cross-cutting in nature [7].

This paper proposes a program synthesis approach for architectural resilience tactics. The approach is extended with the validation of the *detection* and *recovery* mechanisms of each tactic. Code validation increases our confidence that the system code is resilient against some particular types of cyber-attacks [9], [7]. Our approach extensively uses the *Event-B formal method* [2] to model architectural tactics for various quality attributes, a novel *code synthesis approach* based on the EVENTB2JAVA tool [20] to generate **certified code** for the tactics, and *software testing* to check (timing or static) properties of the tactics at the code level. Testing supplements verification and certified program synthesis by allowing us to check if we generated code for the right model (the model we had in mind). Therefore, cyber-resilience is brought into a system from the ground, starting from a robust and abstract software architecture, all the way down to a resilient software implementation. We do not regard security as an after-thought effort but consider that architecture design decisions should drive the implementation of a system to reinforce resilience.

In a nutshell: $(i.)$ we model architectural tactics for availability, performance, and security in EVENT-B. Architectural tactics improve the response of a system to an attack, thus a correct system implementation must correctly implement the tactics to withstand the attacks. $(ii.)$ Resilience requirements often include safety ("something bad does not occur") and liveness ("something good will eventually occur") temporal properties. These are property statements about what a system will do over time, so a stopped system will, in general, not satisfy these properties. A temporal logic property characterises the behaviour of a system by specifying a behaviour about system traces. As EVENT-B does not natively provide support for temporal specifications, this paper proposes an LTL (Linear Temporal Logic) [18] extension to EVENT-B for the said architectural tactics. Our LTL extension covers the *specification* of safety and liveness properties but only offers mechanisms to the

verification of the former. $(iii.)$ We leverage a novel code synthesis approach on top of the EVENTB2JAVA tool [11]. Our program synthesis approach specialises in the considered architectural tactics and our LTL extension to EVENT-B. $(iv.)$ The EVENTB2JAVA tool generates Java implementations of EVENT-B models as Eclipse projects. Our certified code generation approach goes through a dual sanitizing process. First, proof obligations are discharged for the EVENT-B models and the temporal properties to make sure that the models are sound. We use the Rodin platform [3] for writing the EVENT-B models and conducting the proofs. Second, Java unit tests are used to check and animate the behaviour of the EVENT-B models in Java to validate the models.

The **main contributions** of this paper are the following:

- We have formalised several architectural tactics [7] in the EVENT-B formal language for availability, performance, and security. We have conducted the formalisation and discharged the generated proof obligations with the Rodin platform version 3.4.
- We present a novel EVENT-B LTL extension. The extension is realized through a set of temporal patterns that altogether account for the modelling of safety and liveness properties in EVENT-B. We define the semantics of the LTL extension and its translation to EVENT-B. Our work on extending EVENT-B with temporal specifications paves future works that seek to extend EVENT-B with temporal logic natively.
- We have written a series of temporal properties for each architectural tactic, and have encoded them in EVENT-B, following the translation approach above. We have produced certified code implementations for the LTL properties as well as the respective architectural tactics.
- We increase our confidence in the soundness of the code generated for the architectural tactics in 2-ways. First, we discharge the proof obligations of the EVENT-B models of each architectural tactic in Rodin. And, second, we unit-test the code following the syntax of the LTL properties.
- We demonstrate the usability of our approach with the aid of an Autonomous Vehicle System (AVS). We model the whole system in EVENT-B, generate Java code for it, and interface it with the heartbeat implementation.

This paper is organized as follows: Section 2 discusses the background needed to understand the work presented in this paper; it describes the syntax and semantics of EVENT-B and its mathematical language. Section 3 presents our approach to make applications more cyber-resilient. Section 4 presents the EVENT-B modelling of the *heartbeat* architectural tactic. Section 5 presents the syntax and semantics of the EVENT-B LTL extension. Section 6 explains how this extension is encoded in EVENT-B. Section 7 describes how code is generated for LTL trace properties. Although the discussions in Sections 6 and 7 revolve around heartbeat, our ideas are general so that they can be extended to the rest of architectural tactics considered in this paper. Section 8 evaluates the *usability* of our work and discusses the rest of architectural tactics. Section 9 discusses related work, and Section 10 presents conclussions and discusses further work.

We have created a GitHub project https://github.com/-ncatanoc/cyber_resilience to store the EVENT-B models for the architectural tactics that we have written for this paper. It also contains program synthesis implementations of the tactics as Eclipse projects. Each Eclipse project contains respective unit-tests for each tactic in Java.

2 PRELIMINARIES

2.1 EVENT-B

EVENT-B models are called *machines*, composed of a static part defining *observations* (variables, constants, parameters, etc.) about the system and its invariants properties, and a dynamic part defining operations (called *events*) changing the state of the system. Each operation must maintain the system invariants. In EVENT-B, the language for stating properties, essentially predicate calculus plus set theory, and the language for specifying dynamic behaviours are seamlessly integrated. EVENT-B models are development of discrete transition systems, composed of *machines* and *contexts*. Contexts define the static part of a machine (except for variables and their invariants). EVENT-B represents a system as a succession of states connected through a series of transitions called *events*. An EVENT-B machine definition generates a series of Proof Obligations (POs). They are theorems stating a property that must hold in order for a formal model to be consistent. For instance, if an EVENT-B model includes an invariant property that states that some mathematical relation r is a function, then for any event that modifies r, a PO which states that the modified relation r' must be a function is generated.

EVENT-B relies on the *refinement* strategy for software development [4], [13] whereby a machine goes through a series of stages. Each stage adds more details (observations) to the description of a system. Hence, an abstract machine is first developed and verified to satisfy the system's correctness and safety properties. A machine refinement establishes a correspondence between an abstract machine and a (more) concrete machine in a way that the concrete machine fulfills (at least) the same properties that the abstract machine does. To prove that correspondence, *refinement proof obligations* must be discharged (proven) to ensure that each refinement is a faithful sound model of the previous machine, and so that all the machines satisfy the correctness safety property of the most abstract machine.

Figure 1 shows the syntax for events, where x is a set of local variables encoding the parameters of the event, the *guard* $G(c, v, x)$ depends on machine constants and variables, and the event parameters, respectively. A guard is a predicate that needs to hold for the event to be *enabled*. Only enabled events may be triggered (executed). The body of an event is composed of *actions*, that is, assignments of the form $v := E(c, v)$, where v on the left is a machine variable, and E is an expression that depends on all the machine constants c and all the machine variables v.

2.2 Tool Support

We have written the EVENT-B models for the architectural tactics with the Rodin toolset [3], an open-source Eclipse IDE that provides a set of tools for working with EVENT-B: an editor, a proof generator, and several provers.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3168672, IEEE Transactions on Software Engineering

EVENT $name$
ANY x
WHERE
@grd $G(c, v, x)$
THEN
@act $v := E(c, v, x)$
END

Element	Definition
$name$	name of the event
x	event parameters, a set of variables
$G(c, v, x)$	logical predicate
v	a disjoint subset of machine variables
$E(c, v, x)$	sets of expressions

Fig. 1. Basic syntax for events

We have used the EVENTB2JAVA tool [11], [12] to generate code for the EVENT-B models of the architectural tactics. EVENTB2JAVA itself is a Rodin plugin and offers full support for EVENT-B'syntax. EVENTB2JAVA generates code as Eclipse projects, and, in fact, our work on unit-testing the code generated by the EVENTB2JAVA tool (Section 7.1) has been conducted with the Eclipse platform.

3 Our Approach

Our approach relies on the use of formal methods techniques to generate **certified** code for a selected group of architectural tactics modelled in EVENT-B. We have modelled the following architectural tactics: *exception*, *heartbeat*, *pingecho*, and *timestamp* for **availability**; *priority* and *response* for **performance**; and, *authentication* and *discretionary access control* for **security**. Code synthesis of these architectural tactics is attained with the EVENTB2JAVA code generator [20], which generates multi-threaded code implementations of the EVENT-B models. We unit-test the code generated for each of the architectural tactics to validate our intuition on the behaviour of the certified code. One can write unit tests for each event[1], or can unit-test LTL trace properties of all the machine events in Java as described at the end of Section 7.

We next describe the different steps of our approach for program synthesis of resilient code.

(i.) We model common architectural tactics in EVENT-B. Each EVENT-B model includes system invariants and EVENT-B machine events for two different resilience architectural aspects of a system, namely, *detection* of security breaches and *recovery* from them [7].

(ii.) We extend each EVENT-B architectural tactic with LTL specifications. Since the EVENT-B language does not have a built-in mechanism for the modelling and/or refinement of temporal properties, we have conceived and implemented an EVENT-B LTL extension (Section 5), and a translation from that extension into EVENT-B (Section 6).

(iii.) We use program synthesis techniques to generate resilient Java implementations from the EVENT-B models of the architectural tactics. We focus on the EVENT-B encoding of temporal properties of each tactic, and the code generation of certified implementations[2].

(iv.) With the purpose of validating the **certified code** and checking our understanding of it, we write unit tests for the *detection* and *recovery* mechanisms of each tactic. These two mechanisms are encoded as LTL trace properties. Section 7 explains how certified code is generated for each architectural tactic, and how unit tests are written as LTL trace properties. Unit testing in Java supplements formal proof in Rodin. Formal proof serves to check if the architectural tactic in EVENT-B is sound (right), and unit-testing helps us check we model the right tactic.

4 The Event-B Modelling of Heartbeat

We show here the EVENT-B modelling of one of the architectural tactics that we have considered in our work. The *heartbeat* architectural tactic provides a good example to demonstrate our ideas. Heartbeat implements a fault detection mechanism based on the exchanging of periodic messages between a monitoring system and a monitored process. The monitor system determines if the monitored process has failed or not. It sends a beat, which the monitored process should acknowledge before some time limit; otherwise, the monitoring system assumes that the process has failed or is dead. Whenever a process fails, the system may roll-back to a safe state.

We consider two main aspects of each architectural tactic, namely, detection (of a failure) and recovery (from the failure). Our heartbeat EVENT-B model includes 6 events: SEND, RECEIVE, TICK, HASFAILED, FAIL, and ROLLBACK. SEND sends a beat - it sets the BEAT machine variable to TRUE; RECEIVE acknowledges the beat - it sets BEAT to FALSE, it has TSLOT time units to respond to the monitoring system, hence event TICK models a clock. Event HASFAILED checks if the process has failed or not. A process has failed if TIME = 0 and BEAT = TRUE. The ROLLBACK event sets the BEAT and TIME machine variables to their initial machine values regardless of the system has failed or not. Indeed, the FAIL event injects a failure into the system, and hence makes the guard of the HASFAILED event hold. HASFAILED implements the tactic detection and ROLLBACK the recovery one.

```
EVENT SEND
WHERE
 @grd1 BEAT = FALSE
THEN
 @act1 BEAT := TRUE
 @act2 TIME := TSLOT
 // more actions shown in Section 7
END
```

SEND sets BEAT to TRUE so as to state that the system has sent a beat (action @act1), hence, disabling the event guard (making @grd1 FALSE). Event action @act2 sets the time available for the RECEIVE event to respond or acknowledge the SEND event. BEAT is a machine variable, so all the (machine) events can access and edit it. According to EVENT-B semantics, events are executed atomically so only one event can execute at the time, and multiple assignments to the same machine variables are disallowed.

RECEIVE acknowledges SEND by setting BEAT to FALSE (action @act1).

1. EVENTB2JAVA translates each event as a Java class, so testing an event amounts to testing the methods of the class object.

2. The soundness proof of the translation encoded by the EVENTB2JAVA tool has been conducted elsewhere [10].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3168672, IEEE Transactions on Software Engineering


```
EVENT RECEIVE
WHERE
 @grd1 BEAT = TRUE ∧ TIME > 0
THEN
 @act1 BEAT := FALSE
 @act2 TIME := 0
 // more actions shown in Section 7
END
```

The heartbeat system can rollback to a safe state regardless of whether it has failed or not. We model this in EVENT-B by writing a ROLLBACK event with no WHERE condition. Hence, the event may execute under no external condition. ROLLBACK sets BEAT to FALSE so that SEND may execute afterwards. It also restarts the timer.

```
EVENT ROLLBACK
THEN
 @act1 BEAT := FALSE
 @act2 TIME := 0
 // more actions shown in Section 7
END
```

SEND, RECEIVE, and ROLLBACK include additional event actions that are discussed in Section 7. These actions are part of the encoding of our LTL extension that is discussed in Section 5.

5 THE EVENT-B LTL EXTENSION

5.1 The Syntax

This section presents the syntax of our EVENT-B LTL extension (Figure 2), Section 5.2 presents its semantics. An LTL property ϕ can occur under 3 different scopes, **after** or **before** a set of events occur, or **between** the occurrence of two sets of events. Hence, the "**after** <Events> ϕ" temporal property has an **after** scope, and thus ϕ must be evaluated after all the <Events> have occurred. Therefore, ϕ is a trace property. A trace property such as ϕ describes a part of a program execution, and hence, the evaluation of "**after** send **executed** ϕ" should start after the execution of send in the given trace, where send is a Boolean EVENT-B predicate that refers to a particular event.

Therefore, for a given event in EVENT-B, we define three Boolean EVENT-B predicates <event> **enabled**, <event> **disabled**, and <event> **executed**. They state situations in which the predicate holds or not. The two former predicates model the conditions for the event to be enabled or disabled. They are associated with some EVENT-B machine invariants in the translation presented in Section 6. The latter predicate models the situation in which the event has been executed. This is associated with a Boolean condition in the same translation. In general, EVENT-B invariants are written in predicate calculus. They can be as simple as the **true** or **false** Boolean constants, or as complex to include predicates over sets and relations.

We show below an example of an LTL property that uses our temporal extension syntax. It belongs to the *heartbeat* architectural tactic [7]. It states that a beat cannot be re-sent unless it is first received (by the process) or the sender (the system) rolls back to a safe state. We use the "," (comma) symbol to denote the disjunction of any of the events.

```
<scope>   ::=  after <Events> <scope>
             |  before <Events> <LTL>
             |  between <Events> <Events> <LTL>
             |  <LTL>

<Events>  ::=  <Event>
             |  <Event> , <Events>

<LTL>     ::=  always <LTL>
             |  eventually <LTL>
             |  <LTL> until <LTL>
             |  <LTL> unless <LTL>
             |  <Pred>

<Pred>    ::=  true | false
             |  <event> enabled
             |  <event> disabled
             |  <event> executed
             |  <Event> , <Event>
```

Fig. 2. Temporal extension syntax

```
after send executed
(
 always send disabled
  unless rollback executed, receive executed
)
```

The LTL property below belongs to the heartbeat architectural tactic as well. It states that if the system ever fails, then the system cannot send a beat unless it first rolls back to a safe state.

```
after hasFailed enabled
(
 always send disabled
  unless rollback executed
)
```

5.2 The Semantics of the LTL Extension

To evaluate the semantics of properties in the LTL extension, we assume a structure $M = (S, E, \sigma, \rightarrow)$ composed of a set S of states, a set E of events, a *state-based* trace σ, and a transition relation $\rightarrow \subseteq S \times E \times S$. An LTL temporal formula is interpreted over a state-based trace $\sigma = < s_1, s_2, \cdots >$. We say that $(\sigma, i) \vDash \phi$ whenever ϕ holds in the i^{th} state of the trace σ. When we write $\sigma \vDash \phi$, we mean that ϕ holds in the first state of the sequence σ, that is, $\sigma \vDash \phi$ is the same as $(\sigma, 1) \vDash \phi$. We use the expression σ^i, for $i \geq 0$, to denote $s_i, s_{i+1}, \cdots$. We say that $s_{i-1} \xrightarrow{e} s_i$ if the state s_i can be reached from the state s_{i-1} after the execution of *some* event $e \in E$, and thus say that an e-transition between states s_{i-1} and s_i exists.

Finally, we use the notation σ_k^i in this paper to represent the sequence $< s_k, s_{k+1}, \cdots, s_{i-1}, s_i >$, and the notation $\sigma(i)$ to refer to the state s_i. In what follows we assume that e is an event in E. We use the symbol $\Rightarrow$ for logical implication.

We have collected a series of EVENT-B models for different architectural tactics. We have extended these models with LTL specifications as explained in the remainder of this Section. For each considered architectural tactic, for each respective EVENT-B machine, and for each event E in that machine, we include fresh variables E_ENABLED

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3168672, IEEE Transactions on Software Engineering

and E_EXECUTED to the respective machine. We extend each event E to include an assignment that sets variable E_EXECUTED to TRUE. Each machine includes an invariant that associates the E_ENABLED variable to a Boolean formula. Hence, E_ENABLED is set to TRUE or FALSE within each event E'≠E depending on whether E' makes the formula TRUE or FALSE, respectively.

Definition 5.1 (Semantic definition for the scope).

$$\sigma \vDash \textbf{after}\ E\ \phi \quad \text{iff} \quad \forall_{i>0} \cdot s_{i-1} \xrightarrow{e} s_i \Rightarrow \sigma^i \vDash \phi$$
$$\sigma \vDash \textbf{before}\ E\ \phi \quad \text{iff} \quad \forall_{i>0} \cdot s_{i-1} \xrightarrow{e} s_i \Rightarrow \sigma_0^{i-1} \vDash \phi$$

A temporal formula **after** $E\ \phi$ holds for any execution trace after an e-transition occurs. Thus, in the suffix path after the transition occurs, the ϕ trace property must hold. Similarly, a temporal formula **before** $E\ \phi$ must hold for the prefix path before the e-transition. The semantic definition for **between** $E_1\ E_2\ \phi$ (not shown here) combines the two previous definitions for prefix and suffix paths for e-transitions.

Definition 5.2 (Semantic definition for trace properties).

$$\sigma \vDash \textbf{always}\ \phi \quad \text{iff} \quad \forall_i \cdot \sigma^i \vDash \phi$$
$$\sigma \vDash \textbf{eventually}\ \phi \quad \text{iff} \quad \exists_i \cdot \sigma^i \vDash \phi$$
$$\sigma \vDash \psi\ \textbf{until}\ \phi \quad \text{iff} \quad \exists_k \cdot \sigma^k \vDash \phi \wedge (\forall_{i<k} \cdot \sigma^i \vDash \psi)$$
$$\sigma \vDash \psi\ \textbf{unless}\ \phi \quad \text{iff} \quad (\psi\ \textbf{until}\ \phi) \vee (\textbf{always}\ \psi)$$

The definition of **always** ϕ requires that ϕ holds in every state of the trace σ. And, the definition of **eventually** ϕ requires that it holds for some state in the trace. The operator **until** is what it is called "*strong until*" in literature, and **unless** "*weak until*". The latter holds even if ϕ does not as long as ψ holds globally (always).

Definition 5.3 (Semantic definition for Boolean predicates).

$$\sigma \vDash \textbf{true} \quad \text{iff} \quad \text{TRUE}$$
$$\sigma \vDash \textbf{false} \quad \text{iff} \quad \text{FALSE}$$
$$\sigma \vDash \textbf{e enabled} \quad \text{iff} \quad \sigma(0) \vDash \text{E\_ENABLED} = \text{TRUE}$$
$$\sigma \vDash \textbf{e disabled} \quad \text{iff} \quad \sigma(0) \vDash \text{E\_ENABLED} = \text{FALSE}$$
$$\sigma \vDash \textbf{e executed} \quad \text{iff} \quad \sigma(0) \vDash \text{E\_EXECUTED} = \text{TRUE}$$

Our semantic definition assumes the existence of an E_ENABLED Boolean variable for each machine event E. These variables are key to the translation from the LTL extension to EVENT-B. Each variable is associated to a machine invariant in EVENT-B, therefore the variable equals TRUE if and only if the invariant holds, e.g., a machine invariant stating that ROLLBACK_ENABLED = TRUE means that the ROLLBACK event may execute any time under no condition. We also assume the existence of an E_EXECUTED variable for each machine event E, and therefore, our LTL-to-EVENT-B algorithm (Beginning of Section 6) adds an event *action*[3] E_EXECUTED := TRUE to each machine event E.

Events must *maintain* all the machine invariants, hence, an assignment such as ROLLBACK_ENABLED := FALSE will produce an unprovable proof obligation for an INV ROLLBACK_ENABLED = TRUE machine invariant.

Variable SEND_ENABLED is associated with the guard of the SEND event through the invariant SEND_ENABLED = BOOL(BEAT = FALSE), therefore, every event that sets BEAT to TRUE must set SEND_ENABLED to FALSE, and vice-versa; otherwise, Rodin will generate an unprovable proof obligation, and the HEARTBEAT machine will be unsound.

3. An event is composed of potentially many machine variable assignments, called event *actions*.

6 THE LTL-TO-EVENT-B TRANSLATION

We next present the algorithm that translates LTL properties to EVENT-B. To motivate the algorithm we discuss the translation of the LTL trace property below.

after send **executed**
(
always send **disabled**
unless rollback **executed**, receive **executed**
)

The LTL property is encoded as the invariant INV, obtained as results of applying the next algorithm.

- $(i.)$ (the **after** part) Invariant INV is a logical implication whose antecedent is SEND_EXECUTED = TRUE. The SEND event must contain a SEND_EXECUTED := TRUE assignment. Additionally, any other event must contain a SEND_EXECUTED := FALSE assignment.
- $(ii.)$ (the **always** part) Invariant INV includes a consequent with a disjunction SEND_ENABLED = FALSE.
- $(iii.)$ (the **unless** part) Invariant INV includes a consequent with a disjunction (ROLLBACK_EXECUTED = TRUE OR RECEIVE_EXECUTED = TRUE) => SEND_ENABLED = TRUE.
- $(iv.)$ (the **after-always** part) The SEND event includes a SEND_ENABLED := FALSE assignment.
- $(v.)$ (the **always-unless** part) The ROLLBACK event includes a SEND_ENABLED := TRUE assignment. The RECEIVE event includes a SEND_ENABLED := TRUE assignment.
- $(vi.)$ (the **unless** part) The ROLLBACK event includes a ROLLBACK_EXECUTED := TRUE assignment. Any event other than ROLLBACK must include a ROLLBACK_EXECUTED := FALSE assignment.
- $(vii.)$ (the **unless** part) The RECEIVE event includes a RECEIVE_EXECUTED := TRUE assignment. Any event other than RECEIVE must include a RECEIVE_EXECUTED := FALSE assignment.

INV is shown below in full detail. It is the result of applying the first 3 steps of our LTL-to-EVENT-B algorithm. The first step makes it clear that we are furnishing a trace semantics for LTL, hence, having a send **executed** formula under an **after** scope requires us to construct a logical implication whose antecedent is SEND_EXECUTED = TRUE. Since the SEND event includes a SEND_EXECUTED := TRUE assignment, our trace property is evaluated after attaining a state that executes the SEND event.

Step 2 checks that the SEND event is disabled **after** the given condition. The checking is added as a consequent of the logical formula that is built. The third step implements the semantics of **unless**: it holds whenever the prior **always** condition does not. In this case, the **unless** formula is a disjunction (the comma symbol). If the disjunction holds then the **always** condition does not, hence, implementing the semantics of **unless**.

Steps 4 to 7 are not used to code INV, but are written within the machine events, as event assignments, which prevent the events from breaking INV. Step 4 states the relationship between the **after** and the **always** conditions, therefore, the event that relates to the **after** part must include an assignment that makes the condition mentioned in the **after** part TRUE.

Step 5 formulates the semantics of the relationship between the **always** and the **unless** parts. The event that relates to the **unless** part must include an assignment that negates the **always** condition.

The sixth and seventh steps instrument the semantics of the **unless** part. Let us amplify on step 7. For the heartbeat architectural tactic, sending a beat forbids a second beat to be sent, unless (until) the beat is first received or the system rolls back first. Now, let us suppose that a first SEND event executes successfully, a ROLLBACK event executes right afterwards, and a second SEND event executes thereafter. Although a ROLLBACK event has been executed immediately before the second SEND event, a third SEND event cannot execute. This means that the ROLLBACK event should at least include a ROLLBACK_EXECUTED := FALSE assignment.

```
INV ∈ BOOL
INV = TRUE // the invariant always holds
INV = BOOL( SEND_EXECUTED = TRUE => (
 ( ( ROLLBACK_EXECUTED = TRUE ∧
    RECEIVE_EXECUTED = TRUE )
      =>
    SEND_ENABLED = TRUE ) ∨
  SEND_ENABLED = FALSE )
 )
```

The INV = TRUE invariant tells Rodin's proof engines that every event must satisfy INV. Therefore, Rodin generates invariant proof obligations for each event that sets or modifies SEND_EXECUTED, ROLLBACK_EXECUTED, or RECEIVE_EXECUTED. Only when one discharges all the generated proof obligations with Rodin is the model considered to be sound. To achieve the same effect for the "enabled" variables involved in the invariant, one needs to tie the variable to another invariant. For instance, we have added the SEND_ENABLED = BOOL(BEAT = FALSE) invariant to the EVENT-B model of heartbeat. Rodin generates an invariant proof obligation for every event that sets the BEAT variable. Therefore, for every event implementation, SEND_ENABLED is TRUE if and only if BEAT is FALSE.

We show below an extended version of the RECEIVE event that was introduced in Section 4. The version here includes event actions that set the "enabled" and "executed" LTL temporal variables. These assignments are the result of applying the translation steps discussed here for the INV invariant. They are marked as // @inv.

```
EVENT RECEIVE
WHERE
 @grd1 BEAT = TRUE ∧ TIME > 0
THEN
 @act1 BEAT := FALSE
 @act2 TIME := 0
 @act3 SEND_ENABLED := TRUE // @inv
 @act4 RECEIVE_EXECUTED := TRUE // @inv
 @act5 SEND_EXECUTED := FALSE // @inv
 @act6 ROLLBACK_EXECUTED := FALSE // @inv
 // more actions here
END
```

Besides RECEIVE, INV also refers to SEND and ROLLBACK. Section 7 presents the code for SEND.

7 CERTIFIED CODE SYNTHESIS

We discuss here the certified code generated for the SEND event of the HEARTBEAT machine and show how this code can be unit-tested. We show first a more complete version of the SEND event presented in Section 4 that includes event actions that are the result of applying the translation steps discussed in Section 6. Guard @grd1 below states that the monitoring system can only send the monitored system a beat if this one has already acknowledged any previous beat sent to it. Action @act1 disables the SEND event. Action @act2 sets the clock. Action @act4 corresponds to step 4 in the LTL-to-EVENT-B algorithm in Section 6, action @act5 to step 6, and action @act6 to step 7.

```
EVENT SEND
WHERE
 @grd1 BEAT = FALSE
THEN
 @act1 BEAT := TRUE
 @act2 TIME := TSLOT
 @act3 SEND_EXECUTED := TRUE // @inv
 @act4 SEND_ENABLED := FALSE // @inv
 @act5 ROLLBACK_EXECUTED := FALSE // @inv
 @act6 RECEIVE_EXECUTED := FALSE // @inv
 ... // more actions here
END
```

EVENTB2JAVA translates each event E as a Java class that includes two methods, a `guard_e` and a `run_e` method, for the translation of the event guards and the event actions, respectively. EVENTB2JAVA generates the two Java method implementations below for the SEND event in the HEARTBEAT machine.

```
public boolean guard_send() {
  return get_beat().equals(false);
}

public void run_send(){
 if(guard_send()) {
  // some statements
  set_beat(true);
  set_time(Tslot);
  set_send_executed(true);
  set_send_enabled(false);
  set_rollback_executed(false);
  set_receive_executed(false);
  // more statements
 }
}
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3168672, IEEE Transactions on Software Engineering

The two generated methods are certified implementations of the SEND event in the following ways. $(i.)$ All the HEARTBEAT machine proof obligations, including those related to the SEND event, are discharged with Rodin. This ensures that the EVENT-B model of the HEARTBEAT machine is sound. $(ii.)$ EVENTB2JAVA produces a sound translation of EVENT-B models [10]. $(iii.)$ We have unit-tested the Java implementation to check if it behaves as expected (See Section 7.1).

7.1 Unit Testing

We write all the unit tests manually, following the syntax of the LTL formulae that we want to test. And, we use the Eclipse IDE to run the unit tests automatically. Let us start by demonstrating how to unit-test the LTL trace property below.

after send **executed**
(
always send **disabled**
)

This trace property is an invariant, however, EVENTB2JAVA does not translate EVENT-B invariants as code that one can run directly. Instead, we need to use and run the Java implementation of the events involved in the invariant to unit-test the invariant in Java. Unit-testing does not seek to check for soundness of the implementation (this was already achieved with Rodin), but to check if the implementation works as we expected. The unit-test below checks that the HEARTBEAT machine can send a beat right after it is created (first `assert` instruction), yet it cannot send a second beat right after the first one is sent (second `assert` instruction).

```
// the system can beat right after it is created,
// but cannot beat a second time immediately after
@Test
public void send_test_01() {
 send s = new send(machine);
 assertTrue(s.guard_send());
 s.run_send();
 assertFalse(s.guard_send());
}
```

Now, let us extend the previous LTL trace property as below.

after send **executed**
(
always send **disabled**
unless rollback **executed** , receive **executed**
)

We now extend the previous unit test with the last four instructions below. They check that every time the ROLLBACK event runs, the SEND event is enabled (its guard holds). We have created a similar testing code for the RECEIVE event (not shown here).

```
// ... it cannot beat a second time,
// unless it rolls back to a safe state
@Test
public void send_test_02() {
 // it cannot beat a second time,
 send s = new send(machine);
 assertTrue(s.guard_send());
 s.run_send();
 assertFalse(s.guard_send());
 // unless it rolls back
 rollback rb = new rollback(machine);
 assertTrue(rb.guard_rollback());
 rb.run_rollback();
 assertTrue(s.guard_send());
}
```

8 EVALUATION

To evaluate the *usability* of our approach we will demonstrate how the code generated for the heartbeat resilience tactic is interfaced and integrated with the implementation of an autonomous vehicle system (AVS). By usability we mean that the system can be used (executed) for the purpose it was designed for and has some *intended behaviour*, hence, correctness is part of our definition of usability. Checking the correct behaviour of the interaction of two multi-threaded applications - heartbeat and the AVS - is, of course, challenging and error-prone. We chose to model the AVS in EVENT-B, generate certified code for it, and interface the code with the heartbeat implementation.

In what follows, we describe: $(i.)$ the EVENT-B model of the autonomous vehicle, $(ii.)$ its Java implementation, and $(iii.)$ the code instrumentation we have carried out to interface the heartbeat to the AVS implementation.

8.1 The Autonomous Vehicle System (AVS)

The vehicle is an autonomous vehicle that runs over a *lane* that has a *finish line* and lane borders. Lanes can include obstacles and other vehicles called *opponents*. The goal of the vehicle is to reach the finish line in the least amount of time possible, hence avoiding collisions with opponents and obstacles. Therefore, two important aspects of the physics involved in the simulation of the autonomous vehicle are kinetic movement and collision detection. We have modeled a simple two-dimensional collision-free algorithm (not shown here) in EVENT-B to detect the collision of two objects with a uniformly accelerated movement. Table 1 shows the structure of the two EVENT-B machines of the AVS and the respective observations of each machine.

Machine	Observations
STATIC	objects, vehicles, obstacles, positions, lanes
KINETIC	velocity, friction, acceleration, collisions

TABLE 1
AVS machine structure

Figure 3 shows an excerpt of the AVS in EVENT-B. It shows the EVENT-B encoding of the UPDATE_POSITION event which updates the position of a vehicle after some *Elapsed* time. This machine combines observations and events of the two machines in Table 1. Carrier sets OBJECTS and LANES represent all the possible objects and lanes in the modelled system. Variables OBJECTS and LANES are the current objects and the existing lanes, respectively.

Invariants OBSTACLES ∪ VEHICLES = OBJECTS and OBSTACLES ∩ VEHICLES = ∅ ensure that OBSTACLES and VEHICLES form a mathematical *partition* of OBJECTS. Every object has a horizontal and vertical position, POSX and

```
MACHINE VEHICLE
SEES CTXT
VARIABLES OBJECTS LANES OBSTACLES VEHICLES WIDTH HEIGHT
  POSX POSY VEL ACC DRIFT SCORE COLLIDED ACTIVE
INVARIANTS
  OBJECTS ⊆ OBJECTS                LANES ⊆ LANES
  OBSTACLES ∪ VEHICLES = OBJECTS   OBSTACLES ∩ VEHICLES = ∅
  POSX ∈ OBJECTS → ℤ               POSY ∈ OBJECTS → ℤ
  WIDTH ∈ OBJECTS → ℕ              HEIGHT ∈ OBJECTS → ℕ
  VEL ∈ VEHICLES → ℤ               ACC ∈ VEHICLES → ℤ
  DRIFT ∈ VEHICLES → {-1, 0, 1}    SCORE ∈ VEHICLES → ℤ
  COLLIDED ∈ VEHICLES → BOOL
  ACTIVE ∈ (OBSTACLES ∪ VEHICLES) → BOOL
EVENTS

 EVENT INITIALISATION
 THEN
  OBJECTS := ∅  LANES := ∅  OBSTACLES := ∅  VEHICLES := ∅
  POSX := ∅  POSY := ∅  WIDTH := ∅  HEIGHT := ∅
  VEL := ∅  ACC := ∅  DRIFT := ∅  COLLIDED := ∅  ACTIVE := ∅
 END

 EVENT UPDATE_POSITION
  ANY V  ELAPSED
  WHERE
   V ∈ VEHICLES
   ELAPSED ∈ ℕ
  THEN
   POSX(V) := POSX(V) + DRIFT(V) × ELAPSED × 50 ÷ 1000
   POSY(V) := POSY(V) + VEL(V) × ELAPSED ÷ 1000
  END
 // more events
END
```

Fig. 3. Vehicle EVENT-B machine

POSY, which are total functions (the → symbol) mapping some object, e.g. a vehicle or an obstacle, into an integer number that represents the position for the respective axis. Objects are two-dimensional objects with a HEIGHT and a WIDTH. Variables VEL (velocity), ACC (acceleration), and DRIFT (bending from the straight-up position) model the kinetic of the vehicle. Function COLLIDED is a total Boolean function that returns TRUE of a collided vehicle. The machine VEHICLE includes (not shown here) events such as ADD_VEHICLE, which creates a vehicle, UPDATE_VEL which updates the velocity of a vehicle, SET_MAXVEL which sets the maximum velocity of a vehicle, among others.

8.2 Multi-Threaded Mechanisms

EVENTB2JAVA multi-threaded mechanism for the AVS is summarised below. EVENTB2JAVA implements each event as a Java `Thread`, spawn within a for-loop using the standard Java `start()` method. The events communicate through a shared memory mechanism, and indeed, the parameter `this` below is a reference to the KINETIC machine class implementation, which is shared by all the events.

```
events = new Thread[nevents];
events[0] = new SET_MAXVEL_Threaded(this, ...);
events[1] = new SET_DRIFT_Threaded(this, ...);
 ...
events[nevents-1] = ...

for (int i = 0; i < n_events;i++){
 events[i].start();
}
```

EVENTB2JAVA implements a similar multi-threaded synchronisation mechanism for heartbeat. Therefore, once the events for both systems start running, they compete to enter their *Critical Sections*, that is, the event actions. Events execute atomically using a Java `ReentrantLock` concurrency mechanism.

For the evaluation of the usability of the approach presented in this paper, we next show how the SEND-vs-RECEIVE heartbeat mechanism can monitor selected parts of the implementation of the AVS to check if it is running or has crashed. This is achieved through code instrumentation.

8.3 Code Instrumentation

We have used EVENTB2JAVA to translate the VEHICLE machine into Java. EVENTB2JAVA translates the UPDATE_POS event (and any other event) as two methods: `run_update_position` and `guard_update_position`. Hence, interfacing the AVS to the heartbeat architectural tactic implementation requires us to select (at least) two events of the AVS between which we want to check if the vehicle system is alive or not, and relate them to the SEND and RECEIVE events of the heartbeat, respectively. For instance, if we want to check if the AVS is alive between the moment the vehicle is created and the moment its position is eventually updated, then we need to instrument the `run_add_vehicle` method to include code that calls `run_send()`, and the `run_update_position` to make a call to the `run_receive()` method of heartbeat.

Code instrumentation for updating a vehicle's position is shown next for the RECEIVE part of heartbeat. The `update_position` Java class includes a `kinetic` reference to the AVS implementation, and a `heartbeat` reference to the heartbeat system implementation, which are both initialized within the class constructor.

The instrumentation of the `run_add_vehicle(...)` method is shown below. It is placed at the end of the method, just after adding the vehicle. Therefore, `run_send(...)` is called on an object of type `send`. Notice that the call might or might not execute depending on whether `guard_send(...)` holds or not. However, the multi-threaded mechanism presented in Section 8.2 guarantees that eventually `guard_add_vehicle(...)` and `guard_send(...)` will both hold, and so `run_send()` will execute.

```
public void run_add_vehicle(Integer H,
  Integer Vehicle, Integer W, Integer X,
  Integer Y, Integer Max) {
  if(guard_add_vehicle(H,Vehicle,W,X,Y,Max)) {
    // code that adds a vehicle

    // instrumentation begins
    send s = new send(heartbeat);
    s.run_send();
    // instrumentation ends
  }
}
```

The second part of the code instrumentation occurs at the end of method `run_update_position(...)`. An instance of an object `r` of type `receive` is created, and the method `guard_receive(...)` is called on it. If the method may run, thus `r.guard_receive(...)` holds, then some code that reacts to that event executes.

```
public class update_position extends Thread {
 private Kinetic kinetic;
 private HeartBeat heartbeat;

 public update_position(Kinetic k, HeartBeat h)
     { ... }

 public boolean guard_update_position(
     Integer Elapsed, Integer V) { ... }

 public void run_update_position(Integer Elapsed,
                                 Integer V) {
  if(guard_updated_position(Elapsed, V)) {
    // code that updates a vehicle's position

    // instrumentation begins
    receive r = new receive(heartbeat);
    if(r.guard_receive()) {
     // code that reacts to vehicle is alive
    }
    // instrumentation ends
  }
 }

 public void run() {
  while(true) {
   Integer Elapsed = ... // elapsed time
   Integer V = ... // the vehicle
   kinetic.lock.lock(); // Critical Section begins
   run_update_position(Elapsed,Obj);
   kinetic.lock.unlock(); // Critical Section ends
  }
 }
}
```

The class resulting from translating an event includes a third method, a `run()` method that overrides the corresponding Java `Thread` method. Method `run_update_position` is atomic, it is executed within `lock` and `unlock` instructions using a `Reentrant` lock from the Java `concurrent` Library.

Below is the output printed for one of the executions of the instrumented code. The process is non-deterministic, and outputs vary with different executions. The AVS instrumentation prints `Exit` after updating a vehicle position.

```
ADD_VEHICLE executed H: 63 Vehicle: 70
                    W: 63 X: 51 Y: 72 Max: 11
send executed
UPDATE_POSITION executed Elapsed: 27 Obj: 1
Exit
```

8.4 Size of the Model

Table 2 shows some statistics about the size of the AVS model. LOC stands for Lines of (EVENT-B) Code, and POs for the number of Proof Obligations generated by the Rodin platform. The EVENT-B model includes 1 abstract machine and 1 refinement machine. Rodin's provers discharged all the POs automatically for the abstract machine (STATIC). They discharged 91 percent of them for the concrete machine (KINETIC) automatically. We discharged the rest 9 percent manually by just running one of the proof tactics that Rodin offers. For the concrete machine, Rodin failed to discharge automatically some of the POs related to the multiplication and division of integers.

8.5 The Rest of Heartbeat's LTL Properties

We have encoded 6 LTL temporal properties for *heartbeat* in EVENT-B, 6 for *pingecho*, 1 for *exception*, 3 for *timestamp*, and 2 for *discretionary access control*.

Machine	LOC	# POs	% Aut.
STATIC	158	57	100
KINETIC	202	56	91

TABLE 2
AVS: EVENT-B efforts

We show below the remaining 5 LTL temporal properties for heartbeat. The first LTL property is similar to the one introduced in Section 5 except that receive becomes enabled rather than send disabled after send is executed.

after send **executed**
(
always receive **enabled**
unless hasFailed **enabled** , fail **executed**
)

The LTL property below relates fail with hasFailed, therefore, hasFailed is enabled for a failing system. rollback always set the system back to normal.

after fail **executed**
(
always hasFailed **enabled**
unless rollback **executed**
)

rollback is always enabled, even for failing systems. rollback can always run under any circumstances.

after hasFailed **enabled**
(
always rollback **enabled**
)

A failing system cannot send a beat, unless the system rolls-back to a safe state.

after hasFailed **enabled**
(
always send **disabled**
unless rollback **executed**
)

A failing system cannot receive a beat, unless the system rolls-back to a safe state.

after hasFailed **enabled**
(
always receive **disabled**
unless send **executed**
)

8.6 The Rest of the Architectural Tactics

pingecho is a fault detection architectural tactic similar to *heartbeat*, however, pingecho's rollback (recovery) mechanism can only execute if the system has failed.

exception is a tactic that raises exceptions and uses a particular event as exception handling (recover) mechanism. Exceptions can be raised and handled any time, hence, the

tactic does include a notion of time. We show a typical LTL trace property for *exception* below.

```
after cyber_exception executed
(
 always cyber_exception_handling enabled
  unless cyber_exception_handling executed
)
```

timestamp is a tactic that is used to detect incorrect sequence of events. It keeps a record of the *time before* and the *time after* an event occurs; the system fails when the time before is greater than the time after. *timestamp*'s recovery mechanism sets the time to zero. The property below states that the system can rollback to a safe state any time a failure is produced.

```
after hasFailed executed
(
 always recover enabled
  unless recover executed
)
```

priority implements two task queues with respective priorities for the elements in each queue. A first event sends tasks to the first queue, and second event fetches them (depending on their priorities) and places them in the second queue, which are eventually run depending on their priorities. *response* implements a simpler version of *priority* with queues but without priorities.

limit access implements a discretionary access control mechanism with subjects, resources, and permissions. The owner of the resource has all the permissions over it and can grant other users permissions.

The first LTL property below says that after added, a resource can eventually be deleted by its owner.

```
after add_resource executed
(
 always delete_resource enabled
  unless delete_resource executed
)
```

The LTL property below says that after granted, a permission can eventually be revoked by its owner.

```
after grant_permission executed
(
 always revoke_permission enabled
  unless revoke_permission executed
)
```

9 Related Work

In [27], the authors present an EVENT-B refinement trace-based semantics for timed systems. Timed systems are modelled in EVENT-B with the aid of some trigger-response patterns. The proposed semantics caters for concepts such as deadlock freedom and convergence. Our LTL extension in Section 5 is defined in terms of temporal patterns. As future work, we plan to include patterns such as the ones discussed by the authors.

Some previous works focus on architectural-first approaches to cyber-resilience [22], [23]. The general idea is to bring cyber-resilience into a system starting from its design all the way down to code as opposed to deal with cyber-security breaches once the system is deployed. Our approach is a architectural-first approach, and, indeed, the autonomous vehicle system (AVS) that we discussed in Section 8 was originally designed in EVENT-B.

In [21], the authors present a formal methods approach to the specification and verification of security requirements. Security requirements are first specified in logic, then the Alloy model-checking tool [15] is used to check the validity of the logical requirements; if the requirements do not hold, then the model-checker counter-examples are used to suggest valid security policies to the user. Our work differs from theirs in several aspects. Our approach is not based on model-checking techniques and thus we do not use error traces to evolving security requirements. Our approach not only works at the logic-model level but also at the implementation level so software developers can execute temporal logic specifications directly in Java.

In [6], the authors present an approach to verify semi-structured requirements in EVENT-B. Requirements are first classified as data-oriented, constrained-oriented, event-oriented, flow-oriented, and "others". Then, the authors use UML-B, ERS (Event Refinement Structure) diagrams, and structured English to represent the requirements. As pointed out by the authors, their work does not provide support to the formalization of fairness and timing properties (since EVENT-B lacks native support to temporal properties). Our work supplements their work on the formalization and validation of temporal requirements in EVENT-B.

In [8], the authors discuss some research efforts leading to integrate formal methods and agile methodologies. Most of these efforts try to make formal methods more agile. The authors argue that formal methods are oftentimes used to perform sanity checks of specifications and validation of formal models. Some of the areas where agility benefits from the work presented in this paper are unit-testing, incremental development and refactoring. The authors in [19] go further to establish a comparison between formal and agile concepts, e.g., refinement vs refactoring, proofs vs tests, abstract vs concrete, and verification vs validation.

In [25], the authors propose an LTL (Linear Temporal Logic) extension to the Java Modelling Language (JML). They describe the semantics of both safety and liveness properties, however, they do not present appropriate verification techniques for liveness properties. Some parts of our work on LTL are similar to theirs. The advantage of our work is that EVENTB2JAVA translates EVENT-B models to JML too so we can evaluate several LTL formalizations, in EVENT-B but also in Java and JML.

Schneider *et al.* extend the previous work and demonstrate how temporal properties can carry through the refinement mechanism of EVENT-B [24]. Their work can be generalized to events that can be split into several events in the event refinement chain.

10 Conclusions

Cybersecurity efforts tend to protect the right of people, institutions, or states to access their own data privately, yet, often do not account for safety, e.g., for accidental or inadvertent failures that might result from the misuse of

software or hardware. Safety is important as oftentimes security breaches are originated from human errors. A system is safe if it does not exhibit some bad behaviour, oftentimes modelled as system invariants that must hold over state traces. This paper presented an approach for program synthesis of architectural tactics. Our approach is based on formal methods techniques with EVENT-B. The paper mainly focuses on architectural tactics for availability, but we have also worked on performance and security. We focus on two cyber-resilience aspects, namely, *detection* and *recovery*. *Detection* is related with being able to determine if a failure has occurred, and *recovery* is related to transition back to a safe state when a failure occurs.

Section 8 discusses the *usability* of our approach. We want to discuss here two further evaluation aspects, namely, soundness and performance. A soundness proof of the LTL EVENT-B extension, and the translation from the LTL extension to EVENT-B would require the definition of either a *shallow* or *deep* embedding [14], [10] of both EVENT-B and LTL in Logic, and then the execution of proofs that demonstrate that the translation preserves the validity of the LTL properties.

Notice that because our LTL extension directly translates into EVENT-B as Boolean variables and event assignments, we do not need to modify EVENT-B's refinement mechanism to be able to adopt the extension. It would only require a parser or a preprocessor to map the LTL formulae into EVENT-B's syntax.

Regarding perfomance, EVENTB2JAVA is not meant to produce fast code, but correct code. Hence, our paper does not claim that the code that we generate for architectural tactics is faster than pre-existing implementations. We rather claim that our approach can be integrated to existing software development methodologies to design and conceive resilient system architectures.

Broken Access Control [1] was ranked fifth in 2017 OWASP top 10 ranking and first in 2021. We plan to extend our *limit access* model discussed at the end of Section 8.6 for mandatory access control and conduct a case study on the modelling and implementation of a system based on access permissions.

Our *security authentication* EVENT-B model is still very simple. We plan to extend it by modeling hashing and dehashing as functions and incorporating various of the cryptographic properties of hash functions in the logic of EVENT-B.

REFERENCES

[1] OWASP TOP 10. Broken Access Control. https://owasp.org/Top10/A01_2021-Broken_Access_Control/, 2021.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA, 2010.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[4] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1,2):1–24, 2007.

[5] A. Alexeev, D. Henshel, K. Levitt, P. Mcdaniel, B. Rivera, S. Templeton, and Mike Weisman. Constructing a Science of Cyber-Resilience for Military Systems, 2017.

[6] Eman Alkhammash, Michael J. Butler, Asieh Salehi Fathabadi, and Corina Cîrstea. Building Traceable Event-B Models from Requirements. *Science of Computer Programming*, 111:318–338, 2015.

[7] Leonard J. Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2012.

[8] Sue Black, Paul Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, 2009.

[9] Deborah J. Bodeau and Richard D. Graubart. Cyber Resiliency Design Principles. The MITRE Corporation, May 2017.

[10] Néstor Cataño and Shigeo Nishi. Soundness Proof of EventB2Java. In *Seventh Latin-American Symposium on Dependable Computing (LADC)*, volume 0 of *IEEE Digital Library*, pages 25–34, Cali, Colombia, October 19–21 2016. IEEE Computer Society.

[11] Néstor Cataño and Víctor Rivera. EventB2Java: A code generator for Event-B. In *Nasa Formal Methods (NFM)*, volume 9690 of *LNCS*, pages 166–171, Minneapolis, MN, USA, June 7–9 2016. Springer.

[12] Néstor Cataño, Timothy Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. Translating B Machines to JML Specifications. In *27th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, pages 1271,1277, Trento, Italy, March 26-30 2012. ACM.

[13] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[14] Michael Gordon. Mechanizing Programming Logics in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439, New York, NY, USA, 1989. Springer-Verlag.

[15] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The Mit Press, Massachusetts Institute of Technology, 2006.

[16] A. Kott, Benjamin A. Blakely, D. Henshel, Gregory Wehner, James Rowell, Nathaniel Evans, Luis Muñoz-González, Nandi O. Leslie, Donald W. French, Donald Woodard, K. Krutilla, A. Joyce, I. Linkov, C. M. Machuca, J. Sztipanovits, H. Harney, Dennis Kergl, P. Nejib, Edward Yakabovicz, S. Noel, Tim Dudman, P. Trepagnier, Sowdagar Badesha, and Alfred Møller. Approaches to Enhancing Cyber Resilience: Report of the North Atlantic Treaty Organization (NATO) Workshop IST-153. *ArXiv*, abs/1804.07651, 2018.

[17] Sergii Lysenko, Kira Bobrovnikova, Piotr Gaj, Tomas Sochor, and Iryna Forkun. Resilient Computer Systems Development for Cyberattacks Resistance. volume 2853 of *CEUR Workshop Proceedings*, pages 353–361. CEUR-WS.org, 2021.

[18] Amir Pnueli. The temporal logic of programs. In *Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society Press.

[19] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In Shail Arora and Gary T. Leavens, editors, *Companion to OOPSALA*, pages 999–1006. ACM, 2009.

[20] Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code Generation for Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(1):31–52, February 2017.

[21] Quentin Rouland, Brahim Hamid, Jean-Paul Bodeveix, and Mamoun Filali. A Formal Methods Approach to Security Requirements Specification and Verification. In Jun Pang and Jing Sun, editors, *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*, pages 236–241. IEEE, 2019.

[22] Joanna C. S. Santos, Selma Suloglu, Joanna Ye, and Mehdi Mirakhorli. Towards an Automated Approach for Detecting Architectural Weaknesses in Critical Systems. In *ICSE'20: International Conference on Software Engineering*, pages 250–253. ACM, 2020.

[23] Joanna C. S. Santos, Katy Tarrit, and Mehdi Mirakhorli. A Catalog of Security Architecture Weaknesses. In *ICSA'17: International Conference on Software Architecture*, pages 220–223. IEEE Computer Society, 2017.

[24] Steve A. Schneider, Helen Treharne, Heike Wehrheim, and David M. Williams. Managing LTL Properties in Event-B Refinement. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, volume 8739 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2014.

[25] Kerry Trentelman and Marieke Huisman. Extending JML Specifications with Temporal Logic. In Hélène Kirchner and Christophe

Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *Lecture Notes in Computer Science (LNCS)*, pages 334–348. Springer, 2002.

[26] WenXuan Wang, Jun Hu, JianChen Hu, JieXiang Kang, Hui Wang, and ZhongJie Gao. Automatic Test Case Generation from Formal Requirement Model for Avionics Software. In *2020 6th International Symposium on System and Software Reliability (ISSSR)*, pages 12–20, 2020.

[27] Chenyang Zhu, Michael Butler, and Corina Cirstea. Towards Refinement Semantics of Real-Time Trigger-Response Properties in Event-B. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–8, 2019.