A Framework for Mobile Java Applications

Andrew Hughes
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street
Sheffield, UK
S1 4DP
andrew@dcs.shef.ac.uk

ABSTRACT

We present the Dynamic Theory Execution (DynamiTE) framework for creating concurrent object-oriented applications, with semantics grounded in a process calculus. DynamiTE allows a system to be constructed as a series of distinct mobile components called *environs* which can change position during execution, and between which individual processes can migrate.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Design

Keywords

Java, Mobility, Process Calculus, CCS, Ambient Calculus, CaSE, TNT

1. INTRODUCTION

At an implementation level, concurrent systems tend to be designed in a very ad-hoc way, resulting in complex concepts such as interprocess communication and code migration becoming difficult to manage and control. DynamiTE provides a framework which abstracts away the implementation details of such concepts, allowing the programmer to concentrate instead on a set of simpler constructs grounded in the formal theory of the process calculus, TNT. In this paper, we first present an overview of TNT (section 2)[1], before looking at how its concepts are implemented within DynamiTE (section 3) and closing with a consideration of related and future work.

[1]The reader is refered to the cited papers for the exact semantics and examples, as space is limited here

Ja4Mo Workshop/PPPJ 2007, Sep. 5–7, 2007, Lisboa, Portugal.

2. THE THEORETICAL BACKBONE

DynamiTE is based on the process calculus TNT (*Typed Nomadic Time*) [2, 3], which provides a formal abstraction of concurrent process behaviour. Using this theoretical framework, a thread of execution can be described as a series of sequential actions which incorporates internal behaviour, interprocess communication (2.1), multiprocess synchronisation (2.2) and mobility (2.3, 2.4). TNT utilises and combines well-established concepts from existing process calculi, including synchronisation from Milner's CCS [5], mobility from Cardelli and Gordon's Ambient Calculus [1] and global discrete time from Norton, Lüttgen and Mendler's CaSE [7].

Take the following example process, $a.\tau.b.\mathbf{0}$. The . operator is used to prefix a process with an action and is used repetitively to form a complete description of the behaviour of a process. In a formal syntactic definition, this is written $a.P$ to denote an action a followed by another process, P, which may itself be of the form $a.P$. This example states that three actions should be performed, a, τ and then b before the process evolves into the predefined $\mathbf{0}$ process, which represents a process with no explicit behaviour[2].

We generally classify such actions by their observability. The action with the name τ is special, as it denotes arbitrary internal behaviour. Other actions (a and b in the above example) are observable; generally, when comparing two processes, it is usual only to match on such observable behaviour, and discount internal activity.

Such sequential behaviour can be made more interesting by introducing non-deterministic behaviour via three control flow operators, $+$, $\lfloor P \rfloor \sigma(Q)$ and $\lceil P \rceil \sigma(Q)$. The first of these, $+$, is a basic choice operator inherited from CCS. When two processes, P and Q, are connected by this operator, a non-deterministic choice will be made which causes one to execute and the other to be lost. From the process, $a.P+b.Q$ either an a or b action may be performed. Where the action is a, we are left simply with the process P. Likewise, if b is executed, Q remains.

The other two operators interact with TNT's notion of global discrete time, realised by *clock signals*. Such signals are emitted in situations where they can not be pre-empted by the presence of a high-priority action such as τ. All other action prefixes and the $\mathbf{0}$ process can idle, while the clock ticks, representing the passage of time. The two timeout

[2]It still exhibits some behaviour, as it can idle over time.

operators allow processes to respond to the ticks of a specific clock, the left-hand process executing if the clock does not emit a signal and the right-hand executing if it does. In $\lfloor a.P \rfloor \sigma(b.Q)$, we are left with $b.Q$ if the clock σ ticks. If it doesn't, then an a action occurs instead and the resulting process is simply P.

TNT provides two timeout operators, which have different behaviour with respect to processes idling. The above example uses the fragile variant, where, if the left-hand side ($a.P$) can idle over another clock, the ticks of the other clock are treated like actions and cause the process to evolve to become P. The alternate stable operator $\lceil P \rceil \sigma(Q)$ lets the timeout stay in place; it is only removed when an explicit action from P occurs or σ ticks.

A clock may be prevented from ticking by using the Δ_σ operator. More generally, Δ stops all clocks. This transcends up through the binary operators, $+$ and $|$ (introduced below), as they require the constituent processes to both be able to idle in order for the resulting process to do so. For example, σ may not tick over $a.\mathbf{0} + \Delta_\sigma$.

Finally, a process can also be defined with recursive behaviour. The process, $\mu X.a.X$ repeatedly produces a actions. This is achieved by the appearance of μX which binds the variable, X, to the content appearing after the X.. When X occurs in the process body, it results in a substitution. More simply put, when the a action in this example is performed, the process will become simply X. This is then replaced by its value, $a.X$, allowing another a action to occur and so on.

2.1 Interprocess Communication

The constructs we describe above are fine for specifying sequential systems, but the main focus of process calculi is to provide an abstract representation of concurrent systems. TNT, and its predecessors, make provision for this via the parallel composition operator, $|$. When two processes, P and Q are joined with this operator, they are said to execute concurrently.

The actual concurrent operation of the two processes is realised through *interleaving*. A process $a.P \mid b.Q$ will evolve into one of two possible processes, $P \mid b.Q$ or $a.P \mid Q$, by performing either the a or b action respectively. Thus, the behaviour of $|$ is just like that of $+$, except that both sides remain in place, since this represents two concurrent processes rather than a control branch in a single process.

With this mechanism, we can represent two orthogonal processes running at the same time and the different permutations of action sequences possible from applying an interleaved semantics to concurrency. However, to represent truly interesting behaviour, we need to also allow the processes to interact. Action naming again becomes significant here, as we assume that two processes interact if they emit a corresponding pair of actions simultaneously.

We've tended to use actions named a and b above. One of the reasons for this is that these two actions don't pair up. An action a can only pair up with a corresponding co-action, $\overline{a}$. It is common to see the name a as being a reference to a channel a, with the action a being an input and $\overline{a}$ being the output. Indeed, this is how they are used within DynamiTE.

A process such as $a.P \mid \overline{a}.Q$ can evolve to $P \mid \overline{a}.Q$ or $a.P \mid Q$, just as we saw above. However, as a and $\overline{a}$ are both available at the same time, the two processes can synchronise, causing both processes to evolve in one step to $P \mid Q$. Such behaviour can be enforced by restricting the scope of the name a. In the process, $(a.P \mid \overline{a}.Q)/\{a\}$, actions involving a (both a and $\overline{a}$) can only be observed within the brackets due to the presence of the restriction operator, $/\{a\}$. As a result, the two are forced to synchronise with each other.

2.2 Process Synchronisation

Synchronisation is a fundamental part of the calculus, and observable actions, in practise, are used for this purpose. Thus, actions should be restricted at appropriate points to enforce this behaviour. Clearly, combinatorial explosion may result if restriction is not appropriately applied, as each pair of actions will produce three alternatives, rather than one deterministic action.

Another pertinent point is that synchronisation emits an internal action. Recall the behaviour of clocks described above; clock ticks are pre-empted by such internal actions and so communication also takes precedence over time progression. This puts process interaction on an equal footing with the internal behaviour of a single process.

This synergy of process synchronisation and time is interesting, because it allows us to effectively detect when all possible interactions have taken place. A classic example of this is when a process wants to broadcast to an arbitrary number of recipients. How can we construct such a process, given what we've seen above?

The obvious solution is for the process to output on the channel just the required number of times. However, this doesn't give a flexible solution which can handle an arbitrary number. Instead of having a general broadcast agent, we have a process that can transmit to three processes ($\overline{o}.\overline{o}.\overline{o}.P$) and we need a new one for transmitting to four ($\overline{o}.\overline{o}.\overline{o}.\overline{o}.P$).

Alternatively, we can define the process using our recursion operator, $\mu X.\overline{o}.X$. This, of course, works for any number of recipients, but is fundamentally flawed. What happens when no-one wants to receive on o any more? This process will still go on providing an output; note that there is no P process in this version, because the process never continues on to do something else. In a practical implementation, this corresponds to a thread that never terminates.

The solution is to utilise the timeout operator to provide a base case for the recursion. When $\overline{o}$ can synchronise with a recipient, o, the resulting internal action, τ, will stop the clock from ticking. Thus, when the clock does tick, it demonstrates that no further synchronisations can take place and so our broadcast agent can go and do something else, which we simply refer to as P. Formally, this is written as

$$\mu X.\lceil \overline{o}.X \rceil \sigma(P) \mid o.Q \mid o.R$$

where $o.Q$ and $o.R$ are two recipients. We can trivially add more, as prior knowledge of the recipients is no longer required. If any process is listening on o, a synchronisation will take place between it and the broadcast agent. The broadcast agent will then recurse, recreating the original situation with one less recipient. When there are no further recipients, σ will be allowed to tick, causing the broadcast agent to evolve into P.

Such a n-ary process synchronisation mechanism is believed to be novel within the field of mobile process calculi, originating from a non-mobile process calculus (CaSE) and being extended within TNT.

2.3 Structural Mobility

The interactions described so far are all localised. Mobility in TNT is realised by a hierarchy of locations we refer to as *environs*. Processes reside within these environs, and their interaction is limited to within their bounds. Environs also restrict the behaviour of clocks. Each environ has an associated set of clocks, which can tick within that environ and any sub-environs. Outside the environ's bounds, the clock ticks are transformed into internal actions, which then pre-empt the ticks of any clock further up the hierarchy.

Environs are given a name and a security policy in the form of a special 'bouncer'[3] process. Syntactically, they appear as

$$m[\mathbf{0}]^{\overline{\circledcirc}.\overline{\oslash}.\Omega}_{\{\sigma\}}$$

The environ is called m[4] and contains the simple process, $\mathbf{0}$. The clock σ may tick within the bounds of m, but such ticks appear as internal actions outside. The sequence $\overline{\circledcirc}.\overline{\oslash}.\Omega$ represents the bouncer process, which restricts the usage of mobility primitives with respect to m.

These mobility primitives are provided through further syntactic constructs, three of which allow the hierarchy to change during execution and two that allow processes to move (see 2.4). All five must pair up with a corresponding co-primitive (in much the same way as actions match co-actions) provided by the bouncer of the environ concerned. In doing so, they emit a high-priority action, which, like internal actions, pre-empts clock ticks. This allows mobility primitives to be used in a broadcast style, in the same way as we used actions in section 2.2.

Such behaviour is best demonstrated by example. In the following process,

$$n[\circledcirc m.P]^{\Omega}_{\emptyset} \mid m[\mathbf{0}]^{\overline{\circledcirc}.\overline{\oslash}.\Omega}_{\{\sigma\}}$$

$\circledcirc m$ instructs the surrounding environ n to attempt to move inside its sibling, m. It may only do so if the bouncer of m provides the corresponding co-primitive $\overline{\circledcirc}$. This is true in the above, where $n[\circledcirc m.P]^{\Omega}_{\emptyset}$ may move inside the environ m and continue as $n[P]^{\Omega}_{\emptyset}$ within this new environment. This results in

$$m[\mathbf{0} \mid n[P]^{\Omega}_{\emptyset}]^{\overline{\circledcirc}.\Omega}_{\{\sigma\}}$$

$\oslash m$ provides the opposite behaviour, allowing the surrounding environ to leave m, a parent environ. If we assume P in the above expands to $\oslash m.P'$, then the process can again interact with the bouncer and cause n to move outside m, giving

$$n[P']^{\Omega}_{\emptyset} \mid m[\mathbf{0}]^{\Omega}_{\{\sigma\}}$$

which is fairly close to the original process, the exception being that $\circledcirc m.P$ has evolved into P' and the bouncer has become simply Ω. Note that Ω is the equivalent of $\mathbf{0}$ for bouncers, and so no further mobility interactions can involve m, making it immobile (this is the case with the bouncer of n from the start).

Clearly, via these two primitives, the hierarchy may be rearranged arbitrarily. The final structural primitive allows environs to be removed completely[5]. Again, such an operation must be permitted by the bouncer of the environ. This prevents arbitrary destruction of environs. Instead, an environ must effectively be defined as removable on creation.

A bouncer exhibiting the co-action $\overline{\circledast}$ allows an environ to be destroyed. When an ambient with such a bouncer is run in parallel with the process $\circledast m.P$, as in

$$\circledast m.P \mid m[Q]^{\overline{\circledast}.\Omega}_{\{\sigma\}}$$

the environ m will disappear and the process inside will enter the environ above giving

$$P \mid Q$$

The bouncer of the removed environ is simply lost. The clock set is unified with the clock set of the parent environ, so the operation effects both Q (now executing in a different context) and P (which can now see the ticks of any clock previously hidden inside m).

2.4 Process Mobility

The final feature of TNT is process mobility. Unlike the structural mobility described above, this is *objective*; the process which exhibits the mobility primitive does not move itself, but instead causes another process to move. The migrating process is determined by matching the action name mentioned in the mobility primitive with one emitted by another process. Consider the composition

$$on\ a \circledcirc m.P \mid a.Q \mid m[\mathbf{0}]^{\overline{\circledcirc}.\overline{\oslash}.\Omega}_{\{\sigma\}}$$

where the first process may perform $on\ a \circledcirc m$, causing the second to move,

$$P \mid m[\mathbf{0} \mid Q]^{\overline{\circledcirc}.\Omega}_{\{\sigma\}}$$

its continuation Q now evolving inside the environ m.

Subjective movement can still be performed by forking a process in two. For example, suppose Q diverges to become $b.Q' \mid on\ leave \oslash m.\mathbf{0}$, where the process on the right moves the one on the left outside m. This then allows the inverse operation to be performed subjectively,

$$P \mid Q' \mid m[\mathbf{0} \mid \mathbf{0}]^{\Omega}_{\{\sigma\}}$$

to again give a final process which is very similar to the original.

To summarise, the full syntax of TNT is presented below:

$$\begin{array}{rcl} \mathcal{E}, \mathcal{F} & ::= & \mathbf{0} \mid \Omega \mid \Delta \mid \Delta_{\sigma} \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid \mathcal{E} \mid \mathcal{F} \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ & & \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus A \mid m[\mathcal{E}]^{\mathcal{F}}_{\sigma} \mid \mathcal{M}.\mathcal{E} \\ \mathcal{M} & ::= & \circledcirc m \mid \oslash m \mid \circledast m \mid on\ \beta \circledcirc m \mid on\ \beta \oslash m \mid \\ & & \overline{\circledcirc} \mid \overline{\oslash} \mid \overline{\circledast} \end{array}$$

3. MAPPING THEORY TO PRACTICALITY

DynamiTE uses the TNT process calculus described above as the basis for a concurrent object-oriented framework. Within this framework, developers can create concurrent applications simply by implementing the specific behaviour they require in appropriate subclasses. Each syntactic construct is mapped to an appropriate Java class, which provides the required functionality and relates to others via a

[3]Named after the staff who restrict access to a night club. American usage: doorman/woman.

[4]Names may be of any length, but we prefer single letters for formal representations to maintain brevity. The same also applies to channel names.

[5]The opposite of this, creating an environ, is achieved by simply evolving a process into a new environ e.g. $a.b.n[P]^{\Omega}_{\sigma}$

common `Process` superclass. Operation follows a top-down approach; the complete system is represented by a single instance of one of these classes which, in most cases, will be an operator that composes together further instances as appropriate.

The simplest `Process` subclass is the representation of **0**, realised as a class `Nil` which provides process termination. The internal action τ is realised as an abstract class `Tau` and this is where the user can implement arbitrary sequential behaviour as required, by providing a subclass. The observable actions form part of the channel subsystem, described in 3.1.

The + operator is implemented as a class which contains a list of subprocesses from which one is chosen at random. The action to perform is computed by traversing the hierarchy, so restriction is simply a matter of providing appropriate filtering, thus preventing the restricted names from travelling further up the hierarchy.

More interesting is the `Par` class which implements the | operator, as it must allow its subprocesses to operate concurrently. The most obvious way to achieve this is by mapping individual processes onto Java threads. This also means that data can be stored with the process by means of thread-local variables. However, we are keen to offer flexibility in how the individual features of the framework are implemented. Java thread mapping is only one way in which concurrent processing may be implemented and so we abstract away `Par` from the threading implementation as much as possible, thus allowing it to be replaced by other implementations at a later date. For example, concurrent processing could also be provided by distinct processes spawned by the VM or a more complex distributed solution may become apparent.

3.1 The Channel Abstraction

In the same vein, the implementation of synchronisation channels is abstracted in such as way as to allow for differing implementations. Here, the provision of multiple implementations is more prevalent and so a plugin mechanism is already present. Fortunately, Java already has plenty of support for plugin based frameworks (imaging and sound already being implemented in this fashion) and the new `java.util.ServiceLoader` API provided in 1.6 makes this simpler still. This allows the user to have freedom of choice with respect to their chosen channel implementation, which may even be further extended by their own or third-party plugins.

At its simplest, DynamiTE provides a way of testing TNT processes and ensuring they perform as expected. In this respect, the simplest channel plugin is a dummy channel, which need do nothing more than simply exist. More complex solutions are of course possible and are needed to make the framework both usable and interesting.

Although currently there is no realisation of data within the formal layer of the calculus, this only matters to the extent that we wish transmitted data to alter the constructs themselves via substitution[6]. Data can be transferred between processes and used within internal actions without having to be explicitly realised at the formal level. There are a multitude of ways of implementing data transfer, ranging from simple mechanisms like files and sockets to more full-blown interprocess communication protocols such as Java's Remote Method Invocation (RMI), the Common Object Request Broker Architecture (CORBA) and web services. The plugin nature of the channel architecture means that any of these possibilities may be used and more besides.

While the implementations of the channels themselves can provide the input and output mechanisms, interoperability between the two has to take place at a higher level. Thus, the onus is on the parallel implementation, `Par`, to co-ordinate the communication between the two, by virtue of discovering which names are exposed at the point of composition.

A possible simplification becomes apparent here, as some implementations may make use of channel naming. For example, if the channel name refers to a host and port for a TCP/IP implementation, then the sender need only try and connect to see if a recipient is available. Channel names are assumed to be unique, so such a mapping is possible. However, they are not unique to a particular process, making it perfectly plausible for the channel name to occur simultaneously on multiple processes and thus for a competition to occur. There is also the issue of whether they can actually 'see' each other, according to the constraints of the calculus, so the decision should still be left to an appropriate parallel composition operator.

3.2 Signalling

One of the most interesting parts of the DynamiTE framework is the implementation of clock signals. While there have been other attempts to produce frameworks or languages based on process calculi (see section 4), we believe that the rendering of discrete time into such a context is novel.

The first question to answer when attempting to perform such a translation is where to actually locate the clocks. Within TNT, the obvious answer is within each environ, as these are responsible for providing the division between processes which can observe clock ticks and those which can not. For example, the following environ

$$m[P]^{\Omega}_{\{\sigma\}}$$

would be realised as an instance of the `Environ` class with the name m. This instance would maintain a reference to the process P with which it interacts. Not only is the execution of P controlled by the environ (as with the implementations of + and | above), but it also controls when and how the ticks of σ reach P.

Recall our earlier description of the calculus, where we mentioned how clock ticks are always pre-empted by high priority actions, which may arise either from explicit internal actions denoted by τ, implicit internal actions caused by synchronisation or movement. So, in order for the environ to know whether to propagate a clock tick to the process, it must first probe it to find out whether such a high priority action is pending. Clock ticks may also be prevented by the Δ and $\Delta_s igma$ constructs, so these must also be checked for.

Both can actually be achieved in one transaction by making the probe the clock tick. The clock tick is sent down the process hierarchy until it reaches a point at which a decision can be made as to whether the tick should occur or not. If the tick can occur, it is propagated back up the hierarchy, eventually stopping when it reaches its host environ

[6]The π calculus [6] is an obvious example of such behaviour, which goes to the extreme of not only allowing data to be transferred but also references to channels which can then later be used in the language constructs. This, in essence, provides the form of mobility present in the π calculus.

again. The host environ can be determined by the set of clocks associated with each environ, which is also used to calculate the signals to be propagated initially. If the clock is not allowed to tick, then the actual action performed is sent instead.

This algorithm is best explained by a couple of prototypical examples. First, consider

$$m[a.\mathbf{0} + b.\mathbf{0}]^{\Omega}_{\{\sigma\}}$$

where the process inside m has no τ actions, synchronisations, mobility or clock stop operators, and thus clearly allows the clock σ to tick. The environ m iterates over its set of clocks (here just σ), and sends a tick from each to its process, $a.\mathbf{0} + b.\mathbf{0}$.

This process is realised by an instance of the `Sum` class, which composes the two processes together. A clock can only tick over the summation operator if it can tick over both sides, so the result from this instance is simply the result of combining the return value from probing each of the constituent processes.

Both $a.\mathbf{0}$ and $b.\mathbf{0}$ are implemented using instances of the `Prefix` class, which composes a `Channel`[7] or `Tau` instance (unified by the `Action` class) with another instance of a `Process` subclass. In determining whether a clock can tick, it first checks that the action is a channel rather than a `Tau` instance (which would pre-empt the clock), and then probes the `Process` instance. In both these simple cases, this is an instance of `Nil`, which allows clock ticks.

Having determined that the clock may tick, each nested call returns with the σ clock tick, thus propagating it up to the original call in the environ m. Having seen how this operates for a process that can tick, it is simple to see how it differs when something prevents the clock from ticking. If any part of the query returns something other than a clock tick, this will be propagated upwards in preference.

Consider what happens if $a.\mathbf{0}$ is changed to $\tau.\mathbf{0}$. The left-hand side of the summation will receive the τ action from the `Prefix` instance, which then takes priority over the σ from the right-hand side and is propagated to the environ, m. This is the case in any situation where the σ is required to compete against an action, a τ or a mobility primitive. The clock stop operators behave slightly differently in that they don't replace the action, but instead mark the σ action as *stopped*.

Note that a similar method of determining the presence of clock ticks must take place to handle the `STimeout` and `FTimeout` classes. Both sides of the timeout are inspected, and behaviour determined as follows:

1. If the left-hand side can perform a high-priority action, it will be allowed to proceed and the right-hand side need not be considered.

2. Otherwise, the possible actions include unpaired actions (such as a and b) and clock ticks (both from the clock involved in the timeout and from other clocks), one of which is chosen to be performed.

3. Once the chosen action has been performed, the timeout instance will be replaced as appropriate (see section 2).

[7]An abstract class, instances of which are provided by the channel architecture described in 3.1

3.3 Structural Changes

The `Environ` class also places a central role in providing system structure. In section 2.3, we described how processes are organised into environs and the way communication is limited to its bounds. Within DynamiTE, one possible use of environs is to map them to physical or virtual hosts. While a simple testing solution can execute the entire system on a single platform, environs provide a natural form of process distribution which can be leveraged by the framework.

This does however give the initial impression that structural mobility will become very inefficient, if hosts are expected to interact to determine the feasibility of a move and then actually change position during execution. In reality, these issues are minimal. An inward movement is always in relation to a sibling, while an outward movement concerns some parent environ. As the structure of environs is expected to closely match the actual physical structure of the hosts, such interactions should be relatively low cost to perform. Also, a structural movement does not change the contents of the moving environ, only its context. Thus, only later communication with surrounding environs is affected. For example, it may have been able to see a sibling environ before the movement, but is now inside this environ and can receive clock ticks emitted by it.

If hosts do not physically move, then what is the point in allowing such structural changes? The change in clock signalling just mentioned is one effect. In addition, we also make provision for contextual data to be stored at the environ level, in addition to that stored local to a particular thread, and transferred via channels. This gives additional purpose to the use of structural mobility and process migration, which we describe next.

3.4 Migration

The final aspect of DynamiTE that we describe here is the migration of a process from one environ to another, which occurs both as a result of using one of the process mobility operators and from the behaviour of $\circledast$. This is perhaps one of the most interesting aspects, as it represents the movement of code from one environment to another, possibly located in a different physical location.

Migrating an active process is not a simple operation. Not only must any remaining code to be executed be transferred, but any local data must also migrate. TNT does allow us to achieve a significant amount of simplification here. The transferred process is already separated from other code within the system by virtue of the moving process being in the form of a `Prefix` instance. When the action is matched to the one used for the mobility operation, the `Process` instance is transferred to its new location. There is no necessity to deal with code that is currently being executed.

As with concurrency and channel operation, how movement is achieved is designed to be flexible, with provision being made for distribution and code migration to be implemented in different ways. One of the most obvious ways is to serialise the `Process` instance and reconstitute it at its destination. Migrating a process should then just be a case of transmitting the serialised instance, followed by any local data, and beginning execution at the destination. However, this is one area in which we expect further study of the existing literature to enlighten us with more sophisticated ways of achieving such migration.

4. RELATED WORK

There has already been a significant body of research into providing concurrent frameworks, including those based on process calculi. However, we believe our work to be novel in approaching the implementation of both global discrete time, via clock signalling, and mobility.

The π calculus has been the subject of much of this work, primarily due to its status as the most prevalant mobile process calculus. Obliq [4] and Pict [8] are both programming languages with semantics founded in the π calculus, while Nomadic Pict [10] takes this further, introducing distribution not usually present in the π calculus. Within research related to the ambient calculus, a machine framework (PAN citesangiorgi:safeambientsmachine) has been developed and implemented. Process calculi, such as the Seal calculus [9] have also been developed specifically to provide a formal framework for a distributed implementation.

5. CONCLUSIONS AND FUTURE WORK

To conclude, we have presented the overall structure of the DynamiTE framework for concurrent systems, with particular note to its more interesting aspects involving the use of signalling (via clock ticks) and process migration. We have also outlined its underlying theoretical basis in the form of the process calculus TNT, further details of which are provided in the cited references.

We believe that the framework provides a unique way of developing concurrent systems. It provides features which have already proved advantageous in a theoretical setting, such as the n-ary process synchronisation mechanism described in 2.2. The existence of a formal theory for DynamiTE's behaviour gives many advantages over more ad-hoc approaches, allowing the underlying mechanisms to be rigorously examined before being applied to the implementation. For example, the equivalence of two processes may be established clearly and unambiguously in the underlying process calculus and then used to compare the implementation of a process to its specification.

The DynamiTE framework is still in heavy development. At its lowest level, it provides a means of simulating the operations of the TNT process calculus, allowing them to be more clearly understood. In application, it can provide a useful mechanism for structuring concurrent programs, clearly dividing internal behaviour and interprocess communication. The presence of signalling and code migration also means that fairly complex concepts can be leveraged by the programmer in the simple manner provided by the framework.

There are still areas we wish to explore in the future. One such proposition is the addition of data to the clock signals, allowing them not only to act as phasing signals but also as a mechanism for broadcast data. It would also be interesting to further expand on the plugin frameworks mentioned, by providing more complex implementations such as interprocess communication via web services.

Acknowledgements

This work is supported by a doctoral training award from the Engineering and Physical Sciences Research Council (EPSRC).

6. REFERENCES

[1] L. Cardelli and A. D. Gordon. Mobile Ambients. In *Proc. of the 1st Intl. Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[2] A. Hughes. Nomadic Time (Extended Abstract). In R. Schmidt and G. Struth, editors, *Proc. of the PhD Programme at Relational Methods in Computer Science/Applications of Kleene Algebra (RelMiCS/AKA) 2006*, number CS-06-09 in University of Sheffield Technical Reports, pages 60–64, 2006.

[3] A. Hughes. Timed Mobile Systems. Technical Report CS-07-09, University of Sheffield, 2007.

[4] M. Merro, J. Kleist, and U. Nestmann. Mobile Objects as Mobile Processes. *Information and Computation*, 177:195–241, 2002.

[5] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[6] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-86, University of Edinburgh, June 1989.

[7] B. Norton, G. Lüttgen, and M. Mendler. A Compositional Semantic Theory for Synchronous Component-Based Design. In *Proc. of the 14th Intl. Conference on Concurrency Theory (CONCUR '03)*, number 2761 in LNCS, pages 461–476. Springer, 2003.

[8] D. N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, The University of Edinburgh, 1996.

[9] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Proc. of the ICCL '98 Workshop on Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77. Springer, 1999.

[10] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, The University of Cambridge, Mar. 2000.