Intersert: Assertions on Distributed Process Interaction Sessions

Zack Newsham
University of Waterloo
znewsham@uwaterloo.ca

Augusto Born de Oliveira
University of Waterloo
a3olivei@uwaterloo.ca

Jean-Christophe Petkovich
University of Waterloo
j2petkovich@uwaterloo.ca

Ahmad Saif Ur Rehman
QNX Software Systems
asaifurrehman@qnx.com

Guy Martin Tchamgoue
University of Waterloo
gmtchamg@uwaterloo.ca

Sebastian Fischmeister
University of Waterloo
sfischme@uwaterloo.ca

***Abstract*—Program assertions typically operate on available program state such as global and local variables. To support sophisticated assert statements such as invariants on control flow or inter-process communication patterns, developers must design and maintain supporting infrastructure. It is non-obvious how to realize this infrastructure: how to maintain the data, how to access it, how to use it in assertions, how to keep the overhead low enough for embedded systems, and how to manage assertions across a distributed system.**

This work demonstrates the utility of assertions on interaction history among distributed system components and solves the challenges of efficiently maintaining interaction data while providing an expressive interface for assertions. Our toolchain enables developers to program assertions on interaction history written in regular expressions that incorporate inter-process and inter-thread behavior amongst multiple components in a distributed system. We demonstrate that the interaction tracking and property verification systems incur negligible overhead, measured with several benchmarks. This work discusses our toolchain with a real-world safety-critical embedded system.

***Index Terms*—Assertions, Regular Expressions.**

I. Introduction

Assertions are a widely used method for increasing program reliability and enhancing debugging as they permit checking program state against a specified statement at run time. Developers use assertions to check whether their assumptions about the state of a program are true by the time an assertion is executed. This concept is useful for achieving a variety of goals [1] including testing software correctness, detecting software defects, and isolating faults. Assertions operate on program state information, which is commonly encoded in global and local variables. For example, the assertion below would check that an input buffer is large enough to hold the data required by an application.

```
assert(sizeof(input_buffer) >= MIN_BUFFER_SIZE)
```

An important type of information that traditional assertions do not handle well is the history of interactions between threads or processes. Interaction properties that a developer might want to assert include, e.g., that Process A communicates with Process B before opening a file, or that a pair of redundant sensors are both read before their values are used in any calculations. In contrast to assertions on program state, assertions on interaction history require supporting infrastructure. Often, it is the developer who creates and manages this infrastructure, and develops specific state-based checking rules for properties. This additional development effort incurs extra costs and introduces another potential source of defects.

Adaptations of traditional assertions to operate on interaction histories are already been used in modern software. The Canadian Darlington nuclear plant uses interaction history to verify whether a particular set of actions has occurred in the correct sequence [2]. The GNU C Compiler (GCC) and the Linux kernel use assertions on specific program interactions with manually coded supporting infrastructure. For example, GCC 4.4 has 41 assert statements (that the authors are aware of) which check whether a particular action was completed before continuing execution. Finally, the Apache Portable Runtime uses assertions to prevent threads in thread-pools from interacting with task abstractions already taken by other threads. In fact, any standard concurrency problem such as producer-consumer, or reader-writer could benefit from assertions on interaction history.

We propose that, instead of requiring each developer to implement its own infrastructure, the operating system (OS) should provide support for recording and verifying assertions on interactions. This aligns with the original intention of assert statements, which is to increase reliability and provide debugging support. However, it is unclear what the semantics should be and how the OS should maintain the interaction history, how access and control over the information should be provided to developers, how the information should be used in assertions, and how to minimize runtime overhead.

This paper introduces Intersert, an infrastructure for programming assertions on interaction history of threads in a distributed system. Intersert assertions contain regular expressions or *regex* [3] placed on interaction history. This work shows that interaction history can be provided with negligible overhead and that this information is particularly useful in combination with assertions. The contributions of this work are the demonstration of: (1) the utility of exposing interaction history to assertions, (2) the means of exposing the interaction

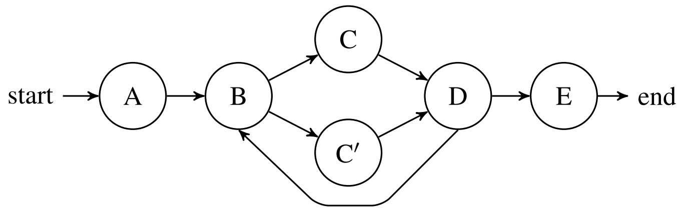


Fig. 1. Interaction diagram of the example application.

history of threads residing on physically distributed nodes, (3) the applicability of regex for checking properties on interaction histories at programming level, and (4) the feasibility of providing this functionality with negligible runtime overhead in a fully working toolchain on a commercial platform used for safety-critical applications.

II. Assertions on Interaction History

To illustrate the use of assertions on interaction history, let us consider the example application of Fig. 1, which represents threads as circles with their interactions as arrows. The application periodically takes sensor readings in Thread A. Each reading is processed by a pipeline of threads until, eventually, Thread E returns the final validated result. Thread B routes the readings through either Thread C or C′ based on the reading quality. Thread D can return the reading to Thread B for further processing, or forward it to Thread E for finalization.

To verify the proper operation of the system in Fig. 1, the developer can use regex assertions on a string that represents interaction history. As data passes through threads, the interaction history is tracked by appending the receiving thread names to a string (e.g., [A, B, C]). For instance, the following assertions could be verified on the system:

- **Check that Thread A initiates all interactions.** This assertion could be checked in any thread. In Thread E, the assertion statement can state the property in POSIX regex as "A.*". If Thread A is not in the interaction history, or if any other thread initiates it, the assertion will fail.
- **Check that a reading that reaches Thread C′ has not yet gone through Thread C.** This assertion ensures that a validated reading is not returned to Thread B. This is checked in Thread C′ with "[ˆC]*", with the caret (ˆ) being the negation symbol. This assertion will fail if Thread C is in the history when Thread C′ receives the reading, signaling an unintended sequence of events.
- **Check that only Thread D initiates interactions with Threads B or E.** This assertion is checked in Thread E by verifying that the regex ".*D[ˆBE].*" does *not* produce a match. This regex states that the only thread that may follow D in the interaction history is either B or E.

All the assertions described above would pass when applied to interaction histories [A, B, C, D, E] or [A, B, C′, D, B, C, D, E]. However, the history, [A, B, C, D, B, C′], will fail the second assertion in Thread C′ as it contains Thread C. This simple example shows the utility of placing assertions on interaction history. The challenge is how to realize this functionality in a user-friendly manner with low overhead, hiding as much complexity as possible from the developer and with support of distributed applications. Our framework, called Intersert, demonstrates how these goals can be achieved.

III. Intersert

This section provides details on the Intersert framework.

A. Description

Intersert provides a variation on standard assert functions. Developers can use intersert() statements to verify properties in the form of a *regex* on the history of interactions among threads and processes on a single or distributed system. Intersert uses *regex*, because it is an intuitive and familiar way to verify properties on sequences of characters, which represent events in our case. More, regex are expressive, which allows users to easily define interaction behaviors. Also, there is wide variety of tools available to match regex properties given a trace.

An *interaction* simply indicates an inter-thread message passing. An *interaction history* is an ordered list of threads that communicate in a continuous interaction. An interaction history is similar to a list of participating threads in a UML sequence diagram [4]. Intersert serializes these lists into strings that a finite state machine scanner can consume. Characters in interaction strings mark the presence of a thread in the history.

The framework uses POSIX compliant regex syntax in intersert() statements. There are three parameters to an intersert() call: the *regex property* the user wants to verify, a *session name* that uniquely specifies the session on which to apply the assertion, and an *expected value* that defines what the regex matcher should return for the assertion to pass. For instance, intersert("AB", "my_session", true) will pass only if the interaction session named `my_session` consists exactly of [A,B]. By default, the violation of a property aborts the program, but user-defined actions may overwrite this behavior.

Since intersert() statements check for properties at a given point in time and not properties that are valid at all times, i.e., invariants, liveness properties (e.g., eventually, A will always happen) are not supported. Model checking [5] is more appropriate for the verification of invariants. Developers construct a full model of the system and use it to exhaustively check properties. This is a significantly more computationally intensive operation and is not the focus of this work.

B. Developer's Perspective

The use of intersert() statements is similar to that of regular assert() statements. As an assert() indicates developer's assumptions about a program state, each intersert() statement captures assumptions about the interaction history prior to the execution of the statement. However, there are three differences between intersert() and assert() statements: (1) threads must uniquely register themselves in a lookup directory to appear in the interaction history, (2) intersert() statements use regex, and (3) operate on a finite *interaction session*.

The lookup directory enables developers to refer to threads as characters in intersert() statements and is broadcast to all

```
1 void A () {
    fill_id("A"); //called once at thread start
3   start_session("my_session");
    data = read_sensor();
5   msg_send(B, data);
  }
7 void B () {
    fill_id("B");  //called once at thread start
9   ...
    msg_receive(&data);
11  if(is_valid(data))
         msg_send(C, data);
13  else
         msg_send(Cprime, data);
15 }
  void Cprime () {
17  fill_id("Cprime");  //called at thread start
    msg_receive(&data);
19  intersert("[^C]*", "my_session", true);
    ...
21  msg_send(D, data)
  }
23 void E () {
    fill_id("E");  //called once at thread start
25  msg_receive(&data);
    intersert("A.*", "my_session", true);
27  intersert(".*D[^BE].*", "my_session", false);
    ...
29  end_session("my_session");
  }
```

Listing 1. Example usage of the intersert() statement.

nodes of the network. Intersert transparently supports any communication between threads residing on different nodes, with no additional effort from users. Thus, to uniquely identify the threads of Fig. 1, one would use the fill_id() function. When fill_id("A") is called at thread start, the calling thread is attached to symbol "A". Thus, whenever this thread participates in an interaction, Intersert records an "A" in the history string. Intersert enforces thread label uniqueness across all nodes.

Listing 1 shows sections of code for the example of Fig. 1, with the exception of Threads C and D. Line 26 shows the first assertion, intersert("A.*", my_session, true), which evaluates to true only if Thread A initiates the history. The assertion, intersert("[^C]*", my_session, true), at Line 19, holds only if Thread C is absent from the history prior to calling Thread C$'$.

C. Interaction Sessions

Interaction sessions are periods of execution during which interaction histories are recorded. Thus, a session also implicitly represents the interaction history recorded during its lifetime. When a session is started with start_session(), a new *baton* with a unique identifier is created to symbolize the new session. A baton can be seen as a special synchronization protocol which grants update rights on the underlying interaction history to its holder. Each call to start_session() takes the session name as parameter. Any intersert() call targeting a session references its name. This naming mechanism allows to verify different properties on different sessions that share some (or all, or none of the same) threads. A thread does not have to hold the baton to make an intersert() statement on its session. This creates a more flexible environment for assertions. Batons are passed along with every interactions between threads. Upon receiving the baton, a thread automatically adds itself to the interaction history. Calling end_session() destroys the named baton and its associated interaction history. This mechanism removes the need for branching in interaction history, guaranteeing linearity in the sequence of participating threads. This also simplifies the expression of properties, while still allowing the verification of a wide range of properties.

Interaction sessions allow users to (1) define boundaries for the interaction history, and (2) define multiple, concurrent interaction histories which are maintained independently. Placing limits on the history is important for practical concerns as the time for verifying properties depends on the length of the history. It is of interest to the developer to keep histories as short as possible. Allowing concurrent sessions is important as periodic interactions, such as the ones initiated by Thread A in Fig. 1, require properties to be verifiable on separate interaction histories. In Listing 1, a new session starts each time Thread A takes a sensor reading at Line 3. Intuitively, this means that multiple readings pass through the system at any given time, and that properties are checked on a per-reading basis on separate history sessions. After Thread E has processed a reading, it closes the session at Line 29.

It is worth noting that, while a session must start at some point to start recording history information, it needs not end. In this case, the system continues to record interactions, and any intersert() call will verify properties over the entire history. This is useful for systems that enter a steady state with no clear session boundaries, but where invariants such as "T_x is never present" are of interest. Infinite history is obviously impossible to implement, hence, Intersert currently relies on user settings (see Section IV) to work around this limitation.

IV. Runtime Support

Intersert mainly consists of three components. The history recorder which uses an online interaction tracker to gather interaction sessions. The second component, which distributes the interaction history across nodes, is implemented inside the kernel as part of the QNET messaging protocol. Finally, the verification mechanism checks the validity of intersert() calls.

A. Recording Interaction Sessions

We implemented Intersert on the QNX Neutrino microkernel, a popular commercial real-time operating system used in safety-critical systems such as nuclear plants and automotive.

In a microkernel, interactions manifest as messages passed between threads. To maintain the interaction history, our runtime system snoops these messages to record associated metadata. Snooping interactions with low enough overhead for use in embedded systems is a challenge. Our approach builds upon the *mTags* infrastructure [6], which attaches user-defined metadata, called *tags*, to processes and threads. By means of a modified message passing routine in the kernel, mTags provides a hook to implement additional actions on messages. Depending on the function added to the hook, one can implement various semantics for tag propagation, such as tag duplication or baton passing. Tag duplication, the default passing semantics, *copies* tags to other threads upon interaction. Baton passing, however, *hands over* the tag from one thread to another. While not default to the mTags system, it was a trivial change to allow for both tags, and batons.

Intersert uses mTags with *baton passing* semantics to implement sessions. Thus, when Thread A, which currently holds a tag, interacts with Thread B, Thread B receives Thread A's tag, while Thread A loses its tag. A call to start_session() creates a new session and instantiates a new tag uniquely identifying that session. Due to memory constraints, the interaction history is stored in a circular buffer. The length of this buffer is configurable, and should be large enough to track all properties of interest. Additional refinements could handle session overflows. The number of concurrent sessions is also limited by the available memory; in our implementation, a default of 32 sessions is used, but can be tuned by the developer.

B. Distributing Session History

Due to the linearity of interaction histories, the history currently on node N_a needs only be transmitted to node N_b when the thread in possession of the *baton* on N_a sends a message to a thread on N_b. As the sender thread relinquishes control of the *baton* at this point, there is a guarantee that no other thread on any node other than N_b will be able to modify the interaction history, until another QNET message pass occurs. Since the QNET protocol requires far less space than the size of the underlying protocol (in our tests TCP/IP), there is space available in each QNET message pass to be utilized by the interaction history, without exceeding the maximum packet size. As such, it was possible to implement this distributed feature with negligible overhead. The changes required to the QNET module were likewise very efficient, only those messages containing transaction histories are affected and this only by the amount of time taken to copy the history.

C. Processing intersert() Statements

At run time, a call to intersert() results in a call to the verifier module, which retrieves the interaction history of the named session. The regex pattern and session name from the call are passed on to the verifier module which checks the regex against the session. The verifier processes the history from its earliest recorded point, visiting each entry. On completion, a truth value is returned and compared against the intersert()'s third argument, triggering an error if they differ.

V. Formal Model

To clarify how Intersert works, we specify a formal model for the creation, deletion and transferring of tags, and show that these are the only events that affect the outcome of an Intersert call. The model assumes a single session but can easily be extended to multiple sessions.

The system state consists of a set of processes P, a set of tags T, and a global clock C. Our lifeline log L is a list of tuples with $L = \langle c, t, p_1, p_2 \rangle$ with a timestamp c taken from C, a tag $t \in T$, and two processes $p_1, p_2 \in P$ for which p_1 is the sender and p_2 the receiver. The log L is ordered by the timestamps c. The assignment function $A : T \rightarrow 2^P$ specifies which tags are currently assigned to which processes. Finally, we use e to indicate the outcome of intersert() statements with $e = \top$ meaning *passed* and $e = \bot$ for *failed*.

The overall system state $S = \langle P, T, A, L, e \rangle$ contains all information and the initial state of the system is that where all processes $p_x \in P$ exist, the exit state may be $e = \top$, but the sets T, A, and L are empty, i.e., no tags, no assignments and no entries in the lifeline log. The system then works through a list of actions E and updates its state. We define $E = \epsilon_1 \epsilon_2 \dots \epsilon_n$ to be a list of allowed actions where $\epsilon_x = \langle \alpha, t \rangle$ with an operation $\alpha \in \{\text{createtag}, \text{deletetag}, \text{settag}, \text{unsettag}, \text{append}, \text{intersert}\}$ and a tag $t \in T$. We use ϵ^* as the suffix as in $\epsilon_x \epsilon^*$ where ϵ^* would be a list of all actions after ϵ_x.

Table I shows the semantics for Intersert actions. Many of the rules are self-explanatory. Rules R_1 to R_4 are basic operations to create, delete, set, and unset a tag. Rule R_5 specifies how the lifelines work. Note that append actions are only created as part of a message pass (c.f., Rule R_6), however, if necessary, a user-level utility could also create such actions. Rule R_6 specifies what happens during a message pass from process p_x to process p_y. A message pass triggers several basic actions, and Rule R_6 lists them as part of ϵ'; essentially it adds the tag (baton) to the receiver, removes it from the sender, and then appends the transaction information in the lifeline log. Finally, Rules R_7 and R_8 detail how intersert() statements work. The processing is straightforward in that, if the assertion a listed in the statement evaluates to true ($\top$), the system will continue (R_7). Otherwise, the state is set to failed ($s'.e = \bot$), the action list cleared, and the execution stopped (R_8).

VI. Case Study

The case study is an abstracted but complete version of a deployed system of European Train Control System (ETCS) Level 2 [7] for a customer of QNX Software Systems. ETCS is a signaling and control system used by European Railways for compatibility between several high speed rail lines. The study demonstrates the applicability of Intersert to real-world data acquisition and control applications.

A common need in data acquisition for safety-critical systems is to eliminate potentially spurious data from sensors to prevent incorrect decisions. A typical solution is to cross-validate and filter measurements before any decisions. The application obtains periodic inputs from sensors installed on two separate nodes. The system triggers specific responses based on the input data. Upon reception, the system temporally orders the input events, filters and cross-validates them to finally make its decision. The system uses redundancy and concurrency to process the data in two parallel streams.

Fig. 2 shows an abstract model of the application. Each node represents a process, and arrows between them indicate message flow. Processes A and B produce new measurements at arbitrary times, and communicate them to C and D. A and B reside on different nodes within the host system, both of which are different from the node containing the remaining threads. C and D agree cooperatively upon the order in which the events occur and identify correlated events.

TABLE I
SEMANTICS FOR INTERSERT ACTIONS.

Rule	**Semantics**	
R_1: Creating a tag	$\langle (createtag, t_x)\epsilon^*, s\rangle \rightarrow \langle \epsilon^*, s'\rangle$	with $s'.T = s.T \cup \{t_x\}$
R_2: Deleting a tag	$\langle (deletetag, t_x)\epsilon^*, s\rangle \rightarrow \langle \epsilon^*, s'\rangle$	with $s'.A(t_x) = \emptyset, s'.T = s.T \setminus \{t_x\}$
R_3: Setting a tag	$\langle (tag, p_x, t_x)\epsilon^*, s\rangle \rightarrow \langle \epsilon^*, s'\rangle$	with $s'.A(t_x) = s.A(t_x) \cup \{p_x\}$
R_4: Unsetting a tag	$\langle (untag, p_x, t_x)\epsilon^*, s\rangle \rightarrow \langle \epsilon^*, s'\rangle$	with $s'.A(t_x) = s.A(t_x) \setminus \{p_x\}$
R_5: Log entry	$\langle (append, p_x, p_y, t_x)\epsilon^*, s\rangle \rightarrow \langle \epsilon^*, s'\rangle$	with $s'.L = s.L \cup \{\langle c, t_x, p_x, p_y\rangle\}$ and c as the current time
R_6: Message pass	$\langle (msg, p_x, p_y))\epsilon^*, s\rangle \rightarrow \langle \epsilon'\epsilon^*, s\rangle$	with $\epsilon' = \{(tag, p_y, A(t_x))(untag, p_x, A(t_x))(append, p_x, p_y, A(t_x))\}$
R_7: Intersert passes	$\dfrac{eval(a) = \top \quad \langle (intersert, a)\epsilon^*, s\rangle}{\langle \epsilon^*, s\rangle}$	with property a and its valuation $eval(a)$
R_8: Intersert fails	$\dfrac{eval(a) = \bot \quad \langle (intersert, a)\epsilon^*, s\rangle}{\langle \emptyset, s'\rangle}$	with $s'.e = \bot$, property a, and its valuation $eval(a)$

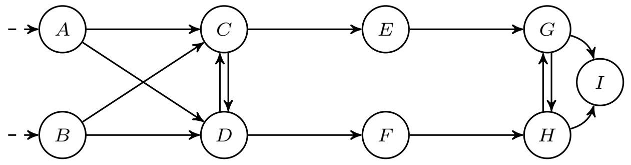


Fig. 2. Process interaction in the case study.

Once in agreement, C and D separately pass on their data to filter processes E and F, respectively. E and F filter the data to remove spurious events and performs other data transformations. Processes G and H cross validate the data and confirm with each other that the results were calculated across the same set of input events. Afterward, G and H pass on their results to I which decides on the course of action.

The following regex properties could be checked with Intersert at the designated processes for this application:

id	**Regex**	**Process**	**id**	**Regex**	**Process**
$P1$	A C D.*E	E, F	$P2$	B D C.*F	E, F
$P3$	.*(G H —H G) I	I	$P4$	.*E G	G
$P5$	[ˆD]*C D.*E	E	$P6$	[ˆC]*D C.*F	F

Property $P1$, checked in both processes E and F, verifies that if A produces a data which is later accessed by C, then C also transmits this data to D. $P2$ checks the same property with B as data source. $P3$ verifies that when G and H agree, I is the next process to receive the data; '—' is the logical OR. $P4$ checks the integrity of the interaction between E and G. With $P5$, if C first receives the data and D validates the data, then E will filter the data. $P6$ is similar to $P5$ with D as receiver.

Before Intersert, the application had no clear mechanism in place to enforce these properties and only assumed them to be correct. This assumption may not always hold, and represents a security glitch for such an application. Checking these properties at run time with Intersert ensures that any deviation from the expected ordering of interactions is promptly detected, which is particularly important in such a safety-critical system. More, it can also be performed with minimal modifications to the source code and low run time overhead.

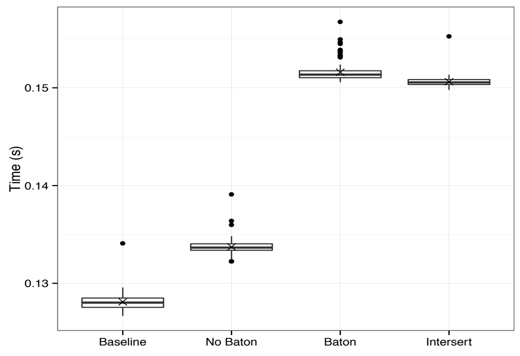
VII. Performance Evaluation

We first evaluated the overhead of the runtime system that tracks interaction sessions. To isolate the overhead, we executed the benchmarks with and without interaction history recording enabled, both in a local and a distributed setting. We also demonstrate the relation between verification times and interaction history lengths, to show that property checking incurs low overhead even in long sessions. In this section, we present the results for two sets of benchmarks: a message-passing benchmark, designed to isolate and measure the overhead of Intersert, and a verification benchmark, designed to demonstrate how the verification time scales with history size.

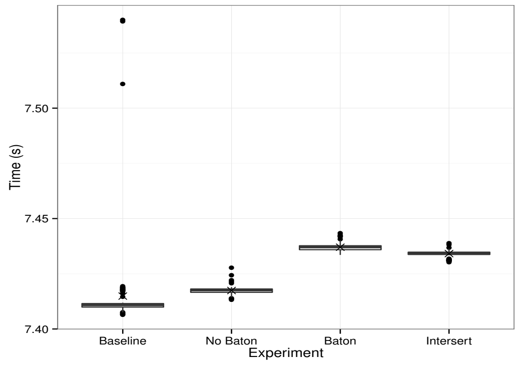
A. Local Inter-Process Messaging Benchmark

This benchmark creates seven processes, named A through G. Each process, except A, creates a messaging channel to which the alphabetically preceding process connects. Then, process A sends a 10 byte message to B, which, upon reception, forwards it to C which also forwards it until process G receives its copy of the message from F. At this point, all processes but G are blocked waiting for replies to their respective messages. Then, G replies to F, which replies to E and so on, until the reply reaches back to A. This pattern is repeated 10 000 times, all of which is timed using the processors' time stamp counter. This benchmark is run 100 times for each variant as described below for a fair sample.

We investigate four alternative setups: (1) baseline, (2) no baton passing, (3) baton passing, and (4) full Intersert. Scenario (1) consists of an unmodified QNX operating system, which serves as a baseline for further measurements. Scenario (2) measures the overhead of mTags on a kernel with mTags support. In scenario (3), we pass batons with each message, to measure the overhead of registering interaction history for Intersert, however no assertions are made. In the final scenario, we add the following call to process G: intersert("AB.*G", my_session, true), which checks that the interaction history consists of A, then B, and finally G. This scenario measures the overhead of a periodic system that makes full use of Intersert,



(a) Pure Messaging



(b) With Workload

Fig. 3. Execution time for 10,000 message chains.

since process A creates a new baton with each message it sends, and an intersert() made each time G receives a message.

Fig. 3 shows box-plots with an added $\times$ to indicate the mean for the four scenarios, under two load conditions. All data shown are collected on a Pentium 4, 3.2GHz with 2GB of RAM, running QNX Neutrino 6.5.0. The scenarios appears on the x-axis, while the y-axis shows the execution time in seconds. In Fig. 3(a), processes did nothing but sending messages, and therefore the entirety of the measured time relates to message passing and the overheads incurred by Intersert. In Fig. 3(b), a busy loop of approximately $100\mu s$ is added to each process to simulate a small amount of work. This loop executes between receiving a message and forwarding it or replying in the case of process G.

Fig. 3(a) shows that the mean execution time of the baseline scenario is $0.1281s$ or, approximately $2.135\mu s$ per message pass, while the mean execution time with no baton passing is $0.1337s$, or $2.228\mu s$ per message pass. From this, we can extrapolate that every message pass will have an *absolute* overhead of approximately $0.093\mu s$, which should be negligible in most applications. We believe that the implementation of similar functionality in other systems (e.g., other microkernels or MPI systems) could be achieved with similar overhead.

The mean execution time of the baton passing and the Intersert scenarios are $0.1515s$ and $0.1506s$, respectively. This *speedup* in the Intersert scenario over the baton passing case is due to lower baton creation overhead: in both cases process A must (1) create a baton, (2) attribute the baton to the caller thread, and (3) set that baton as passable with no restrictions. This takes three system calls in the baton passing scenario, but is abstracted by a single call to start_session() in Intersert, leading to this small speedup. A similar pattern is observed in Fig. 3(b), where a small workload is added. If the threads do some work other than messaging, the relative overhead of the Intersert-capable versions is diminished; in this case, the full Intersert version has an overhead of 0.259% over the baseline.

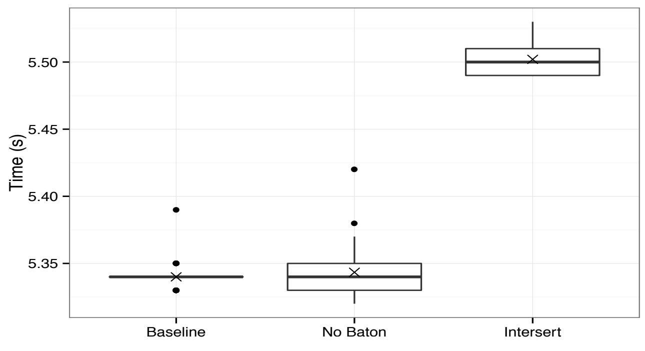


Fig. 4. Execution time of 10,000 distributed message chains.

Similarly to the local benchmarks, we measured the overhead of the distributed Intersert in three scenarios: (1) baseline, (2) no baton passing, and (3) full Intersert. In this case, we created three processes, A, B, and C, with A and C residing on host 1 and B on host 2. Messages are passed in sequence from A to C at which point C performs an Intersert call that verifies the property "ABC". As such, each trial results in 2 QNET message passes and responses. The y-axis in Fig. 4 shows the execution time for 10 000 repetitions with 50 trials per scenario. The Intersert version has a mean execution time of $5.502s$, which, despite being a statistically significant increase from the baseline of $5.34s$, is negligible in practice.

B. Verification Benchmark

To further demonstrate the feasibility of checking Intersert properties with low overhead, we check the property ".*AB" on a series of synthetic histories generated following the pattern [C, C, C, ..., C, A, B]. Histories ranging in size from 10,000 to 110,000 are obtained by increasing the number of C entries at the start of the pattern. Since all histories end with [A,B], the intersert() will always pass. This also means that the full interaction history will have to be iterated through, which is the worst case for any given assertion.

Fig. 5 shows the verification times for different history lengths. The y-axis shows the median of 15 executions for each history. We plot the median of 15 execution time measurements for each history length. As expected, the execution time grows linearly with the size of the histories. Even with the longest history, the worst execution time observed was $17.21ms$. This demonstrates that our system could be used at run time with little overhead. We note that the complexity of

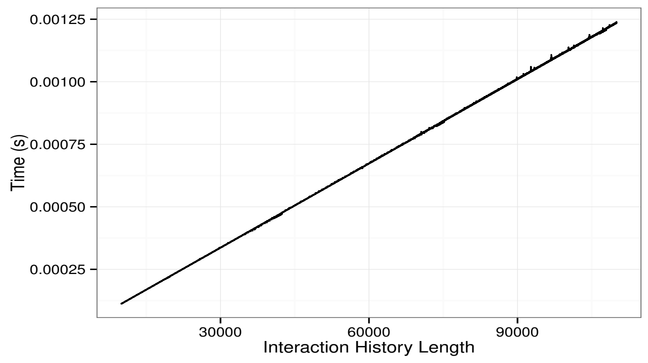


Fig. 5. Execution times for checking ".*AB" with different history lengths.

TABLE II
INCORPORATING LTL INTO INTERSERT().

Description	LTL	Intersert with LTL
Both A and B exist	$A \wedge B$	intersert_ltl("A & B")
Either A or B exists	$A \vee B$	intersert_ltl("A — B")
B must follow A	$A \rightarrow XB$	intersert_ltl("A –> XB")
A does not exist	$\sim A$	intersert_ltl("∼A")
A exists before B	$A\ R \sim B$	intersert_ltl("A R ∼B")

the regex property also affects the verification time of properties. As arbitrarily complex properties can be created and checked, execution times are bound to vary widely. Therefore, we suggest developers investigate that the execution times for verifying their properties fit within their timing constraints.

VIII. DISCUSSION

This section discusses interesting observations about checking interaction history and potential limitations of intersert().

LTL as an Alternative to Regex. In addition to regex, we also explored the use of Linear Temporal Logic (LTL) [8] as a means to express properties on interaction history. Table II shows how some of the properties we support with regex could be expressed in LTL. The first column describes the properties, the middle column shows equivalent LTL properties, and the last column an intersert() variant of the property. The syntax is that of Spot [9], the LTL checking library we use.

While LTL is more expressive than regex, its use in intersert() has a few problems. For instance, it may be unknown whether a property will yield a verdict. Liveness properties, as discussed in Section III, usually fall in this category. Thus, the property "GF A" (always eventually A) is unverifiable at run time. Therefore, much of the added expressiveness of LTL is not useful for our purposes. Alternative logics such as past time LTL (ptLTL) [10] could be used to avoid such ambiguities, and is part of our future work.

Intersert Limitations. Currently, Intersert only supports a limited number of concurrent sessions and a finite number of entries in the interaction history. Intersert uses *tags*, encoded as bit field metadata in the thread control block, to represent its interaction sessions. The width of the bit field bounds the number of concurrently active tags and thus, that of concurrent sessions. This limitation is not a hard one, however, as the user can trivially widen the field to increase the number of sessions. Similarly, the user can increase the size of interaction histories per session by configuring the history buffer.

Now, Intersert relies on the user to bound the number of sessions and history length. Automatic configuration of these parameters based on a high-level specification might be an interesting future work. Yet, until now, we have not seen this specific need in our case studies and examples as safety-critical systems are usually well understood prior to implementation and only a couple of kilobytes of memory already drastically increase the number of sessions and length of the history. In our implementation, an additional tag incurs the overhead of 64x bytes, where x is the length of the history session. An additional entry in the session consumes 64 bytes of memory.

Intersert and Single-Threaded Applications. The general concepts of the Intersert framework and intersert() statements are also applicable to single-threaded programs. Our implementation of intersert() statements only uses interaction between processes and threads, however this can easily be extended. For instance, intersert() statements may be used to check interaction history of messages passed in an object system like Smalltalk [11]. Furthermore, using aspect-oriented programming, a user could weave a runtime support system similar to that of Intersert into regular applications.

Intersert on Other Interactions. Presently, Intersert does not permit assertions on resources. Adding resource interactions, such as file access, to interaction history would allow for a finer grained control over resource access. When a thread holding a session accesses a resource it would add that resource to the history. Assertions on file access could be used to shut down threads that attempt to access certain files.

IX. RELATED WORK

The Intersert framework relates to past works in the areas of runtime verification (RV) and information flow monitoring.

Runtime verification and extended assertions. Program verification is achieved either online [8], [12] or statically [5]. However, while some propose to use sophisticated assertions at programming level, our work relates specifically to those providing software assert-based verification.

Some works use aspect orientation for RV. RMOR [13] monitors C programs using state machines, as opposed to LTL or regex. Volker and Bodden [14] verify LTL properties on events they generate at arbitrary pointcuts, e.g., method calls or data access, of Java programs using AspectJ. Allan et al. [15] also use AspectJ, and, similarly to Intersert, check regex on execution traces to decide whether to execute a function.

Partial translation verification [16] uses LTL to capture the requirements of models, then translates the LTL into C assertions to verify the correctness of generated model-based codes. Trace Analyzer (TaZ) [12] translates LTL properties into finite-state automata, called observers, used to check whether a Java process conforms to the LTL formula. Java-Mac [17] defines events and relations in the Primitive Event Definition Language, and uses runtime monitors to check Java program executions against a defined formal specification.

Other approaches [18], [19] use annotation-based techniques. Necula and Lee [20] propose a compiler that check memory-safety properties of assemblers using annotations. sPSL [21], a subset of the Property Specification Language (PSL), proposes an assert-based verification of C programs using LTL properties written in PSL.

Some tools such as DBRover [22] translate LTL into executable code for run time monitoring. ASAP [23] creates assertions from first-order logic and partial functions to detect faults at run time. Reinbacher [10] uses a dedicated hardware to collect execution traces and check ptLTL [10] assertions at run time. Intersert uses regex, but ptLTL would be an appropriate alternative logic. Also, Intersert requires no extra hardware support, but an additional kernel module.

The Intersert framework differs from prior work as it concentrates on the interaction behavior of multi-threaded applications, and *transparently* supports closed-source software. Additionally, Intersert is shown to have very low overhead, being, therefore, usable in an embedded application.

Information propagation. Information propagation relates to tag propagation in Intersert, although it is not the main contribution of this paper. Thus, Asbestos [24] implements information flow control by propagating labels among processes. seL4 [25] uses *badges* to define the capability of a process. TaintDroid [26] uses message-based taint tracking to detect information leak in mobile devices. Techniques like Datatomography [27] and Histar [28], use memory-location tagging to track information flow. Some of these systems could be used in Intersert, but suffer from prohibitive overhead.

Other existing approaches such as Jif [29], JFlow [30], and Flume [31] propose information flow control to enforce security policies and detect faults. The goal of Intersert is to support assertions based on process and thread, not to enforce security policies or uncover faults.

X. Conclusion

Program assertions are a common means for adding runtime checks to applications. Assertions that verify properties on interaction history would be useful in a wide variety of systems. To enable such assertions, we introduce Intersert, an infrastructure for checking interaction behavior between processes and threads with regex at run time.

The work resulted in a number of surprising insights and results: (1) placing assertions and interaction history is useful, (2) exposing interaction history can be achieved with negligible overhead, and (3) it is possible to push the complexity of this work into a toolchain that eases the use of such assertions.

This work's results open up several possible and interesting avenues for further work with some already highlighted in Section VIII. Others include extending the amount of history information used beyond interaction on local hosts as well as synthesizing the assertions from high-level specifications.

References

[1] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation," Microsoft Research, Tech. Rep. MSR-TR-2006-54, May 2006.

[2] Ontario Power Generation Inc., "SDS1 Software Design Description, NK38-MAN-68258-001, Rev06," 2002.

[3] J. E. F. Friedl, *Mastering Regular Expressions*, 2nd ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[4] OMG UML, "OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2," 2007.

[5] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 21:1–21:54, Oct. 2009.

[6] A. B. de Oliveira, A. Saif Ur Rehman, and S. Fischmeister, "mTags: Augmenting Microkernel Messages with Lightweight Metadata," *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 2, pp. 67–79, Jul. 2012.

[7] European Railway Agency, *ERTMS/ETCS Functional Requirements Specification*, 5th ed. ERA, 2007.

[8] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, 2011.

[9] Spot Library, "Spot library," Apr. 2012, http://spot.lip6.fr/wiki/.

[10] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski, "Past time LTL runtime verification for microcontroller binary code," in *FMICS*. Springer-Verlag, 2011, pp. 37–51.

[11] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[12] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *ASE*. IEEE, 2001.

[13] K. Havelund, "Runtime verification of c programs," in *TestCom/FATES*, 2008, pp. 7–22.

[14] V. Stolz and E. Bodden, "Temporal assertions using AspectJ," *Electron. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 109–124, May 2006.

[15] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA*. ACM, 2005, pp. 345–364.

[16] M. Staats and M. P. Heimdahl, "Partial Translation Verification for Untrusted Code-Generators," in *ICFEM*. Springer, 2008, pp. 226–237.

[17] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-MaC: A run-time assurance approach for Java programs," *Form. Methods Syst. Des.*, vol. 24, no. 2, pp. 129–155, Mar. 2004.

[18] G. Canet, P. Cuoq, and B. Monate, "A value analysis for C programs," in *SCAM*. IEEE, 2009, pp. 123–124.

[19] E. Denney and B. Fischer, "Annotation Inference for Safety Certification of Automatically Generated Code," in *ASE*. IEEE, 2006, pp. 265–268.

[20] G. C. Necula and P. Lee, "The design and implementation of a certifying compiler," *SIGPLAN Not.*, vol. 33, no. 5, pp. 333–344, May 1998.

[21] P. H. Cheung and A. Forin, "A C-Language Binding for PSL," in *ICESS*. Springer-Verlag, 2007, pp. 584–591.

[22] D. Drusinsky, "Monitoring temporal rules combined with time series," in *CAV, vol. 2725 of LNCS*. Springer, 2003, pp. 114–118.

[23] I. D. Curcio, "ASAP - A Simple Assertion Pre-Processor," *SIGPLAN Not.*, vol. 33, no. 12, pp. 44–51, Dec. 1998.

[24] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and Event Processes in the Asbestos Operating System," in *SOSP*. ACM, 2005.

[25] NICTA, "seL4 Reference Manual API version 1.2," 2013.

[26] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *OSDI'10*. USENIX.

[27] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and Visualizing Full Systems with Data Flow Tomography," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 211–221, 2008.

[28] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," in *OSDI*. USENIX, 2006.

[29] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, 2000.

[30] A. C. Myers, "JFlow: practical mostly-static information flow control," in *POPL*. ACM, 1999, pp. 228–241.

[31] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in *SOSP*. ACM, 2007, pp. 321–334.