A Reduced Complexity Design Pattern For Distributed Hierarchical Command and Control System

Heechul Yun
University of Illinois at Urbana-Champaign
201 N. Goodwin
Champaign, IL 60801
heechul@illinois.edu

Po-Liang Wu
University of Illinois at Urbana-Champaign
201 N. Goodwin
Champaign, IL 60801
wu87@illinois.edu

Maryam Rahmaniheris
University of Illinois at Urbana-Champaign
201 N. Goodwin
Champaign, IL 60801
rahmani1@illinois.edu

Cheolgi Kim
University of Illinois at Urbana-Champaign
201 N. Goodwin
Champaign, IL 60801
cheolgi@illinois.edu

Lui Sha
University of Illinois at Urbana-Champaign
201 N. Goodwin
Champaign, IL 60801
lrs@illinois.edu

ABSTRACT

Cyber Physical Systems (CPS) get a lot of attention due to the strong demand for the integration of physical devices and computing systems. There are many design aspects involved in CPS, such as efficiency, real-time, reliability and security. One of the major issues is system integration and verification. In many safety critical systems verification plays an essential role in system design. However, the high complexity for the composition of diverse systems is a major challenge for system verification. In this paper, we focus on command and control systems for search and rescue missions and propose a systematic design pattern called *Interruptible RPC* to compose complex systems while keeping the verification costs low. This has been made possible due to the reduced state space of the systems designed using our pattern. Therefore, the system models can be efficiently verified using available verification tools. In our experiments, the search and rescue system based on *Interruptible RPC* pattern had fewer states than the asynchronous one by several orders of magnitude.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*

General Terms

Verification

Keywords

Complexity, Command and Control System, Remote Procedure Call

1. INTRODUCTION

As computing devices deeply and pervasively affect our daily lives, new requirements arise for the seamless integration of physical objects and computing systems. *Cyber-physical systems* (CPSs) are systems that tightly combine physical elements and computation resources through communication technologies. Some applications for cyber-physical systems include health care, advanced automotive systems, and avionics [1]. Since CPSs interact with various physical components and systems overall system design becomes a critical issue with respect to robustness, reliability, efficiency, and real-time properties. Moreover, integrating these diverse systems will inevitably add significantly to the verification cost. Therefore, new system-level composition techniques are required to provide systematic design patterns with low complexity [2].

In many command and control systems such as the Search and Rescue system [3][1], robustness and reliability are the major concerns. Therefore, there is a strong demand to formally verify the correctness of systems with some program verification tools such as UPPAAL and Maude [4, 5]. However, in many systems, the components are physically distributed and interact asynchronously, due to the lack of a common clock. Verifying such asynchronous systems is very difficult, or even impossible, because the state space grows exponentially due to message interleaving resulting from asynchronous communication. This kind of problem is known as state explosion [6, 7].

The problem of state space explosion due to distributed system designs can be seen in different systems. In database systems, one of the most successful approaches to address this problem is the use of atomic transactions [8]. Atomicity translates all the possible results generated by the distributed database system into an equivalent serialized one. Since atomicity solves the operation interleaving problem, the complexity of the distributed database systems is significantly reduced. However, the *all or nothing* atomicity property decreases flexibility. Moreover, the roll-back mechanism is sometimes extremely difficult or even impossible in systems with physical constraints, such as command and control systems.

[1]The detail discussion is provided in the next section.

On the other hand, imprecise computing [9] (or anytime computing) discusses the tradeoff between the responsiveness and robustness of the systems. Imprecise computing provides the flexibility of receiving a usable, approximate result. However, there is no formal and verified model that exploits the potential usage of imprecise computing in other system designs. The technical problem is to provide sufficient flexibility/efficiency while maintaining the simplicity [10].

In this paper, we focus on the design methodology of providing formal description of the basic structure for composing complex systems, called building block, which is flexible, easy to extend, and only introduces low complexity and verification cost. In order to achieve the design goal, we present an architectural design pattern – *interruptible RPC* – for verifiable command and control systems. The interruptible RPC provides bounded asynchrony, which limits the possible message interleaving due to asynchronous communication, and prevents state explosion. Based on the proposed building blocks, we propose a composition method for designing complex hierarchical command and control systems while maintaining low complexity. The experiment results show that the proposed design pattern achieves significant state space reduction. In the system configuration with two sensors, the number of states of the system based on the proposed design pattern is at least ten times smaller than that of the asynchronous one. Moreover, the reduction efficiency increases as the number of nodes and the size of queue increase.

The contributions of this paper can be summarized as follows:

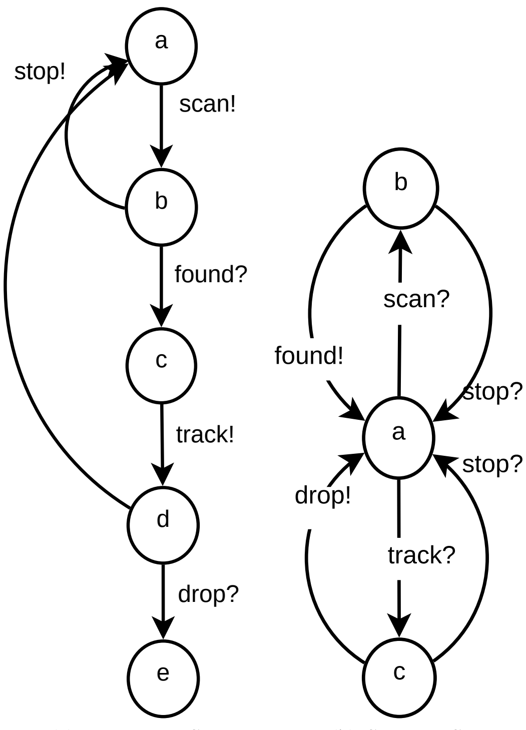
- An architectural design pattern called *interruptible RPC*, which is flexible and only introduces bounded asynchrony.
- A composition methodology to compose complex hierarchical command and control systems while maintaining low complexity.
- The quantative evaluation of the system complexity.

The rest of the paper is organized as follows. In Section 2 we present our motivating example. Section 4 presents the *interruptible RPC* pattern. Section 5 describes the composition method. We present evaluation results in Section 6. We review related work in Section 7 and we conclude in Section 8.

2. COMMAND AND CONTROL SYSTEM EXAMPLE: SEARCH AND RESCUE SYSTEM

In this section, we present a simplified search and rescue system as a motivating example. The example system is a computerized helicopter that searches for, and rescues people lost in ocean. The system is a distributed system consisting of a *manager node* and multiple *sensor nodes*. A sensor node is a physical system that includes (a) an infrared sensor, (b) a microprocessor to analyze sensor data, and (c) a communication device to send and receive data to/from the manager. The manager is a cyber system which is a part of the main flight control system (FCS). The manager chooses the search area and sends corresponding commands to sensors. The sensors notify the manager upon detecting human targets in the designated area.

Fig.1 shows the synchronous version of the rescue system design with a manager and a sensor. This system is synchronous since message send (denoted as $msg!$) and receive (denoted as $msg?$) happens in a lockstep fashion – the formal definition of synchronous system is described in Section 3.1.



(a) Manager FSM (b) Sensor FSM

Figure 1: Fully synchronous network of automata modeling search and rescue system

The mission is divided into a *scanning phase* and a *tracking phase*. In the first phase, the manager sends the *scan* command to the sensor. If the sensor finds a target, it reports back to the manager by sending a *found* message and the manager moves to the tracking phase by sending the command *track*. On receiving the *track* command, the sensor follows the identified target until the rescue mission completes. The important factor in this model is that the manager can stop the mission during scanning or tracking phase (b or d state in Fig. 1(a)).

Verification of synchronous systems is relatively easy. In this example, the total number of states of the system is only five since all transitions are synchronized between the manager and the sensor. However, in real distributed systems it is difficult to use synchronous design because of physical limitation such as communication delay or performance issues. On the other hand, asynchronous systems are notoriously complex since they have to deal with potential message interleaving which does not exist in synchronous systems. Fig.2 shows message interleaving when all commands and responses are asynchronously sent. We use A and B to denote different search areas. In this figure, the manager first sends *scan A* command. If nothing is found within a certain timeout interval, it sends a *stop A* command to cancel the current execution and immediately sends *scan B* command to start a new search. However, there is a possible scenario in which the sensor finds an object in area *A* before receiving the *scan B* command and sends back the

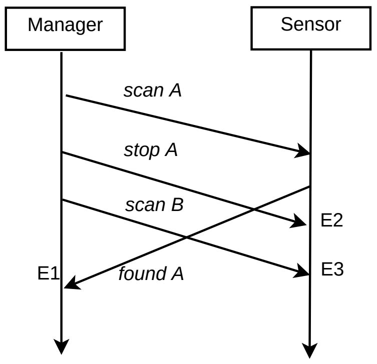


Figure 2: Message interleaving. $stopA, scanB$ **and** $foundA$ **are interleaved and must be considered in both the manager and the sensor (*E1*, *E2* and *E3*).**

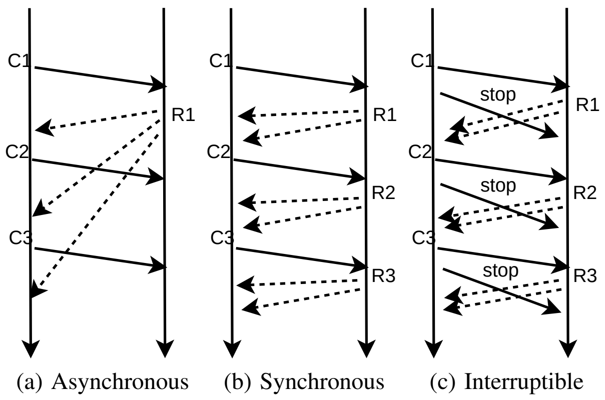


Figure 3: Comparison of RPC patterns.

report to the manager. In this case, both the manager and the sensor must handle the interleaving of messages: the manager should make a decision – i.e. whether not to process *found A* response at location *E1*, and the sensor should make a decision – whether to handle *stop A* and *scan B* commands at location *E2* and *E3*. Therefore, message interleaving adds complexity to system design.

The complexity of message interleaving can be shown as the number of states explored when we perform model checking of the system. We can model asynchronous communication between any two finite state machines (FSMs) using two directional queues – the formal definition of asynchronous system is described in Section 3.2. Model checking of such systems can easily suffer from a state space explosion problem because the number of states grows exponentially as the queue size grows. Let $S(Q_{k,x})$ be the total number of states of a queue where k is the size of the queue and x is the number of distinct messages in the queue ($k \geq 1, x \geq 1$). Then, $S(Q_{k,x}) = 1 + x + x^2 + ... + x^k = \frac{x^{k+1}-1}{x-1}$ in worst case. If there are c distinct commands and r distinct responses, then the maximum number of states of the two queues can be $(\frac{c^{k+1}-1}{c-1}) \cdot (\frac{r^{k+1}-1}{r-1})$ assuming there is no order enforcement between the commands and responses. Exponential number of states of the queues represent the maximum possible number of message interleaving which is a major source of state explosion in distributed systems. Our evaluation shows that the asynchronous system with a queue size of four has more than four orders of magnitude of states compared to a fully synchronous system.

Moreover, the asynchronous model easily results in deadlocks, that is un-acceptable in most system designs. For example, if the manager in Fig.2 expects *found B* message and does not have a transition function of *found A* message then the manager FSM can't make any further progress. However, due to the state explosion problem, it is difficult to check if there can be a deadlock by most program verification tools. Even if we find a deadlock, fixing it correctly is still a big challenge for system designers. Moreover, after fixing the deadlock, the designers have to re-verify the correctness of the system. Therefore, we need a systematic way to reduce the number of states in the system to make model checking feasible and to minimize the risk of undesirable situations such as deadlock.

We realized that a command and control system resembles the remote procedure call(RPC) pattern. In Fig.1, *scan-found* and *track-drop* pair can be regarded as two separate RPCs. The synchronous RPC pattern reduces complexity and verification costs significantly since it does not allow interleaving of commands and responses. The only exception is the *stop* command. The *stop* command allows the system to cancel the current command and issue a new command that is a typical requirement in many command and control systems for adapting to environment changes or emergency alerts.

This motivates us to introduce interruptible RPC pattern as a building block for command and control systems. The interruptible RPC pattern is synchronous RPC allowing bounded asynchrony – the ability to stop the current executing command. It can be thought of as a tradeoff between strict synchronous RPC and fully asynchronous RPC [11, 12, 13].

The Fig.3 shows the conceptual difference between asynchronous RPC, synchronous RPC, and interruptible RPC. In Fig.3(a), response $R1$ can be interleaved with multiple commands: $C2$ and $C3$. We call this *inter-command interleaving* and it causes exponential state space growth since previous command executions affect the current command. In Fig.3(b), responses never interleave with any commands. Therefore, there's no additional state space growth caused by unexpected interleaving. In Fig.3(c), only the *stop* command can be interleaved with the responses. Therefore, while there is interleaving, which must be verified for correct operation of the system, the added complexity is bounded. We call this *intra-command interleaving*.

The goal of the proposed interruptible RPC model is to provide a low complexity design pattern for command and control systems with the consideration of the physical nature and the tradeoff between flexibility and simplicity. We describe the details of interruptible RPC in the next section.

3. SYSTEM MODEL

To quantify the system complexity in terms of the number of states, we first define a formal model of computation and communication for distributed command and control systems.

3.1 Synchronous system

The computation and communication model of the synchronous system is similar to the hand-shake synchronization model of UPPAAL [4].

A single FSM M_i is a tuple $\langle S_i, l_i^0, \Sigma, E_i \rangle$ where

- S_i is a set of locations of M_i,
- $l_i^0 \in S_i$ is the initial locations,
- Σ is the alphabet, whose elements are input ($a?$), output ($a!$), and/or local (a) actions, and
- $E_i \subseteq S_i \times \Sigma \times S_i$ is the set of edges.

In the above definition and the later description, a stands for an event passing between machines encapsulating the source and the destination. We write $l_i \xrightarrow{\sigma} l_i'$ for a state transition such that $\langle l_i, \sigma, l_i' \rangle \in E$, $l_i \in S_i$ and $l_i' \in S_i$ where $\sigma \in \Sigma$.

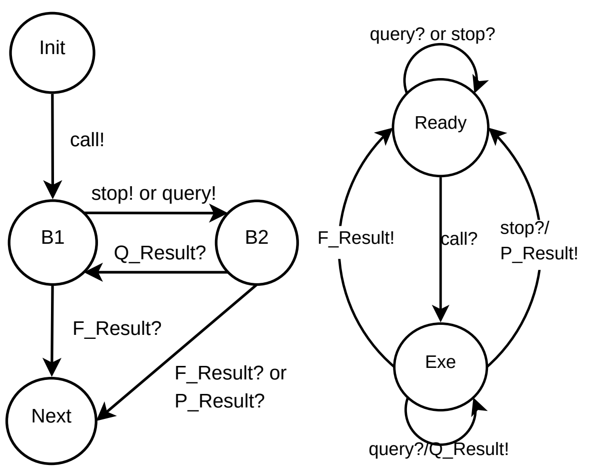
A synchronous system is a set of finite state machines $M_1, ..., M_n$. A state in the system is defined as a vector of current locations of the machines and denoted $\vec{l}$. There are two transition rules in the system: local transition rule where one FSM make its own move, and synchronized transition rule where two FSMs make simultaneous move. In the latter case, the two FSMs are synchronized using input actions, and output actions. Let l_i is ith element of vector $\vec{l}$ and $\vec{l}[l_i'/l_i]$ represents l_i is replaced by l_i'. The transition rules are as follows:

- *Local*: $\vec{l} \xrightarrow{a} \vec{l}[l_i'/l_i]$ if $l_i \xrightarrow{a} l_i'$
- *Sync*: $\vec{l} \xrightarrow{a} \vec{l}[l_i'/l_i][l_j'/l_j]$ if $l_i \xrightarrow{a!} l_i'$ and $l_j \xrightarrow{a?} l_j'$

3.2 Asynchronous system

In asynchronous model, a system is composed of a set of FSMs communicating through queues. we introduce queues to model the communication delay in the distributed system. Let $Q_{i,j}$ denotes a queue connecting from M_i to M_j. The queue is served as an unidirectional communication channel. $q_{i,j}$ represents an instance of $Q_{i,j}$, having a sequence of events. If a queue is a concatenation of two event sequences q, q', it is represented by $q \cdot q'$. Then, the distributed system can be defined as a parallel composition $M_1|...|M_n|Q_{i_1,j_1}|...|Q_{i_m,j_m}$ of a set of finite state machines $M_1, ..., M_n$ and queues $Q_{i_1,j_1}, ..., Q_{i_m,j_m}$. A queue represents the sequence of messages sent by the originator, M_{ik}, but has not been processed yet by the destination, M_{jk} where $1 \leq k \leq m$. The state of a distributed system is a pair $\langle \vec{l}, \vec{q} \rangle$ where $\vec{l}$ denotes a vector of current locations of machines, and $\vec{q}$ is a vector of assignments of the queues in the network. q_{ij} stands for an element of $\vec{q}$ connecting M_i and M_j. Let l_i is ith element of vector $\vec{l}$ and $\vec{l}[l_i'/l_i]$ represents l_i is replaced by l_i'. Also, let q_i and $q[q_i'/q_i]$ are defined in the same way. Then, the transition rules are *local transition*, *send transition*, and *receive transition* as defined in the following:

- *Local*: $\langle \vec{l}, \vec{q} \rangle \xrightarrow{a} \langle \vec{l}[l_i'/l_i], \vec{q} \rangle$ if $l_i \xrightarrow{a} l_i'$
- *Send*: $\langle \vec{l}, \vec{q} \rangle \xrightarrow{a!} \langle \vec{l}[l_i'/l_i], \vec{q}[q_{ij}'/q_{ij}] \rangle$ if $l_i \xrightarrow{a!} l_i'$, $q_{ij}' = q_{ij} \cdot a$
- *Receive*: $\langle \vec{l}, \vec{q} \rangle \xrightarrow{a?} \langle \vec{l}[l_i'/l_i], \vec{q}[q_{ij}'/q_{ij}] \rangle$ if $l_i \xrightarrow{a?} l_i'$, $q_{ij} = a \cdot q_{ij}'$



(a) Caller FSM (b) Callee FSM

Figure 4: FSM of interruptible RPC pattern

4. INTERRUPTIBLE RPC PATTERN

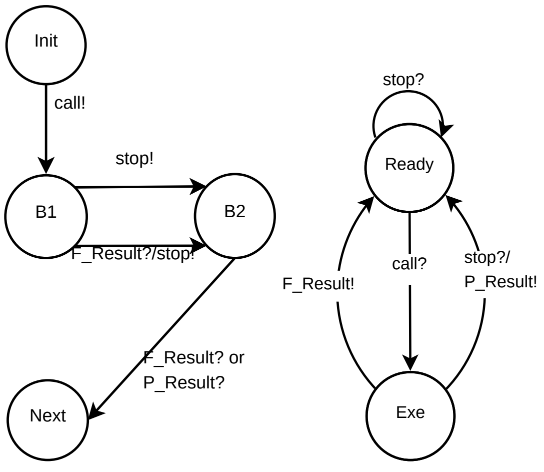
Fig. 4 shows our proposed interruptible RPC pattern for command and control systems. The syntax of the model is described in Table 1. To simplify the explanation, we show the model for a single command and present the composition procedure in the next section. The basic scenario follows the synchronous RPC pattern and provides bounded asynchrony. The caller actively sends a command to the callee and waits for the result. If the callee finishes execution without receiving the *stop* command, callee will send full result and go back to the ready state. at the caller side, on receiving the full result, the caller will go to the *next* state and invokes the next command. On the other hand, the caller may need to stop the current execution due to environment changes or emergency alerts. Therefore, we provide bounded asynchrony, which allows the caller to send the *stop* command while in blocking state *B1*. After sending the *stop* command, the caller waits for the partial result in state *B2*. When the callee receives the *stop* command, it will stop the execution and send the partial result back. Due to the transmission delay, the *stop* command may arrive at callee side after the callee has already finished the execution (the message interleaving between *stop* and full result). In that case, the caller will accept the full result after sending the *stop* command. Take the search and rescue system in Section 2 as an example, the caller plays the role of a manager and the callees play the role of sensors. However, the callee may fail and do not receive and send messages. We can introduce timeout mechanism to handle this problem. Nevertheless, completely modeling all the possible failure between caller and callee is a challenging problem. We will put fault-tolerance mechanism in the future work. Another worthwhile point to mention is that this pattern also provides a query command to let the caller actively keep track of the execution status in the callee.

We modeled the proposed interruptible RPC in Maude and verified the properties. According to Maude, the proposed model only introduces 17 states, so it can be exhaustedly verified by most model checking methods. The proposed model contains three verified properties.

- There is no *inter-command interleaving*, and the only *intra-command interleaving* is between *stop* command and full re-

$msg!$	Send msg to communication queue
$msg?$	Receive msg from communication queue
$A?/B!$	If receive A then send B
F_Result	Full result, generated after finishing the execution
P_Result	Partial result, generated after receiving $stop$
Q_Result	Query result, generated after receiving $query$

Table 1: FSM operation syntax



(a) Caller FSM (b) Callee FSM

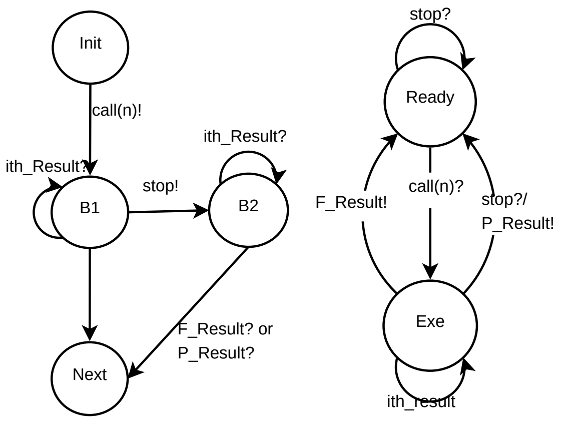
Figure 5: FSM of interruptible RPC pattern for first response

sult. The caller will receive full result instead of partial result if the callee has already finished the execution.

- If the callee finishes the execution or the caller sends *stop* commands, then the caller goes to the *next* state and the callee goes to ready state without leaving pending messages in the queue. Consequently, there is no deadlock in this building block.
- The size of the queue is bounded by two since the callee can only receive two consecutive commands: *call!-stop!* and *call!-query!*.

4.1 Distributed Interruptible RPC Pattern

To extend the current model to support distributed command and control systems, a caller serves as a manager to control multiple distributed callees. We consider the system structure in two ways. First, the callees are independent, which means that the execution of one callee does not depend on the executions of other callees. Therefore, the caller creates a thread or a process to control each independent callee. Since there is no message passing between the callees, the complexity of the caller is the sum of the complexity of the callee models. Second, the callees coordinate on the same job, and the caller controls the callees according to the responses it receives from them. In order to easily extend the proposed model to support this scenario while maintaining low complexity, all callees have to stay in the same state when there are no pending messages left in the queue. All the callees can only accept execute the same command at the same time. After all the callees finish the execution



(a) Caller FSM (b) Callee FSM

Figure 6: FSM of interruptible RPC pattern with sub-results

of the command and send the full results back, the caller can go to the *next* state and issue the next command.

In this manner, this pattern prevents extra interleaving between callees and keeps the system complexity low. By providing interruption in the form of bounded asynchrony, we meet the primary goal of concurrency and flexibility. The limitation of concurrency is a worthwhile tradeoff since otherwise we will have a unverifiable system due to state explosion. As we show in Section 6, when comparing an unbounded asynchronous system, the complexity difference can be several orders of magnitude.

4.2 Variations of Interruptible RPC Pattern

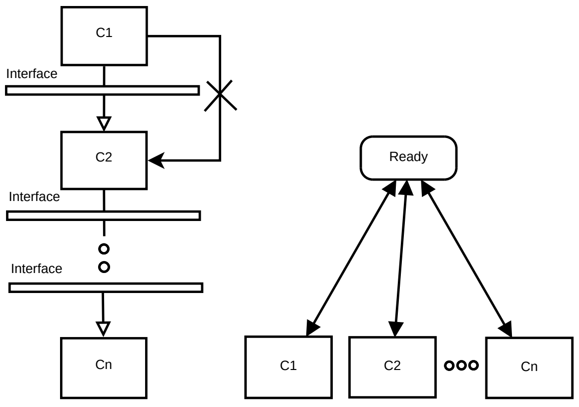
Our proposed model is flexible to support different design scenarios with slight modification. Now we present some useful variations. We will show how to use these variations to compose a search and rescue system in the next section.

Interruptible RPC for first response Consider a set of callees coordinated for the same work. After receiving the first response from one of the callees, the caller can stop current execution and issue a new command. As shown in Fig.5, the caller waits for the first full result from one of the callees and sends the *stop* command. On receiving the results from the other callees, caller goes to the *next* state and issues new commands.

Interruptible RPC with sub-results Fig.6 shows the modified interruptible RPC pattern in which a callee is asked to provide *n* sub-results specified by the *call(n)* command. Therefore, the caller can keep track of the execution status of the callee without introducing more unnecessary commands. Once the caller is satisfied with the quality of the sub-result, it can send *stop* command and invoke the next command. In this scenario, the caller has the flexibility to determine the flow of execution according to the quality of the sub-results.

5. COMPOSITION OF INTERRUPTIBLE RPC

In this section, we describe how our interruptible RPC pattern can be used in designing more complicated systems with multiple commands. Also we describe the complexity of composed system.



(a) Caller (b) Callee

Figure 7: Composition of multiple commands (C1,...,Cn).

5.1 Composition Methodology

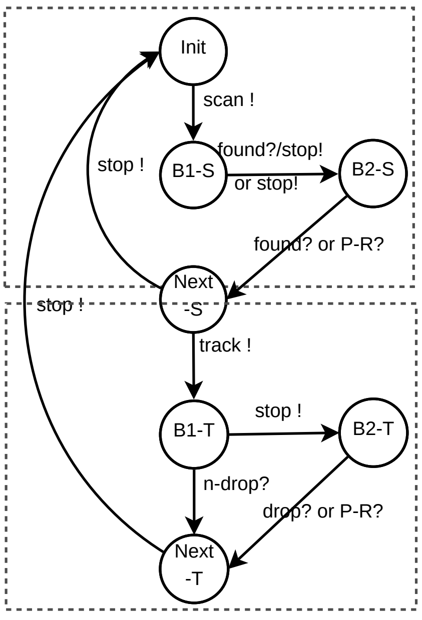
Fig. 7 shows the multi-command model on the caller side can be modeled as a sequential composition of state machines of commands that are built based on the interruptible RPC pattern. On the other hand, the composition on the callee side is in a parallel style and can also be easily composed with the interruptible RPC pattern. However, in reality the commands are executed in a sequential manner and there is no interleaving between the execution steps of different commands. The order of execution of commands on the callee side is enforced by the caller. Moreover, the only dependency between the commands of the proposed pattern is in the *interface* between the commands. This interface is the output of one command used as the input of the next command. The order of commands is the run-time order in which they are executed. Each command can start only when the previous command has completed its execution.

Fig. 8 shows a search and rescue system with two commands, *scan* and *track*. The system is composed using the interruptible RPC pattern and the composition methodology illustrated in Fig. 7. In the first phase, the mission manager sends out the *scan* command to both of the sensors. When a target is detected by any of the sensors, the mission manager moves to the tracking phase. In this phase the sensors will follow the target and depending on the application can report back the location, direction and speed of the target. In this model each sensor remains in the tracking mode until it receives a *stop* message from the mission manager or it loses the target. The sensor model is shown on the right side of Fig. 8. Similar to the original interruptible RPC model, on receiving a *stop* command, the sensors report back their partial result or full result if they have already finished the execution.

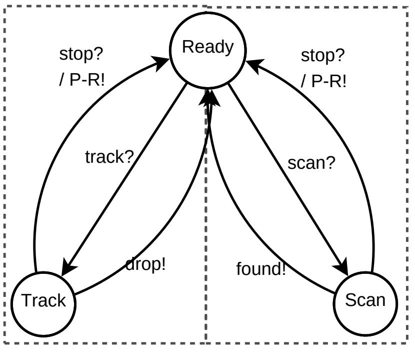
For the scan command, we use the building block that accepts the first response from the sensors, as shown in Fig. 5 and for the track command, we use the original building block. The provided building blocks are flexible and useful for composing complex systems.

5.2 System Complexity

The verification cost of the system consists of two phases: verification of the commands, $C_{command}$, and verification of the composition, $C_{composition}$. These costs are *added* together instead of being *multiplied* due to the bounded asynchrony property provided by our proposed design pattern.



(a) Manager FSM



(b) Sensor FSM

Figure 8: Interruptible RPC pattern based search and rescue system model

For the first part, $C_{command}$, when composing the system using the building blocks, the proposed composition methodology prevents *inter-command interleaving*. Therefore, there is no need to verify the different possible interleaving of execution steps of the commands in our system (Fig. 7). The verification cost is the *sum* of the verification cost of the commands.

In contrast, the asynchronous model may result in significant *inter-command interleaving*. Therefore, the complexity of verifying all possible interleaving is the *product* of verification cost of the commands that is much larger than the proposed pattern and can easily cause the state space of the system to explode.

The second part, $C_{composition}$, is the verification cost that is paid by the designer to find the correct wiring of the building blocks. This cost depends on the approach that the designer takes to obtain the correct composition for the system. It should be mentioned that the focus of our proposed methodology is $C_{command}$. $C_{composition}$ is beyond the scope of this paper, but we intend to address it in future work.

Table 2: State space and execution time comparison of search and rescue system. async(n) denote asynchronous system of queue size n. *NF* means it didn't finish within 12 hours.

(a) State space comparison (**number of states**)

	synchronous	interruptible	async(2)	async(3)	async(4)	async(5)
1 sensor	5	29	70	121	195	275
2 sensors	11	133	1410	4424	11816	23732
3 sensors	22	701	28691	184418	782417	*NF*

(b) Execution time comparison (**msec**)

	synchronous	interruptible	async(2)	async(3)	async(4)	async(5)
1 sensor	0	1	4	7	12	16
2 sensors	1	13	160	549	1571	3276
3 sensors	1	120	6586	49307	1096480	*NF*

Based on the proposed design pattern, after verifying each command separately ($C_{command}$), the designers only need to verify the correct wiring of the building blocks ($C_{composition}$), that is independent of the individual commands. In this manner, the correctness of the composed system is guaranteed and there is no need to verify the whole system. The proposed design pattern not only provides the designers with a library that consists of building blocks and multiple options for each of the blocks but also greatly reduces the design and verification costs. The next section contains more details on the system design and verification cost.

6. EVALUATION

Table 2 shows experimental results of modeling the search and rescue system described in Section 2 with different patterns: synchronous, asynchronous RPC, and the proposed interruptible RPC.

For each pattern we varied the number of sensors from 1 to 3 sensors in the model. Synchronous RPC pattern is based on synchronous model described in Section 3.1. Interruptible RPC and asynchronous RPC patterns are both based on asynchronous model described in Section 3.2. For asynchronous RPC pattern, we varied the queue size from 2 to 5. Note that interruptible RPC uses queues of size two since the pattern do not need more than two messages in the queue. We used Maude [5] to model the systems and all measured in a Core2Duo 2.7 GHz workstation with 4GB RAM. In the asynchronous RPC based system, we added additional transitions to discard non acceptable messages to prevent deadlock – for example, when the *stop* and *found* are interleaved, we discarded the *found*.

From Table 2(a), we see that the state space growth rate of the three patterns as the number of sensors increase. It is evident that asynchronous RPC patterns grow much faster than the other two. The growth rate of 'interruptible' pattern is about an order of magnitude slower, for each sensor addition, than 'async(4)' – asynchronous with queue size of four. Also, the asynchronous RPC pattern suffers severely from state space explosion as the queue size grows. Therefore, model checking of even reasonably sized asynchronous RPC systems will be infeasible. On the other hand, the interruptible RPC pattern shows reasonable state space growth rate. Moreover, it is not affected by the queue size since the maximum interleaving is bounded by the design.

Table 2(b) shows a comparison of the execution times. It also shows a trend similar to Table 2(a). However, the growth rate is even faster. For async(4) configuration, the growth rate is two orders of magnitude for each sensor addition; For async(5), it failed to finish within 12 hours for three sensors.

7. RELATED WORK

After the introduction of software patterns [14], the use of patterns is well accepted in many software areas from designing small component to designing system architecture [15, 16, 17]. While these patterns are useful for reducing design and development cost, they are informally described in general. Thus, they do not formally guarantee the correctness for wide-range of similar problems. Since CPS demand rigorous verification of designed systems, formally describable design pattern have been an active research area recently.

Physically Asynchronous Logically Synchronous (PALS) system is a formal architectural pattern which provides architectural abstraction of time triggered synchronous systems over physically asynchronous systems [18]. The pattern is formally specified and simplifies the formal verification of real-time distributed system by providing a correctness preserving transformation of the synchronous design into the equivalent asynchronous system. While PALS share similar objective with us, our approach is different in two key aspects. First, our model is event (message) triggered system while PALS is time triggered system. Second, we narrowly focus on RPC style communication of hierarchical command and control systems.

RPC is widely used as a client/server model to request service through network. The synchronous RPC has the advantage of low complexity and verification cost. However, the synchronous model lacks flexibility, because the system can be blocked during execution. In contrast, asynchronous RPC model provides high flexibility and concurrency and is widely adopted in client-sever systems designs [11]. However, asynchronous RPC model suffers from serious state explosion problem due to message interleavings. Although some programming templates support abort or cancel mechanism in asynchronous RPC, they lack formal description of the behavior for aborting RPC [12].

Interruptible RPC is a formally described pattern which allow bounded asynchrony and can be fully analyzed without causing state space explosion.

8. CONCLUSION

This research is motivated by the state space explosion problem due to message interleaving in distributed system designs. We focus on the command and control systems for search and rescue and propose a design pattern, *Interruptible RPC*, as a building block of system designs. We show the proposed pattern has low complexity and verification costs in terms of number of states explored by the program verification tool. In addition, we propose a composition methodology to compose complex command and control systems without introducing state space explosion. In our experiments, we model a search and rescue system and we show that the proposed design pattern can reduce verification space and time by several orders of magnitude. The reduction is more significant when the size of the model (number of sensors, queue size) increases.

In future work, we will exploit the fault-tolerance mechanism. Moreover, we shall further explore the hierarchical structure of command and control systems and provide the formal description and proof of the complexity.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for useful feedback, Abdullah Al-Nayeem, Mu Sun, and Sibin Mohanm for their comments and help that improved the quality of this paper. We also thank Lockheed Corporation Martin for providing valuable opinions. This effort is funded in part by Lockheed Martin 2009-00524 and by ONR N00014-08-1-0896 (ONR).

10. REFERENCES

[1] E. A. Lee, "Cyber physical systems: Design challenges," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8, Jan 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html

[2] Stankovic, J. A., Lee, Insup, Mok, Aloysius, Rajkumar, and Raj, "Opportunities and obligations for physical computing systems," *Computer*, vol. 38, no. 11, pp. 23–31, 2005.

[3] "COSPAS-SARSAT Search and Rescue System." [Online]. Available: http://www.nasa.gov/centers/goddard/pdf/105930main_cospas.pdf

[4] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," *Lecture Notes in Computer Science*, vol. 3098, pp. 87–124, 2004.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude – A High-Performance Logical Framework*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model checking*. Springer, 1999.

[7] N. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[8] J. E. Moss, *Nested transactions: an approach to reliable distributed computing*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1985.

[9] J. W.S.Liu, W.-K. Shih, K.-J. Lin, B. R., and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, pp. 83–94, 1994.

[10] S. Lui, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[11] A. L. Ananda, B. H. Tay, and E. K. Koh, "A survey of asynchronous remote procedure calls," *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 2, pp. 92–109, 1992.

[12] Microsoft, *Asynchronous RPC (Windows)*. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa373550%28VS.85%29.aspx

[13] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," in *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 188–197.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented design*. Addison-Wesley Reading, MA, 1995.

[15] F. Buschmann, *Pattern-oriented software architecture: a system of patterns*. Wiley, 2002.

[16] P. Avgeriou and U. Zdun, "Architectural patterns revisited–a pattern language," in *10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. Citeseer, 2005, pp. 1–39.

[17] B. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[18] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. Miller, and D. Cofer, "A formal architecture pattern for real-time distributed systems," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2009, pp. 161–170.