Abstract

An operational formal specification of the Tramel system is presented. Tramel is used by NASA’s Jet Propulsion Laboratory to support asynchronous inter-task communication of distributed software across varying architectures and operating systems. Security analysis of communications between non-Tramel programs and Tramel is explored using an operational trace-based specification model.

1. Introduction

Tramel, the Task Remote Asynchronous Message Exchange Layer, was created by the Flight System Testbed group at NASA’s Jet Propulsion Laboratory, California Institute of Technology [10,1,2]. Tramel supports the inter-task communication of software distributed across many architectures with potential communication rates across interplanetary distances. The architecture was developed to support rapid prototyping of planetary spacecraft systems. The goals of the designers include the properties of robustness and reliability. This paper formally analyzes security properties of communications between non-Tramel programs and the Tramel system.

This formal analysis is based on an operational formal method that centers on capturing the externally visible behavior of a concurrent system [7]. The operational formal method contains both a specification and verification component, is both modular and fully abstract, and is based solely on first-order logic. It has been successfully applied to several software systems found in the industrial domain [8,9] and the tabular specification notation is a natural origin for test case generation. The operational formal method has been shown to be relatively easy to teach; it is currently taught in several undergraduate core curriculum courses [6].

An overview of the Tramel system is presented in section 2. An outline of the specification model of the operational formal method is presented in section 3. The specifications of Tramel nodes are given in section 4. A formal analysis of security properties for communications between non-Tramel programs and the Tramel system is presented in section 5. Section 6 contains a comparison with other trace-based specification notations. Lastly, a summary is presented in section 7.

2. Tramel Overview

A goal of the Flight System Testbed (FST) is to generalize and optimize system-level spacecraft interfaces to support the rapid prototyping and integration testing of spacecraft under development. In terms of time, monetary value, and quality assurance, it is extremely advantageous to begin testing a developing spacecraft with software simulations of the proposed physical product. As the software simulations become replaced with actual hardware or flight software, the development environment of FST guarantees that this replacement will be unnoticed by the users of the FST. This guarantee of incremental development and smooth integration of subsystems is mainly possible due to the use of a media-independent message passing system for subsystem communication called Tramel. All subsystems, whether simulated or actual, share the common interface of Tramel messages. Future work of the FST includes the formal specification of the Tramel messaging functions.

In the Tramel system, the general term node, is used to denote a task, thread, or process. A set of nodes that can exchange messages defines a message space. Each message space is comprised of at least one region and each region is controlled by a registrar that is responsible for monitoring the failure status of all nodes and propagating message space configuration changes. These changes can be of the following forms:

1. registration of a new node of a specified name
2. termination of a specified node
3. notification of a new subscription to a specified subject
4. notification that a subscription to a specified subject has terminated

Any one of these changes is propagated to all other nodes in that region and to the other registrars located in this message space.

The Tramel application programming model is based on event-driven processing; an event loop cycles until the detection of a system event causes the execution of the corresponding application code. The list of potential system events includes receipt of an original or reply message or the expiration of a time limit corresponding to the expected receipt of a reply message. Nodes can either publish messages on a particular subject or send private messages to other nodes.

A small example of Tramel code which is taken from the performance benchmarking programs that are contained within the Tramel distribution package follows. This c-style code fragment highlights a function that is called upon receipt of a message from another Tramel node and causes this node to send an acknowledgement of the receipt and terminate.

```c
static Tr_Outcome handleTestR( Tr_NodeState node, Tr_NodeId id, void *userData) {
    Tr_Outcome outcome;
    int subject, size=0;
    outcome = Tr_sendMsg( node, id, subject, NULL, size, NONE );
    if ( outcome != Succeeded )
        return outcome;
    return Tr_shutDownNode( node );
}
```

3. Specification Model

The specification of a node consists of a collection of the externally visible “action” specifications of the node. These externally visible actions constitute the node’s externally visible behavior. An action represents an interaction between this node and the other nodes of the concurrent program. The specification of an action describes when that action may occur in terms of the current value of the node trace and the effect of the occurrence of this action on the node trace by the addition of another element to the trace sequence. These specifications do not contain any information concerning the possible behavior of other nodes in the system. Therefore, these specifications are modular: a modification of the implementation of any one node would not cause a change in the specifications of the other nodes in the program.

When defining a trace-based semantics that is based on channel traces, some authors have resorted to using temporal logic instead of first-order logic to overcome incompleteness. However, incompleteness is a characteristic of channel traces, not of trace-based semantics in general. The use of node traces as presented in this specification model has been shown to be (relatively) complete and uses only first-order logic. Also, this node trace model is a full abstraction of the operational model in that it contains sufficient information to prove essential properties of the model without containing information concerning the internals of a process. Lastly, a corresponding verification model exists for either deriving the implementation code or proving that a particular implementation satisfies its specification [7].

3.1 Trace Notation

The variable \( h \) is used to represent the node trace sequence. A subscript \( i \) is used to identify the node to which this trace sequence is associated. Generally, node trace sequences are initialized to the empty sequence \( \epsilon \). Trace operations are defined as follows.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(#h)</td>
<td>length of ( h )</td>
</tr>
<tr>
<td>(h' \wedge h)</td>
<td>(h') concatenated to ( h)</td>
</tr>
<tr>
<td>(h')</td>
<td>reverse of ( h)</td>
</tr>
<tr>
<td>(h[k])</td>
<td>(k^{th}) element of ( h)</td>
</tr>
<tr>
<td>(h/i)</td>
<td>restriction of the elements of ( h) to those elements involving ( N_i)</td>
</tr>
<tr>
<td>(h_{ext})</td>
<td>restriction of the elements of ( h) to those elements involving nodes external to the program ( P)</td>
</tr>
</tbody>
</table>

3.2 Node Specifications

Formally, an individual node execution is represented by a sequence of the form:

\[ h_i \rightarrow _{\alpha 1} h'_i \rightarrow _{\alpha 2} h''_i \rightarrow _{\alpha 1} \ldots \]

where \( h_i \) is the \( i^{th}\) node trace sequence and \( \alpha_k \) is an element from the set of externally visible actions of the node \( N_i\). All possible node trace sequences for the node \( N_i\) define the behavioral model of the node.

Typically, actions are some form of communication: (a)synchronous message passing or shared variable communication. An action is defined in terms of a change of value of the corresponding node trace. A
specification of an action, which includes the current and extended node trace, will be written in two parts: an enabling and an effect. The effect specifies the change in value of the node trace by concatenating another element(s) to the trace. The enabling part specifies when this action may occur as a guard to the trace update.

To write the specification of a node, the actions which comprise the behavior of that node are determined. For each action, all possible enabling conditions and their effect on the node trace are listed. Tabular notation is used to represent the node specification. Each table is a visual representation of the potential changes of a node trace. Specification tables have the following form.

<table>
<thead>
<tr>
<th>N_i</th>
<th>action</th>
<th>enable</th>
<th>effect</th>
</tr>
</thead>
</table>

The enable condition is typically written in terms of the last element of the node trace sequence at any point in time, namely \( h_i[0] \). Therefore, an individual element listed as the enable condition means that the last element of the current sequence is asserted to be of this form. Similarly, the effect lists the element which is added to the node trace sequence for a particular action. Each node participates in an initial action which initializes a node’s trace sequence. This action will be omitted from the table when the node trace sequence is initialized to \( \epsilon \).

A sequence element either has the form, \(<!, A, j, k, X>, \) or \(<?, A, j, X>, \) where the first component represents the type of communication (either receive (?) or send (!)), \( A \) represents the externally visible action, and \( j \) names the process to which this element is associated. If the element represents a send, a \( k \) component is included when the name of the process that receives this message is known. Lastly, \( X \) represents the data sent or received. If \( X \) is listed as \( \lambda \), then it is the empty data value. To conserve table space, the \( j \) component is omitted from the elements when its value is readily apparent. For example, using the following specification table, the \( j \) component is omitted because it is known to have the value \( i \) which is given in the name of the table.

The notation, \( N_i \; sat \; T_j \), indicates that \( N_i \)’s node trace sequence values satisfy the predicate \( t_i \) composed of \( N_i \)’s specification table entries (\( T_i \)) at any point during the execution of \( N_i \). The predicate \( t_i \) is constructed as follows: \( \wedge_j \; enable_j \Rightarrow effect_j \) where \( j \) ranges over the rows of node \( N_i \)’s specification table, \( T_i \).

Given the following specification table for a node \( N_i \) which receives a value from another node in the program and forwards that value to the node \( N_k \):

<table>
<thead>
<tr>
<th>( N_i )</th>
<th>Receive</th>
<th>Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon \lor &lt;!,Send,k,X&gt; )</td>
<td>&lt;?,Receive,X&gt;</td>
<td>&lt;!,Send,k,X&gt;</td>
</tr>
</tbody>
</table>

the corresponding predicate of the specification table is as follows.

\[
(h_i = \epsilon \lor h_i[0] = <!,Send,k,X>) \Rightarrow h_i = h_i \land <?,Receive,X> \land h_i[0] = <?,Receive,X> \Rightarrow h_i = h_i \land <!,Send,k,X>
\]

3.3 (De)Compositionality

The individual node traces, \( h_i \), are combined to determine a concurrent program trace, \( h \), representing the externally visible behavior of the concurrent system consisting of the composition of nodes. The combination of the node traces must obey compatibility, e.g.: if \( N_i \) sends a particular value to \( N_j \), then this value must be recorded on both sequences in an appropriate location. Essentially, compatibility (or mutual consistency) ensures that the set of node trace sequences under consideration could arise during the execution of the corresponding set of nodes. In fact, any program trace sequence can be used to define individual node traces by projecting out and concatenating each element of \( h \) that involves that particular node.

It is this observation that forms the basis of the rule for composing the individual node specifications into a concurrent program specification, where \( P \) is composed of nodes \([N_1 \parallel N_2 \parallel ... \parallel N_n]\).

\[
N_i \; sat \; T_i, \; i = 1, ... , n \exists h, [ h[1] = h_1 \land t_1 \land h[2] = h_2 \land t_2 \land ... \land h[n] = h_n \land t_n ]
\]

\( P \; sat \; T \)

The following inference rule can be used to weaken the process specification, \( t_i \).

\[
N_i \; sat \; T_i, \; t_i \Rightarrow t_k
\]

\( N_i \; sat \; T_k \)

This concurrent program may also be a subsystem of a larger concurrent program. If so, the program trace sequence can be modified to represent the externally visible actions of the program by eliminating all actions involving pairs of nodes that are components of this program. These nodes must be eliminated since the program sequence may only contain actions involving an internal node and a node external to this program.

Program trace sequences consisting only of externally visible actions of the program can be decomposed into
partial node trace sequences. The elements of the original sequences are partitioned and associated with individual node traces on the basis of the externally visible actions of a node. The original ordering of the elements in the program trace must be preserved in the node trace. During the process of iterative refinement stages that continually add implementation details, additional nodes will be added to the concurrent program. Each additional node will have a set of externally visible actions which are disjoint from the concurrent program actions. Those nodes that inherit externally visible actions from the concurrent program actions may also increase their set of actions by adding externally visible actions involving this node and one of the newly created nodes.

\[ P \text{ sat } T \]
\[ \exists h_i \ [ h_i = h_i^{\text{ext}} \land i' \Rightarrow t_i] \]
\[ N_i \text{ sat } t_i \]

The subscript \( i \) ranges over all nodes defined in the program, \( P \). The notation \( i' \) defines the predicate constructed from the specification table \( T \) of \( P \) consisting only of those externally visible actions of the node \( N_i \). Any references to actions in the remaining enabling conditions of the specification table \( T \) of \( P \) which are not shared by \( P \) and \( N_i \) must be eliminated since they can not appear in node \( N_i \)'s specification table, \( T_i \). Finally, we can not assume that the enabling conditions as denoted in the predicate \( i' \) refer to the last element of the node trace sequence, \( h_i \). However, they do refer to a recent prior element; so the enabling condition listed in the \( k^{\text{th}} \) row of \( P \)'s specification table, \( \text{enable}_{k} \), must be modified as follows: ( \( \exists j. 0 \leq j < \#h_i. \text{Min}(j) \land h_i'[j] = \text{enable}_{k} \)).

This inference rule ensures that the order of elements in the program trace sequence \( h \) is preserved in the individual node trace sequences \( h_i \) for those externally visible actions shared by both the node \( N_i \) and the program \( P \). The rule also ensures that the assertion \( i' \) corresponding to the portion of \( P \)'s specification table \( T \) of the shared externally visible action of \( N_i \) and \( P \) implies the assertion \( t_i \) corresponding to the new specification table \( T_i \).

4. Tramel Specification

4.1 Registrars

Each region within a message space is administered by a registrar which is responsible for message space

<table>
<thead>
<tr>
<th>Table 1. Registrar specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R_i</strong></td>
</tr>
<tr>
<td>&lt;?, run, &lt;N_o,net_o&gt;&gt;</td>
</tr>
</tbody>
</table>
| | \[\forall i. 1 \leq i \leq n . \]
| | \[<!,regestr,N_o, node_o,net_o>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j . \]
| | \[<!,regestr,R_k, N_o, node_o,net_o>\] |
| | \[\forall k \in IS \land \forall i. 1 \leq i \leq n . \]
| | \[<!,regestr,N_i, subjct_o, net_o>>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j \land \forall i \in IS . \]
| | \[<!,regestr,R_k, N_o, subjct_o>>\] |

<table>
<thead>
<tr>
<th><strong>R_i</strong></th>
<th><strong>requestMsg (rqmsg)</strong></th>
<th><strong>unSubscribe (unsub)</strong></th>
</tr>
</thead>
</table>
| <?,rqmsg, <N_o,subjct_o>> | \[\forall i. 1 \leq i \leq n \land i \neq o . \]
| | \[<!,rqmsg,N_o, subjct_o>>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j . \]
| | \[<!,rqmsg,R_k, N_o, subjct_o>>\] |

<table>
<thead>
<tr>
<th><strong>R_i</strong></th>
<th><strong>run</strong></th>
<th><strong>shutdownNode (sdn)</strong></th>
</tr>
</thead>
</table>
| <?,sdn, <N_o,net_o>> | \[\forall i. 1 \leq i \leq n \land i \neq o . \]
| | \[<!,sdn, N_o, node_o,net_o>>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j . \]
| | \[<!,sdn,R_k, N_o, node_o,net_o>>\] |

<table>
<thead>
<tr>
<th><strong>R_i</strong></th>
<th><strong>requestMsg (rqmsg)</strong></th>
<th><strong>unSubscribe (unsub)</strong></th>
</tr>
</thead>
</table>
| <?,rqmsg, <N_o,subjct_o>> | \[\forall i. 1 \leq i \leq n \land i \neq o . \]
| | \[<!,rqmsg,N_o, subjct_o>>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j . \]
| | \[<!,rqmsg,R_k, N_o, subjct_o>>\] |

<table>
<thead>
<tr>
<th><strong>R_i</strong></th>
<th><strong>run</strong></th>
<th><strong>shutdownNode (sdn)</strong></th>
</tr>
</thead>
</table>
| <?,sdn, <N_o,net_o>> | \[\forall i. 1 \leq i \leq n \land i \neq o . \]
| | \[<!,sdn, N_o, node_o,net_o>>\] |
| | \[\forall k. 1 \leq k \leq m \land k \neq j . \]
| | \[<!,sdn,R_k, N_o, node_o,net_o>>\] |
configuration changes: node registrations, terminations, subscriptions, and unsubscriptions. The following specification tables list the externally visible actions for the registrar along with their effect and enabling conditions.

Table 1 lists all the actions associated with a registrar. Once the registrar receives a request from a particular node for either registration (run), termination (shutDownNode), subscription (requestMsg), or unsubscription (unSubscribe), the registrar must broadcast this information to all other nodes in the region and to all other registrars in the message space. The action of registration not only registers the node in the region but also automatically subscribes the node to a particular set of subjects; this set is denoted in the specification table as IS.

4.2 Nodes

A node can send a message to a specified node, refer a message it has received to some other specified node, subscribe to all published messages concerning a particular subject, publish a message, and reply to a message.

Table 2 lists the actions corresponding to the act of registration and sending messages. Before any messages can be sent, the node N_j must request a node identification number from the region’s registrar by notifying the registrar that this node wishes to “run” (action regstr). If a node receives notification that a new node has been registered, then each node in the message space sends its name, identification number, and network identification to the newly registered node (second row). The last row of the table reflects that a node can send a private message to any other registered node (sendMsg).

Table 3 lists the actions of receiving and forwarding a message. A node can receive a message from any other node in the message space either through private communication or through publication and can potentially respond to that message (recve). The referMsg action represents the receipt of a message that is ultimately forwarded to another node in the message space.

Table 4 lists the actions of publication and termination. The set, S, represents all nodes which have subscribed to a particular subject. The publish action causes the published message to be broadcast to all nodes in this set. Notice, the node N_j is not required to have subscribed to the subject of the published message. Termination of a node is accomplished by sending notification to the registrar of the impending action (shutDownNode).

Table 5 finishes the specification with the actions of subscription and unsubscription. Any node that has previously registered can subscribe (requestMsg)/unsubscribe(unSubscribe) to a particular subject by notifying the registrar.

5. Communications External to Tramel

A tunnel, the name given to the ability for non-Tramel programs to exchange Tramel messages, is a simple Tramel node that has one additional non-Tramel communication channel. The tunnel can run as either a client or a server. The tunnel registers as a Tramel node when the socket connection is made and unregisters when the socket connection is broken. A tunnel node can subscribe to messages of a particular subject, unsubscribe from subjects, and terminate. The tunnel is provided with an initial list of subscriptions to subjects that a non-Tramel program may subscribe. Each message that is sent to the tunnel from the non-Tramel program is published as a Tramel message on the indicated subject.
and each message received by the tunnel from Tramel is written on the socket connection. The messages received from the non-Tramel program have a given format that is parsed by the tunnel and reformatted into Tramel messages. The code that creates a tunnel and was used to verify the specification of a tunnel is found in the Appendix. The specification for a tunnel node contains some of the externally visible actions of a normal Tramel node (regstr, referMsg) and two new actions. These two actions, getMsg and putMsg, represent the communications between the tunnel and the non-Tramel program. The specification table for these additional actions follows (Table 5).

When the tunnel is established, it registers with the system in the same manner as all other Tramel nodes. The tunnel is instructed to receive messages from the non-Tramel program when given a getMsg action and to send any messages it receives from Tramel nodes to the non-Tramel program by means of a putMsg action. A getMsg action instructs the tunnel to either subscribe to a particular subject, unsubscribe a subject, publish a message to all nodes subscribed to a particular subject, or terminate. Any message received by the tunnel, by publication or private communication, is forwarded to the non-Tramel program using a putMsg action. If the tunnel is unable to forward the message, it terminates.

Using the specification tables for a tunnel, we can establish some necessary security properties for communications that are external to Tramel. Let \( h \) represent the node trace sequence for the tunnel.

### Table 4. Node specification (cont)

<table>
<thead>
<tr>
<th>( N_j )</th>
<th>publish (pub)</th>
<th>shutDownNode (sdn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall k \in S. \exists i. 0 \leq i &lt; #h_j. \max(i) \land h_j[i] = &lt;\text{regstr}, &lt;N_j,node_j,net_j&gt;&gt; \land \text{subject} \leftrightarrow S \land \forall l. i &lt; #h_j. h_j[l] \neq &lt;\text{sdn}, R_k, &lt;T_j,node_j,net_j&gt;&gt; \lor \forall l. i &lt; #h_j. h_j[l] \neq &lt;\text{unsub}, R_k, &lt;T_j,node_j,net_j&gt;&gt; )</td>
<td>( \forall k \in S \land k \neq j \land &lt;\text{pub}, N_k, &lt;\text{subject},msg&gt;&gt; \land &lt;\text{regstr}, &lt;T_j,node_j,net_j&gt;&gt; \land \forall l. i &lt; #h_j. h_j[l] \neq &lt;\text{sdn}, R_k, &lt;T_j,node_j,net_j&gt;&gt; \lor &lt;\text{unsub}, R_k, &lt;T_j,node_j,net_j&gt;&gt; )</td>
<td>( &lt;\text{sdn}, R_k, &lt;N_j,node_j,net_j&gt;&gt; )</td>
</tr>
</tbody>
</table>

#### Table 5. Node specification (cont)

<table>
<thead>
<tr>
<th>( N_j )</th>
<th>requestMsg (rqmsg)</th>
<th>unSubscribe (unsub)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists i. 0 \leq i &lt; #h_j. \max(i) \land h_j[i] = &lt;\text{rqmsg}, &lt;N_j,node_j,net_j&gt;&gt; \land \forall l. i &lt; #h_j. h_j[l] \neq &lt;\text{sdn}, R_k, &lt;N_j,node_j,net_j&gt;&gt; )</td>
<td>&lt;\text{rqmsg}, R_k, &lt;N_j,subject&gt;&gt;</td>
<td>&lt;\text{unsub}, R_k, &lt;N_j,subject&gt;&gt;</td>
</tr>
</tbody>
</table>

1. \( \forall i. 0 \leq i < \#h, h_i[i] \neq <\text{sdmsg}, N_k, <T_j,subject,msg>> \Rightarrow \exists k. 0 \leq k < \#h, (\max(k) \land h[k] = <\text{regstr}, <T_j,node_j,net_j>> \land k < i \land \forall l. k < l < i. h[l] \neq <\text{sdn}, R_k, <T_j,node_j,net_j>> \land h[l] \neq <\text{unsub}, R_k, <T_j,subject>> \) |

The first property states that non-Tramel programs are unable to send private messages to Tramel nodes. However, it should be noted that Tramel nodes can send private messages to non-Tramel programs; this fact comes from the specification table given for tunnels. So, it is possible for a subversive Tramel node to disclose any information of the Tramel program to outsiders.

2. \( \forall i. 0 \leq i < \#h, h_i[i] = <\text{putmsg}, N_j, <\text{subject},msg>> \Rightarrow \exists k. 0 < k < \#h, (\max(k) \land h[k] = <\text{regstr}, <T_j,node_j,net_j>> \land k < i \land \forall l. k < l < i. h[l] \neq <\text{sdn}, R_k, <T_j,node_j,net_j>> \land h[l] \neq <\text{unsub}, R_k, <T_j,subject>> \) |

The second property states that non-Tramel programs can only receive messages from a tunnel that is registered as a Tramel node.

3. \( \forall i. 0 \leq i < \#h, h_i[i] = <\text{pub}, N_k, <\text{subject},msg>> \Rightarrow \exists k. 0 \leq k < \#h, (\max(k) \land h[k] = <\text{regstr}, <T_j,node_j,net_j>> \land k < i \land \forall l. k < l < i. h[l] \neq <\text{sdn}, R_k, <T_j,node_j,net_j>> \land h[l] \neq <\text{unsub}, R_k, <T_j,subject>> \) |

The third property states that non-Tramel programs can only publish messages by means of a tunnel that is registered as a Tramel node.
Table 6. Tunnel specification

<table>
<thead>
<tr>
<th>Tunnel Specification</th>
<th>getMsg</th>
<th>putMsg</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;?,getmsg,&lt;subject,msg&gt;&gt;</code></td>
<td><code>&lt;!,rqmsg,R_k,&lt;T_j,subject&gt;&gt;</code></td>
<td><code>&lt;!,pub,N_k,&lt;subject,msg&gt;&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>∨ ( ∃ i.0 ≤ i &lt; #h_i.h_i[i]=&lt;!,rqmsg,R_k,&lt;T_j,subject&gt;&gt; ⇒ &lt;!,unsub,R_k,&lt;T_j,subject&gt;&gt; )</code></td>
<td><code>∨ ( ∀ k ∈ S. ∃ i.0 ≤ i &lt; #h_i.h_i[i]=&lt;!,sdn,R_k,&lt;T_j,node_j,net_j&gt;&gt; ∧ max(i) ∧ subject ↔ S ∧ ∀ l.i &lt; l &lt; #h_i.h_i[i] ≠ &lt;!,unsub,R_k,&lt;T_j,sub&gt;&gt; ∨ h_i[i] ≠ &lt;!,sdn,R_k,&lt;T_j,node_j,net_j&gt;&gt; ) ⇒ (k ≠ j ∨ &lt;!,pub,N_k,&lt;subject,msg&gt;&gt; ) )</code></td>
</tr>
<tr>
<td><code>&lt;?,sdmsg,N_k,subject,msg&gt;&gt;</code></td>
<td><code>&lt;!,putmsg,N_k,&lt;subject,msg&gt;&gt;</code></td>
<td><code>&lt;!,sdn,R_k,&lt;T_j,node_j,net_j&gt;&gt;</code></td>
</tr>
</tbody>
</table>

The fourth property states that non-Tramel programs can only subscribe to subjects by means of a tunnel that is registered as a Tramel node.

\[ ∀ i.0 ≤ i < #h_i. ( h_i[i]=<!,rqmsg,R_k,<T_j,subject>> ⇒ ∃ k.0 ≤ k < #h_i.(max(k) ∧ h[k]=<!,regstr,<T_j,node_j,net_j>>) ∧ k < i ∧ ∀ l.i < l < #h_i.h_i[l] ≠ <!,sdn,R_k,<T_j,node_j,net_j>> ) ) \]

The last property states that non-Tramel programs can only receive messages from other Tramel nodes by means of a tunnel.

\[ (∀ i.0 ≤ i < #h_i. ( h_i[i]=<!,rqmsg,R_k,<T_j,subject>> ⇒ ∃ k.0 ≤ k < #h_i.(max(k) ∧ h[k]=<!,regstr,<T_j,node_j,net_j>>) ∧ k < i ∧ ∀ l.i < l < #h_i.h_i[l] ≠ <!,sdn,R_k,<T_j,node_j,net_j>> ) ) \]

6. Comparisons of the Trace-based Specification Model

Leveson’s AND/OR tables [3] are a tabular representation of the disjunctive normal form of a Boolean expression. Specifications are written in RSML, Requirements State Machine Language, which uses state transitions to capture the functional behavior of the software. Transition conditions are translated into AND/OR tables. Functions were chosen to guarantee completeness and consistency and tools are provided to support this analysis. However, this characteristic of the functions chosen forces specifications to be deterministic. AND/OR tables can become difficult to use when transition conditions are written using first-order logic operators such as implication and equivalence. It is clear that the implication operator is used frequently in the tabular form of the presented operational specification model. The application of AND/OR tables focuses only on the requirements specification stage of the software life cycle whereas the operational formal method, of which the specification model presented is a component, can be applied throughout the entire software engineering life cycle.

Parnas proposes the use of tabular representations of relations [5] for creating program specifications which describe a set of state sequences in a finite state machine. A variety of table formats are used to define the mathematical functions and relations that capture the state transitions of the software system. The table format is purported to aid in the understanding of multidimensional expressions and simplify the inspection of requirements specification documents. Numerous rules are provided for changing table formats in order to provide the most readable function definition. The main emphasis for providing a tabular notation is to support the construction of readable systems requirements documents. The operational formal presented here supports the refinement of the specifications into code as well as the rules for verifying that a particular implementation satisfies its specification. It is unclear how Parnas’ tables are refined into implementation code.
Also, no support is given for application during the other phases of software development.

7. Summary

A specification model is presented for the Tramel system which is based on capturing the externally visible behavior of Tramel registrars and nodes and communications with non-Tramel nodes. This model can be used to aid the development of Tramel applications while providing the basis to analyze the security and safety properties of the application. The Flight System Testbed members set the formal specification of Tramel message passing as a future goal [1]. These specifications give the team members the ability to formally specify the Tramel subsystem interfaces which can be used as part of the requirements specifications of the subsystem under current development.

Acknowledgments

My sincerest gratitude goes to Brent Auernheimer of California State University who first encountered the Tramel system while supported under an Associated Western Universities (AWU) summer research fellowship at the NASA’s Jet Propulsion Laboratory, California Institute of Technology. It was from his experience that I learned of the Tramel system and the significant role that it plays in prototyping and testing potential NASA flight systems.

References


Appendix

The code for establishing a tunnel that provides non-Tramel software controlled access to a Tramel message space was written by Scott Burleigh at NASA’s Jet Propulsion Laboratory, the California Institute of Technology. The portion of the code that creates a Tramel tunnel node is listed below.

```c
static Tr_Outcome getMsg(Tr_NodeState ns, int fd, void *client_data)
{
    Tr_SockMsgHdr hdr;
    TunnelState *tunnel;
    char *content;
    tunnel = Tr_getNodeData(ns);
    switch (readFromSocket(fd, (char *) &hdr, sizeof hdr))
    {
        case 0: /* Broken connection. */
            Tr_shutDownNode(ns);
            return Succeeded;
        case -1: /* I/O error. */
            return Failed;
        hdr.length = ntohl(hdr.length);
        hdr.subject = ntohl(hdr.subject);
        if (hdr.length < TUNNEL_TERMINATE)
        {  
            fprintf(stderr,"Tunnel received invalid message length %d; quitting.
            hdr.length);
            return Failed;
        }
        switch (hdr.length)
        {
            case TUNNEL_UNSUBSCRIBE:
                Tr_unsubscribe(ns, hdr.subject);
                return Succeeded;
```
case TUNNEL_SUBSCRIBE:
    return Tr_subscribe(ns, hdr.subject);

case TUNNEL_TERMINATE:
    return Failed;

case 0:
    content = NULL;
    break;

default:
    if (hdr.length > tunnel->bufferSize)
    {
        if (tunnel->buffer)
        {
            free(tunnel->buffer);
            tunnel->buffer = NULL;
            tunnel->bufferSize = 0;
        }
        tunnel->buffer = malloc(hdr.length);
        if (tunnel->buffer = NULL)
        {
            fprintf(stderr, "Tunnel unable to allocate \n buffer for message of length %d(\%-4s); quitting: \n", hdr.length, (char *) &hdr.length);
            return Failed;
        }
        tunnel->bufferSize = hdr.length;
    }

switch(readFromSocket(fd, tunnel->buffer, hdr.length))
{
 case 0: /* Broken connection. */
    Tr_shutDownNode(ns);
    return Succeeded;
 case -1: /* I/O error. */
    return Failed;
}

content = tunnel->buffer;

return Tr_publish(ns, hdr.subject, content,
    hdr.length, NONE);   }

static Tr_Outcome putMsg(Tr_NodeState ns, Tr_Msg msg,
 void *client_data)
{
 int fd = (int) client_data;
 int Tr_SockMsgHdr hdr;
 hdr.subject = Tr_subjectOf(msg);
 hdr.subject = htonl(hdr.subject);
 contentLength = Tr_contentLengthOf(msg);
 hdr.length = htonl(contentLength);
 switch (writeToSocket(fd, (char *) &hdr, sizeof hdr))
 {
 case 0: /* Broken connection. */
    Tr_shutDownNode(ns);
    return Succeeded;
 case -1: /* I/O error. */
    return Failed;
 }

if (contentLength == 0)
{
    return Succeeded;
}

switch (writeToSocket(fd, Tr_contentOf(msg),
    contentLength))
{
 case 0: /* Broken connection. */
    Tr_shutDownNode(ns);
    return Succeeded;
 case -1: /* I/O error. */
    return Failed;
}

return Succeeded;

static Tr_Outcome initTunnel(Tr_NodeState ns,
 void *initData)
{
 Tr_Outcome outcome;
 TunnelState *tunnel;

 Tr_setNodeData(ns, initData); /* Dynamic buffer. */
 tunnel = (TunnelState *) initData;
 outcome=Tr_handleInput(ns, tunnel->fd, getMsg, NULL);
 if (outcome != Succeeded)
 {
     fprintf(stderr, "tunnel unable to handleInput\n");
     return Failed;
 }

outcome=Tr_handleMsg(ns,TR_ALL_OTHERS,
    putMsg,(void *) tunnel->fd);
 if (outcome != Succeeded)
 {
     fprintf(stderr, "tunnel unable to handleMsg\n");
     return Failed;
 }

return Succeeded;

static int tunnelServer(char *applicationName,
 char *authorityName,
 char *regionName,
 char *nodeName,
 unsigned short portNbr)
{ int dialupFd = -1;
    TunnelState tunnel = { NULL, 0, -1 }; 
    Tr_Outcome outcome; 

    while (1) 
    { 
        dialupFd = offerSocket(portNbr);
        if (dialupFd == -1) 
        { 
            fputs("tunnel can’t become server", stderr);
            break;
        }

        tunnel.fd = acceptClient(dialupFd);
        close(dialupFd);
        dialupFd = -1;
        if (tunnel.fd < 0) 
        { 
            break; 
        }

        outcome = Tr_run(applicationName, authorityName, 
        regionName, nodeName, nodeSpec, initTunnel, &tunnel, NULL);
        close(tunnel.fd);
        if (outcome != Succeeded) 
        { 
            break;
        }
    }

    if (tunnel.buffer) 
    { free(tunnel.buffer); 
    }

    if (dialupFd != -1) 
    { close(dialupFd); 
    }

    return -1;
}

static int tunnelClient( char *applicationName, char *authorityName, 
*regionName, char *nodeName, char *nodeSpec, unsigned short portNbr, 
char *hostName)
{ 
    TunnelState tunnel = { NULL, 0, -1 }; 
    Tr_Outcome outcome; 

    while (1) 
    { 
        while (1) 
        { 
            tunnel.fd = connectToSocket(hostName, portNbr);
            switch (tunnel.fd)
            { 
            case -2: /* No success yet. */
                snooze(1);
                continue;
            case -1:
                fputs("tunnel can’t become client", stderr);
                return -1;
            }
            outcome = Tr_run(applicationName, authorityName, 
            regionName, nodeName, nodeSpec, initTunnel, 
            &tunnel, NULL);
            close(tunnel.fd);
            if (outcome != Succeeded) 
            { 
                break;
            }
            }
        }

        if (tunnel.buffer) 
        { free(tunnel.buffer); 
        }

        if (dialupFd != -1) 
        { close(dialupFd); 
        }

        return -1;
    }

static int openTunnel(char *applicationName, char *authorityName, 
*regionName, char *nodeName, char *nodeSpec, int portNbr, char *hostName)
{ 
    unsigned short normalizedPort;

    signal(SIGPIPE, SIG_IGN);
    normalizedPort = portNbr;
    if (hostName)
    { 
        return tunnelClient(applicationName, authorityName, 
        regionName, nodeName, nodeSpec, normalizedPort, hostName);
    }

    return tunnelServer(applicationName, authorityName, 
    regionName, nodeName, nodeSpec, normalizedPort); 
}