A Parametric Rely-guarantee Reasoning Framework for Concurrent Reactive Systems

Yongwang Zhao[1], David Sanán[2], Fuyuan Zhang[2], and Yang Liu[2]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
[2] School of Computer Science and Engineering, Nanyang Technological University, Singapore
Email: zhaoyw@buaa.edu.cn

Abstract. Reactive systems are composed of a well defined set of event handlers by which the system responds to environment stimulus. In concurrent environments, event handlers can interact with the execution of other handlers such as hardware interruptions in preemptive systems, or other instances of the reactive system in multicore architectures. The rely-guarantee technique is a suitable approach for the specification and verification of reactive systems. However, the languages in existing rely-guarantee implementations are designed only for "pure programs", simulating reactive systems makes the program and rely-guarantee conditions unnecessary complicated. In this paper, we decouple the system reactions and programs using a rely-guarantee interface, and develop *PiCore*, a parametric rely-guarantee framework for concurrent reactive systems. *PiCore* has a two-level inference system to reason on events and programs associated to events. The rely-guarantee interface between the two levels allows the reusability of existing languages and their rely-guarantee proof systems for programs. In this work we show how to integrate in *PiCore* two existing rely-guarantee proof systems. This work has been fully mechanized in Isabelle/HOL. As a case study, we have applied *PiCore* to the concurrent buddy memory allocation of a real-world OS, providing a verified low-level specification and revealing bugs in the C code.

1 Introduction

Nowadays high-assurance systems are often designed as *concurrent reactive systems* (CRSs) [3]. CRSs react to their computing environment by executing a sequence of commands under an input event. Some examples of CRSs are operating systems (OSs), control systems, and communication systems, whose implementation follows an event-driven paradigm. The rely-guarantee technique [16] represents a fundamental approach to compositional reasoning of *concurrent programs* with shared variables, where programs are represented in imperative languages with extensions for concurrency. Whilst rely-guarantee provides a general framework and can certainly be applied for CRSs, the languages in existing mechanizations of rely-guarantee (e.g. [28,23,18,20,24]) are imperative and designed only for pure programs, i.e, programs following a flow of procedure calls from an entry point. Examples of reactive systems mentioned above are far more complex than pure programs because they involve many different agents and also heavy interactions with their environment. Without statements for such system behavior, we often use imperative programs to simulate them and make the formal specifi-

cation cumbersome, in particular for rely-guarantee conditions. The motivation of this paper will be presented in detail in Section 2.

In this paper, we propose *PiCore*, a two-level event-based rely-guarantee framework for CRSs (Section 3). *PiCore* detaches the specification and the logic of the reactive aspect of systems from event behaviours. Rather than creating yet another framework for modelling and reasoning on events behaviour, *PiCore* allows to reuse existing rely-guarantee frameworks. The top level introduces the notion of "events" [6,2] into the rely-guarantee method for system reactions. This level defines the events composing a system, and how and when they are triggered. It specifies the language, semantics, and mechanisms to reason on sequences of events and their execution conditions. The second level focuses on the specification and reasoning of the behaviour of the events in the first level. *PiCore* parameterizes the second level using a rely-guarantee interface and thus allows to easily reuse existing rely-guarantee frameworks. This design allows *PiCore* to be independent of the language used to model the behaviour of events.

We have integrated two existing languages and their rely-guarantee proof systems into the *PiCore* framework. As a result we create two instances of *PiCore*: π*IMP* and π*CSimpl* (Section 4). π*IMP* integrates the *HOL-Hoare_Parallel* library in Isabelle/HOL that uses a general imperative language [23]. π*CSimpl* integrates the *CSimpl* language in [24]. *CSimpl* is a generic and realistic imperative language by extending *Simpl* [25] and providing a rely-guarantee proof system in Isabelle/HOL. *Simpl* is able to represent a large subset of C99 code and has been applied to the formal verification of seL4 OS kernel [17] at C code level.

We have developed the *PiCore* framework and its integration with the two languages in Isabelle/HOL, the sources are available at http://lvpgroup.github.io/picore/. As a case study, we have applied *PiCore* to the formal specification and mechanized proof of the concurrent buddy memory allocation of a real-world OS, Zephyr RTOS [1] (Section 5). The formal specification represented in π*IMP* is fine-grained providing a high level of detail. It closely follows the Zephyr C code, covering all the data structures and imperative statements present in the implementation. We use the rely-guarantee proof system in π*IMP* for the formal verification of functional correctness and invariant preservation in the model, revealing three bugs in the C code.

2 Motivation and Approach Overview

Reactive systems respond to continuous stimulus from their computing environment [12] by changing their state and, in turn, affecting their environment by sending back signals to it or initiating other operations. We consider *concurrent reactive systems (CRSs)*, which may involve many different competitive agents executing concurrently with shared resources due to multicore setting, task preemption or embedded interrupts, e.g. concurrent OS kernels [7,27] and interrupt driven control systems, where the execution of handlers is not atomic. Moreover, the configuration and context of underlying hardware of systems are not usually encoded in programs, which represent only a portion of the whole system behaviour. For instance, although interrupt handlers (e.g. kernel services and scheduling) in OS kernels are programmed in the C language,

when and how interrupts are triggered and which handlers are invoked to react with an interrupt are out of the handler code.

In the setting of imperative languages, CRSs are usually modelled as the parallel composition of reactive systems, each of which is simulated by a *while(true)* loop program with reading data from its environment and invoking the relevant handlers in the loop body (e.g. [4]). First, it is non-deterministically decided by the environment which event handler is triggered and what are the arguments of the handler for this triggering. Second, some critical properties, such as noninterference of OS kernels [21], concern the traces of reactions rather than the program states only. Without native supports in language semantics, the *while* loop programs have to use auxiliary logical/program variables to simulate the two non-determinisms together and store the event context of each reactive system. This will make the program and the rely-guarantee conditions unnecessary complicated, in particular for realistic CRSs with many event handlers.

The reason of above problems are lack of a rely-guarantee approach for system reactions and, as a result, the mixture of system and program behavior together. In this paper, we take the level of abstraction and reusability of the rely-guarantee method a step further by decoupling the two levels using a rely-guarantee interface. The result is a flexible rely-guarantee framework for CRSs, which is able to integrate existing rely-guarantee implementations at program level being unchanged. At system reaction level, we consider a reactive system as a set of event handlers called *event systems* responding to stimulus from the environment. Fig. 1 illustrates an *event*, which has an event name, a list of input parameters, a guard condition to determine the conditions triggering the event, and an imperative program as its body. In addition to the input parameters, an event has a special parameter κ which indicates the execution context, e.g. the thread invoking the service and the external devices triggering the interrupt. The execution of an event system concerns the continuous evaluation of the guards of the events with special arguments. From the set of events for which the associated guard condition holds in the current state, one event is non-deterministically selected to be triggered, and its body executed. After the event finishes, the evaluation of the guards starts again looking for the next event to be executed. We call *reactive semantics* to the semantics of event systems and they store the event context, i.e. which event is currently executing. A CRS is modeled as the *parallel composition* of event systems which interleaves their execution.

EVENT *alloc [Ref p, Nat size, Int tout]* @ κ
WHEN
  $p \in$ ´*mem-pools* $\wedge$ *timeout* ≥ -1
THEN
  .....
  IF timeout $>$ *0 THEN*
    ´*endt* := ´*endt*(*t* := ´*tick* + *timeout*)
  FI;
  .....
END

Fig. 1: An Example of Event

As shown in the Zephyr case study in Section 5, the formal specification of CRSs with support for reactions and their composition is much simpler than those represented by pure programs. Furthermore, *PiCore* supports verifying total correctness of events, whose execution is usually assumed to be terminating, as well as the properties of event systems, whose execution is often non-terminating.

3 *PiCore*: The Rely-guarantee Framework

This section introduces the event language in *PiCore* as well as its rely-guarantee proof system, the soundness of proof rules and invariant verification.

3.1 The Event Language

Event:
$\mathcal{E}$ ::= **Event** (l, g, P) $(Basic\ Event)$
| $\lfloor P \rfloor$ $(Triggered\ Event)$
Event System:
$\mathcal{S}$ ::= $\{\mathcal{E}_0, ..., \mathcal{E}_n\}$ $(Event\ Set)$
| $\mathcal{E} \triangleright \mathcal{S}$ $(Event\ Sequence)$
Parallel Event System:
$\mathcal{PS}$ ::= $\mathcal{K} \to \mathcal{S}$

Fig. 2: Abstract Syntax of *PiCore*

The abstract syntax of *PiCore* and its semantics are shown in Fig. 2 and 3 respectively. The syntax for events distinguishes basic events pending to be triggered from already triggered events that are under execution. A basic event is defined as **Event** (l, g, P), where l is the event name, g the guard condition, and P the body of the event. **Event** (l, g, P) is triggered when g holds in the current state. Then, its body begins to be executed (BASICEVT rule in Fig. 3) and it becomes a triggered event $\lfloor P \rfloor$. The execution of $\lfloor P \rfloor$ just simulates the program P (see TRGDEVT rule in Fig. 3). $\perp$ is the notation to represent the termination of programs. Instead of defining a language for programs, *PiCore* reuses existing languages and their rely-guarantee proof systems, which will be discussed in Section 4. Events are parametrized in the meta-logic with the list of input parameters of the event $plist$, and the event system identifier κ that the event belongs to. These parameters are not part of the syntax of events to make the rely and guarantee relations more flexible, allowing to define different instances of the relations for different values of $plist$ and κ.

The event system has two forms that we call *event sequence* and *event set*. The event sequence models the sequential execution of events. The execution of an event set consists of a continuous evaluation of the guards of the events in the set. When there is an event **Event** (l, g, P) in the set where g holds in the current state, the event is triggered (EVTSET rule) and its body P executed (EVTSEQ1 rule). After P finishes, the evaluation of the guards starts again looking for the next event to be executed (EVTSEQ2 rule). A CRS is modeled by a parallel composition of event systems with shared states. It is a function from $\mathcal{K}$ to event systems, where $\mathcal{K}$ indicates the identifiers of event systems. This design is more general and could be applied to track executing events. For instance, we use $\mathcal{K}$ to represent the core identifier in multicore systems.

The semantics of *PiCore* is defined via transition rules between configurations. We define a configuration $\mathcal{C}$ in *PiCore* as a triple $(\sharp, s, x)$ where $\sharp$ is a specification, s is a state, and $x : \mathcal{K} \to \mathcal{E}$ is an event context. The event context indicates which event is currently being executed in an event system k. Transition rules in events, event systems, and parallel event systems have the form $\Sigma \vdash (\sharp_1, s_1, x_1) \xrightarrow{\delta}_{\square} (\sharp_2, s_2, x_2)$, where $\delta = t@\kappa$ is a label indicating the type of transition, the subscript "$\square$" ($_e$, $_{es}$ or $_{pes}$) indicates the transition objects, and Σ is used for some static configuration for programs (e.g. an environment for procedure declarations). Here t indicates a program action c

$$\text{[BasicEvt]}\quad \frac{body(\alpha) \neq \bot \quad s \in guard(\alpha) \quad x' = x(k \mapsto \textbf{Event}\ \alpha)}{\Sigma \vdash (\textbf{Event}\ \alpha, s, x) \xrightarrow{\textbf{Event}\ \alpha@\kappa}_e (\lfloor body(\alpha) \rfloor, s, x')}$$

$$\text{[TrgdEvt]}\quad \frac{\Sigma \vdash (P, s) \longrightarrow_p (P', s')}{\Sigma \vdash (\lfloor P \rfloor, s, x) \xrightarrow{c@k}_e (\lfloor P' \rfloor, s', x)}$$

$$\text{[EvtSet]}\quad \frac{i \leq n \quad \Sigma \vdash (\mathcal{E}_i, s, x) \xrightarrow{\mathcal{E}_i@\kappa}_e (\mathcal{E}_i', s, x')}{\Sigma \vdash (\{\mathcal{E}_0,\ ...,\ \mathcal{E}_n\}, s, x) \xrightarrow{\mathcal{E}_i@\kappa}_{es} (\mathcal{E}_i' \triangleright \{\mathcal{E}_0,\ ...,\ \mathcal{E}_n\}, s, x')}$$

$$\text{[EvtSeq1]}\quad \frac{\Sigma \vdash (\mathcal{E}, s, x) \xrightarrow{t@\kappa}_e (\mathcal{E}', s', x') \quad \mathcal{E}' \neq \lfloor \bot \rfloor}{\Sigma \vdash (\mathcal{E} \triangleright \mathcal{S}, s, x) \xrightarrow{t@\kappa}_{es} (\mathcal{E}' \triangleright \mathcal{S}, s', x')}$$

$$\text{[EvtSeq2]}\quad \frac{\Sigma \vdash (\mathcal{E}, s, x) \xrightarrow{t@\kappa}_e (\lfloor \bot \rfloor, s', x')}{\Sigma \vdash (\mathcal{E} \triangleright \mathcal{S}, s, x) \xrightarrow{t@\kappa}_{es} (\mathcal{S}, s', x')}$$

$$\text{[Par]}\quad \frac{\Sigma \vdash (\mathcal{PS}(\kappa), s, x) \xrightarrow{t@\kappa}_{es} (\mathcal{S}', s', x') \quad \mathcal{PS}' = \mathcal{PS}(\kappa \mapsto \mathcal{S}')}{\Sigma \vdash (\mathcal{PS}, s, x) \xrightarrow{t@\kappa}_{pes} (\mathcal{PS}', s', x')}$$

Fig. 3: Operational Semantics of *PiCore*

or an occurrence of an event $\mathcal{E}$. $@\kappa$ means that the action occurs in event system κ. Environment transition rules have the form $\Sigma \vdash (\sharp, s, x) \xrightarrow{e}_{\square} (\sharp, s', x')$. Intuitively, a transition made by the environment may change the state and the event context but not the specification. The parallel composition of event systems is fine-grained since small steps in events are interleaved in the semantics of *PiCore*. This design relaxes the atomicity of events in other approaches (e.g., Event-B [2]).

A *computation* of *PiCore* is a sequence of transitions. We define the set of computations of all parallel event systems with static information Σ as $\Psi(\Sigma)$, which is a set of lists of configurations inductively defined as follows. The singleton list is always a computation (1). Two consecutive configurations are part of a computation if they are the initial and final configurations of an environment (2) or action transition (3). The operator $\#$ in $e\#l$ represents the insertion of element e in list l.

$$\begin{cases} (1)[(\mathcal{PS}, s, x)] \in \Psi(\Sigma) \\ (2)(\mathcal{PS}, s_1, x_1)\#cs \in \Psi(\Sigma) \Longrightarrow (\mathcal{PS}, s_2, x_2)\#(\mathcal{PS}, s_1, x_1)\#cs \in \Psi(\Sigma) \\ (3)\Sigma \vdash (\mathcal{PS}_2, s_2, x_2) \xrightarrow{\delta}_{pes} (\mathcal{PS}_1, s_1, x_1) \wedge (\mathcal{PS}_1, s_1, x_1)\#cs \in \Psi(\Sigma) \\ \qquad \Longrightarrow (\mathcal{PS}_2, s_2, x_2)\#(\mathcal{PS}_1, s_1, x_1)\#cs \in \Psi(\Sigma) \end{cases}$$

Computations for events and event systems are defined in a similar way. We use $\Psi(\Sigma, \mathcal{PS})$ to denote the set of computations of a parallel event system $\mathcal{PS}$. The function $\Psi(\Sigma, \mathcal{PS}, s, x)$ denotes the computations of $\mathcal{PS}$ starting up from an initial state s and event context x.

3.2 Rely-guarantee Proof System

We consider the verification of two different kinds of properties in the rely-guarantee proof system for reactive systems: pre and post conditions of events and invariants in the fine-grained execution of events. We use the former for the verification of functional correctness of the event, where the pre and post conditions have to be respectively satisfied only before and after the execution of the event. The latter is used on the verification of safety properties concerning the small steps inside events and that must be preserved by any internal step of the event. For instance, in the case of Zephyr RTOS, a safety

property is that memory blocks do not overlap each other even during internal steps of the *alloc* and *free* services. Other critical properties can also be defined considering the execution trace of events, e.g. noninterference [21,19,22].

A rely-guarantee specification in *PiCore* is a quadruple $\langle pre, R, G, pst\rangle$, where pre is the precondition, R is the rely condition, G is the guarantee condition, and pst is the post condition. The assumption and commitment functions are denoted by A and C respectively. For each computation $\varpi \in \Psi(\Sigma, \mathcal{E})$, we use ϖ_i to denote the configuration at index i. $\sharp_{\varpi_i}$, s_{ϖ_i}, and z_{ϖ_i} represents an element of $\varpi_i = (\sharp, s, x)$.

$$A(\Sigma, pre, R) \equiv \{\varpi \mid s_{\varpi_0} \in pre \wedge (\forall i < len(\varpi) - 1.\ (\Sigma \vdash \varpi_i \xrightarrow{e} \varpi_{i+1}) \longrightarrow (s_{\varpi_i}, s_{\varpi_{i+1}}) \in R)\}$$

$$C(\Sigma, G, pst) \equiv \{\varpi \mid\ (\forall i < len(\varpi) - 1.\ (\Sigma \vdash \varpi_i \xrightarrow{\delta}_e \varpi_{i+1}) \longrightarrow (s_{\varpi_i}, s_{\varpi_{i+1}}) \in G) \wedge (\sharp_{last(\varpi)} = \lfloor\bot\rfloor \longrightarrow s_{\varpi_n} \in pst)\}$$

We define validity of rely-guarantee specification for events as

$$\Sigma \models \mathcal{E}\ \mathbf{sat}\ \langle pre, R, G, pst\rangle \equiv \forall s, x.\ \Psi(\Sigma, \mathcal{E}, s, x) \cap A(\Sigma, pre, R) \subseteq C(\Sigma, G, pst)$$

Intuitively, validity represents that the set of computations $cpts$ starting at the configuration $(\mathcal{E}, s, x)$, with $s \in pre$ and environment transitions in a computation $cpt \in cpts$ belonging to the rely relation R, is a subset of the set of computations where action transitions belong to the guarantee relation G and if an event terminates, then the final states belongs to pst. Validity for event systems and parallel event systems are defined in a similar way.

Next, we present the rely-guarantee proof rules in *PiCore* and their soundness w.r.t the validity. The proof rules are shown in Fig. 4, which gives us a relational proof method for concurrent systems. We first define $stable(f, g) \equiv \forall x, y.\ x \in f \wedge (x, y) \in g \longrightarrow y \in f$. Thus, $stable(pre, rely)$ means that the precondition is stable when the rely condition holds. Rules may stability of the precondition with regards the rely relation $stable(pre, R)$ to ensure that the precondition holds during the environment transitions.

The TRGDEVT inference rule says that a triggered event $\lfloor P \rfloor$ satisfies the rely-guarantee specification if the program P satisfies the specification. This rule is directly derived from the semantics for triggered events in Fig. 3, where triggered events modifies the state according to how the program modifies the state. A basic event satisfies its rely-guarantee specification (inference rule BASICEVNT) if its body satisfies the rely-guarantee strengthening the precondition with the guard of the event. Since the occurrence of an event does not change the state, it is necessary that the guarantee relation includes the identity relation to accept stuttering transitions.

Regarding the proof rules for event systems, sequential composition of events is modeled by EVTSEQ rule, which is similar to that of the sequential command in imperative languages. In order to prove that an event set satisfies its rely-guarantee specification, we have to prove eight premises (EVTSET rule in Fig. 4). It is necessary that each event together with its specification is derivable in the system (Premise 1). Since the event set behaves as itself after an event finishes, each event postcondition has to imply each event precondition (Premise 2), and the precondition for the event set has to imply the preconditions of all events (Premise 3). An environment transition for the event set corresponds to a transition from the environment of any event i in the event

$$\text{[BasicEvt]}\quad \frac{\Sigma \vdash \mathit{body}(\alpha)\ \mathbf{sat}\ \langle \mathit{pre} \cap \mathit{guard}(\alpha), R, G, \mathit{pst}\rangle \quad \mathit{stable}(\mathit{pre}, R) \quad \forall s.\ (s,s) \in G}{\Sigma \vdash \mathbf{Event}\ \alpha\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

$$\text{[Conseq]}\quad \frac{\mathit{pre} \subseteq \mathit{pre}'\ R \subseteq R'\ \ G' \subseteq G\ \ \mathit{pst}' \subseteq \mathit{pst} \quad \Sigma \vdash \sharp\ \mathbf{sat}\ \langle \mathit{pre}', R', G', \mathit{pst}'\rangle}{\Sigma \vdash \sharp\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

$$\text{[TrgEvt]}\quad \frac{\Sigma \vdash P\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}{\Sigma \vdash (\lfloor P \rfloor)\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

$$\text{[EvtSeq]}\quad \frac{\Sigma \vdash \mathcal{E}\ \mathbf{sat}\ \langle \mathit{pre}, R, G, m\rangle \quad \Sigma \vdash \mathcal{S}\ \mathbf{sat}\ \langle m, R, G, \mathit{pst}\rangle}{\Sigma \vdash (\mathcal{E} \triangleright \mathcal{S})\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

$$\text{[EvtSet]}\quad \frac{\begin{array}{l}(1)\forall i \le n.\ \Sigma \vdash \mathcal{E}_i\ \mathbf{sat}\ \langle \mathit{pres}_i, \mathit{Rs}_i, \mathit{Gs}_i, \mathit{psts}_i\rangle \quad (2)\forall i, j \le n.\ \mathit{psts}_i \subseteq \mathit{pres}_j \\ (3)\forall i \le n.\ \mathit{pre} \subseteq \mathit{pres}_i \quad (4)\forall i \le n.\ R \subseteq \mathit{Rs}_i \quad (5)\forall i \le n.\ \mathit{Gs}_i \subseteq G \\ (6)\forall i \le n.\ \mathit{psts}_i \subseteq \mathit{pst} \quad (7)\mathit{stable}(\mathit{pre}, R) \quad (8)\forall s.\ (s,s) \in G\end{array}}{\Sigma \vdash (\{\mathcal{E}_0,\ \ldots,\ \mathcal{E}_n\})\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

$$\text{[Par]}\quad \frac{\begin{array}{l}(1)\forall \kappa.\ \Sigma \vdash \mathcal{PS}(\kappa)\ \mathbf{sat}\ \langle \mathit{pres}_\kappa, \mathit{Rs}_\kappa, \mathit{Gs}_\kappa, \mathit{psts}_\kappa\rangle \quad (2)\forall \kappa.\ \mathit{pre} \subseteq \mathit{pres}_\kappa \quad (3)\forall \kappa.\ R \subseteq \mathit{Rs}_\kappa \\ (4)\forall \kappa.\ \mathit{Gs}_\kappa \subseteq G \quad (5)\forall \kappa.\ \mathit{psts}_\kappa \subseteq \mathit{pst} \quad (6)\forall \kappa, \kappa'.\ \kappa \neq \kappa' \longrightarrow \mathit{Gs}_\kappa \subseteq \mathit{Rs}_{\kappa'}\end{array}}{\Sigma \vdash \mathcal{PS}\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle}$$

Fig. 4: Rely-guarantee Proof Rules for *PiCore*

set (Premise 4). The guarantee condition Gs_i of each event must be in the guarantee condition of the event set, since an action transition of the event set is performed by one of its events (Premise 5). The postcondition of each event must be in the overall postcondition (Premise 6). The last two refer to stability of the precondition and identity of the guarantee relation.

The parallel rule in Fig. 4 establishes compositionality of the proof system, where verification of the parallel specification can be reduced to the verification of individual event systems and then to the verification of individual events. It is necessary that each event system $\mathcal{PS}(\kappa)$ satisfies its specification $\langle pres_\kappa, Rs_\kappa, Gs_\kappa, psts_\kappa\rangle$ (Premise 1). The precondition for the parallel composition implies all the event system's preconditions (Premise 2). An environment transition Rs_κ for the event system κ corresponds to a transition from the overall environment R (Premise 3). Since an action transition of the concurrent system is performed by one of its event system, the guarantee condition Gs_κ of each event system must be a subset of the overall guarantee condition G (Premise 4). The overall postcondition must be a logical consequence of all postconditions of event systems (Premise 5). An action transition of an event system κ should be defined in the rely condition of another event system κ', where $\kappa \neq \kappa'$ (Premise 6).

Finally, the soundness theorem for a specification $\sharp$ relates rely-guarantee specifications proven on the proof system with its validity. The proof of the theorem is presented in detail in Appendix A.

Theorem 1 (Soundness). $\Sigma \vdash \sharp\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle \Longrightarrow \Sigma \models \sharp\ \mathbf{sat}\ \langle \mathit{pre}, R, G, \mathit{pst}\rangle$

3.3 Invariant Verification

In many cases, we would like to show that CRSs preserve certain data invariants. Since CRSs may not be closed systems, i.e. their environment may change the system state

that is represented by rely conditions of CRSs, the reachable states of CRSs are dependent on both the initial states and the environment. We define as follows that a CRS $\mathcal{PS}$ with static information Σ, starting up from a set of initial states $init$ under an environment R, preserves an invariant inv when its reachable states satisfy the predicate:

$$\forall s_0\ x_0\ \varpi.\ \varpi \in \Psi(\Sigma, \mathcal{PS}, s_0, x_0) \cap A(\Sigma, init, R) \longrightarrow (\forall i < len(\varpi).\ inv(s_{\varpi_i}))$$

In this definition, ϖ denotes an arbitrary computation of $\mathcal{PS}$ from a set of initial states $init$ and under an environment R. It requires that all states in ϖ satisfy the invariant inv.

To show that inv is preserved by a system $\mathcal{PS}$, it suffices to show the invariant verification theorem as follows. This theorem indicates that (1) the system satisfies its rely-guarantee specification $\langle init, R, G, post\rangle$, (2) inv initially holds in the set of initial states, and (3) each action transition as well as each environment transition preserve inv. Later, by the proof system of *PiCore*, invariant verification is decomposed to the verification of individual events.

Theorem 2 (Invariant Verification). *For formal specification $\mathcal{PS}$ and Σ, a state set $init$, a rely condition R, and inv, if*

- $\Sigma \vdash \mathcal{PS}$ **sat** $\langle init, R, G, post\rangle$.
- $init \subseteq \{s.\ inv(s)\}$.
- $stable(\{s.\ inv(s)\}, R)$ *and* $stable(\{s.\ inv(s)\}, G)$ *are satisfied.*

then inv is preserved by $\mathcal{PS}$ w.r.t. $init$ and R.

4 Integrating Concrete Languages

We present the rely-guarantee interface of *PiCore* framework in this section as well as the integration of the *IMP* and *CSimpl* languages.

4.1 Rely-guarantee Interface of *PiCore* Framework

To implement flexible integration of languages for programs of events body, *PiCore* provides a rely-guarantee interface that the program languages must respect. The interface is an abstraction for common rely-guarantee components required by *PiCore* (Fig. 5). These components are represented as a set of *parameters* and their *assumptions*. The language, semantics, proof rules and soundness proof of *PiCore* in Section 3 are developed using this interface.

Following this interface, third-part languages and their rely-guarantee proof systems are embedded into *PiCore* as *interpretations* using an *adapter* that implements the interface. Since the languages may have existed for years, they are not necessary to be completely consistent with the interface. For each language that we want to integrate it is necessary to provide a *rely-guarantee adapter* to bridge the differences of rely-guarantee components between *PiCore* and the languages. The adapter implements the interface by delegating functionality of the event language to the integrated language. This architecture allows to integrate existing languages without modifying their specification, semantics, and rely-guarantee inference system.

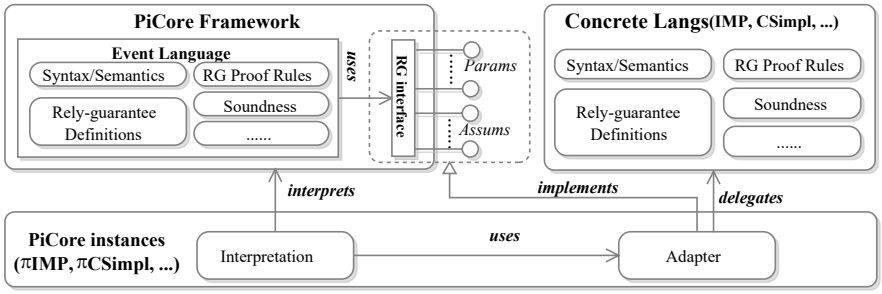

Fig. 5: *PiCore* Framework and its Integration with Imperative Languages

The interface requires specifications and assumptions for four differentiated elements: language definition (syntax and semantics), rely-guarantee definitions (computation and rely-guarantee validity), rely-guarantee proof rules, and their soundness.

As a parametric framework, *PiCore* does not define the syntax for languages of programs. It only requires a notation to represent the termination of programs, which is denoted as $\perp$ in *PiCore* (Parameter 1 in Table 1). *PiCore* also needs the transition relations representing the event behaviour (event action) and the environment (Parameters 2 and 3). To reason about event behaviors, *PiCore* assumes that (1) program $\perp$ cannot take a step to another state (Assumption 1 in Table 2), (2) if a program P takes an action transition, the program is changed in the next configuration (Assumption 2), and (3) environment transitions do not change the program itself (Assumption 3).

Since the body of events in *PiCore* is specified using external languages, computations and the reasoning of events are dependent on those languages. *PiCore* requires the specification for *computation* of programs (Parameters 4 and 5) and assumes that (1) a computation of any program is not empty (Assumption 4), (2) if ϖ is a computation of a language and the program of its first configuration is P, then ϖ is a computation for the program P (Assumption 5), and (3) there are three constructions for computation of programs (Assumption 6), which is similar to the definition of events we have presented in Section 3.

Finally, the interface requires the components related to the validity of rely-guarantee specification and the proof rules (Parameters 6 – 9). The definitions of the assume/commit functions and the validity are similar to those in *PiCore* (see Section 3), and are relaxed to be not necessarily equivalent. *PiCore* requires that the rely-guarantee proof rules in languages are sound (Assumption 10). Other rely-guarantee components, such as rely and guarantee condition, are defined in the above parameters at the same time.

4.2 Integrating the *IMP* and *CSimpl* languages

To integrate a language and its rely-guarantee framework into *PiCore*, we first create an adapter for the language providing the *PiCore* interface. For each parameter in the interface, there is a corresponding definition (or function) in the adapter instantiating the

Table 1: Parameters of *PiCore*

No.	Name	Notation	No.	Name	Notation
(1)	Terminating statement	$\bot$	(2)	Program transition	$\Sigma \vdash (P,s) \longrightarrow_p (Q,t)$
(3)	Environment transition	$\Sigma \vdash (P,s) \xrightarrow{e}_p (Q,t)$	(4)	Computations	$\Psi(\Sigma)$
(5)	Computations of a program	$\Psi(\Sigma, P)$	(6)	Assume	$A(\Sigma, pre, R)$
(7)	Commit	$C(\Sigma, G, pst)$	(8)	Validity	$\Sigma \models P\ \mathbf{sat}\ \langle pre, R, G, pst\rangle$
(9)	Proof rule	$\Sigma \vdash P\ \mathbf{sat}\ \langle pre, R, G, pst\rangle$			

Table 2: Assumptions of Parameters

(1)	$\neg(\Sigma \vdash (\bot, s) \longrightarrow_p (P,t))$	(2)	$\neg(\Sigma \vdash (P,s) \longrightarrow_p (P,t))$
(3)	$\Sigma \vdash (P,s) \xrightarrow{e}_p (Q,t) \Longrightarrow P = Q$	(4)	$[] \notin \Psi(\Sigma)$
(5)	$\varpi_0 = (P,s) \wedge \varpi \in \Psi(\Sigma) \Longrightarrow \varpi \in \Psi(\Sigma, P)$		
(6)	$(\exists P\ s.\ \varpi = [(P,s)]) \vee (\exists P\ t\ xs\ s.\ \varpi = (P,s)\#(P,t)\#xs \wedge (P,t)\#xs \in \Psi(\Sigma)) \vee$ $(\exists P\ s\ Q\ t\ xs.\ \varpi = (P,s)\#(Q,t)\#xs \wedge \Sigma \vdash (P,s) \longrightarrow_p (Q,t) \wedge (Q,t)\#xs \in \Psi(\Sigma)))$ $\Longrightarrow \varpi \in \Psi(\Sigma)$		
(7)	$\Sigma \models P\ \mathbf{sat}\ \langle pre, R, G, pst\rangle \Longrightarrow \forall s.\ \Psi(\Sigma, P, s) \cap A(\Sigma, pre, R) \subseteq C(\Sigma, G, pst)$		
(8)	$(\forall i < len(\varpi) - 1.\ (\Sigma \vdash \varpi_i \xrightarrow{e}_p \varpi_{i+1}) \longrightarrow (s_{\varpi_i}, s_{\varpi_{i+1}}) \in R) \wedge s_{\varpi_0} \in pre$ $\Longrightarrow \varpi \in A(\Sigma, pre, R)$		
(9)	$\varpi \in C(\Sigma, G, pst) \Longrightarrow (\forall i < len(\varpi) - 1.\ (\Sigma \vdash \varpi_i \longrightarrow_p \varpi_{i+1}) \longrightarrow (s_{\varpi_i}, s_{\varpi_{i+1}}) \in G)$ $\wedge\ (\sharp_{last(\varpi)} = \lfloor \bot \rfloor \longrightarrow s_{\varpi_n} \in pst)$		
(10)	$\Sigma \vdash P\ \mathbf{sat}\ \langle pre, R, G, pst\rangle \Longrightarrow \Sigma \models P\ \mathbf{sat}\ \langle pre, R, G, pst\rangle$		

parameter. Moreover, the adapter provides the necessary set of lemmas and theorems to show that the instances of the interface specifications satisfy the interface assumptions.

In the mechanized implementation of *PiCore* in Isabelle/HOL, we use *locales* to create the framework, where parameters and assumptions of *PiCore* are represented as *parameters* and *assumptions* of locales. Locales are the Isabelle's approach for dealing with parametric theories. They may be instantiated by assigning concrete data to parameters, and conclusions of locales will be propagated to the current theory or the current proof context. This is called *locale interpretation*. Using the notion of locales, we create *PiCore* instances by interpreting the *PiCore* locale using adapters for *IMP* and *CSimpl*.

Since the definitions of rely-guarantee components in *IMP* [23] are consistent with the *PiCore* interface, except that there is no static information Σ in *IMP*, the adapter for *IMP* is straightforward from its rely-guarantee specification, we omit the details here and the interested reader can refer to the Isabelle/HOL sources.

More interesting is *CSimpl* that supports most of the features of real world programming languages including exceptions, and is substantially more complex than *IMP*. Here, we show the adapter for *CSimpl*. The language and its rely-guarantee proof system are presented in detail in [24]. The abstract syntax of *CSimpl* is defined as in Fig. 6 in terms of states, of type *'s*; a set of fault types, of type *'f*; and a set of procedure names of type *'p*. Type *('s,'p,'f) config* defines the configuration used in its transition semantics and *('s,'p,'f) body* denoted as Γ defines the procedure declarations as mapping from procedure names to *CSimpl* programs. *('s,'p,'f,'e) confs* defines the type of computations. To support reasoning about procedure invocations, *CSimpl* uses the notation Θ to maintain the rely-guarantee specification for procedures. The validity in *CSimpl* requires that each procedure satisfies its specification.

datatype ('s, 'p, 'f) *com* = *Skip* | *Throw* | *Basic* 's ⇒ 's | *Spec* ('s × 's) *set*
| *Seq* ('s ,'p, 'f) *com* ('s,'p, 'f) *com* | *Cond* 's *bexp* ('s,'p,'f) *com* ('s,'p,'f) *com*
| *While* 's *bexp* ('s,'p,'f) *com* | *Call* 'p | *DynCom* 's ⇒ ('s,'p,'f) *com*
| *Guard* 'f 's *bexp* ('s,'p,'f) *com* | *Catch* ('s,'p,'f) *com* ('s,'p,'f) *com*
| *Await* 's *bexp* ('s,'p,'f) *com*
datatype ('s,'f) *xstate* = *Normal* 's | *Abrupt* 's | *Fault* 'f | *Stuck*
type-synonym('s,'p,'f) *config* = ('s,'p,'f)*com* × ('s,'f) *xstate*
type-synonym ('s,'p,'f) *body* = 'p ⇒ ('s,'p,'f) *com option*
type-synonym ('s,'p,'f,'e) *confs* = ('s,'p,'f,'e) *body* ×(('s,'p,'f,'e) *config*) *list*

Fig. 6: Syntax and state definition of the CSimpl Language [24]

In the adapter, we first use the pair (Γ, Θ) to instantiate the environment Σ in *PiCore*. We instantiate the termination statement as the $Skip$ command in *CSimpl*. The program transition in *CSimpl* is $\Gamma \vdash_c (P, s) \longrightarrow (Q, t)$, is adapted as $(\Gamma, \Theta) \vdash_{cI} (P, s) \longrightarrow (Q, t) \equiv \Gamma \vdash_c (P, s) \longrightarrow (Q, t)$. *CSimpl* semantics for programs can transit from a *Normal* state to a different type. However, it does not allow transitions from a non *Normal* state to a different type of state. Therefore, the environment transition in *CSimpl* is defined as follows.

$$\begin{cases} \Gamma \vdash_c (P, Normal\ s) \longrightarrow_e (P, t) \\ (\forall t'.\ t \neq Normal\ t') \Longrightarrow \Gamma \vdash_c (P, t) \longrightarrow_e (P, t) \end{cases}$$

To adapt the restricted environment transition, we first define the environment transition in the adapter as $(\Gamma, \Theta) \vdash_{cI} (P, s) \longrightarrow_e (P, t)$, which allows any state transition and is compatible with that in the interface. Then, we restrict the rely condition in the definition of proof rules in the adapter to bridge this difference, which will be discussed later. Based on the transition functions, the computation function Ψ of the adapter is defined in the same form as in *CSimpl*.

The rely-guarantee specification in *CSimpl* is in the form $[p, R, G, (q, a)]$, where the postcondition (q, a) is a pair of state sets. The set q constrains the final state if the program terminates as *Skip* representing a normal state, whilst a constrains abrupt terminations in an exception with the command *Throw*. The assume and commit functions in *CSimpl* are like *PiCore*, but considering the fault states and abrupt termination. The validity function of *CSimpl* is defined in the same form as in *PiCore*. For procedure invocations, *CSimpl* defines another validity function using the general one, which also requires that each procedure satisfies its rely-guarantee specification.

We define the *assume*, *commit* and *validity* functions in the adapter as the same form as in *PiCore*. In *CSimpl* preconditions are over normal states. For type consistency *PiCore* does not impose that restriction, but rather it is enforced by the adapter to bridge the difference, which will be discussed later. *PiCore* does restrict the final statement to *Skip* thus exceptions have to be handled at program level. This restriction is motivated by the second assumption in the rule EVTSET for *PiCore* proof system in Figure 4, since postconditions of events must imply their preconditions, and preconditions in *CSimpl* are set of normal states, a final configuration of an event cannot throw an exception.

Finally, based on the definition of proof rules $\Gamma,\Theta \vdash_{/F}$ P sat [q, R, G, q,a] in *CSimpl*, we define that in the adapter as follows. (1) The validity in *CSimpl* only concerns preconditions of *Normal* states, so we restrict the precondition p to *Normal*. (2) Pro-

grams of an event body cannot throw exceptions to the event level, so final states when reaching the final statement *Skip* are *Normal*. Thus, we restrict the postcondition q to *Normal*. (3) Events assume the normal execution of their program body, and furthermore the program cannot fall into a *Fault* state. So we assume the *Fault* set F to be empty. In addition, the program P should satisfy its rely-guarantee specification in *C-Simpl*. (4) The environment transition in *CSimpl* does not allow transitions from a non *Normal* state to a different type of state, we represent it in the rely condition R. (5) Finally, the rely-guarantee specification for each procedure in Θ has to be satisfied.

$$(\Gamma,\Theta) \vdash_I P\ sat_p\ [p, R, G, q] \equiv \overbrace{(p \subseteq Normal\ `\ UNIV)}^{(1)} \wedge \overbrace{(q \subseteq Normal\ `\ UNIV)}^{(2)} \wedge$$
$$\overbrace{(\Gamma,\Theta \vdash_{/\{\}} P\ sat\ [\{s.\ Normal\ s \in p\}, R, G, \{s.\ Normal\ s \in q\}, UNIV])}^{(3)} \wedge$$
$$\overbrace{(\forall (s,t) \in R.\ s \notin Normal\ `\ UNIV \longrightarrow s = t)}^{(4)} \wedge \overbrace{(\forall (c,p,R,G,q,a) \in \Theta.\ \Gamma \models_{/\{\}} (Call\ c)\ sat\ [p, R, G, q, a])}^{(5)}$$

To interpret the *PiCore* framework using the adapter, we have to show that the assumptions in Table 2 are preserved on the adapted definitions. The preservation of assumptions 1 – 9 are straightforward. To show assumption 10, we prove that

$$(\Gamma,\Theta) \vdash_I P\ sat_p\ [p, R, G, q] \Longrightarrow (\Gamma,\Theta) \models_I P\ sat_p\ [p, R, G, q]$$

5 Concurrent Memory Management of Zephyr RTOS

In this section, we use π*IMP*, the instantiation of *PiCore* with *IMP*, to formally specify and verify the concurrent memory management of Zephyr RTOS. The size of the C code is $\approx$ 400 lines of code. During the formal verification, we found 3 bugs in the C code of Zephyr: *an incorrect block split*, *an incorrect return*, and *non-termination of a loop* in the *k_mem_pool_alloc* service (see Appendix B). The first two bugs are critical and have been repaired in the latest release of Zephyr.

The buddy memory allocation can split large blocks into smaller ones to fit as best as possible the requested size. This allows blocks of different sizes to be allocated and released efficiently while limiting memory fragmentation concerns. The memory is organized by levels, each "level n" block is a quad-block that can be split into four smaller "level (n+1)" blocks of equal size. This process is repeated until blocks reach a minimum level for which splitting is not possible. In our formal specification, we define the structure of a memory pool as illustrated in Fig. 7. The top of the figure shows the real memory of the first block at level 0.

Thread preemption and fine-grained locking make kernel execution of memory services to be concurrent. Zephyr provides two kernel services *k_mem_pool_alloc* and *k_mem_pool_free*, for memory allocation and release respectively. The main part of the C code of *k_mem_pool_alloc* is shown in Appendix B. When an application requests for a memory block, Zephyr first computes two levels necessary for the operation. One is $alloc\_l$, the level with the size of the smallest block that will satisfy a request. The other level is $free\_l$ and it is the lowest level containing free memory blocks, with $free\_l \leqslant alloc\_l$. Due to concurrency, when a service tries to allocate a free block *blk* from level $free\_l$, blocks at that level may be allocated or merged into a bigger block by other

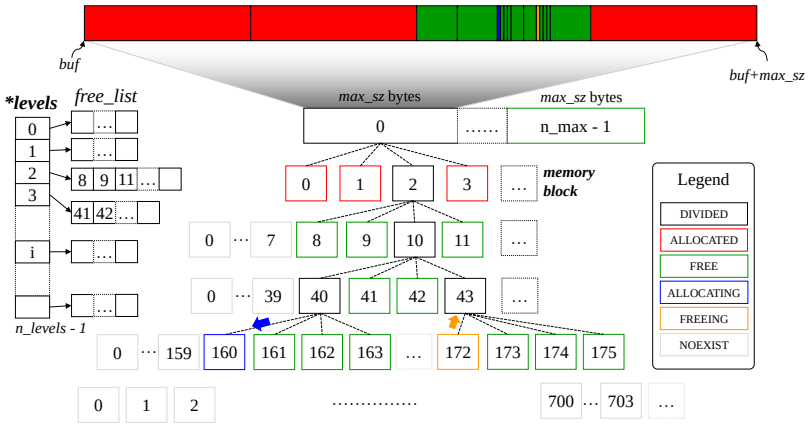

Fig. 7: Structure of Memory Pools

concurrent threads. In such case the service will back out to retry. If blk is successfully locked for allocation, then it is broken down to level $alloc\_l$. Allocation supports a *time-out* parameter to allow threads waiting for that pool for a period of time when the call does not succeed. If the allocation fails and the timeout is not *K_NO_WAIT*, the thread is suspended and the context is switched to another thread.

We define a rich set of *invariants* on the kernel state clarifying the constraints and consistency of quad trees, free block lists, memory pool configuration, and waiting threads. From the well-shaped properties of quad trees, we can derive a critical property to prevent memory leaks, i.e., memory blocks cover the whole memory address of the pool, but not overlap each other. Memory blocks of a memory pool mp are a partition of the pool where for any memory address $addr$ in the address space of a memory pool, i.e. $addr < n\_max * max\_sz$, there is one and only one memory block whose address space contains $addr$. The predicate is defined as follows.

addr-in-block *mp addr i j* ≡
i < *length* (*levels mp*) ∧ *j* < *length* (*bits* (*levels mp* ! *i*)) ∧ (*is_memblock(bits* (*levels mp* ! *i*) ! *j*))
∧ *addr* ∈ {*j* ∗ (*max-sz mp div* (4 ^ *i*)) ..< *Suc j* ∗ (*max-sz mp div* (4 ^ *i*))}
mem-part *s* ≡ ∀ *p*∈*mem-pools s*. ***let*** *mp* = *mem-pool-info s p* ***in***
(∀ *addr* < *n-max mp* ∗ *max-sz mp*. (∃ !(*i*,*j*). *addr-in-block mp addr i j*))

From the invariants of the well-shaped bitmap, we derive the general property for the memory partition.

Theorem 3 (Memory Partition). *For any kernel state s, If the memory pools in s are consistent in their configuration, and their bitmaps are well-shaped, the memory pools satisfy the partition property in s, i.e.*

inv_mempool_info $s \wedge$ ***inv_bitmap*** $s \wedge$ ***inv_bitmap0*** $s \wedge$ ***inv_bitmapn*** $s \Longrightarrow$ ***mem_part*** s

In the formal specification, we consider a scheduler $\mathcal{S}$ and a set of threads $t_1, ..., t_n$. A user thread t_i invoke allocation/release services, thus the event system for t_i is

$$esys_{t_i} \equiv (\bigcup blk.\ \{mem\_pool\_free[blk]@t_i\}) \cup$$
$$(\bigcup (p, sz, tmout).\ \{mem\_pool\_alloc[p, sz, tmout]@t_i\})$$

which is a set of *alloc* and *free* events, where the input parameters for these events correspond with the arguments of the service implementation in the C code. Together with the threads we model the event service for the scheduler $esys_{sched}$ consisting of a unique event *sched* whose argument is a thread t to be scheduled when it is in the *READY* state. The formal specification of the memory management is thus defined as: Sys-Spec $\equiv \lambda$ k. case k of $(\mathcal{T}\ t_i) \Rightarrow esys_{t_i} \mid \mathcal{S} \Rightarrow esys_{sched}$. This is much simpler than the specification obtained from a non-event oriented language.

We refer readers to Appendix C for the main part of the C code of *k_mem_pool_free* and to Appendix D or the Isabelle/HOL sources for the complete specification of the service. Events are parametrized by a thread identifier used to control access the execution context of the thread invoking it.

Using the compositional reasoning of π*IMP*, correctness of Zephyr memory management can be specified and verified with the rely-guarantee specification of each event. The functional correctness of a kernel service is specified by its pre/post conditions. The preservation of invariants, memory configuration, and separation of local variables is specified in the guarantee condition of each service. Although *IMP* does not have proof rules for loop termination, we use a logical variable α to parametrize the loop invariants and prove the termination of loop statements in Zephyr by finding a convergent relation to show that the number of iterations is finite.

The guarantee condition for both memory services is defined as:

Mem-pool-free-guar $t \equiv \overbrace{Id}^{(1)} \cup (\overbrace{gvars\_conf\_stable}^{(2)} \cap$
$\{(s,r).\ (\overbrace{cur\ s \neq Some\ t \longrightarrow gvars\text{-}nochange\ s\ r \wedge lvars\text{-}nochange\ t\ s\ r}^{(3.1)})$
$\wedge\ (\overbrace{cur\ s = Some\ t \longrightarrow inv\ s \longrightarrow inv\ r}^{(3.2)})\ \wedge\ (\overbrace{\forall t'.\ t' \neq t \longrightarrow lvars\text{-}nochange\ t'\ s\ r}^{(4)})\ \})$

This relation states that an step from *alloc* or *free* may not change the state (1), e.g., selecting branch on a conditional statement. If it changes the state then: (2) static configuration of memory pools in the model does not change; (3.1) if the scheduled thread is not the thread invoking the event then its local variables do not change; (3.2) if it is, then the relation preserves the memory invariant; (4) a thread does not change the local variables of other threads.

Using *PiCore* and *IMP* proof rules we verify that the invariant is preserved by all the events. Additionally, we prove that when starting in a valid memory configuration given by the invariant, and if the service does not return an error code, then it returns a valid memory block with size bigger or equal than the requested capacity.

A property verification is carried out by inductively applying the proof rules for each system event and discharging the proof obligations the rules generate. Typically, these proof obligations require to prove stability of the pre and postcondition to check that changes of the environment preserve them, and showing that a statement modifying a state from the precondition gets a state belonging to the postcondition. A detailed proof sketch of the *free* service is shown in Appendix D.

6 Evaluation and Conclusion

Evaluation. We use Isabelle/HOL as the specification and verification system. All derivations of our proofs have passed through the Isabelle proof kernel. We use ≈ 9,200 lines of specification and proof (*LOSP*) to develop the *PiCore* framework. The *IMP* language and its rely-guarantee proof system consist of ≈ 2,400 *LOSP*, and *CSimpl* ≈ 15,000 *LOSP*. The two parts of specification and proof are completely reused in π*IMP* and π*CSimpl* respectively. The adapter of *IMP* is ≈ 650 *LOSP* including new proof rules and their soundness as well as a concrete syntax. The adapter of *CSimpl* is ≈ 400 *LOSP*. Finally, we develop ≈ 17,600 *LOSP* for the Zephyr case study, 40 times than the lines of the C code due to the in-kernel concurrency, where invariant proofs represent the largest part.

Related works. The rely-guarantee approach has been mechanized in Isabelle/HOL (e.g. [23,26,14,13,24]) and Coq (e.g. [18,20]). In [14,13], an abstract algebra of atomic steps is developed, and rely/guarantee concurrency is an interpretation of the algebra. To allow a meaningful comparison of rely-guarantee semantic models, two abstract models for rely-guarantee are developed and mechanized in [26]. The two works do not consider the concrete imperative languages for rely-guarantee. The works [23,20] mechanize the rely-guarantee approach for simple imperative languages. Later, a rely-guarantee proof system for *CSimpl* [24], a generic and realistic imperative language by extending *Simpl*, is developed in Isabelle/HOL. These mechanizations focus on imperative languages for pure programs. Two of them [23,24] with mechanization of proof system in Isabelle/HOL have been integrated in *PiCore*.

Refinement of reactive systems [5] and the subsequent Event-B approach [2] propose a refinement-based formal method for system-level modeling and analysis. In [15], an Event-B model is created to mimic rely-guarantee style reasoning for concurrent programs, but not to provide a rely-guarantee framework for Event-B. The rely-guarantee reasoning for event-based applications has been studied in [8,11,10,9]. The definition of events is similar to *PiCore*. They extend a simple, sequential, imperative language by primitives for announcing and consuming events, *announce(e)* and *consume(e(x))* where *e* is an event. Therefore, events are triggered by imperative programs in another event. This is very different from the reactive semantics in *PiCore* where the system is non-deterministically executed simulating a real reactive system. Moreover, the language to specify events in these works is a simple imperative language, whilst *PiCore* has an open interface for the integration and reusability of different languages and frameworks.

Conclusion and future work. In this paper, we propose an event-based rely-guarantee framework for concurrent reactive systems. This approach is open to the specification of event behaviours. It provides an interface to integrate systems for specification and reasoning at that level that eases formal methods reusability. We have mechanized the integration of the *IMP* and *CSimpl* languages and their proof systems into *PiCore* in the Isabelle/HOL theorem prover. We show the simplicity of events to represent concurrent reactive systems and the ability of *PiCore* for realistic systems in the verification of the concurrent buddy memory allocation of Zephyr RTOS. As future work, we plan to extend *PiCore* to support more event structures and step-wise refinement.

References

1. The Zephyr Project. https://www.zephyrproject.org/, accessed: December 2018
2. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae 77(1-2), 1–28 (2007)
3. Aceto, L., Ingólfsdóttir, A., Larsen, K., Srba, J.: Reactive Systems - Modeling, Specification and Verification. Cambridge University Press (2007)
4. Andronick, J., Lewis, C., Morgan, C.: Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System. In: Proceedings Workshop on Models for Formal Analysis of Real Systems MARS. pp. 10–24 (2015)
5. Back, R.J., Sere, K.: Superposition Refinement of Reactive Systems. Formal Aspects of Computing 8(3), 324–346 (1996)
6. Back, R.J., Sere, K.: Stepwise Refinement of Action Systems. Structured Programming 12, 17–30 (1991)
7. Chen, H., Wu, X., Shao, Z., Lockerman, J., Gu, R.: Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In: 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 431–447. ACM (2016)
8. Dingel, J., Garlan, D., Jha, S., Notkin, D.: Towards a Formal Treatment of Implicit Invocation Using Rely/Guarantee Reasoning. Formal Aspects of Computing 10(3), 193–213 (Mar 1998)
9. Fenkam, P., Gall, H., Jazayeri, M.: Composing Specifications of Event Based Applications. In: International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 67–86. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
10. Fenkam, P., Gall, H., Jazayeri, M.: Constructing Deadlock Free Event-Based Applications: A Rely/Guarantee Approach. In: International Symposium of Formal Methods Europe (FME). pp. 636–657. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
11. Garlan, D., Jha, S., Notkin, D., Dingel, J.: Reasoning About Implicit Invocation. In: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 209–221. ACM, New York, NY, USA (1998)
12. Harel, D., Pnueli, A.: On the Development of Reactive Systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. pp. 477–498. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
13. Hayes, I.J.: Generalised Rely-guarantee Concurrency: An Algebraic Foundation. Formal Aspects of Computing 28(6), 1057–1078 (Nov 2016)
14. Hayes, I.J., Colvin, R.J., Meinicke, L.A., Winter, K., Velykis, A.: An Algebra of Synchronous Atomic Steps. In: nternational Symposium on Formal Methods (FM 2016). pp. 352–369. Springer International Publishing (2016)
15. Hoang, T.S., Abrial, J.R.: Event-B Decomposition for Parallel Programs. In: Second International Conference of Abstract State Machines, Alloy, B and Z (ABZ). pp. 319–333. Springer Berlin Heidelberg (2010)
16. Jones, C.B.: Tentative Steps Toward a Development Method for Interfering Programs. ACM Transactions on Programming Languages and System 5(4), 596–619 (October 1983)
17. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal Verification of an OS Kernel. In: Proceedings of ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP'09, ACM Press, Big Sky, Montana, USA (2009)
18. Liang, H., Feng, X., Fu, M.: A Rely-guarantee-based Simulation for Verifying Concurrent Program Transformations. In: 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 455–468. ACM Press (2012)
19. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and Guarantees for Compositional Noninterference. In: 24th Computer Security Foundations Symposium (CSF). pp. 218–232. IEEE Press (2011)

20. Moreira, N., Pereira, D., de Sousa, S.M.: On the Mechanisation of Rely-Guarantee in Coq. Tech. rep., Universidade do Porto (2013)
21. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for Operating System Kernels. In: 2nd International Conference on Certified Programs and Proofs (CPP). pp. 126–142. Springer (December 2012)
22. Murray, T., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In: 29th IEEE Computer Security Foundations Symposium (CSF). IEEE Press (2016)
23. Nieto, L.P.: The Rely-Guarantee Method in Isabelle/HOL. In: 12th European Symposium on Programming (ESOP). pp. 348–362. Springer Berlin Heidelberg (2003)
24. Sanán, D., Zhao, Y., Hou, Z., Zhang, F., Tiu, A., Liu, Y.: CSimpl: A Rely-Guarantee-Based Framework for Verifying Concurrent Programs. In: 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 481–498. Springer Berlin Heidelberg (April 2017)
25. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technischen Universitat Munchen (2006)
26. van Staden, S.: On Rely-Guarantee Reasoning. In: International Conference on Mathematics of Program Construction. pp. 30–49. Springer International Publishing (2015)
27. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A Practical Verification Framework for Preemptive OS Kernels. In: 28th International Conference on Computer Aided Verification (CAV). pp. 59–79. Springer (July 2016)
28. Xu, Q., de Roever, W.P., He, J.: The Rely-guarantee Method for Verifying Shared Variable Concurrent Programs. Formal Aspects of Computing 9(2), 149–174 (1997)

Appendix A Proof of Soundness

The soundness of rules for events is straightforward and proved by the assumptions in the interface. To prove soundness of rules for event systems. First, we show how to decompose a computation of event systems into computations of its events.

We first define an equivalent relation on computations as follows. Here, we concern the state, event context, and transitions, but not the specification of a configuration.

Definition 1 (Simulation of Computations). *A computation ϖ_1 is a simulation of ϖ_2, denoted as $\varpi_1 \asymp \varpi_2$, if $len(\varpi_1) = len(\varpi_2)$ and $\forall i < len(\varpi_1) - 1.\ s_{\varpi_{1_i}} = s_{\varpi_{2_i}} \wedge x_{\varpi_{1_i}} = x_{\varpi_{2_i}} \wedge (\varpi_{1_i} \xrightarrow{\delta} \varpi_{1_{i+1}}) = (\varpi_{2_i} \xrightarrow{\delta} \varpi_{2_{i+1}})$.*

In order to decompose computations of event systems to those of events, we define serialization of events based on the simulation of computations.

Definition 2 (Serialization of Events). *A computation ϖ of event systems is a serialization of a set of events $\{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$, denoted by $\varpi \lll \{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$, iff there exist a set of computations $\varpi_1, ..., \varpi_m$, where for $1 \leq i \leq m$ there exists $1 \leq k \leq n$ that $\varpi_i \in \Psi(\Sigma, \mathcal{E}_k)$, such that $\varpi \asymp \varpi_1 \# \varpi_2 \# ... \# \varpi_m$.*

Then, we can decompose a computation of an event system into a set of computation of its events as follows.

Lemma 1. *For any computation ϖ of an event system $\mathcal{S}$, $\varpi \lll evts(\mathcal{S})$.*

The soundness of the EVTSEQ rule is proved by two cases. For any computation ϖ of "$\mathcal{E} \triangleright \mathcal{S}$", the first case is that the execution of event $\mathcal{E}$ does not finish in ϖ. In such a case, $\varpi \lll \{\mathcal{E}\}$. By the first premise of this rule, we can prove the soundness. In the second case, the execution of event $\mathcal{E}$ finishes in ϖ. In such a case, we have $\varpi = \varpi_1 \# \varpi_2$, where $\varpi_1 \lll \{\mathcal{E}\}$ and $\varpi_2 \lll evts(\mathcal{S})$. By the two premises of this rule, we can prove the soundness. The soundness of the EVTSET rule is complicated. From Lemma 1, we have that for any computation ϖ of the event set, $\varpi \asymp \varpi_1 \# \varpi_2 \# ... \# \varpi_m$, for $1 \leq i \leq m$ there exists $1 \leq k \leq n$ that $\varpi_i \in \Psi_{\mathcal{E}}(\mathcal{E}_k)$. When ϖ is in $A(\Sigma, pre, R)$, from $\forall i \leq n, j \leq n.\ psts_i \subseteq pres_j$, $\forall i \leq n.\ pre \subseteq pres_i$, and $\forall i \leq n.\ R \subseteq Rs_i$, we have that there is one k for each ϖ_i that ϖ_i is in $A(pres_k, Rs_k)$. By the first premise in the EVTSET rule, we have ϖ_i is in $C(\Sigma, Gs_k, psts_k)$. Finally, with $\forall i \leq n.\ Gs_i \subseteq G$ and $\forall i \leq n.\ psts_i \subseteq pst$, we have that ϖ is in $C(\Sigma, G, pst)$.

To prove the soundness of the PAR rule, we first use *conjoin* as follows to decompose a computation of parallel event systems into computations of its event systems. Computations of a set of event systems can be combined into a computation of the parallel composition of them, iff they have the same state and event context sequences, as well as they do not have the action transition at the same time. The resulting computation of $\mathcal{PS}$ also has the same state and event context sequences. Furthermore, in this computation a transition is an action transition on core κ if this is the action in the computation of event system κ at the corresponding position; a transition is an environment transition if this is the case in all computations of event systems at the corresponding position. By the definition, we have that the semantics is compositional as shown in Lemma 2.

Definition 3. *A computation ϖ of a parallel event system $\mathcal{PS}$ and a set of computations $\widehat{\varpi} : \mathcal{K} \to \Psi_{\mathcal{S}}$ conjoin, denoted by $\varpi \propto \widehat{\varpi}$, iff*

- $\forall \kappa.\ len(\varpi) = len(\widehat{\varpi}(\kappa))$.
- $\forall \kappa, j < len(\varpi).\ s_{\varpi_j} = s_{\widehat{\varpi}(\kappa)_j} \wedge x_{\varpi_j} = x_{\widehat{\varpi}(\kappa)_j}$.
- $\forall \kappa, j < len(\varpi).\ \sharp_{\varpi_j}(\kappa) = \sharp_{\widehat{\varpi}(\kappa)_j}$.
- *for any $j < len(\varpi) - 1$, one of the following two cases holds:*
  - $\varpi_j \xrightarrow{e} \varpi_{j+1}$, *and* $\forall \kappa.\ \widehat{\varpi}(\kappa)_j \xrightarrow{e} \widehat{\varpi}(\kappa)_{j+1}$.
  - $\varpi_j \xrightarrow{t@\kappa_1} \varpi_{j+1}$, $\widehat{\varpi}(\kappa_1)_j \xrightarrow{t@\kappa_1} \widehat{\varpi}(\kappa_1)_{j+1}$, *and* $\forall \kappa \neq \kappa_1.\ \widehat{\varpi}(\kappa)_j \xrightarrow{e} \widehat{\varpi}(\kappa)_{j+1}$.

Lemma 2. *The semantics of* PiCore *is compositional, i.e.,* $\Psi(\Sigma, \mathcal{PS}, s, x) = \{\varpi \mid (\exists \widehat{\varpi} \mid (\forall \kappa.\ \widehat{\varpi}(\kappa) \in \Psi(\Sigma, \mathcal{PS}(\kappa), s, x)) \wedge \varpi \propto \widehat{\varpi})\}$.

Finally, the soundness of the PAR rule is proved by a similar way to [28,23].

Appendix B The C Source Code and Bugs of Memory Allocation in Zephyr

```
1
2  static int pool_alloc(struct k_mem_pool *p, struct k_mem_block *block,
3                        size_t size)
4  {
5    size_t lsizes[p->n_levels];
6    int i, alloc_l = -1, free_l = -1, from_l;
7    void *blk = NULL;
8
9    /* Walk down through levels, finding the one from which we
10    * want to allocate and the smallest one with a free entry
11    * from which we can split an allocation if needed.  Along the
12    * way, we populate an array of sizes for each level so we
13    * don't need to waste RAM storing it.
14    */
15   lsizes[0] = _ALIGN4(p->max_sz);
16   for (i = 0; i < p->n_levels; i++) {
17     if (i > 0) {
18       lsizes[i] = _ALIGN4(lsizes[i-1] / 4);
19     }
20
21     if (lsizes[i] < size) {
22       break;
23     }
24
25     alloc_l = i;
26     if (!level_empty(p, i)) {
27       free_l = i;
28     }
29   }
30
31   if (alloc_l < 0 || free_l < 0) {
32     block->data = NULL;
33     return -ENOMEM;
34   }
35
36   /* Iteratively break the smallest enclosing block... */
37   blk = alloc_block(p, free_l, lsizes[free_l]);
38
39   if (!blk) {
40     /* This can happen if we race with another allocator.
```

```
41        * It's OK, just back out and the timeout code will
42        * retry.  Note mild overloading: -EAGAIN isn't for
43        * propagation to the caller, it's to tell the loop in
44        * k_mem_pool_alloc() to try again synchronously.  But
45        * it means exactly what it says.
46        */
47       return -EAGAIN;
48     }
49
50     for (from_l = free_l;
51          level_empty(p, alloc_l) && from_l < alloc_l;
52          from_l++) {
53       blk = break_block(p, blk, from_l, lsizes);
54     }
55
56     /* ... until we have something to return */
57     block->data = blk;
58     block->id.pool = pool_id(p);
59     block->id.level = alloc_l;
60     block->id.block = block_num(p, block->data, lsizes[alloc_l]);
61     return 0;
62   }
63
64   int k_mem_pool_alloc(struct k_mem_pool *p, struct k_mem_block *block,
65                        size_t size, s32_t timeout)
66   {
67     int ret, key;
68     s64_t end = 0;
69
70     __ASSERT(!(_is_in_isr() && timeout != K_NO_WAIT), "");
71
72     if (timeout > 0) {
73       end = _tick_get() + _ms_to_ticks(timeout);
74     }
75
76     while (1) {
77       ret = pool_alloc(p, block, size);
78
79       if (ret == 0 || timeout == K_NO_WAIT ||
80         ret == -EAGAIN || (ret && ret != -ENOMEM)) {
81         return ret;
82       }
83
84       key = irq_lock();
85       _pend_current_thread(&p->wait_q, timeout);
86       _Swap(key);
87
88       if (timeout != K_FOREVER) {
89         timeout = end - _tick_get();
90
91         if (timeout < 0) {
92           break;
93         }
94       }
95     }
96
97     return -EAGAIN;
98   }
```

During the formal verification, we found 3 bugs and an integrity issue in the C code of Zephyr, which are shown as below.

(1) Incorrect block split: this bug is located in the loop in Line 51 of the *k_mem_pool_alloc* service. The *level_empty* function checks if there are blocks in the free list at level *alloc_l*. Concurrent threads may release a memory block at that level making *level_empty(p, alloc_l)* to return *false* and stopping the loop. In such case, it allocates a

memory block of a bigger capacity at a level i but it still sets the level number of the block as *alloc_l* at Line 59. The service allocates a larger block to the requesting thread causing an internal fragmentation of $max\_sz/4^i - max\_sz/4^{alloc\_l}$ bytes. When this block is released, it will be inserted into the free list at level *alloc_l* but not level i causing an external fragmentation of $max\_sz/4^i - max\_sz/4^{alloc\_l}$. The bug is fixed by removing the condition *level_empty(p, alloc_l)* in our specification.

(2) Incorrect return from *k_mem_pool_alloc*: this bug is found at Line 80. When a suitable free block is allocated by another thread, the *pool_alloc* function returns *-EAGAIN* at Line 47 to ask the thread to retry the allocation. When a thread invokes *k_mem_pool_alloc* in *FOREVER* mode and this case happens, the service returns *-EAGAIN* immediately. However, a thread invoking *k_mem_pool_alloc* in *FOREVER* mode should keep retrying forever when failed. We repair the bug by removing the condition $ret == -EAGAIN$ at Line 80. As explained in the comments of the C Code (Lines 40 - 46), *-EAGAIN* should not be returned to threads invoking the service. Moreover, the *return -EAGAIN* at Line 97 is actually the case of time out. Thus, we introduce a new return code *TIMEOUT* in our specification.

(3) Non-termination of *k_mem_pool_alloc*: The loop statement at Lines 76 - 95 should terminate in certain cases, which are actually violated in the C code. When a thread requests a memory block in *FOREVER* mode and the requested size is larger than *max_sz*, the maximum size of blocks, the loop at Lines 76 - 95 will never finish since *pool_alloc* always returns *-ENOMEM*. The reason is that the "*return -ENOMEM*" at Line 33 does not distinguish two cases, $alloc\_l < 0$ and $free\_l < 0$. In the first case, the requested size is larger than *max_sz* and the kernel service should return immediately. In the second case, there are no free blocks larger than the requested size and the service tries forever until some free block available. We repair the bug by splitting the *if* statement at Lines 31 - 34 into two cases and introducing a new return code *ESIZEERR* in our specification. Then, we change the condition by *ESIZEERR* at Lines 79 - 80.

Appendix C The C Source Code of Memory Release in Zephyr

```
1
2  static void free_block(struct k_mem_pool *p, int level, size_t *lsizes, int bn)
3  {
4    int i, key, lsz = lsizes[level];
5    void *block = block_ptr(p, lsz, bn);
6
7    key = irq_lock();
8
9    set_free_bit(p, level, bn);
10
11   if (level && partner_bits(p, level, bn) == 0xf) {
12     for (i = 0; i < 4; i++) {
13       int b = (bn & ~3) + i;
14
15       clear_free_bit(p, level, b);
16       if (b != bn &&
17           block_fits(p, block_ptr(p, lsz, b), lsz)) {
18         sys_dlist_remove(block_ptr(p, lsz, b));
19       }
20     }
21
22     irq_unlock(key);
```

```
23      free_block(p, level-1, lsizes, bn / 4); /* tail recursion! */
24      return;
25    }
26
27    if (block_fits(p, block, lsz)) {
28      sys_dlist_append(&p->levels[level].free_list, block);
29    }
30
31    irq_unlock(key);
32  }
33
34  void k_mem_pool_free(struct k_mem_block *block)
35  {
36    int i, key, need_sched = 0;
37    struct k_mem_pool *p = get_pool(block->id.pool);
38    size_t lsizes[p->n_levels];
39
40    /* As in k_mem_pool_alloc(), we build a table of level sizes
41     * to avoid having to store it in precious RAM bytes.
42     * Overhead here is somewhat higher because free_block()
43     * doesn't inherently need to traverse all the larger
44     * sublevels.
45     */
46    lsizes[0] = _ALIGN4(p->max_sz);
47    for (i = 1; i <= block->id.level; i++) {
48      lsizes[i] = _ALIGN4(lsizes[i-1] / 4);
49    }
50
51    free_block(get_pool(block->id.pool), block->id.level,
52      lsizes, block->id.block);
53
54    /* Wake up anyone blocked on this pool and let them repeat
55     * their allocation attempts
56     */
57    key = irq_lock();
58
59    while (!sys_dlist_is_empty(&p->wait_q)) {
60      struct k_thread *th = (void *)sys_dlist_peek_head(&p->wait_q);
61
62      _unpend_thread(th);
63      _abort_thread_timeout(th);
64      _ready_thread(th);
65      need_sched = 1;
66    }
67
68    if (need_sched && !_is_in_isr()) {
69      _reschedule_threads(key);
70    } else {
71      irq_unlock(key);
72    }
73  }
```

Appendix D Specification and Proof Sketch of *k_mem_pool_free*

The formal specification of *k_mem_pool_free* (in *black* color) and its rely-guarantee proof sketch (in *blue* color) are shown as follows.

Mem-pool-free-pre $t \equiv \{\!|\, \acute{}inv \wedge \acute{}allocating\text{-}node\ t = None \wedge \acute{}freeing\text{-}node\ t = None \,|\!\}$
EVENT Mem-pool-free *[Block b]* @ $(\mathcal{T}\ t)$
WHEN
pool b $\in \acute{}$*mem-pools*

```
  ∧ level b < length (levels (´mem-pool-info (pool b)))
  ∧ block b < length (bits (levels (´mem-pool-info (pool b))!(level b)))
  ∧ data b = block-ptr (´mem-pool-info (pool b))
            ((ALIGN4 (max-sz (´mem-pool-info (pool b)))) div (4 ^ (level b))) (block b)
THEN
  Mem-pool-free-pre t ∩ {| g |}    (* g is the guard condition of the event *)
  (* here we set the bit to FREEING, so that other thread cannot mem-pool-free the same block
    it also requires that it can only free ALLOCATED block *)
  t ► AWAIT (bits ((levels (´mem-pool-info (pool b))) ! (level b))) ! (block b
                    = ALLOCATED THEN
        ´mem-pool-info := set-bit-freeing ´mem-pool-info (pool b) (level b) (block b);;
        ´freeing-node := ´freeing-node (t := Some b)
      END);;
  mp-free-precond2 t b ≡ {| ´inv ∧ ´allocating-node t = None ∧  g ∧ ´freeing-node t = Some b |}
  t ► ´need-resched := ´need-resched(t := False);;
  mp-free-precond3 t b ≡ (mp-free-precond2 t b) ∩ {|´need-resched t = False|}
  t ► ´lsizes := ´lsizes(t := [ALIGN4 (max-sz (´mem-pool-info (pool b)))]);;
  mp-free-precond4 t b ≡
    mp-free-precond3 t b ∩ {|´lsizes t = [ALIGN4 (max-sz (´mem-pool-info (pool b)))]|}
  FOR (t ► ´i := ´i(t := 1)); ´i t ≤ level b;  (t ► ´i := ´i(t := ´i t + 1)) DO
    t ► ´lsizes := ´lsizes(t := ´lsizes t @ [ALIGN4 (´lsizes t ! (´i t − 1) div 4)])
  ROF;;
  mp-free-precond5 t b ≡ mp-free-precond3 t b ∩
    {|(∀ ii<length (´lsizes t). ´lsizes t ! ii = (ALIGN4 (max-sz (´mem-pool-info (pool b))))
                                                  div (4 ^ ii)) ∧ length (´lsizes t) > level b|}
  (* = = = start: free-block(pool, level, lsizes, block); = = =*)
  t ► ´free-block-r := ´free-block-r (t := True);;
  mp-free-precond6 t b ≡ mp-free-precond5 t b ∩ {|´free-block-r t = True|}
  t ► ´bn := ´bn (t := block b);;
  mp-free-precond7 t b ≡ mp-free-precond6 t b ∩ {|´bn t = block b|}
  t ► ´lvl := ´lvl (t := level b);;
  mp-free-loopinv t b α
  WHILE ´free-block-r t DO
  mp-free-cnd1 t b α ≡ mp-free-loopinv t b α ∩ {| α > 0 |}
    t ► ´lsz := ´lsz (t := ´lsizes t ! (´lvl t));;
  mp-free-cnd2 t b α ≡ mp-free-cnd1  t b α ∩ {| ´lsz t = ´lsizes t ! (´lvl t) |}
    t ► ´blk := ´blk (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bn t));;
  mp-free-cnd3 t b α ≡ mp-free-cnd2  t b α ∩
                      {| ´blk t = block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bn t) |}
    t ► ATOM
    {V1} (V1 ∈ mp-free-cnd3  t b α ∩ {|´cur = Some t|})
      ´mem-pool-info := set-bit-free ´mem-pool-info (pool b) (´lvl t) (´bn t);;
    {V2} (V2 = V1(|mem-pool-info :=
                    set-bit-free (mem-pool-info V1) (pool b) (lvl V1 t) (bn V1 t)|))
      ´freeing-node := ´freeing-node (t := None);;
    {V3} (V3 = V2(|freeing-node := (freeing-node V2)(t := None)|))
      IF ´lvl t > 0 ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t) THEN
      (V3 ∈ {|NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)|})
    mergeblock-loopinv V3  t b α ≡
    {V. let minf0 = (mem-pool-info V3)(pool b);  lvl0 = (levels minf0) ! (lvl V3 t);
```

```
      minf1 = (mem-pool-info V)(pool b);  lvl1 = (levels minf1) ! (lvl V3 t) in
    (bits lvl1 = list-updates-n (bits lvl0) ((bn V3 t div 4) * 4) (i V t) NOEXIST)
    ∧ (free-list lvl1 = removes (map (λii. block-ptr minf0 (lsz V3 t)
                                  ((bn V3 t div 4) * 4 + ii)) [0..<(i V t)]) (free-list lvl0))
    ∧ (wait-q minf0 = wait-q minf1) ∧ (∀ t'. t' ≠ t ⟶ lvars-nochange t' V V3)
    ∧ (∀ p. p ≠ pool b ⟶ mem-pool-info V p = mem-pool-info V3 p)
    ∧ (∀ j. j ≠ lvl V3 t ⟶ (levels minf0)!j = (levels minf1)!j)
    ∧ (V,V3)∈gvars-conf-stable ∧  i V t ≤ 4  ∧  ∧  α = 4 - i V t ..... }
    FOR ´i := ´i(t := 0); ´i t < 4; ´i := ´i(t := ´i t + 1) DO
     mergeblock-loopinv V3  t b α ∩ {| α > 0 |}
       {V4} (V4 ∈ mergeblock-loopinv V3  t b α ∩ {| α > 0 |} )
       ´bb := ´bb (t := (´bn t div 4) * 4 + ´i t);;
       {V5} (V5 ≡ V4(|bb := (bb V) (t:=(bn V4 t div 4) * 4 + i V4 t)|) )
       ´mem-pool-info := set-bit-noexist ´mem-pool-info (pool b) (´lvl t) (´bb t);;
       {V6} (V6 ≡ V5(| mem-pool-info :=
                        set-bit-noexist (mem-pool-info V5) (pool b) (lvl V5 t) (bb V5 t) |) )
       ´block-pt := ´block-pt (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bb t));;
       {V7} (V7 ≡ V6(|block-pt := (block-pt V6)
                      (t:=block-ptr (mem-pool-info V6 (pool b)) (lsz V6 t) (bb V6 t))|) )
       IF ´bn t ≠ ´bb t ∧ block-fits (´mem-pool-info (pool b)) (´block-pt t) (´lsz t) THEN
          ´mem-pool-info := ´mem-pool-info ((pool b) :=
               remove-free-list (´mem-pool-info (pool b)) (´lvl t) (´block-pt t))
       FI
    ROF;;
     mergeblock-loopinv V3  t b α ∩ {| α = 0 |}
    ´lvl := ´lvl (t := ´lvl t - 1);;
    ´bn := ´bn (t := ´bn t div 4);;
    ´mem-pool-info := set-bit-freeing ´mem-pool-info (pool b) (´lvl t) (´bn t);;
    ´freeing-node := ´freeing-node (t := Some (|pool = (pool b), level = (´lvl t),
          block = (´bn t), data = block-ptr (´mem-pool-info (pool b))
           (((ALIGN4 (max-sz (´mem-pool-info (pool b))))) div (4 ^ (´lvl t)))) (´bn t) |))
  ELSE
    {V3} ∩ - {|NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)|}
    IF block-fits (´mem-pool-info (pool b)) (´blk t) (´lsz t) THEN
       ´mem-pool-info := ´mem-pool-info ((pool b) :=
          append-free-list (´mem-pool-info (pool b)) (´lvl t) (´blk t) )
    FI;;
    ´free-block-r := ´free-block-r (t := False)
  FI
 END  (* END of ATOM *)
OD  (* END of WHILE  free_block_r  DO *)
mp-free-precond9 t b ≡ Mem-pool-free-pre t ∩ {| g |}
(* === end of : free-block(pool, level, lsizes, block); ===*)
t ▶ ATOMIC
{Va} (Va ∈ mp-free-precond9  t b ∩ {|´cur = Some t|})
 stm9-loopinv Va t b α ≡
  {V. inv V ∧ cur V = cur Va ∧ tick V = tick Va ∧ (V,Va)∈gvars-conf-stable
    ∧ freeing-node V t = freeing-node Va t ∧ allocating-node V t = allocating-node Va t
    ∧ (∀ p. levels (mem-pool-info V p) = levels (mem-pool-info Va p))
    ∧ (∀ p. p ≠ pool b ⟶ mem-pool-info V p = mem-pool-info Va p)
```

```
      ∧ (∀ t'. t' ≠ t ⟶ lvars-nochange t' V Va)
      ∧ α = length (wait-q (´mem-pool-info (pool b))) }
    WHILE wait-q (´mem-pool-info (pool b)) ≠ [] DO
      stm9-loopinv Va t b α ∩ {| α > 0 |}
      ´th := ´th (t := hd (wait-q (´mem-pool-info (pool b))));;
      (* -unpend-thread(th); *)
      ´mem-pool-info := ´mem-pool-info (pool b := ´mem-pool-info (pool b)
          (|wait-q := tl (wait-q (´mem-pool-info (pool b)))|));;
      (* -ready-thread(th); *)
      ´thd-state := ´thd-state (´th t := READY);;
      ´need-resched := ´need-resched(t := True)
    OD;;
    stm9-loopinv Va t b α ∩ {| α = 0 |}
    IF ´need-resched t THEN
      reschedule    (* _reschedule_threads(key) *)
    FI
  END  (* END of ATOM *)
END
Mem-pool-free-post t ≡ {| ´inv ∧ ´allocating-node t = None ∧ ´freeing-node t = None|}
```