Static analysis of multithreaded dynamic programs

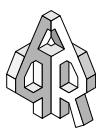
A π-calculus based approach

Aurelien Deharbe

Laboratoire d'Informatique de Paris 6 - APR team

EJCP 2013

Ph.D advisor : F. Peschanski



Context

Compositional static analysis

```
#define N_EVENT                 5

struct dev_data {
        spinlock_t                      lock;
        atomic_t                        count;
        enum ep0_state                  state;          /* P: lock */
        struct usb_gadgetfs_event       event [N_EVENT];
        unsigned                        ev_next;
        struct fasync_struct            *fasync;
        u8                              current_config;

        /* drivers reading ep0 MUST handle control requests (SETUP)
         * reported that way; else the host will time out.
         */
        unsigned                        usermode_setup : 1,
                                        setup_in : 1,
                                        setup_can_stall : 1,
                                        setup_out_ready : 1,
                                        setup_out_error : 1,
                                        setup_abort : 1;
        unsigned                        setup_wLength;

        /* the rest is basically write-once */
        struct usb_config_descriptor    *config, *hs_config;
        struct usb_device_descriptor    *dev;
        struct usb_request              *req;
        struct usb_gadget               *gadget;
        struct list_head                epfiles;
        void                            *buf;
        wait_queue_head_t               wait;
        struct super_block              *sb;
        struct dentry                   *dentry;

        /* except this scratch i/o buffer for ep0 */
        u8                              rbuf [256];
};

static inline void get_dev (struct dev_data *data)
{
        atomic_inc (&data->count);
}

static void put_dev (struct dev_data *data)
{
        if (likely (!atomic_dec_and_test (&data->count)))
                return;
        /* needs no more cleanup */
        BUG_ON (waitqueue_active (&data->wait));
        kfree (data);
}

static struct dev_data *dev_new (void)
{
        struct dev_data         *dev;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev)
                return NULL;
        dev->state = STATE_DEV_DISABLED;
        atomic_set (&dev->count, 1);
        spin_lock_init (&dev->lock);
        INIT_LIST_HEAD (&dev->epfiles);
        init_waitqueue_head (&dev->wait);
        return dev;
}

/*-------------------------------------------------------------------------*/

/* other /dev/gadget/$ENDPOINT files represent endpoints */
enum ep_state {
        STATE_EP_DISABLED = 0,
        STATE_EP_READY,
        STATE_EP_ENABLED,
        STATE_EP_UNBOUND,
```

Context

Compositional static analysis

```
#define N_EVENT                 5

struct dev_data {
        spinlock_t                      lock;
        atomic_t                        count;
        enum ep0_state                  state;          /* P: lock */
        struct usb_gadgetfs_event       event [N_EVENT];
        unsigned                        ev_next;
        struct fasync_struct            *fasync;
        u8                              current_config;

        /* drivers reading ep0 MUST handle control requests (SETUP)
         * reported that way; else the host will time out.
         */
        unsigned                        usermode_setup : 1,
                                        setup_in : 1,
                                        setup_can_stall : 1,
                                        setup_out_ready : 1,
                                        setup_out_error : 1,
                                        setup_abort : 1;
        unsigned                        setup_wLength;

        /* the rest is basically write-once */
        struct usb_config_descriptor    *config, *hs_config;
        struct usb_device_descriptor    *dev;
        struct usb_request              *req;
        struct usb_gadget               *gadget;
        struct list_head                epfiles;
        void                            *buf;
        wait_queue_head_t               wait;
        struct super_block              *sb;
        struct dentry                   *dentry;

        /* except this scratch i/o buffer for ep0 */
        u8                              rbuf [256];
};

static inline void get_dev (struct dev_data *data)
{
        atomic_inc (&data->count);
}

static void put_dev (struct dev_data *data)
{
        if (likely (!atomic_dec_and_test (&data->count)))
                return;
        /* needs no more cleanup */
        BUG_ON (waitqueue_active (&data->wait));
        kfree (data);
}

static struct dev_data *dev_new (void)
{
        struct dev_data         *dev;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev)
                return NULL;
        dev->state = STATE_DEV_DISABLED;
        atomic_set (&dev->count, 1);
        spin_lock_init (&dev->lock);
        INIT_LIST_HEAD (&dev->epfiles);
        init_waitqueue_head (&dev->wait);
        return dev;
}

/*-------------------------------------------------------------------------*/

/* other /dev/gadget/$ENDPOINT files represent endpoints */
enum ep_state {
        STATE_EP_DISABLED = 0,
        STATE_EP_READY,
        STATE_EP_ENABLED,
        STATE_EP_UNBOUND,
```

Context

Compositional static analysis

```c
#define N_EVENT                 5

struct dev_data {
        spinlock_t                      lock;
        atomic_t                        count;
        enum ep0_state                  state;          /* P: lock */
        struct usb_gadgetfs_event       event [N_EVENT];
        unsigned                        ev_next;
        struct fasync_struct            *fasync;
        u8                              current_config;

        /* drivers reading ep0 MUST handle control requests (SETUP)
         * reported that way; else the host will time out.
         */
        unsigned                        usermode_setup : 1,
                                        setup_in : 1,
                                        setup_can_stall : 1,
                                        setup_out_ready : 1,
                                        setup_out_error : 1,
                                        setup_abort : 1;
        unsigned                        setup_wLength;

        /* the rest is basically write-once */
        struct usb_config_descriptor    *config, *hs_config;
        struct usb_device_descriptor    *dev;
        struct usb_request              *req;
        struct usb_gadget               *gadget;
        struct list_head                epfiles;
        void                            *buf;
        wait_queue_head_t               wait;
        struct super_block              *sb;
        struct dentry                   *dentry;

        /* except this scratch i/o buffer for ep0 */
        u8                              rbuf [256];
};

static inline void get_dev (struct dev_data *data)
{
        atomic_inc (&data->count);
}

static void put_dev (struct dev_data *data)
{
        if (likely (!atomic_dec_and_test (&data->count)))
                return;
        /* needs no more cleanup */
        BUG_ON (waitqueue_active (&data->wait));
        kfree (data);
}

static struct dev_data *dev_new (void)
{
        struct dev_data         *dev;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev)
                return NULL;
        dev->state = STATE_DEV_DISABLED;
        atomic_set (&dev->count, 1);
        spin_lock_init (&dev->lock);
        INIT_LIST_HEAD (&dev->epfiles);
        init_waitqueue_head (&dev->wait);
        return dev;
}

/*----------------------------------------------------------------------*/

/* other /dev/gadget/$ENDPOINT files represent endpoints */
enum ep_state {
        STATE_EP_DISABLED = 0,
        STATE_EP_READY,
        STATE_EP_ENABLED,
        STATE_EP_UNBOUND,
```

Properties :

Deadlocks ?

Memory leaks ?

Termination ?

Context

Compositional static analysis

```c
#define N_EVENT                 5

struct dev_data {
        spinlock_t                      lock;
        atomic_t                        count;
        enum ep0_state                  state;          /* P: lock */
        struct usb_gadgetfs_event       event [N_EVENT];
        unsigned                        ev_next;
        struct fasync_struct            *fasync;
        u8                              current_config;

        /* drivers reading ep0 MUST handle control requests (SETUP)
         * reported that way; else the host will time out.
         */
        unsigned                        usermode_setup : 1,
                                        setup_in : 1,
                                        setup_can_stall : 1,
                                        setup_out_ready : 1,
                                        setup_out_error : 1,
                                        setup_abort : 1;
        unsigned                        setup_wLength;

        /* the rest is basically write-once */
        struct usb_config_descriptor    *config, *hs_config;
        struct usb_device_descriptor    *dev;
        struct usb_request              *req;
        struct usb_gadget               *gadget;
        struct list_head                epfiles;
        void                            *buf;
        wait_queue_head_t               wait;
        struct super_block              *sb;
        struct dentry                   *dentry;

        /* except this scratch i/o buffer for ep0 */
        u8                              rbuf [256];
};

static inline void get_dev (struct dev_data *data)
{
        atomic_inc (&data->count);
}

static void put_dev (struct dev_data *data)
{
        if (likely (!atomic_dec_and_test (&data->count)))
                return;
        /* needs no more cleanup */
        BUG_ON (waitqueue_active (&data->wait));
        kfree (data);
}

static struct dev_data *dev_new (void)
{
        struct dev_data         *dev;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev)
                return NULL;
        dev->state = STATE_DEV_DISABLED;
        atomic_set (&dev->count, 1);
        spin_lock_init (&dev->lock);
        INIT_LIST_HEAD (&dev->epfiles);
        init_waitqueue_head (&dev->wait);
        return dev;
}

/*----------------------------------------------------------------------*/

/* other /dev/gadget/$ENDPOINT files represent endpoints */
enum ep_state {
        STATE_EP_DISABLED = 0,
        STATE_EP_READY,
        STATE_EP_ENABLED,
        STATE_EP_UNBOUND,
```

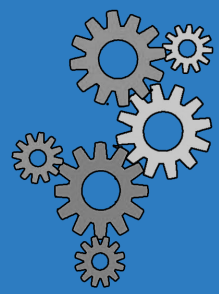
Properties :

Deadlocks ?

Memory leaks ?

Termination ?

Abstraction



Automatic verification

Context

Compositional static analysis

```
#define N_EVENT                 5

struct dev_data {
        spinlock_t                      lock;
        atomic_t                        count;
        enum ep0_state                  state;          /* P: lock */
        struct usb_gadgetfs_event       event [N_EVENT];
        unsigned                        ev_next;
        struct fasync_struct            *fasync;
        u8                              current_config;

        /* drivers reading ep0 MUST handle control requests (SETUP)
         * reported that way; else the host will time out.
         */
        unsigned                        usermode_setup : 1,
                                        setup_in : 1,
                                        setup_can_stall : 1,
                                        setup_out_ready : 1,
                                        setup_out_error : 1,
                                        setup_abort : 1;
        unsigned                        setup_wLength;

        /* the rest is basically write-once */
        struct usb_config_descriptor    *config, *hs_config;
        struct usb_device_descriptor    *dev;
        struct usb_request              *req;
        struct usb_gadget               *gadget;
        struct list_head                epfiles;
        void                            *buf;
        wait_queue_head_t               wait;
        struct super_block              *sb;
        struct dentry                   *dentry;

        /* except this scratch i/o buffer for ep0 */
        u8                              rbuf [256];
};

static inline void get_dev (struct dev_data *data)
{
        atomic_inc (&data->count);
}

static void put_dev (struct dev_data *data)
{
        if (likely (!atomic_dec_and_test (&data->count)))
                return;
        /* needs no more cleanup */
        BUG_ON (waitqueue_active (&data->wait));
        kfree (data);
}

static struct dev_data *dev_new (void)
{
        struct dev_data         *dev;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev)
                return NULL;
        dev->state = STATE_DEV_DISABLED;
        atomic_set (&dev->count, 1);
        spin_lock_init (&dev->lock);
        INIT_LIST_HEAD (&dev->epfiles);
        init_waitqueue_head (&dev->wait);
        return dev;
}

/*----------------------------------------------------------------------*/

/* other /dev/gadget/$ENDPOINT files represent endpoints */
enum ep_state {
        STATE_EP_DISABLED = 0,
        STATE_EP_READY,
        STATE_EP_ENABLED,
        STATE_EP_UNBOUND,
```

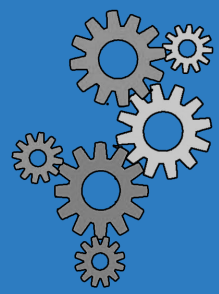
Properties :

Deadlocks ?

Memory leaks ?

Termination ?

Abstraction



Automatic verification

Static analysis of multithreaded programs

Abstract interpretation [Min, 2012]
Software Model Checking (e.g. : Promela/Spin [Holtzmann, 2005])
Compositional verification (e.g. : Mage [Abramsky et al., 2004])
...

Our approach

π-calculus based abstractions

About π-calculus

A process algebra that is expressive and compositional

Modeling

- Parallelism
- Concurrency
- Recursion
- Higher order functions
- Dynamic creation of resources

About π-calculus

A process algebra that is expressive and compositional

Modeling

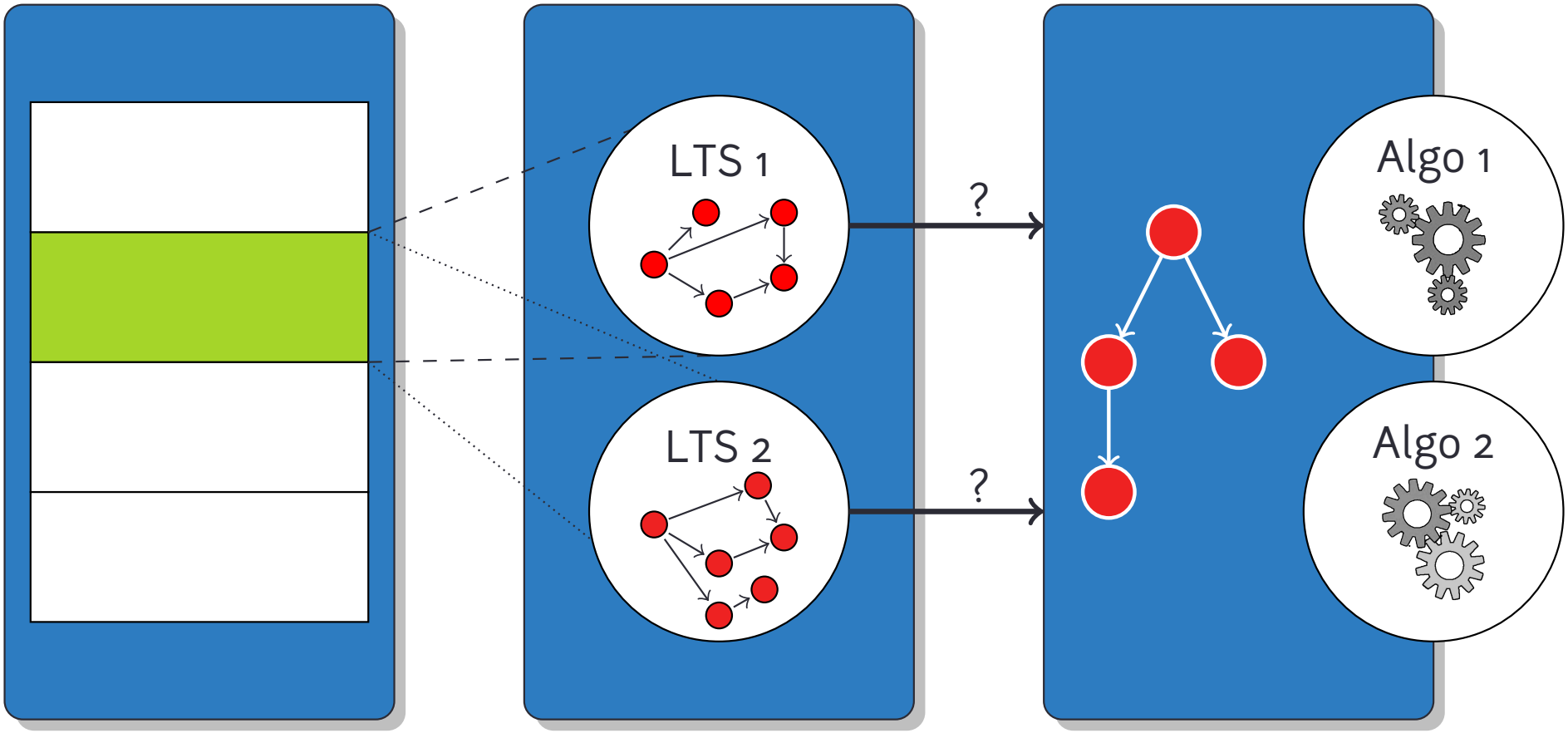
- Parallelism
- Concurrency
- Recursion
- Higher order functions
- Dynamic creation of resources

But...

...name creation and name input do not allow to build ground automata (because alpha-conversion)

Our approach

Introducing an intermediate abstraction to deal with dynamic resources



The Omniscent Garbage Collector Framework (OGC)

An algorithmic framework to deal with dynamic resources, in an automatic verification context

The Omniscent Garbage Collector Framework (OGC)

An algorithmic framework to deal with dynamic resources, in an automatic verification context

Applications

- Pure names management
- Thread identities attribution
- Memory cells attribution
- Fresh names in programming languages

Future work

Improvements and correctness of OGC algorithms

Extension to a π-calculus with values

Verification of properties related to dynamic resources

- Model checking (with a logic for resources)
- Divergence checking