Dynamic Software Update for Behavioural Properties of Concurrent Programs

Gabrielle Anderson
Electronics and Computer Science,
University of Southampton, UK

g.a.anderson@soton.ac.uk

ABSTRACT

I provide two verification techniques for concurrent code. The first guarantees that values obtained from shared resources are of the expected type, and that accesses to shared resources will not deadlock. The second enforces access control on sequences of actions using policy automata. I also provide the first definition and proof of safe dynamic software update for behavioural properties of concurrent systems. I include techniques that reduce verification costs of behavioural properties and update safety.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis, Operational Semantics; D.2.4 [**Software/ Program Verification**]: Concurrent Programming; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Software Corrections, Enhancement

1. INTRODUCTION

In a world where multi-core processors are becoming ubiquitous it is necessary to be able to write concurrent software which takes advantage of such hardware. Correctly developing concurrent (multi-threaded) programs using existing languages and tools, however, is notoriously difficult. On one hand there are the traditional problems of type errors and deadlock, which are errors which most programmers want to ensure don't occur. On the other hand there are specific properties which the program should have, such as communicating according to a protocol or maintaining separation of information which would lead to a conflict of interest. The first question addressed in my thesis concerns how to state and verify that multi-threaded programs conform to desired semantic behavioural properties, with an emphasis on how to re-use parts of proofs between similar systems and on reducing the cost of verifying the properties. In particular I explore deadlock freedom and type-safe usage of values returned from shared resources. An example of the latter property is ensuring that when a thread is expecting to receive a message in a particular format that it does so. I also consider how to guarantee that sequences of actions performed by a concurrent program on shared resources have a certain property, for example ensuring that in all possible executions of a program that all files which are opened are closed again, and that closed files are never read from or written to.

For most applications, software requirements are rarely static and systems are never perfect. Almost all real-life programs will need to be updated at some point after their release; it may be to fix bugs, to add functionality, or to simply make the system more efficient. The naive and simplest method used to update software is to halt the execution of the code, modify this code in some way, and then restart the program. This methodology is certainly both intrusive and disruptive but can also be catastrophic. The restart causes an interruption to service which is in, in its best form, simply an inconvenience. Indeed, restarting a home PC is not usually considered to be especially problematic. In some systems, however, the downtime is simply unacceptable. For example, control software for autonomous devices and safety-critical systems often rely on uninterrupted service. After the launch of the ESA Cassini-Huygens probe to Titan, a software error was discovered that prevented one of the two receivers on Cassini from being used. This prevented the orbital Cassini from receiving data from the Huygens probe, and as a consequence this caused a loss of half of the photos taken by the probe. It was not possible to take down the control software running the orbiter as it would have risked careening out of control; if it had been possible to update the software whilst it was still running it might have been possible to return the orbiter to full functionality. Additionally, without further orchestration of how and when changes are made, the unconstrained updates can cause a multitude of errors. The second question addressed in my thesis concerns what methods should be used to modify programs at run time and how to maintain behavioural properties after updates. Whilst it is possible to write unconstrained updates that modify live code and don't cause errors, to do so consistently requires great skill on the part of the programmer and often is mere serendipity. Hence in order to free the programmer from such accidental complexity I explore static analyses that guarantee that updates do not invalidate behavioural properties of a program; I explore how to maintain the properties of a program that are developed in the first half of the thesis.

$$
\begin{array}{l}
[c \mapsto \emptyset] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{recv}(c) + 4 \\
\quad \rightarrow [c \mapsto \mathtt{true}] \quad () \parallel \mathsf{recv}(c) + 4 \\
\quad \rightarrow [c \mapsto \emptyset] \quad () \parallel \mathtt{true} + 4 \\
\quad \rightarrow [c \mapsto \emptyset] \quad () \parallel \mathtt{error}
\end{array}
$$

Figure 1: Message Passing Error From Not 'Matching Up'

$$
\begin{array}{l}
[c \mapsto \mathtt{true}] \quad \mathsf{send}(c, 3) \parallel \mathsf{recv}(c) + 4 \\
\quad \rightarrow [c \mapsto \mathtt{true}, 3] \quad () \parallel \mathsf{recv}(c) + 4 \\
\quad \rightarrow [c \mapsto 3] \quad () \parallel \mathtt{true} + 4 \\
\quad \rightarrow [c \mapsto 3] \quad () \parallel \mathtt{error}
\end{array}
$$

Figure 2: Message Passing Error From Inconsistent Messages In Queue

In this paper I discuss the methodology of research using programming language analyses (Section 2). In Section 3 I describe two different behavioural analyses for concurrent programs: Variably Typed Return and Concurrent Policy Automata. In Section 4 I discuss how to perform dynamic software update for multi-threaded programs and maintain the safety properties proved in the Variably Typed Return section. I conclude and reflect in Section 5. I include a discussion of related work throughout.

2. RESEARCH USING PROGRAMMING LANGUAGE ANALYSES

The prevailing methodology of the field of program analysis may seem different to that of other sciences, but under the surface it follows a standard approach. The first aspect is to find some interesting phenomena and to observe them. This requires homing in on specific problems rather than generic ones such as writing correct programs. The second aspect is to create abstractions of the phenomena, which generally consists of modelling them mathematically. Abstractions should remove the incidental complexity from the situation and leave only the interesting phenomena. This is to simplify the final aspect of the methodology, proving natural properties. When investigating phenomena, for example systems which send messages back and forth over channels, there are natural questions we may ask, such as whether the receiver can properly process all the messages it will receive, or whether a program will hang forever waiting to receive a value. We can model these properties and attempt to prove them over the abstractions.

Evaluation in the field of programming analysis generally consists of considering how general an approach is, the efficiency of the analysis, what simplifications and restrictions must be made, how straightforward the abstraction is to use and understand, and what other insights the abstraction illuminates.

Characterising the behaviour of concurrent programs, and the update of software at runtime, are problems which are relevant in many technical settings. In order to follow the above methodology which focuses on removing incidental complexity, instead of focusing on a particular programming language I consider the problems using a concurrent lambda

$$
\begin{array}{l}
[c \mapsto \emptyset] \quad \mathsf{send}(c, 1); \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, 2); \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad (\mathsf{recv}(c) + 3); \mathtt{if}\ \mathsf{recv}(c)\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto 1] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, 2); \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad (\mathsf{recv}(c) + 3); \mathtt{if}\ \mathsf{recv}(c)\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto 1, 2] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad (\mathsf{recv}(c) + 3); \mathtt{if}\ \mathsf{recv}(c)\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto 2] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad (1 + 3); \mathtt{if}\ \mathsf{recv}(c)\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto 2] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad \mathtt{if}\ \mathsf{recv}(c)\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto \emptyset] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, \mathtt{false}) \parallel \\
\qquad \mathtt{if}\ 2\ \mathtt{then}\ 4\ \mathtt{else}\ 5 \\
\quad \rightarrow [c \mapsto \emptyset] \quad \mathsf{send}(c, \mathtt{true}) \parallel \mathsf{send}(c, \mathtt{false}) \parallel \mathtt{error}
\end{array}
$$

Figure 3: Message Passing Error From Interleaving

calculus. This language is an abstraction which can represent key features and behaviour of many common languages such as Java and C#. We then may apply our results to any program which can be represented in our setting.

3. BEHAVIOURAL PROPERTIES OF CONCURRENT PROGRAMS

In my thesis I consider behavioural properties of concurrent programs in two different settings. The first consists of a generalisation of message passing systems, where I consider how to prove communications safety and deadlock-freedom (Section 3.1). The second consists of expanding Policy Automata, which use finite state automata to describe incorrect behaviours, to multi-threaded programs (Section 3.2)

3.1 Variably Typed Return

Traditionally, values obtained from shared resources such as channels and shared memory are either treated as untyped, and must be dynamically type checked before they can be used, or are assumed to always be of one specific type. If values of different types are sent and received on shared channels then various errors can occur.

In order to describe correct and erroneous programs I introduce a little notation. Alongside standard programming language commands, a thread contains send and receive actions, where $\mathsf{send}(c, v)$ denotes sending a value on channel c, and $\mathsf{recv}(c)$ denotes receiving a value from channel c. A program consists of the parallel composition of several threads, using the $\parallel$ operator. The shared state of a system is a map from channels to their message queues. For example, the state $[c \mapsto 1, \mathtt{true}, 3]$ denotes that a channel c has three items in its queue, where the head of the queue is 1 and the tail of the queue is 3.

Consider the system and evaluation in Figure 1. Here an error occurs because the type of the value sent and the type of the value received, booleans and integers respectively, do not 'match up'. In order to rule out such erroneous programs

$$
\begin{array}{ll}
[c \mapsto \emptyset] & \mathsf{send}(c, \mathtt{true}) \parallel \neg\mathsf{recv}(c);\ \mathsf{recv}(c) + 4 \\
\rightarrow [c \mapsto \mathtt{true}] & () \parallel \neg\mathsf{recv}(c);\ \mathsf{recv}(c) + 4 \\
\rightarrow [c \mapsto \emptyset] & () \parallel \neg\mathtt{true};\ \mathsf{recv}(c) + 4 \\
\rightarrow [c \mapsto \emptyset] & () \parallel \mathsf{recv}(c) + 4 \\
\not\rightarrow &
\end{array}
$$

Figure 4: Deadlock from Not 'Matching Up'

it is not sufficient to look solely at the programs themselves, however. The shared state also affects the safety of message passing (Figure 2). In addition, even if the send and receive types match up, and the system starts with an empty state, errors can still occur from certain interleavings of multiple threads (Figure 3). Here the interleaving behaviour leads to an error, even though both senders are providing values, in order, which the receiver is expecting. Deadlocks can occur in a similar ways (Figure 4).

Work on Session Typing [18] allows values of different types to be safely sent and received on shared channels, and to prove the absence of deadlock due to communication behaviour. Early work considers these properties for systems where only two threads share a synchronous channel [17,18], such as in the first two above examples. Additional work on Session Typing includes subtyping [15], asynchronous communication [14], and systems where more than two threads share a communications channel [8, 19]. The information about the message passing behaviour revealed by Session Typing analyses can also be used in order to prove other properties such as buffer bounds [12]. A detailed review of Session Typing literature can be found in [13].

The contributions of my thesis are as follows:

I expand Session Typing work on communications safety (an absence of the above type errors) and deadlock freedom to generalised resources and resource accesses, where a resource access can return values of different types [3]. This generalised approach is referred to as *Variably Typed Return*. I define a formal model which describes how to access resources in a general setting. I describe properties of the model which are sufficient to guarantee that programs always receive values from shared resources of a type which they expect, and that deadlocks due to accesses of these shared resources never occur.

Variably Typed Return exposes the essence of prior formalisms and proofs of Session Typing systems. In particular it indicates how to prove the above properties in systems with semantics different to those traditionally considered by Session Typing analyses, such as those with non-blocking message passing or fixed sized queues. The general case is computationally expensive as it reduces to model checking, however in specific systems, such as those presented above, the cost of checking can be reduced to linear in the size of the program. In addition, the definition of the general safety property in this approach exposes a commonality between this property and access control performed using policy automata (Section 3.2).

3.2 Concurrent Access Policies

Finite state policy automata can be used to represent access control. They can guarantee that various properties are maintained, such as correct use of mutual exclusion, prevention of spoofing attacks by web servers, enforcing Chinese Wall properties, and prevention of denial of service attacks. The work in [5–7] presents a methodology for efficiently model checking the policies against the actions of a program. Of particular note is their approach of transforming resource usages into Basic Process Algebra (BPA) and using a weakened form of model checking to make their approach complete as well as sound. In [6], Bartoletti et al. posit that their approach could be simply extended to multi-threaded programs by transforming into Basic Parallel Processes [11] rather than BPA [6]. Recent work in equivalence decidability [20], however, shows that trace equivalence, on which their approach is based, is undecidable for BPPs.

The contributions of my thesis are as follows:

I extend work by Bartoletti et al. on local access policies to multi-threaded programs by representing multi-threaded effects using trace equivalent single-threaded effects. I refer to this approach as *Concurrent Policy Automata*. I show how to apply this approach to concurrent programs. Specifically I show how to represent a specific class of concurrent programs as trace-equivalent sequential programs, and prove that this representation is well defined This permits us to use existing model checking work. In order to make the transformation from concurrent to sequential programs feasible it was necessary to restrict concurrent programs to only permit recursion at a per-thread level. Whilst this restriction ensures simply that concurrent programs have a finite state space, it may be possible to use some less stringent restrictions which achieve the same goal. The result that we can use existing model checking techniques, however, make the approach simple to use and implement.

I also identify a promising techniques which may reduce the model checking cost. I exploit the insight that we need only model check for safety with respect to some policy up until the point an execution becomes blocked and investigate whether the space which needs to be model checked can be reduced. This method is a variant of a symbolic model checking technique, where the reachability analysis prunes sections of the search space which cannot occur in practice due to blocked actions (such as one thread acquiring a mutex lock before it is released by another) [21]. I show how it is sufficient to consider only the parts of the shared state which can influence the blocking actions, such as the mutexes, rather than the entire state, such as the mutexes and the shared variables. In future I hope to collaborate with others to incorporate these technique into a model checker and to determine experimentally the savings which can be obtained.

4. DYNAMIC SOFTWARE UPDATE

The goal of dynamic software update (DSU) is to permit modification of software without shutting down the system, and to guarantee that some properties of the system which held before the update continue to hold after the update. This removes for the programmer the complexity of having to worry about the safety of her patch, as such an analysis will guarantee it.

$[c \mapsto \emptyset] \quad \mathsf{send}(c, \mathtt{true}); \mathsf{send}(c, 1); f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2)) \parallel \mathsf{recv}(c); \mathsf{recv}(c); g(\neg\mathsf{recv}(c); \mathsf{recv}(c) + 3)$

$\rightarrow \ldots$

$\rightarrow [c \mapsto 1] \quad f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2)) \parallel \mathsf{recv}(c); g(\neg\mathsf{recv}(c); \mathsf{recv}(c) + 3)$

$\rightarrow [c \mapsto \emptyset] \quad f(\mathsf{send}(c, false); \mathsf{send}(c, 2)) \parallel g(\neg\mathsf{recv}(c); \mathsf{recv}(c) + 3)$

$\rightarrow^{u} [c \mapsto \emptyset] \quad f(\mathsf{send}(c, \mathtt{true}); \mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 3)) \parallel g(\neg\mathsf{recv}(c); \mathsf{recv}(c) + 3)$

$\rightarrow \ldots$

$\rightarrow [c \mapsto \emptyset] \quad \mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 3) \parallel \mathsf{recv}(c) + 3$

$\rightarrow [c \mapsto \mathtt{false}] \quad \mathsf{send}(c, 3) \parallel \mathsf{recv}(c) + 3$

$\rightarrow [c \mapsto \emptyset] \quad \mathsf{send}(c, 3) \parallel \mathtt{false} + 3$

$\rightarrow [c \mapsto \emptyset] \quad \mathsf{send}(c, 3) \parallel \mathtt{error}$

Figure 5: Update Error From Not 'Matching Up'

$[c \mapsto \emptyset] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$

$\rightarrow [c \mapsto \mathtt{false}] \quad \mathsf{send}(c, 2); \underline{X} \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$

$\rightarrow \ldots$

$\rightarrow^{u} [c \mapsto \mathtt{false}, 2] \quad \mu\underline{X}.f(\mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g((\mathsf{recv}(c) + 3); \underline{X}')$

$\rightarrow \ldots$

$\rightarrow [c \mapsto \mathtt{false}, 2] \quad \mu\underline{X}.f(\mathsf{send}(c, 2); \underline{X}) \parallel (\mathsf{recv}(c) + 3); \underline{X}'$

$\rightarrow [c \mapsto 2] \quad \mu\underline{X}.f(\mathsf{send}(c, 2); \underline{X}) \parallel (\mathtt{false} + 3); \underline{X}'$

$\rightarrow [c \mapsto 2] \quad \mu\underline{X}.f(\mathsf{send}(c, 2); \underline{X}) \parallel \mathtt{error}; \underline{X}'$

Figure 6: Update Error From Messages In The Queue

There are two main approaches in existing work on DSU. The first approach focuses on a systems approach, with informal guarantees. In [1], Ajmani et al. focus on the update of object oriented systems, and use Simulation Objects to interface between old and new versions of classes, where the old and new classes may have different class signatures (method types, internal fields, etc.). In [10], Chen et al. use Virtualization Machine Monitors to run multiple operating systems at the same time, permitting one to handle the old code and data, and one to handle the updated code and data. In [25], Subramanian et al. also focus on the update of object oriented systems, with an aim to reduce the cost of the update infrastructure as much as possible, and do so by only performing updates on objects when no objects of that class are in use. The second approach focuses on formal models and properties which can be proved about those models. Early work considers how to modify module based single-threaded programs, when the changes do not change the type signature (the types of the parameters and of the return values) of module functions [9]. This approach is used informally in KSplice [4] which is a practical implementation of DSU for the Linux kernel. In [24] this approach is expanded to C-like single-threaded programs and permit updates which change the type signatures of functions. Transactional approaches to DSU are introduced in [23], and built upon in [22] to permit updates which change the type signatures of functions in multi-threaded programs.

Designing multi-threaded programs is non-trivial: standard problems include synchronisation issues, deadlock and race conditions. These properties cannot be fully characterised using simple typing. The existing work on DSU of multi-threaded programs with behavioural properties focuses on regression testing [16]. This ensures that any update does not change the result of the regression tests, and reducing the number of tests required to get total coverage. Whilst this approach is powerful in development processes which do not require hard guarantees, it cannot provide a proof of absence of errors.

I explore how to perform DSU on systems which have formal behavioural properties, by focusing on how to coordinate the timing and scope of updates to guarantee type safety in systems with Variably Typed Return. In addition I develop techniques which reduce the cost of checking safety and maximise the window of time in the code's execution when updates can be applied.

In order to describe correct and erroneous updates to programs I introduce some additional notation. I use $\mu\underline{X}.e$ to denote recursion, where we substitute $\mu\underline{X}.e$ for $\underline{X}$ in e. I use $f(e)$ to annotate a region of code e with a name f. In the following examples I update named regions, replacing the entire body of the region with a new body of code (as in [9]).

Consider the system and evaluation in Figure 5. The reduction annotated u denotes where the update takes place. In this example the update replaced the code of f, but not the code of g, in such a way that the two threads no longer 'matched up'. This leads to an error similar to Figure 5.

There are, however, less trivial errors which can occur due to the timing of an update (Figure 6). Here, the code in the updated regions 'matches up', as one repeatedly sends an integer and one repeatedly receives an integer, but the update still leads to an error as the values in the message queue at the time of the update have not yet been consumed [2].

Instead of permitting updates to occur at any point in execution, which can lead to errors seen in Figure 6, we can instead restrict these points and prove the safety of updates

$$[c \mapsto \emptyset] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

$$\rightarrow [c \mapsto \mathtt{false}, 2] \quad \underline{X} \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow [c \mapsto \mathtt{false}, 2] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

$$\rightarrow [c \mapsto \emptyset] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow^u [c \mapsto \emptyset] \quad \mu\underline{X}.f(\mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g((\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

Figure 7: Update Success: Updates Can Only Occur When In Starting Configuration

$$[c \mapsto \emptyset] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

$$\rightarrow [c \mapsto \mathtt{false}, 2] \quad \mu\underline{X}.f(\mathsf{send}(c, \mathtt{false}); \mathsf{send}(c, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow^u [c \mapsto \mathtt{false}, 2] \quad \mu\underline{X}.f(\mathsf{send}(d, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

$$\rightarrow [c \mapsto \emptyset, d \mapsto 2, 2] \quad \mu\underline{X}.f(\mathsf{send}(d, 2); \underline{X}) \parallel \mu\underline{X}'.g(\neg\mathsf{recv}(c); (\mathsf{recv}(c) + 3); \underline{X}')$$

$$\rightarrow^u [c \mapsto \emptyset, d \mapsto 2, 2] \quad \mu\underline{X}.f(\mathsf{send}(d, 2); \underline{X}) \parallel \mu\underline{X}'.g((\mathsf{recv}(d) + 3); \underline{X}')$$

$$\rightarrow \ldots$$

Figure 8: Update Success: Updates Occur Separately

in these situations. This ensures, assuming the programmer is changing from one message passing program which is verifiable using standard Session Typing techniques to another, that the update will be safe, and hence the programmer need not worry about how the update may interact with the existing protocol.

Consider the same starting program as in Figure 6, but with the restriction that updates can only occur when the shared state is empty, and the code is at the same point as that at which it started. We explore a reduction of such a system in Figure 7. The update converts one starting configuration into another, and we can easily show that such a configuration is safe and deadlock free using existing techniques. This approach has some disadvantages, however; there is no guarantee that the system will ever reach the initial configuration, and hence an update can be delayed indefinitely.

Consider instead the same starting program but with different restrictions. The update to the sender thread can occur whenever it reaches the initial configuration in terms of code, but it can ignore the message queue. The receiver can only update when the message queue for c is empty. We consider a reduction of such a system in Figure 8. When the update occurs, it changes the program so that it communicates on a new channel d. Hence the receiver will use up all the values in c's message queue, and will then migrate to the new system. With these restrictions the updates occur separately and will never be delayed indefinitely. The general form of this restriction is requiring the thread to be at the starting point in terms of code, and that all channels on which a thread receives are empty. As the producer never receives it can therefore update as soon as it reaches the top of its main loop.

The contributions of my thesis are as follows:

I develop a formal model for systems with Variably Typed Return which can be updated at runtime. I extend the natural safety properties of Variably Typed Return, generalised from Session Typing analyses, to systems with dynamic software update, and prove them. The general case is again computationally expensive, but the cost can also be reduced using techniques described in work on Variably Typed Return.

I identify how to define and perform updates for message passing systems which make use of a concurrent communications protocol abstraction known as Global Session Types. In an arbitrary program which uses message passing, when applying an update it is necessary to examine significant portions of the runtime state, as well as the program counters of the threads, in order to guarantee that updates can be safety applied. This inspection is necessary to rule out errors such as those displayed in Figure 6. I identify several restrictions on when updates can occur, prove safety and deadlock freedom in such cases. In both these cases the cost of the check is linear in the size of the modified program.

5. CONCLUSION

I define two analyses for behavioural properties of concurrent programs. The first focuses on guaranteeing that, when accessing shared resources, programs will always receive values of the type which they expect and will never deadlock due to accesses to shared resources. The second focuses on ensuring that sequences of actions occur according to a finite state machine specification. I provide the first definition and proof of safe dynamic software update for concurrent programs and show how to maintain type safe access of shared resources (the former safety property). I show how to verify these properties in general, and how to reduce the cost of verification in specific cases such as message passing systems. I provide a compelling example in the form of message passing systems with Global Session Types [8].

6. REFERENCES

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular Software Upgrades for Distributed Systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 452–476. Massachusetts Institute of Technology, 2006.

[2] Gabrielle Anderson and Julian Rathke. Migrating protocols in multi-threaded message-passing systems. In *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*, pages 8:1–8:5, New York, New York, USA, 2009. ACM.

[3] Gabrielle Anderson and Julian Rathke. Resource Access with Variably Typed Return. In *Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, number Section 4, pages 51–57, Saarbrucken, 2011.

[4] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, Nuremberg, Germany, 2009. ACM.

[5] Massimo Bartoletti. Usage Automata. In *Proceedings of the Workshop on Issues in the Theory of Security. Lecture Notes in Computer Science*, pages 52–69, Berlin, 2009. Springer.

[6] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Model Checking Usage Policies. In Christos Kaklamanis and Flemming Nielson, editors, *Trustworthy Global Computing: 4th International Symposium*, volume Trustworth, pages 19–35. Springer Berlin / Heidelberg, 2008.

[7] Massimo Bartoletti, Pierpaolo Degano, Gian-Luigi Ferrari, and Roberto Zunino. Local Policies For Resource Usage Analysis. *ACM Transactions on Programming Languages and Systems*, 31(6):1–43, August 2009.

[8] Lorenzo Bettini, Mario Coppo, Marco De Luca, Mariangiola Dezani-ciancaglini, and Nobuko Yoshida. *Global Progress in Dynamically Merged Multiparty Sessions*, pages 418–433. Springer Berlin / Heidelberg, 5201 edition, 2008.

[9] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing Dynamic Software Updating. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, pages 1–17, Poland, 2003.

[10] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, Ottawa, Ontario, Canada, 2006. ACM.

[11] Scren Christensen, Yoram Hishfeld, and Faron Moller. Bisimulation Equivalence is Decidable for Basic Parallel Processes. In *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715, pages 143–157, London, UK, 1993. Springer-Verlag.

[12] Pierre-malo Deni and Nobuko Yoshida. Buffered Communication Analysis in Distributed Multiparty Sessions. In *CONCUR 2010 - Concurrency Theory*, pages 343–357, Paris, 2010. Springer.

[13] Mariangiola Dezani-ciancaglini and Ugo De Liguoro. Sessions and Session Types : An Overview. In *Proceedings of the 6th international conference on Web services and formal methods*, pages 1–28, Bologna, Italy, 2010. Springer-Verlag.

[14] Simon Gay and Vasco Vasconcelos. Asynchronous Functional Session Types. Journal article, University of Glasgow, 2007.

[15] Simon J Gay and Malcolm J Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[16] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient systematic testing for dynamically updatable software. In *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*, pages 9:1–9:5, New York, New York, USA, 2009. ACM.

[17] Kohei Honda. *Types for Dyadic Interaction*, volume 715, pages 509–523. Springer Berlin / Heidelberg, 1993.

[18] Kohei Honda, Makoto Kubo, and Vasco Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOPâ98, volume 1381 of LNCS*, volume 171, pages 122–138, July 1998.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 43, pages 273–284, San Francisco, California, USA, January 2008. ACM.

[20] Hans Hüttel, Naoki Kobayashi, and Takashi Suto. Undecidable equivalences for basic parallel processesâ. *Inf. Comput.*, 207(7):812–829, July 2009.

[21] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[22] Iulian Neamtiu and Michael Hicks. Safe and Timely Dynamic Updates to Multi-threaded Programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, volume 44, pages 13–24, Dublin, Ireland, 2009. ACM.

[23] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, pages 37–49, New York, New York, USA, 2008. ACM.

[24] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4):183–194, 2005.

[25] Suriya Subramanian, Michael Hicks, and Kathryn S Mckinley. Dynamic Software Updates : A VM-centric Approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, number 1, pages 1–12, Dublin, Ireland, 2009. ACM.