Dynamic Relational Behaviour for Large-Scale Systems

Kirsten Winter[1,2] Robert Colvin[1,2] R. Geoff Dromey[1,3]

[1] The ARC Centre for Complex Systems,
[2] The University of Queensland, Brisbane, Australia
School of Information Technology and Electrical Engineering
[3] Griffith University, Brisbane, Australia

Abstract

In this paper we introduce a syntax and semantics for capturing complex relational behaviour commonly found in natural language system requirements. The syntax is an extension of Behaviour Trees*, a modelling notation used for capturing user requirements from natural language in a structured way. The underlying semantics of the extended notation is based on that of relational databases, thereby allowing the expressive power of database queries to be combined with the event- and state-based dynamic behaviour of Behavior Trees. To be a practical method for developing large-scale complex systems, the language is formal and hence supported by simulation and model checking tools.*

1. Introduction

The following natural language functional requirement from a real project (names disguised to satisfy confidentiality obligations) is typical of the complex and subtle behaviour exhibited by large-scale systems:

> *The mission system shall operate an XX as a relay platform between another XX collecting YY information and a control site for a relay XX to mission XX range of up to 5km.* (1)

With such natural language descriptions of complex behaviour there is a higher risk of ambiguity, misinterpretation, and propagation of latent defects. For example, in this 33 word sentence, after rigorous analysis and formalisation is carried out there are at least 10 major defects/issues. If we are to enhance the dependability of systems built to satisfy such requirements we need to, in the first instance, construct a formal specification that accurately and unambiguously reflects, corrects and clarifies the original intent. If, as an added benefit, the resulting formal specification can be executed, it opens up the option of using simulation and model-checking to enhance the dependability of the system.

We have demonstrated elsewhere [7, 6] that Behavior Trees offer an effective means for constructing small to medium state-based specifications in which the specification is built out of the requirements by integrating their behaviour. In parallel with these studies, in our work with industry on very large-scale systems over the last six years, the focus has been on accurately capturing and integrating requirements, but not on producing an executable specification.

The strategy we have employed in the industry work involves use of a single defining form which entails strengthening the original natural language requirements using prepositions and the six questions who, what, why, where, when, and how, in order to clarify intent and remove the risk of ambiguity. The non-executable example in Fig. 1 uses this defining form. This form is important for initially capturing requirements involving complex behaviour because it uses the same vocabulary as the original requirement and is therefore easy for stakeholders to understand and validate. This representation is, however, not directly suitable for execution.

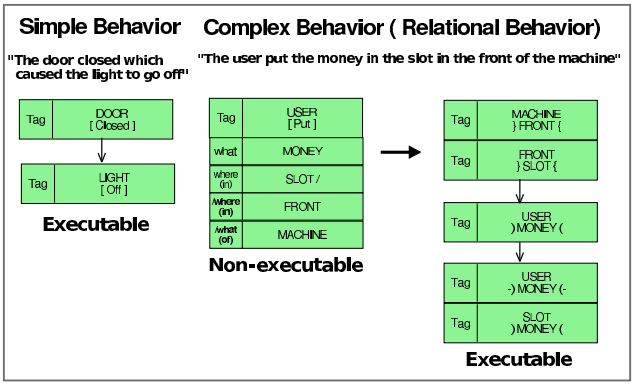

Figure 1. Formalising and comparing simple and complex (relational) behaviour.

To achieve our long-term goal of being able to simulate

1530-0803/09 $25.00 © 2009 Crown Copyright
DOI 10.1109/ASWEC.2009.41

and possibly model-check the integrated Behavior Tree representation of large-scale industry systems that exhibit complex behaviour like that shown above, we need to use a different representation. A constraint on any such model is that it should be compatible with the component-state model of Behavior Trees. In this model, basic behaviour involves components realising states, changing states, responding to events, sending and receiving messages and testing the current state of a component. More complicated behaviour is constructed by composing either sequentially, in parallel, or conditionally, basic behaviour of components.

What is fundamentally different between simple and complex behaviour? It is not simply longer sentence descriptions of behaviour. In Fig. 1, components realise states then pass control to other components that realise states. With complex behaviour things are different. The primary focus is on two things: (1) components dynamically establishing relations with other components and (2) components dynamically breaking relations with other components.

A component which, through its behaviour, forms a relation with another component, achieves a relation realisation which is analogous to a component, through its behaviour, changing state. This way of thinking about complex behaviour, or, more explicitly, *relational behaviour*, gives us a practical way forward. We can translate the non-executable representation of relational behaviour, which is important for validation, into an executable form built on making and breaking relations. In describing the relational behaviour of the user, the requirement in Fig. 1 talks not only about *dynamic* relations but also about *static* relations. The machine "has" or "contains" or ultimately forms a relationship with "front". Similarly front has a static relationship with "slot". These static relations represent preconditions for the dynamic behaviour that takes place.

In dynamic terms, the user initially has a relationship with the money. Then, when the user puts the money in the slot, the user no longer has a relationship with the money and now the slot "contains" or has a relationship with the money.

In this paper we address these issues by introducing an extension of the Behavior Trees notation to handle the translation of complex requirements in a concise way. In Sect. 2 we review Behavior Trees, and in Sect. 3 we provide an extended syntax which captures relational behaviour. In Sect. 4 we provide the formal semantics for the new notation, and discuss related work and conclude in Sections 5 and 6.

2. Background

2.1. Introduction to Behavior Trees

Behavior Trees (BTs) are a graphical notation used for capturing requirements. The core idea is that individual requirements may be modelled individually and separately from other requirements, and then *integrated* together to form a complete, structured representation of the system. For a large system, the number of individual requirements may number in the thousands, and the BT notation provides a convenient language for dealing with this complexity. We can use the following analogy: if the system is a picture, the BT process treats the requirements as pieces of a jigsaw puzzle of that image; while traditional approaches develop a new image "free-hand", based on intuition and experience gained from observing each piece of the puzzle individually. There are several benefits to the BT approach, in particular that *traceability* is maintained, so that if a stakeholder needs to know how a particular requirement (jigsaw piece) is captured in the system, it can immediately be found. The task of constructing the system may also be divided among multiple modellers, thus reducing the amount of time required for, and the cognitive overload associated with, modelling large scale systems.

Type declarations and other structural information about the system model are captured in a *Composition Tree* (CT). We do not introduce the notion of CTs here but refer the interested reader to [19].

2.2. Notation

The syntax of the BT notation comprises nodes and edges. A node can be either a state realisation, a selection, a guard, or an event. A component C has a state and may have attributes, which can be accessed using the dot notation, *C.attribute*. The node is specialized by its *behaviour*, B, which can be an identifier describing a state, an event, or a channel name, or can be an expression which refers to attributes of the component. More precisely, as shown in Fig. 2, a node can be

(a) a state realisation, modelling C being in a state if B is a state name, or updating C's attribute if B is an update expression over the attribute;

(b) a selection (or condition) on C's state if B is a state name, or a selection on the state of one of C's attributes if B is an expression over the attribute; in both cases, the control flow terminates if the condition is not satisfied;

(c) a guard; the control flow can pass the guard when C is in state B, otherwise it is blocked until the state realisation occurs;

(d-e) an internal event modelling communication and data flow between components within the system, where *B* specifies an event; the control flow can pass the internal input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs;

(f-g) an external event modelling communication and data flow between the system and its environment, where *B* specifies an event; the control flow can pass the external input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs.

A node may also be labeled with one or more *flags*, used to indicate control flow. Flags refer to a *matching node*, which is a node of the same type and with the same component name and behaviour. A flag can specify

1. a reversion ^ in the case where the node is a leaf node, indicating that the control flow loops back to the matching node,
2. a reference node =>, indicating that the flow continues from the matching node,
3. a synchronization point =, where the control flow waits until all other threads with a matching synchronization point have reached the synchronization point.
4. killing of a thread −−, which kills the thread that starts with the matching node.

Each node has also a *tag* which allows the user to relate the BT nodes to the original requirements specification. This tag has no consequences for the semantics but is used for traceability.

The control flow of the system is modelled by either a normal or a branching edge. Fig. 3 shows the different types of normal edges. As an example, we use state realisation nodes in the figure, however, there are no restrictions on the node types. A normal arrowed edge models sequential flow between two steps (Fig. 3a). If two nodes are connected by a line without an arrow head (Fig. 3b), or linked

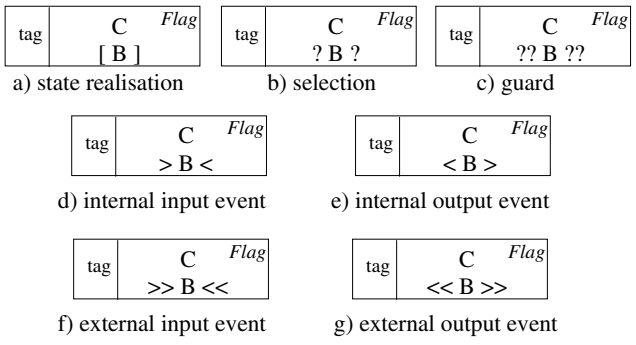

Figure 2. Different node types

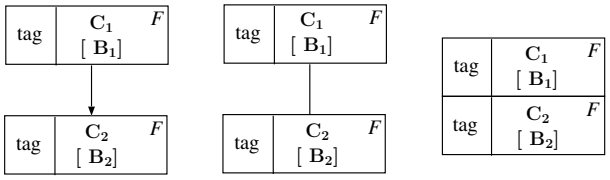

Figure 3. Sequential behaviour

together without an edge (Fig. 3c), the two steps occur together atomically. The notation in Fig. 3b also implies that the upper node is executed before the lower node (which is not necessarily the case when the syntax of Fig. 3c is used).

Fig. 4 shows the two types of branching edges: concurrent and alternative. Concurrent branching (Fig. 4a) models threads running in parallel. As an example the threads in the figure start with a guard node. The branches, however, can start with any node type. We show only two trees in the branching, although in general there may be two or more.

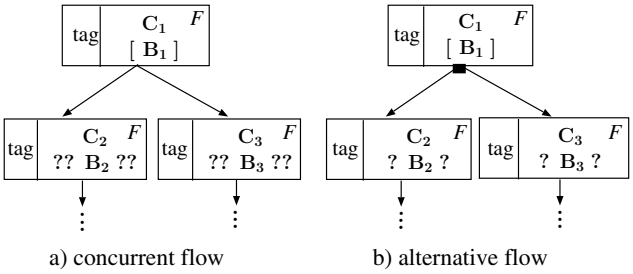

Figure 4. Branching behaviour

In alternative branching (Fig. 4b), the control flow follows only one of the branches. Alternative branches can comprise either selections only (for example, as shown in Fig. 4b) or only other node types but no selections. Alternative branching over selections operates as a non-deterministic choice over the branches with a satisfied selection condition B_i. If none of the selections is satisfied the behaviour terminates. Alternative branching over non-selections behaves like a non-deterministic choice that is unguarded.

As described in [18], the BT syntax also includes notation for standard set operations, some of which are shown in Fig. 5: Assume *C* is a component with an attribute *S* which is a set, then a) models adding of an element *x* to *S*, b) the removal of element *x* from set *S*, and c) demonstrates how to query the cardinality of set *S* (in this case '*is the cardinality of S less than n*').

Additionally to standard set operations, the syntax also provides two constructs for describing behaviour to be performed on all members or on one member of a reference set

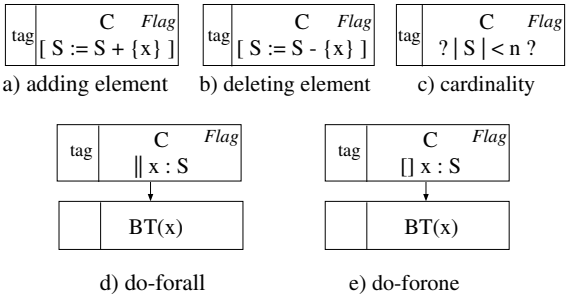

Figure 5. Set Operations

S. Fig. 5d) models execution of a tree BT(x) for all members *x* of set *S*, and Fig. 5e) models execution of BT(x) for some member *x* of set *S* (where *x* is chosen non-deterministically). Note that the sub-tree BT(x) is parametrised with *x* (i.e., it depends on *x*). These constructs are referred to as *do-forall* and *do-forone*, respectively.

2.3. Example

As an example of using the notation to capture requirements, consider the following system for handling student enrolments in degrees at a university.

R1. Each student at the university may sign up for a degree. Each degree has a pre-determined set of courses.

R2. After signing up to a degree, the student becomes active at the university.

R3. If an active student has not yet passed enough courses to finish the degree, they may begin enrolling in courses. If they have passed the required amount of courses for the degree, the student graduates.

R4. A student may enrol in any course that is in their degree which they have not already passed. A student may enrol in up to 4 courses at a time.

R5. At some point after students begin enrolling, the university closes enrolments and announces the start of semester. Students start studying for every course in which five or more students are enrolled.

R6. After studying, the students are examined in their courses when the university announces the start of examination period.

R7. After being examined, it is recorded whether they passed or failed. The student is then no longer enrolled in the course. After all results are recorded, the semester ends and the student becomes active again.

A modeller using BTs would now model each requirement as its own (small) tree. For instance, consider the translation of requirement R6 in Fig. 6. We treat the university announcement as an event, *ExamPeriodStart*, which causes each student to leave the state *study* and enter the state *Examined*. Because the message is sent to all students at the same time, we add the flag = to the event node to signify a synchronisation.

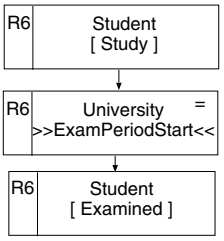

Figure 6. BT modelling requirement R6

Now consider modelling requirement R7. We must decide how to "record" the pass or fail of the course. We may decide to keep an attribute which maintains the set of courses the student is currently enrolled in, and has previously passed or failed. This is shown explicitly in Fig. 7. However, choosing to maintain only the courses the student is enrolled in will cause problems if a course needs to know the set of students that have enrolled in it. Therefore we also maintain attribute *EnrolledS* as an attribute of *Courses*. This results in a rather large and cumbersome tree, which contains too much implementation detail for the purposes of requirements capture and communication with the user.

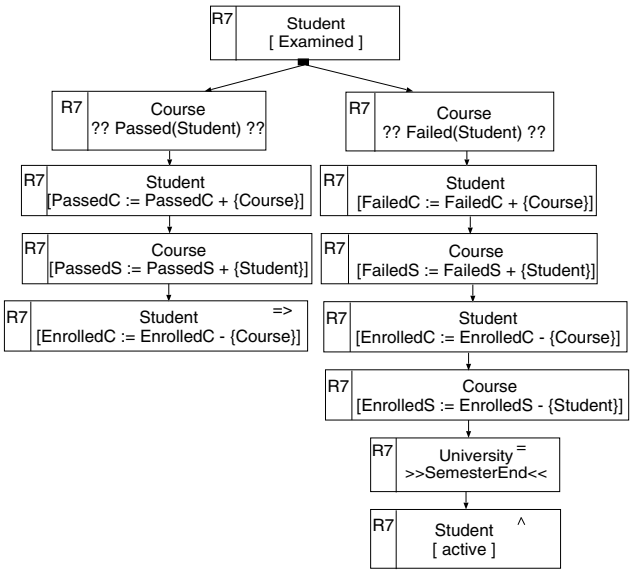

Figure 7. Relations as Attributes

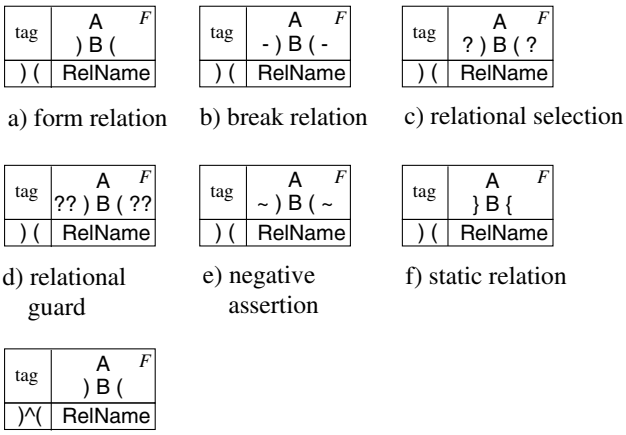

Figure 8. Syntax Elements

In the next section we will extend the BT syntax to handle relations, so that such information can be represented more clearly and compactly.

3. Relational Behavior Trees

Functional requirements can be based around the fact that components are related to each other; we say they form *component relations*. Component relations are dynamic entities in the sense that they can change over time and their status may influence the flow of control of the overall system. We extend the notation of Behavior Trees with new syntax elements enabling the user to swiftly model behaviour of systems based on component relations. We call this extension *relational Behavior Trees*.

3.1. Notation

In our framework, component relations have a similar status to the state of a component. In a similar fashion as changing its state a component can change its relations. It follows that the syntax elements for relations are designed analogously to the syntax elements for states, namely realisation/creation, selection, and guard. Notably though, a component can only be in one state at a time, but it can have more than one relation.

The syntax for a component *forming* a relationship with another component is shown in Fig. 8a): Component *A* forms the relation *RelName* with component *B*.

When a component changes its state it (implicitly) overwrites the previous state realisation. Similarly, component relations can be (explicitly) deleted. Fig. 8b) shows syntax to express that component *A* *breaks* the relation *RelName* with component *B*. If the relation *RelName* has not been established between *A* and *B* this node has no effect.

Forming and breaking component relations can be tested by means of selections and also as guards. Fig. 8c) shows the syntax for a *relational selection*: *If* component *A* is related to component *B* via relation *RelName* this node is satisfied. The flow of control in the Behavior Tree at this point depends on the existence of the specified relation between *A* and *B*. It continues if the relation is established and it terminates if the relation is not established.

Fig. 8d) shows the syntax for a *relational guard*. Similarly to guards on state realisations (see Section 2), a relational guard is triggered *when* a relation is formed. The node in Fig. 8d) models that the flow of control is blocked for as long as the relation *RelName* between *A* and *B* has not been formed and continues when *A* and *B* form this relation.

Fig. 8e) shows the syntax for asserting that two component *must not* be in a relation. Component relations may also be known to be static entities that cannot change. This can be expressed using the syntax shown in Fig. 8f). The use of static relations helps to simplify the analysis of a relational BT, and checks can be run more efficiently (see Section 4.4 for more detail).

To capture the intention behind a functional requirement we sometimes like to specify which of the two components is actively establishing the relation to be formed, and which is passive. The convention is that the name at the top in the BT node indicates the active component. The component name in the behaviour part of a BT node specifies the passive component. However, the symbol)^(allows the user to model the converse. The node in Fig. 8g) models that component *B* has established relation *RelName* with component *A*.

It is optional in our notation to specify the name of a relation. If *RelName* is omitted (and with it the lower part of the node) we assume the default relation "*is an element of*" between the given components.

Since component relations form a link between two components the user can naturally access information about the relation via both components. That is, if component *A* has formed the relation *RelName* to component *B* then both *A* and *B* know about this fact. We allow the user to retrieve this information from both sides via a *projection*, namely *A^RelName* and *B^RelName*. A projection works like a filter that is applied to a relation. It yields partial information of the relation that is relevant in a given "context", i.e., a particular component. That is, the expression *A^RelName* yields all components which *A* is related to via *RelName*, and similarly, *B^RelName* yields the set of components *B* is related to via *RelName*.

Using the graphical syntax of Behavior Trees we place the component's name (i.e., *A* or *B* etc.) at the top in the node, setting the context of the projection, and refer to the projection itself without repeating the component's name. An example is shown in Fig. 9 where we query if the cardi-

nality of the projection *A^RelName* is greater than 3. Note that a projection of a relation, as the relation itself, results in a set and therefore set operations (as in Fig. 5) can be applied.

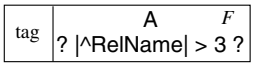

Figure 9. BT node using a projection

If the user wants to explicitly access the first entries in the relation pairs and component *A* might be either first or second entry it can be model using a subscript: $RelName_1$ provides the set of components that are first entries in any of the pairs in *RelName*. Similarly, $RelName_2$ specifies the components that are second entries. Consequently, $A\hat{}RelName_1$ yields all first entries that are related to component *A*.

In the following we demonstrate with our student example how this syntax can be used to capture functional requirements.

3.2. An Example

Our BT model of the student's behaviour (as described through the functional requirements in Section 2.3) constitutes the seven integrated single requirements corresponding to R1 – R7. The full model is shown in Fig. 10 and is described in the following (for convenience we repeat the requirements). Please note that the tag at the left hand side of each BT node refers to the requirement the node originates from. Types and data structures of the components and relations have been specified in the Component Tree (cf. Section 2.1) of the model which is omitted here. We assume that the student's behaviour is embedded into a larger University system in which some relations, e.g., the relation between *Degree* and a number of courses, have already been populated.

R1. Each student at the university may sign up for a degree. Each degree has a pre-determined set of courses.

R2. After signing up to a degree, the student becomes active at the university.

We model *University* as a component which has an attribute *Students* which is a set. Since each student behaves in a similar fashion we utilise the *do-forall* construct over all elements in this set. It serves as the initial node in the tree and implicitly creates a "forest" of sub-trees, each of which models the behaviour of a particular student in *Students*. All sub-trees behave concurrently.

Furthermore, the component *University* has an attribute called *Degrees* which is also a set. One element of this set is chosen, *Degree*, using the *do-forone* operator. *Student* forms a relation *Signup* with the chosen *Degree*. After that

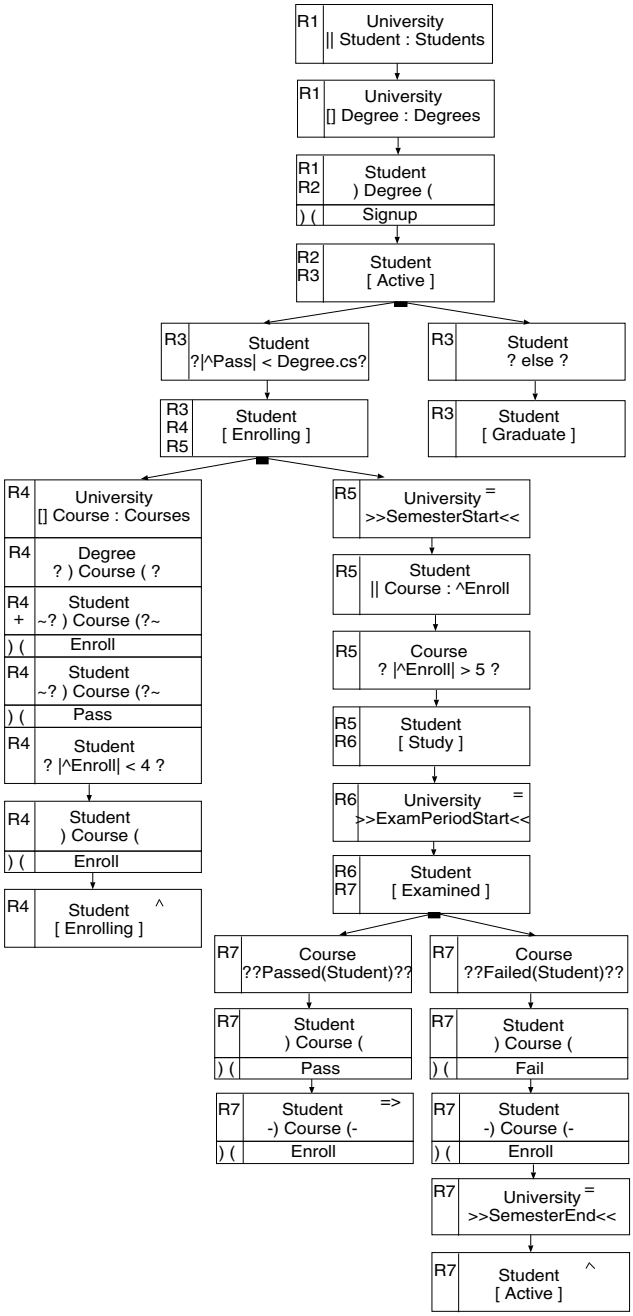

Figure 10. Student behaviour model

Student realises the state *Active*.

R3. If an active student has not yet passed enough courses to finish the degree, they may begin enrolling in courses. If they have passed the required amount of courses for the degree, the student graduates.

We use alternative branching to model that the student either enrolls in courses or may graduate. To decide whether enough courses have been passed we query the cardinality

of the relation *Pass* that has been formed between *Student* and courses. In our case the cardinality of relation *Pass* yields the number of courses the student has passed. Note that we query the projection *Pass* in the context of *Student* (i.e., from his/her point of view) by making *Student* the component of the selection node, and thus get only the number of courses which *Student* (i.e., this particular student) has passed. Each *Degree* has an attribute *cs* which gives the number of courses that have to be passed in order to graduate in this degree (as specified in the composition tree). The selection $?\mid Pass \mid < Degree.cs?$ is thus satisfied if the number of courses the student has passed is less than that required to graduate from the chosen degree.

R4. A student may enrol in any course that is in their degree which they have not already passed. A student may enrol in up to 4 courses at a time.

We use the *do-forone* operator to model the choice of a course the student wants to enroll. We query if the chosen *Course* is in the *Degree*. We also query that *Student* has not yet enrolled in[1] and passed the *Course*(i.e., *Student* is not related to *Course* via *Enroll* and *Pass*), and that the number of courses the *Student* is enrolled in is less than 4 (similar to above where we use the cardinality of the projection of relation *Enroll* in the context of *Student*). If all conditions are satisfied *Student* forms the relation *Enroll* with the chosen *Course*. The behaviour then reverts back to the node $Student[Enrolling]$ so that further courses can be chosen for enrollment.

R5. At some point after students begin enrolling, the university closes enrolments and announces the start of semester. Students start studying for every course in which five or more students are enrolled.

The *University* closes the enrollment period by sending the message *SemesterStart*. As an effect the alternative branching structure in the BT will continue with the right branch. This external input event is also synchronising the "forest" of student sub-trees that behave concurrently. That is, for each student in the university the semester starts at the same time.

For all courses the student is enrolled in (we use the *do-forall* construct) we check the cardinality of the projection of relation *Enroll*, however, this time from the perspective of the component *Course*. The expression $\mid Enroll \mid$ results now in the number of students that are enrolled in this particular course. If the cardinality is greater than 5 *Student* begins studying.

R6. After studying, the students are examined in their courses when the university announces the start of examination period.

The component *University* sends a second external message that synchronises all students. This time the beginning of the exam period is announced and all students change their state from *Study* to *Examined*.

R7. After being examined, it is recorded whether they passed or failed. The student is then no longer enrolled in the course. After all results are recorded, the semester ends and the student becomes active again.

Component *Course* sends out messages that indicate passing or failing of each student (this is an abstraction of what actually happens). The message $Passed(Student)$ is a parametrised event which indicates that this particular *Student* has passed. Alternatively, a message $Failed(Student)$ can be send in case of failure. As a consequence *Student* either forms the relation *Pass* with the course (left branch) or the relation *Fail* (right branch) and breaks the relation *Enroll* after that to indicate that he/she is no longer enrolled in the course.

Component *University* announces the end of the semester via an external message *SemesterEnd*. Similarly to the other two university messages, this is used to synchronise the behaviour of all students. After that the student's behaviour reverts back to its state *Active*.[2]

We have demonstrated that the new notation is straightforward to apply, and avoids the overhead associated with the component/state model as in Fig. 7. In the next section we provide a formal semantics for the extension.

4. Semantics

The semantics of BTs (without relational behaviour) is given via an operational semantics for a CSP-like process algebra in [3]. State realisations are modelled as updates of the state, much like an assignment in an imperative programming language. For instance, $C[s]$ is treated as $C := s$. Guards and selections test the values in the state (as '*if*' statements in imperative languages), while events and synchronisations are treated more abstractly, as in CSP [11]. The box/arrow notation gives the control flow, whether a choice or parallel. The semantics of relations is given in terms of sets of tuples, with relations acting as a special type of component, and therefore the notation fits with existing tools such as the simulator [17] and model checker [10]. In Sect. 4.1–Sect. 4.3 we show how we may map relational behaviour into the process algebra semantics, and in Sect. 4.4 we describe how the notation can be model checked.

[1] Note that this check is not enforced by the requirements but is added by the modeller and thus the node is tagged with a '+'.

[2] Note that we use the macro flag to avoid repeating the last three nodes on the left.

4.1. Relations

For the purposes of defining the semantics we can map relational behaviour into the same model of state as that used for components. Essentially, each relation is treated as a component (a *variable* in the underlying semantics), the type of which is a set of tuples containing elements in the relation. For instance, the relation *Enroll* is a set of *student*/*course* pairs, i.e.,

$$Enroll \mathrel{\widehat{=}} \mathbb{P}(Students \times Courses)$$

If student s_1 is enrolled in courses $cs101$ and $cs102$, while student s_2 is enrolled only in $cs101$, this is represented as:

$$Enroll = \{(s_1, cs101), (s_1, cs102), (s_2, cs101)\} \quad (2)$$

This set-of-tuples representation is essentially that used in relational database models [2]. By using this as the underlying model for relational behaviour in BTs, we obtain the potential of full expressibility of languages such as SQL, which we can combine in a dynamic behaviour setting with state-based behaviour and events.

4.2. Relation nodes

Forming a relationship between two components (Fig. 8a) is modelled by adding the pair to the set. For instance, if student s_1 enrols in $cs102$, this pair is added to the *Enroll* set, i.e.,

$$Enroll := Enroll \cup \{(s_1, cs102)\}$$

Breaking a relationship (Fig. 8b) is subtracting the pair from the set, while testing for existence/nonexistence of a relationship (Fig. 8c, d and e) is a test on set membership, i.e.,

$$Enroll := Enroll \setminus \{(s_1, cs102)\}$$
$$(s_1, cs102) \in Enroll \qquad (s_1, cs102) \notin Enroll$$

Static relationships (Fig. 8f) imply a pre-populated relation which may be queried in the usual way. The active component in relation forming (Fig. 8g) does not have a direct bearing on the semantics, since it does not matter which component established the relationship when testing for membership. However, if such information is required, this can be achieved by adding a third entry to the tuple which names the establishing component.

4.3. Relation projections

As introduced in Sect. 3.1, we allow relations to be projected with respect to a particular place in the tuple (column in the table), and/or to a component. For instance, the set of students that have enrolled in *any* course is written $Enroll_1$, since students appear in the first place of the tuples. The set of students enrolled in course c is given by $c\hat{}Enroll_1$. However, the projection to column 1 is redundant, since it is clear from the types that the projection to c must refer to the students. In such cases, we may omit the column projection. The semantics are given below, for a binary relation R of type $\mathbb{P}(T_1 \times T_2)$.

$$\begin{aligned} R_1 &= \{x : T_1 \mid (\exists\, y : T_2 \bullet (x, y) \in R)\} \\ R_2 &= \{y : T_2 \mid (\exists\, x : T_1 \bullet (x, y) \in R)\} \\ C\hat{}R_1 &= \{x : T_1 \mid (x, C) \in R\} \\ C\hat{}R_2 &= \{x : T_2 \mid (C, x) \in R\} \\ C\hat{}R &= \begin{cases} C\hat{}R_1 & \text{if } C \in T_2 \wedge C \notin T_1 \\ C\hat{}R_2 & \text{if } C \in T_1 \wedge C \notin T_2 \\ C\hat{}R_2 \cup C\hat{}R_2 & \text{if } C \in T_1 \wedge C \in T_2 \end{cases} \end{aligned}$$

For example, given $Enroll : \mathbb{P}(Students \times Courses)$ is currently populated as in (2), the set of courses in which student s_1 is enrolled is determined as below.

$$\begin{aligned} & s_1\hat{}Enroll \\ = \;& s_1\hat{}Enroll_2 \qquad (\text{since } s_1 \in Students) \\ = \;& \{x : Courses \mid (s_1, x) \in Enroll\} \\ = \;& \{cs101, cs102\} \end{aligned}$$

4.4. Model Checking

Model checking is a means to automatically analyse the full state space of the system [1]. Standard BTs can be translated into model checker languages, e.g., the SAL [4] input notation as documented in [9, 10]. This provides a powerful tool for debugging BT models, in particular in the presence of concurrent (i.e., threaded) behaviour.

We extend the translation from BTs to SAL ([9, 10]) in order to capture the concept of component relations. For reasons of efficiency we simply represent each relational entry by a Boolean variable - simulating the characteristic function of the relation. Its value is assigned to `true` when the relation is formed, and it is assigned to `false` when the relation is broken. That is,

$$(s_1, c_1) \in Enroll \Rightarrow \texttt{Enroll\_s1\_c1 = true}$$
$$(s_1, c_1) \notin Enroll \Rightarrow \texttt{Enroll\_s1\_c1 = false}$$

Note that the name of the Boolean variable `Enroll_s1_c1` is generated from the relation name, the name of the first component in the relation pair and the name of the second component. Note that (s_1, c_1) and (c_1, s_1) are treated as two different entries in the relation *Enroll*.

Selection and guards over relations are translated similarly to selections and guards in standard BTs: They are reflected in the guard of a SAL transition. Static relations are simply translated into constants.

In order to avoid the rather complex cardinality operator that is provided for the SAL notation (cf. [13, 5]) we simply maintain a counter for the cardinalities if needed. That is, if the cardinality is used anywhere in the model, a corresponding counter is created. That means for our student example we create two cardinality counters for the relation *Enroll*. Forming the relation *Enroll* between s_1 and c_1 then leads to three updates in the SAL code, namely

```
Enroll_s1_c1' = true;
Card_Enroll_s1' = Card_Enroll_s1 + 1;
Card_Enroll_c1' = Card_Enroll_c1 + 1;
```

Breaking a relation is translated correspondingly in decreasing the cardinality variables, however, this update is guarded and becomes effective only if the relation has been established before, otherwise the value remains unchanged.

The *do-forall* and *do-forone* constructs are simply encoded such that *do-forall* leads to n concurrent branches (assuming the reference set contains n elements) whereas *do-forone* leads to n alternative branches. These can then be translated using the standard BT to SAL translation.

Quantification (as in *do-forall* and *do-forone*), and in particular *nested* quantification (as used in our example), can quickly lead to a model of massive size when the reference sets are large. This becomes a problem when applying model checking where the size of the state space to be checked can grow exponentially. In some cases we can mitigate the problem using model checkers that are based on Petri Nets [12], e.g., [8, 16]. These tools allow the user to efficiently analyse concurrent models which consists of many *equivalent* sub-processes. In future work we will investigate the translation into Petri Nets and the use of Petri Nets tools for the analysis of relational BTs.

5. Related Work

The underlying semantics of the language is based on the relational database model (see, e.g., [2]). This treats a relation between n entities as a set (or *table*) of n-place tuples which represent particular instances of the relationship. This model is well-known and straightforward, and has well-developed languages for querying information. For instance, assume the relationship *Enroll* is a table in which the first column is labelled *Student* and the second *Course*. In a database query language like SQL, the list of courses in which a student $s1$ is enrolled can be retrieved by the following query:

```
SELECT Course FROM Enrolled
WHERE Student = s1
```

With the syntax of relational behaviour for BTs, we have provided a convenient and intuitive graphical representation for common queries, and, in the context of describing complex behaviour, various methods for handling the information in a dynamically changing setting.

Relations are used in many other modelling languages. For instance, in state-based formal specification languages such as *Z* [14], relations can be modelled directly as sets of tuples in the relational database style. However, such languages are not as convenient for requirements translation and non-specialist client interaction in the style of BTs.

UML [15] represents static relationships through class diagrams, however these diagrams state what relationships can be formed, not how they are formed. Dynamic system behaviour is given in UML through other diagram types, such as Activity Diagrams, but it is not clear how they can make use of relationships defined in class diagrams. In addition, there is no single authoritative source for the formal semantics of dynamic behaviour in UML, and it does not support direct requirement translation in the same manner as Behavior Trees.

6. Discussion

In this paper we have introduced a syntax and semantics for capturing complex relational behaviour commonly found in natural language system requirements. We have incorporated the semantics of relational databases, which allows queries of static sets of data, into Behavior Trees, where relations dynamically change and elements may engage in other state- and event-based behaviour. The semantics is also executable, and hence amenable to simulation and model checking.

For reasons of space we have not given syntax to handle relations between an arbitrary number of components and values. For example, we may allow the *Enroll* relation between courses and students to include a 'grade' slot recording the student's mark for that course, and/or a slot including the tutor for that student in that course. The semantics given in Sect. 4 can be extended to handle these cases straightforwardly as three- or four-place relations. In practice, however, it is sometimes more convenient to break n-way relations into the relevant binary relations.

The relational notation we have presented in this paper supports compact, expressive, and traceable translations, which provides a firm basis for interacting with the client in resolving ambiguities and conflicts in the requirements. We argue that this is not the case for languages which provide only state or event based constructs, and which do not support one-requirement-at-a-time modelling.

Acknowledgements. The authors acknowledge the support of the Australian Research Council (ARC) Discovery Grant DP0345355, and would like to thank their colleagues in the Dependable Complex Computer Systems research group for helpful discussions as well as the anonymous re-

viewers for their valuable comments.

References

[1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[3] R. Colvin and I. J. Hayes. A semantics for Behavior Trees. ACCS Technical Report ACCS-TR-07-01, ARC Centre for Complex Systems, April 2007.

[4] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Proc. of Int. Conf. on Computer-Aided Verification, (CAV 2004)*, volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[5] J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In Z. Liu and J. He, editors, *Proc. of Int. Conf. on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, pages 678–696. Springer, 2006.

[6] R. Dromey. Formalizing the transition from requirements to design. In H. Jifeng and Z. Liu, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Component-Based Development, pages 156–187. World Scientific Publishing Co., Inc., 2006.

[7] R. G. Dromey. From requirements to design: Formalizing the key steps. In *Proc. of Int. Conf. on Software Engineering and Formal Methods (SEFM 2003)*, pages 2–13. IEEE Computer Society, 2003.

[8] B. Grahlmann. The PEP Tool. In O. Grumberg, editor, *Proc. of Int. Conf. on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 440–443. Springer, 1997.

[9] L. Grunske, P. A. Lindsay, N. Yatapanage, and K. Winter. An automated failure mode and effect analysis based on high-level design specification with behavior trees. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proc. of Int Conf. on Integrated Formal Methods (IFM 2005)*, volume 3771 of *LNCS*, pages 129–149. Springer, 2005.

[10] L. Grunske, K. Winter, and N. Yatapanage. Defining the abstract syntax of visual languages with advanced graph grammars-a case study based on Behavior Trees. *Journal of Visual Language and Computing*, 19(3):343–379, 2008.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[12] J. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):221–252, 1977.

[13] G. Smith and L. Wildman. Model checking z specifications using SAL. In M. H. H. Treharne, S. King and S. Schneider, editors, *Proc. of Int. Conf. of B and Z Users (ZB 2005)*, volume 3455 of *LNCS*, pages 85–103. Springer, 2005.

[14] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[15] P. Stevens, J. Whittle, and G. Booch, editors. *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications*, volume 2863 of *LNCS*. Springer, 2003.

[16] K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In O. Grumberg, editor, *Procs. of Int. Conf. on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 472–475. Springer, 1997.

[17] L. Wen, R. Colvin, K. Lin, J. Seagrott, N. Yatapanage, and R. G. Dromey. "Integrare", a Collaborative Environment for Behavior-Oriented Design. In Y. Luo, editor, *Proc. of Int. Conf. on Cooperative Design, Visualization, and Engineering (CDVE 2007)*, volume 4674 of *LNCS*, pages 122–131. Springer, 2007.

[18] S. Zafar, R. Colvin, K. Winter, N. Yatapanage, and R. G. Dromey. Early validation and verification of a distributed role-based access control model. In *Proc. of Asia-Pacific Software Engineering Conference (APSEC 2007)*, pages 430–437. IEEE Computer Society, 2007.

[19] Behavior engineering. http://www.behaviorengineering.org.