A Parametric Rely-Guarantee Reasoning Framework for Concurrent Reactive Systems

Yongwang Zhao[1,2(✉)], David Sanán[3], Fuyuan Zhang[4], and Yang Liu[3]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
[2] Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, China
zhaoyw@buaa.edu.cn
[3] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore
[4] MPI-SWS, Kaiserslautern, Germany

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

This work has been supported in part by the National Natural Science Foundation of China (NSFC) under the Grant No.61872016, and the National Satellite of Excellence in Trustworthy Software Systems and the Award No. NRF2014NCR-NCR001-30, funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

© Springer Nature Switzerland AG 2019
M. H. ter Beek et al. (Eds.): FM 2019, LNCS 11800, pp. 161–178, 2019.
https://doi.org/10.1007/978-3-030-30942-8_11

sequence of commands under an input event. Some examples of CRSs are operating systems (OSs), control systems, and communication systems, which implementation follow 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. [18,20,23,24,28]) 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 dedicated statements for such system behavior, we often use imperative programs to simulate them, making the formal specification cumbersome, in particular the rely-guarantee conditions. A more detailed motivation will be presented in detail in Sect. 2.

In this paper, we propose *PiCore*, a two-level event-based rely-guarantee framework for CRSs (Sect. 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" [2,6] 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 composing the first level. *PiCore* parametrizes the second level using a rely-guarantee interface, allowing 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*: *πIMP* and *πCSimpl* (Sect. 4). *πIMP* integrates the *HOL-Hoare_Parallel* library in Isabelle/HOL that uses a general imperative language [23]. *πCSimpl* 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 https://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] (Sect. 5). The formal specification represented in *πIMP* 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 *πIMP* 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 the 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 sharing data with its environment and invoking the relevant handlers in the loop body (e.g. [4]). First, The environment non-deterministically decides 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 execution traces of reaction sequences rather than program states only. Without native support in the 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 cause of the above problems is the 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 while being unchanged. At the 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 additional parameter κ which indicates the execution context, e.g. the thread invoking the service and the external devices

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

triggering the interrupt. The execution of an event system concerns the continuous evaluation of guards of the events with their input arguments. From the set of events for which their 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 guards starts again looking for the next event to be executed. We call the semantics of event systems *reactive semantics*, where the event context shows the event currently being executed. A CRS is modeled as the *parallel composition* of event systems that are concurrently executed.

As shown in the Zephyr case study in Sect. 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:
$\mathcal{PS}$ ::= $\mathcal{K} \rightarrow \mathcal{S}$

Fig. 2. Abstract syntax of *PiCore*

The abstract syntax of *PiCore* and its semantics are shown in Figs. 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. When **Event** (l, g, P) is triggered, 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 Sect. 4. Events are parametrized in the meta-logic as "$\lambda(plist, \kappa)$. **Event** (l, g, P)", where *plist* is the list of input parameters, and κ is the event system identifier that the event belongs to. These parameters are not part of the syntax of events to make the guard g and the event body P, as well as the rely and guarantee relations, more flexible, allowing to define different instances of the relations for different values of *plist* and κ.

An event system has two forms that we call *event sequence* and *event set*. Event sequences model a sequential execution of events, and event sets model

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

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

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

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

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

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

Fig. 3. Operational semantics of *PiCore*

the continuous execution of events from the evaluation of the guards of the events in the set. When the system is not executing any event, one event whose guard condition holds in the current state is non-deterministically chosen to be 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 $x : \mathcal{K} \rightarrow \mathcal{E}$ is an event context. The event context indicates which event is currently being executed in an event system κ. 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 or an occurrence of an event $\mathcal{E}$. $@\kappa$ means that the action occurs in event system κ. The program transition is denoted as $\longrightarrow_p$ in the TrgdEvt rule. Environment transition rules have the form $\Sigma \vdash (\sharp, s, x) \xrightarrow{env}_\square (\sharp, s', x')$. Intuitively, a transition made by the environment may change the state but not the event context nor 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.

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

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 [19,21,22].

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 x_{ϖ_i} represent the projection of each component in the tuple $\varpi_i = (\sharp, s, x)$.

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

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

We define validity of rely-guarantee specification for events as

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

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

[BasicEvt]

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

[Conseq]

$$\frac{pre \subseteq pre' \quad R \subseteq R' \quad G' \subseteq G \quad pst' \subseteq pst \quad \Sigma \vdash \sharp\ \textbf{sat}\ \langle pre', R', G', pst'\rangle}{\Sigma \vdash \sharp\ \textbf{sat}\ \langle pre, R, G, pst\rangle}$$

[TrgEvt]

$$\frac{\Sigma \vdash P\ \textbf{sat}\ \langle pre, R, G, pst\rangle}{\Sigma \vdash (\lfloor P \rfloor)\ \textbf{sat}\ \langle pre, R, G, pst\rangle}$$

[EvtSeq]

$$\frac{\Sigma \vdash \mathcal{E}\ \textbf{sat}\ \langle pre, R, G, m\rangle \quad \Sigma \vdash \mathcal{S}\ \textbf{sat}\ \langle m, R, G, pst\rangle}{\Sigma \vdash (\mathcal{E} \rhd \mathcal{S})\ \textbf{sat}\ \langle pre, R, G, pst\rangle}$$

[EvtSet]

$$\frac{\begin{array}{l}(1)\forall i \leq n.\ \Sigma \vdash \mathcal{E}_i\ \textbf{sat}\ \langle pres_i, Rs_i, Gs_i, psts_i\rangle \quad (2)\forall i, j \leq n.\ psts_i \subseteq pres_j \\ (3)\forall i \leq n.\ pre \subseteq pres_i \quad (4)\forall i \leq n.\ R \subseteq Rs_i \quad (5)\forall i \leq n.\ Gs_i \subseteq G \\ (6)\forall i \leq n.\ psts_i \subseteq pst \quad (7) stable(pre, R) \quad (8)\forall s.\ (s,s) \in G\end{array}}{\Sigma \vdash (\{\mathcal{E}_0,\ ...,\ \mathcal{E}_n\})\ \textbf{sat}\ \langle pre, R, G, pst\rangle}$$

[Par]

$$\frac{\begin{array}{l}(1)\forall \kappa.\ \Sigma \vdash \mathcal{PS}(\kappa)\ \textbf{sat}\ \langle pres_\kappa, Rs_\kappa, Gs_\kappa, psts_\kappa\rangle \quad (2)\forall \kappa.\ pre \subseteq pres_\kappa \quad (3)\forall \kappa.\ R \subseteq Rs_\kappa \\ (4)\forall \kappa.\ Gs_\kappa \subseteq G \quad (5)\forall \kappa.\ psts_\kappa \subseteq pst \quad (6)\forall \kappa, \kappa'.\ \kappa \neq \kappa' \longrightarrow Gs_\kappa \subseteq Rs_{\kappa'}\end{array}}{\Sigma \vdash \mathcal{PS}\ \textbf{sat}\ \langle pre, R, G, pst\rangle}$$

Fig. 4. Rely-guarantee proof rules for *PiCore*

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 validity. The proof rules are shown in Fig. 4, which give 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 assume stability of the precondition with regards to the rely relation $stable(pre, R)$ to ensure that the precondition holds after 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 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.

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

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*. $\{s \mid P(s)\}$ denotes the set of states s satisfying P.

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** $\langle init, R, G, post\rangle$.
- $init \subseteq \{s \mid inv(s)\}$.
- $stable(\{s \mid inv(s)\}, R)$ *and* $stable(\{s \mid inv(s)\}, G)$ *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 a flexible integration of languages for programs on event bodies, *PiCore* provides a rely-guarantee interface that 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 *assumptions* to guarantee the correctness of the proof system, since the language, semantics, proof rules and soundness proof of *PiCore* in Sect. 3 are developed using this interface.

Following this interface, third-party languages and their rely-guarantee proof systems are embedded into *PiCore* as *interpretations* using an *adapter* that implements the interface. Since these languages may have existed for years, they are not necessary completely consistent with the *PiCore* interface. Hence, for each language that we want to integrate in *PiCore* 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 makes it possible to integrate existing languages without modifying their specification, semantics, and rely-guarantee inference system.

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*

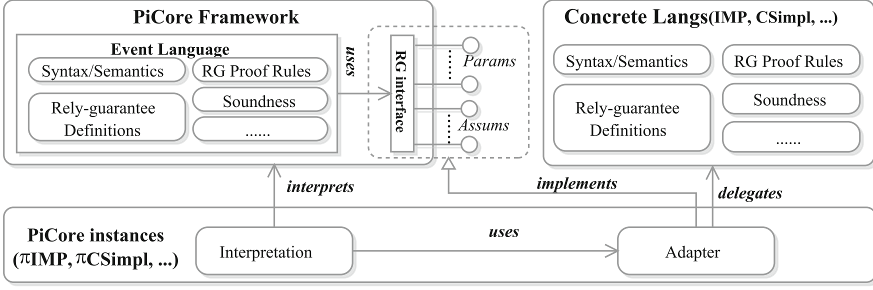

Fig. 5. *PiCore* framework and its integration with imperative languages

assumes that (1) program $\bot$ 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 Sect. 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 validity are similar to those in *PiCore* (see Sect. 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.

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	$\Sigma \vdash (P, s) \xrightarrow{env}_p (Q, t)$	(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	$\Sigma \models P$ **sat** $\langle pre, R, G, pst\rangle$
(9)	Proof rule	$\Sigma \vdash P$ **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)	$\Sigma \vdash (P, s) \xrightarrow{env}_p (Q, t) \Longrightarrow P = Q$	(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)	$\Sigma \models P$ **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)	$(\forall i < len(\varpi) - 1.\ (\Sigma \vdash \varpi_i \xrightarrow{env}_p \varpi_{i+1}) \longrightarrow (s_{\varpi_i}, s_{\varpi_{i+1}}) \in R) \wedge s_{\varpi_0} \in pre$ $\Longrightarrow \varpi \in A(\Sigma, pre, R)$		
(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)	$\Sigma \vdash P$ **sat** $\langle pre, R, G, pst \rangle \Longrightarrow \Sigma \models P$ **sat** $\langle pre, R, G, pst \rangle$		

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 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. Using *locale interpretations*, 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. 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*; a set of procedure names of type *'p*, and a set of simulation events *'e* (simulation events are not addressed in this work). Type *('s,'p,'f,'e) config* defines the configuration used in its transition semantics and *('s,'p,'f,'e) body* denoted as Γ defines the procedure declarations as mapping from procedure

datatype ('s, 'p, 'f, 'e) *com* = *Skip* | *Throw* | *Basic* 's 'e ⇒ 's | *Spec* ('s × 's) *set* 'e
| *Seq* ('s ,'p, 'f, 'e) *com* ('s, 'p, 'f, 'e) *com* | *Await* 's *bexp* 'e ('s, 'p, 'f, 'e) *com*
| *Cond* 's *bexp* ('s, 'p, 'f, 'e) *com* ('s, 'p, 'f, 'e) *com*
| *While* 's *bexp* ('s, 'p, 'f, 'e) *com* | *Call* 'p | *DynCom* 's ⇒ ('s, 'p, 'f, 'e) *com*
| *Guard* 'f 's *bexp* ('s, 'p, 'f, 'e) *com* | *Catch* ('s, 'p, 'f, 'e) *com* ('s, 'p, 'f, 'e) *com*

datatype ('s,'f) *xstate* = *Normal* 's | *Abrupt* 's | *Fault* 'f | *Stuck*
type-synonym('s, 'p, 'f, 'e) *config* = ('s, 'p, 'f, 'e)*com* × ('s,'f) *xstate*
type-synonym ('s, 'p, 'f, 'e) *body* = 'p ⇒ ('s, 'p, 'f, 'e) *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]

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.

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 $\Gamma \vdash_c (P, s) \longrightarrow (Q, t)$, and it 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 any other state. Therefore, the environment transition in *CSimpl* is defined as follows.

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

To adapt the restricted environment transition, we first define the environment transition in the adapter as $(\Gamma, \Theta) \vdash_{cI} (P, s) \longrightarrow_{env} (P, t)$, 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 Fig. 4, since postconditions of events must imply their preconditions, and preconditions in *CSimpl* are sets of normal states, a final configuration of an event cannot throw an exception.

Finally, based on the definition of the proof rules $\Gamma,\Theta \vdash_{/F}$ P sat [q, R, G, q,a] in *CSimpl*, we define proof rules 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) Programs 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 *CSimpl*. (4) The environment transition in *CSimpl* does not allow transitions from a non *Normal* state to a different state, we represent it in the rely condition R. (5) Finally, the rely-guarantee specification for each procedure in Θ has to be satisfied.

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

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 (for more detail refer to [29]). 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. 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.

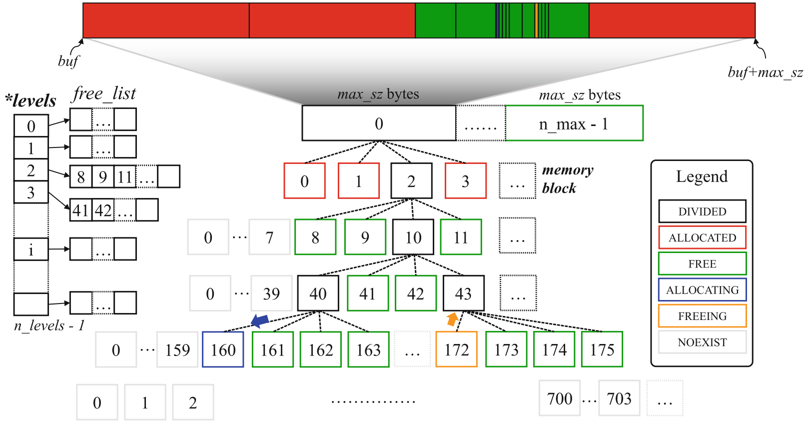

Fig. 7. Structure of memory pools

Thread preemption and fine-grained locking make kernel execution of memory services concurrent. Zephyr provides two kernel services *k_mem_pool_alloc* and *k_mem_pool_free*, for memory allocation and release respectively. When an application requests a memory block, Zephyr first computes a value $free_l$ that is the lowest level containing free memory blocks. 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 concurrent threads. In such a case the service will back out to retry. Allocation supports a *timeout* 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 derive a critical property to prevent memory leaks: memory blocks cover the whole memory address of the pool, but do 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.

$\textbf{addr-in-block}\ mp\ addr\ i\ j \equiv$
$i < length\ (levels\ mp) \wedge j < length\ (bits\ (levels\ mp\ !\ i))$
$\wedge\ (is_memblock(bits\ (levels\ mp\ !\ i)\ !\ j))$
$\wedge\ addr \in \{x \mid j * (max\text{-}sz\ mp\ div\ (4\ \hat{}\ i)) \leqslant x < Suc\ j * (max\text{-}sz\ mp\ div\ (4\ \hat{}\ i))\}$
$\textbf{mem-part}\ s \equiv \forall p{\in}mem\text{-}pools\ s.\ \textbf{\textit{let}}\ mp = mem\text{-}pool\text{-}info\ s\ p\ \textbf{\textit{in}}$
$(\forall\ addr < n\text{-}max\ mp * max\text{-}sz\ mp.\ (\exists!(i,j).\ addr\text{-}in\text{-}block\ mp\ addr\ i\ j)\ )$

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:*

$$\textit{\textbf{inv_mempool_info}}\ s \wedge \textit{\textbf{inv_bitmap}}\ s \wedge \textit{\textbf{inv_bitmap0}}\ s \wedge \textit{\textbf{inv_bitmapn}}\ s \Longrightarrow \textit{\textbf{mem_part}}\ s$$

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

$$\begin{aligned} esys_{t_i} \equiv & (\bigcup blk.\ \{mem_pool_free[blk]@t_i\}) \cup \\ & (\bigcup (p, sz, tmout).\ \{mem_pool_alloc[p, sz, tmout]@t_i\}) \end{aligned}$$

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. Events are parametrized by a thread identifier t_i used to control access to the execution context of the thread invoking it. 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.

Using the compositional reasoning of *πIMP*, 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:

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

This relation states that a step from *alloc* or *free* may not change the state (1), e.g., selecting a 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 to the requested capacity.

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 *πIMP* and *πCSimpl* 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 more 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. [13,14,23,24,26]) and Coq (e.g. [18,20]). In [13,14], 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]. None of both work consider any concrete imperative languages for rely-guarantee. The works [20,23] mechanize the rely-guarantee approach for simple imperative languages. Later, a rely-guarantee proof system is developed in Isabelle/HOL for *CSimpl* [24], a generic and realistic concurrent imperative language by extending the sequential language *Simpl* [25]. These mechanizations focus on imperative languages for pure programs, of which two of them [23,24] 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]. 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 usefulness 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, Cambridge (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 Comput. **8**(3), 324–346 (1996)
6. Back, R.J., Sere, K.: Stepwise refinement of action systems. Struct. Program. **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 Comput. **10**(3), 193–213 (1998)
9. Fenkam, P., Gall, H., Jazayeri, M.: Composing specifications of event based applications. In: Pezzè, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 67–86. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36578-8_6
10. Fenkam, P., Gall, H., Jazayeri, M.: Constructing deadlock free event-based applications: a rely/guarantee approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 636–657. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_35
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 (1998)
12. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 13, pp. 477–498. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_17
13. Hayes, I.J.: Generalised rely-guarantee concurrency: an algebraic foundation. Formal Aspects Comput. **28**(6), 1057–1078 (2016)

14. Hayes, I.J., Colvin, R.J., Meinicke, L.A., Winter, K., Velykis, A.: An algebra of synchronous atomic steps. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 352–369. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_22
15. Hoang, T.S., Abrial, J.-R.: Event-B decomposition for parallel programs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 319–333. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_24
16. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
17. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Proceedings of ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, Big Sky, Montana, USA, pp. 207–220. ACM Press (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. Universidade do Porto, Technical report (2013)
21. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 126–142. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_12
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: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 348–362. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36575-3_24
24. Sanán, D., Zhao, Y., Hou, Z., Zhang, F., Tiu, A., Liu, Y.: CSimpl: a rely-guarantee-based framework for verifying concurrent programs. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 481–498. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_28
25. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technical University Munich (2006)
26. van Staden, S.: On rely-guarantee reasoning. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 30–49. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_2
27. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 59–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_4
28. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997)
29. Zhao, Y., Sanán, D.: Rely-guarantee reasoning about concurrent memory management in Zephyr RTOS. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 515–533. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_29