Tool-Assisted Verification of Behaviour Networks

Christopher Armbrust*, Lisa Kiekbusch*, Thorsten Ropertz*, and Karsten Berns*

*Robotics Research Lab, Department of Computer Science,
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
Email: {armbrust, kiekbusch, ropertz, berns}@cs.uni-kl.de

Abstract—This paper deals with the problem of assisting developers when verifying properties of complex behaviour-based systems. A central aspect of behaviour-based systems is the interaction between the behaviours, as a lot of the functionality of a system typically arises from this interaction. Hence, verification has to deal with the specialities of behaviour interaction. Previous work has introduced a concept for modelling behaviour-based systems as networks of finite-state automata and for applying model checking as verification technique. As the manual verification of large networks is tedious and error-prone, the work at hand introduces a concept for assisting developers by partly automating the verification process. The applicability of the presented approach is demonstrated using the behaviour-based control system of an autonomous bucket excavator.

I. INTRODUCTION

The application of behaviour-based approaches in robot control systems offers several advantages: While in classic architectures one specific component deals with fulfilling a certain task, each functionality in a behaviour-based system (BBS) is typically realised by a set of behaviours. This results in smaller, simpler components, which are easier to develop, implement, and maintain. Furthermore, it is possible to split generic and robot-specific parts, which allows for reusing code on different platforms. [1] describes how BBS differ from purely reactive or planner-based systems, while [2] explains how BBS can be used for executing complex, sequential, and hierarchically structured tasks.

The downside is that with increasing distribution comes increasing need for a sophisticated interaction of the different behaviours. Sound guidelines for the development and implementation of BBS can improve the quality of the systems; but for proving their correctness, verification techniques are necessary. “Correct” in this context means that the behaviours are connected using special signals (see Sec. III-A) in such a way that the resulting system fulfills the intended tasks.

This paper continues previous research about behaviour network verification conducted at the Robotics Research Lab. Figure 1 gives an overview of the underlying concept. Following certain design methodologies (see [3]), behaviour networks are developed and integrated into the control systems of robots. In [4], an approach for modelling behaviour networks as networks of finite-state automata and the application of model checking for verification have already been described. However, the processes of defining queries for the model checker and analysing the resulting traces were purely manual, which made them tedious and error-prone. The paper at hand presents a concept for providing the user with a graphical user interface in which it is possible to select properties from a given subset and adapt them to the system that shall be verified. These properties are automatically transformed into observer automata, which are integrated into the formal model of the behaviour network and evaluated by a model checker.

The remainder of this paper gives an overview of work in the field of verification (Sec. II) and introduces the approach for assisting developers during the verification of complex behaviour networks (Sec. III). Section IV shows how the proposed concepts have been realised and demonstrates their application to the verification of the behaviour-based control network of an autonomous bucket excavator. Finally, a conclusion and an outlook on future work are given in Sec. V.

II. RELATED WORK

It is a known problem that software developers have difficulties in using model checking techniques for the verification of their software because they are unfamiliar with the notation processes and notations. Therefore, a lot of work has already been done in the field of defining properties for software systems in a more comprehensive fashion. [5], [6], [7] propose approaches based on patterns that describe frequently used properties informally and provide the corresponding formulae in several notations, e.g. CTL, LTL.
ACTL, and regular expressions. The formulae can then be used as input for several kinds of model checking tools. The authors state that using their pattern-based approaches significantly shortens the time a developer needs for software verification. This is an aim that the authors of the work at hand also target at by providing tool support that uses special patterns representing properties, here called query graphs (see Sec. [III-B]). Continuing the above-mentioned approach, [8] defines properties in a natural language and automatically translates them to temporal logic formulae. But a formal definition of required properties becomes difficult due to the ambiguity of natural languages. The authors of [9] present a plug-in for the Eclipse development environment, which supports the software engineer in using model checking techniques to solve some typical program analysis problems directly on the source code. In their approach a developer needs no knowledge of model checking or formal notation. However, he is restricted to the given options. A comparable approach is followed by the authors of the paper at hand and the property template. This approach is followed by the authors of the paper at hand [10] extends the approach of [6] by providing the property patterns as templates presented in an extended finite-state automaton notation and as natural language phrases. This approach allows for indicating all options that must be considered when choosing a property template. The approach presented in the paper at hand is similar to the two last-mentioned approaches. It introduces a simplification of the specification of properties of BBS by providing a graphical user interface and a generation method which creates corresponding finite-state automata. The concept of these so-called observer automata is an often-used approach to simplify model checking tasks and solve complex questions (see e.g. [11]).

The work at hand also uses timing diagrams for an easier understanding of properties and requirements as for example described in [12].

III. CONCEPT

This section introduces the type of behaviour networks upon which the work at hand is based and the properties that shall be verified for a given system. Furthermore, the modelling of behaviour networks as networks of finite-state automata is described as well as the transformation and integration process of the said properties.

A. Behaviour Networks

The availability of a well-defined behaviour-based architecture with development guidelines improves the structure of a system and facilitates modelling and verification. The architecture used for the work at hand is the iB2C[1] which has been implemented using the software frameworks MCA2-KL[2] and FINROC[3] (see [13]). Only a very brief introduction to

\[ B = (f_a, f_r, F) \]

Each iB2C behaviour is defined as \( B = (f_a, f_r, F) \), with \( f_a \) calculating its activity vector \( \vec{a} \), \( f_r \) calculating its target rating \( r \), and \( F : \vec{a} = F(\vec{e}, \vec{i}) \) transferring input vector \( \vec{e} \) and activation \( i \) into the output vector \( \vec{a} \). The so-called activation \( i = s \cdot (1 - i) \) is combined of a behaviour’s stimulation \( s \) (used by another behaviour to gradually enable it) and inhibition \( i = ||\vec{i}||_\infty \) (used by other behaviours to gradually disable it). The activity vector \( \vec{a} = (a_1, a_2, \ldots, a_q)^T \) is composed of the behaviour’s activity \( a \) (which indicates the degree of influence a behaviour intends to have in a network) and the so-called derived activities \( a_1, a_2, \ldots, a_q \) with \( a_i \leq a, \forall i \in \{0, 1, \ldots, q - 1\} \), which a behaviour can use to transfer only a part of its activity to other behaviours. A behaviour’s activity is limited by its activation, i.e. \( a \leq t = s \cdot (1 - i) \).

The target rating describes how unsatisfied a behaviour is with the current situation. \( s, i, a, r \), and \( r \) constitute the behaviour signals, which are limited to \([0, 1]\) and are part of each behaviour’s interface. By contrast, there is no limitation of \( \vec{e} \) and \( \vec{i} \) and they differ among behaviours. Figure 2 depicts the symbol of a behaviour.

There are various ways of connecting iB2C behaviours with each other. For example, a behaviour \( B_0 \) can stimulate or inhibit another behaviour \( B_1 \) with its activity. In this case, the \( a \) output of \( B_0 \) is connected to the \( s \) input (or \( i \) input, respectively) of \( B_1 \). In BBS, the coordination of behaviours is crucial and there are various ways for realising it (see [14]). Two special coordination behaviours are used in the iB2C: the fusion behaviour and the conditional behaviour stimulator.

A fusion behaviour \( B_{\text{Fusion}} \) combines the outputs of \( n_c \) competing behaviours \( B_{\text{Input}_d} \) (with \( a_d, r_d \), and \( \vec{a}_d \)) according to:

\[ B_{\text{Fusion}} = (f_a, f_r, F) \]

The symbol of a fusion behaviour for \( n_c = 2 \) competing behaviours is shown in Fig. 3.
to one of three fusion methods. For the work at hand, only the maximum fusion method is relevant, which yields $\bar{a}_{\text{Fusion}} = \bar{d}_{e}$, $a_{\text{Fusion}} = \max(d(a_d) \cdot d_{\text{Fusion}})$, and $r_{\text{Fusion}} = r_e$, where $e = \arg \max(d(a_d))$. The symbol of a maximum fusion behaviour is depicted in Fig. 3.

The purpose of a conditional behaviour stimulator (CBS) is to get active (or inactive) depending on other behaviours that are connected to its ports. Each connected signal is compared to a corresponding threshold according to a certain relation ($<, \leq, =, \geq, >, \neq$). A condition assigned to the input signal is fulfilled depending on the relation and the condition’s type. There are three types of conditions:

1) **Permanent**: The corresponding relation has to be fulfilled during the whole time when the behaviour shall be active, i.e. the condition is fulfilled if and only if the relation is fulfilled.

2) **Ordering**: The corresponding relation has to be fulfilled at some point in time before the behaviour shall get active. The condition will stay fulfilled independent of whether the relation stays fulfilled or not.

3) **Enabling**: The corresponding relation has to be fulfilled at the exact point in time when the behaviour shall get active. After that, the condition stays fulfilled independent of the fulfillment of the relation.

Conditions are either **input conditions** or **feedback conditions**. If a CBS is activated, it gets active as soon as all of its input conditions are fulfilled. It then starts checking whether all of its feedback conditions are also fulfilled. If this is the case, it gets inactive and the cycle is restarted. Figure 4 depicts the symbol of a CBS. Details about this special behaviour can be found in [15].

![Fig. 4. The symbol of a CBS depicting the three different types of ports (enabling, ordering, and permanent) for input conditions (top) and feedback conditions (bottom). As a CBS is a behaviour, it also features the standard behaviour ports.](image)

To facilitate the handling of a large number of interconnected behaviours, **behaviour groups** encapsulate a number of behaviours or further groups and act as new behaviours in a network. Their interface to the outside is the same as that of a standard behaviour. Hence, they can be connected to other behaviours in the same way as single behaviours.

**B. Properties of Behaviour Networks**

During the verification process, it shall be checked whether the behaviour network in question features certain properties or not. Commonly used properties like deadlock and liveness are not applicable in this context due to the used modelling technique which focuses on the network structure. For example, checking for deadlocks would always return a negative result since the target rating is overapproximated by indeterministic value switching in order to abstract from behaviour-internal computations. The properties used in this work follow [6], but are more specific to the needs of behaviour networks. Typical properties are defined in the following.

Let $bs_B \in \{s_B, i_B, t_B, a_B, r_B\}$ be a behaviour signal of behaviour $B$, $\otimes \in \{<, \leq, =, \geq, >, \neq\}$ a relation symbol, and $t \in \{0, 1\}$ a threshold. A basic **term** is then defined as $t = (bs_B \otimes t)$. Further terms can be built by defining $t_0 \land t_1$ and $t_0 \lor t_1$ (with $t_0$ and $t_1$ being terms) also as terms.

Based on these terms, the following **temporal properties** can be defined. They are explained below.

- **synchronous_before** ($t_{src} < t_{dst}$)
- **asynchronous_before** ($t_{src} \neq t_{dst}$)
- **synchronous_paired_before** ($t_{src} \neq t_{dst}$)
- **asynchronous_paired_before** ($t_{src} \neq t_{dst}$)
- **requires_non-strict** ($t_{src} \leq t_{dst}$)
- **requires_settled** ($t_{src} = t_{dst}$)
- **synchronous_requires_once** ($t_{src} < t_{dst}$)
- **asynchronousRequires_once** ($t_{src} < t_{dst}$)
- **globally(t)**
- **eventually(t)**

![Fig. 5. A timing diagram of the temporal property synchronous_before.](image)

Here, $t_{src}$ is called **source term**, $t_{dst}$ is called **destination term**. The **synchronous_before** property states that $t_{src}$ has to hold at least once before each occurrence of $t_{dst}$. Additionally, the two are allowed to start holding at the same time (→ “synchronous”). Figure 5 shows a corresponding timing diagram. Sequences marked with OK fulfill the property. For the sequences marked with E (error) the property is not fulfilled. The red marked sequences (OK2) depict the synchronous mode of the property allowing $t_{src}$ and $t_{dst}$ to start holding at the same time. The **asynchronous_before**

---

4 “Start holding” means that a term did not evaluate to true before and now evaluates to true.
property is defined in a similar way. But here, \( t_{\text{dst}} \) is not allowed to start holding at the same time as \( t_{\text{src}} \) of the same cycle ( \( \rightarrow \) “asynchronous”). Consequently, the red marked sequences in Fig. 5 would describe an error case in which the property is not fulfilled. The \textit{synchronous paired before} property requires \( t_{\text{src}} \) to start holding before \( t_{\text{dst}} \) is fulfilled. In contrast to the \textit{synchronous before} property, \( t_{\text{src}} \) is not allowed to start holding more than once between two consecutive occurrences of \( t_{\text{dst}} \). Therefore, sequence OK1 depicted in Fig. 5 violates the property because \( t_{\text{dst}} \) does not start holding between the last two occurrences of \( t_{\text{src}} \). Again, there exist a synchronous and an asynchronous variant of the property, which differ in whether or not \( t_{\text{src}} \) and \( t_{\text{dst}} \) are allowed to start holding at the same time. The \textit{requires} property is available in two types, namely as \textit{requires non-strict} and \textit{requires strict}. In both cases, \( t_{\text{dst}} \) may only start holding if \( t_{\text{src}} \) also holds. Strict in this case means that \( t_{\text{dst}} \) may only hold as long as \( t_{\text{src}} \) holds. This is not required for the non-strict variant. Finally, there are the properties \textit{synchronous requires once} and \textit{asynchronous requires once}. As the names suggest, in both cases \( t_{\text{src}} \) has to hold at least once before \( t_{\text{dst}} \) is allowed to hold. Similar to above, “synchronous” means that the two are allowed to start holding at the same time, “asynchronous” that they are not.

The already mentioned temporal properties are binary and require \( t_{\text{src}} \) to differ from \( t_{\text{dst}} \) to be meaningful. There are also unary temporal properties like the \textit{globally(t)} and the \textit{eventually(t)} properties. They are satisfied if and only if \( t \) holds all the time or at some point in time, respectively.

Apart from temporal properties, also other properties can be defined. For example, \textit{priority}(B_0, B_1) (with behaviours \( B_0 \) and \( B_1 \)) is defined as follows: \( B_0 \) is said to have higher priority than \( B_1 \) if and only if \( B_0 \) can inhibit \( B_1 \) and \( B_1 \) cannot inhibit \( B_0 \). The \textit{priority} has been defined like this as a behaviour’s activity indicates the maximum influence the behaviour can have within a network.

To illustrate this, some example behaviour networks shall be given along with explanations about properties that hold or do not hold. Figure 6 depicts two behaviours \( B_0 \) and \( B_1 \). The former stimulates the latter with its activity. The properties \textit{requires non-strict}(\( a_{B_0} = 1 \), \( a_{B_1} = 1 \)) and \textit{requires strict}(\( a_{B_0} = 1 \), \( a_{B_1} = 1 \)) both hold as \( B_1 \) can only get active if it is stimulated (due to \( a \leq t \), see Sec. III.A). In Fig. 7 a small network consisting of three behaviours \( B_0 \), \( B_1 \), and \( FB \) is shown. It fulfils the property \textit{priority}(\( B_0 \), \( B_1 \)) as there is an inhibiting connection from \( B_0 \) to \( B_1 \). Hence, it is possible that \( B_1 \) is activated (\textit{eventually}(\( t_{B_1} = 1 \))), but this is not always the case (\textit{globally}(\( t_{B_1} = 1 \))). \( B_0 \) has the ability to overrule \( B_1 \) at the maximum fusion by increasing its activity above 0.5 (because then \( a_{B_1} \) falls below 0.5). Furthermore, the property \textit{requires strict}(\( (a_{B_0} = 1) \lor (a_{B_1} = 1), a_{FB} = 1 \)) holds.

In the last example (see Fig. 8), a CBS is connected with two behaviours \( B_0 \) and \( B_1 \). The former is connected with an enabling input condition, the latter with an enabling feedback condition. As a result, the CBS will get active if \( a_{B_0} = 1 \) and will then start checking its feedback condition. If \( a_{B_1} = 1 \), it will get inactive and check its input condition again. \textit{requires non-strict}(\( a_{B_0} = 1 \), \( a_{CBS} = 1 \)) is fulfilled as the CBS can only get active if the enabling input condition is fulfilled. \( B_0 \) does not have to stay active (enabling), hence \textit{requires strict}(\( a_{B_0} = 1 \), \( a_{CBS} = 1 \)) does not hold. If the input condition was permanent, \textit{requires strict}(\( a_{B_0} = 1 \), \( a_{CBS} = 1 \)) would also be fulfilled.

The properties defined above can be represented graphically by \textit{query graphs}. A query graph is a directed graph \( D \) consisting of a set of \textit{query vertices} \( V \) and a set of \textit{query edges} \( E \). A query vertex is defined by the name corresponding to the represented behaviour, a qualified name to uniquely determine the represented behaviour, the term to be represented, and a flag to distinguish vertices representing terms and vertices that model a conjunction or disjunction. A query edge is defined by the source and destination vertices, the property to be represented, and a flag signalling whether the edge represents a temporal property or is just a construct to model a disjunction or conjunction. Figure 9(a) illustrates a simple query graph representing the binary temporal property \textit{requires strict}(\( (a_{B_0} = 1) \lor (a_{B_1} = 1), a_{FB} = 1 \)) corresponding to the behaviour network shown in Fig. 7. The yellow edge represents the \textit{requires strict} property. \( t_{\text{dst}} \) of the property is represented by the vertex named \( FB \). Its \( t_{\text{src}} \) is given by the \textit{OR} vertex and the connected vertices representing terms. Since the \textit{OR} vertex is no vertex representing a term, its predecessor vertices representing terms have to be regarded. The dotted lines are connection edges, which signal that their source terms have to be combined by a disjunction according to their destination vertex (\textit{OR}). Thus, \( t_{\text{src}} \) of the yellow edge equals \( (a_{B_0} = 1) \lor (a_{B_1} = 1) \) as defined by the
property. Figure 9 depicts the query graph representing the unary temporal property \( \text{eventually}(a_{B_1} = 1) \). Since a unary temporal property requires just one term, the query edge’s source and destination vertices are identical. Therefore, \( t_{\text{src}} \) equals \( t_{\text{dst}} \).

(a) Binary temporal property
\[
\text{requires_strict}(a_{B_0} = 1) \lor (a_{B_1} = 1) \land (a_{FB} = 1).
\]

(b) Unary temporal property
\[
\text{eventually}(a_{B_1} = 1).
\]

Fig. 9. Query graph examples representing binary and unary temporal properties.

C. Mapping Behaviour Networks to Finite-State Automata

For modelling and verification, the UPPAAL toolbox is used (see [16]). A UPPAAL system is built up of a set of automata, each of which is a parametrised instantiation of a so-called template. Automata consist of locations and edges connecting them. Locations can be marked as committed. As long as at least one automaton of a system is in a committed location, time does not pass and the next transition must involve an outgoing edge of at least one of the committed locations. An edge can have a guard (side-effect free Boolean expression to determine whether an edge is enabled), updates (assignments), and synchronisations. The latter are realised using channels (marked with “!” for sending and “?” for receiving). Figure 10 depicts an example automaton.

Fig. 10. A simple automaton with three locations (Location_0: initial, Location_1: normal, Location_2: committed), a guard (i1 > 3), two updates (i1 = 0, i2 = 3), and two channel synchronisations (c1?: receiver, c2!: sender).

Each behaviour within an iB2C network is represented by a set of five finite-state automata. These automata deal with the calculation of the input and output behaviour signals of the behaviour. For example, one automaton models the stimulation of a behaviour (StimulationInterface), another indicates whether a behaviour is currently active or inactive (ActivityCalculation). To reduce the state space, all behaviour signals have been reduced from the interval [0, 1] to the set \{0, 1\}. The interaction between the automata of one behaviour as well as of different behaviours is realised using synchronisation channels (see Fig. 11). In order to model a complete network, an automated process traverses all behaviours and their connections in a given network and creates the automata with the correct synchronisation channels. Detailed information about the modelling of iB2C networks as networks of synchronised automata can be found in [4].

D. Mapping Properties to Queries

UPPAAL’s verifier can process two types of formulae: state formulae, which indicate whether an automaton is in a certain location or not, and path formulae, which are used to evaluate traces. Thereby, the path formulae can be categorised into reachability, liveness, and safety properties. Reachability properties state that a specific condition holds in some state of the model’s potential execution sequences. In contrast, safety properties require that the condition holds invariantly. Liveness properties denote that for all possible execution sequences the condition is eventually fulfilled. For efficiency reasons, UPPAAL interdicts the nested usage of path formulae. Therefore, more complex properties cannot be encoded directly using formulae supported by UPPAAL. Instead, they are expressed using observer automata in combination with reachability and safety properties concerning the observer. Observer automata monitor a model’s progress and transition into dedicated failure or acceptance states. A model satisfies a temporal property if the corresponding observer automaton never reaches a failure state or is able to reach an acceptance state. Figure 12 illustrates the observer automata implementing the synchronous_before property. The observer consists of two automata monitoring \( t_{\text{src}} \) and \( t_{\text{dst}} \) of the model. As already mentioned, the property requires \( t_{\text{src}} \) to hold before \( t_{\text{dst}} \) holds. Therefore, the ready flag is set if \( t_{\text{src}} \) holds (src=1) to signal that \( t_{\text{dst}} \) is allowed to hold. The flag is reset by the automaton handling \( t_{\text{dst}} \) if \( t_{\text{dst}} \) is fulfilled (dst=1). Automaton b contains a failure location named ERROR. It is reached if \( t_{\text{dst}} \) starts holding without a preceding
occurrence of \( t_{\text{src}} \), which is indicated by the \( \text{ready} \) flag not being set. This example requires no dedicated acceptance state since the property is fulfilled in all other cases. The synchronous mode is implemented by using the automaton handling \( t_{\text{src}} \) prior to the automaton handling \( t_{\text{dst}} \). Thus, if \( t_{\text{src}} \) and \( t_{\text{dst}} \) start holding at the same time, the \( \text{ready} \) flag is set before automaton \( b \) reacts. To check whether a given \texttt{UPPAAL} model fulfils the \textit{synchronous before} property, the observer automata are instantiated and the reachability of the failure location is verified by checking the formula \( E <_a > (\text{automaton}_b.\text{ERROR}) \). If the \texttt{UPPAAL} verifier provides a negative result, then the failure state is not reachable and the property is fulfilled. To improve the readability of the result, the \texttt{UPPAAL} query is negated, leading to the formula \( A[!] (\text{automaton}_b.\text{ERROR}) \).

![Observer automata implementing the synchronous before property.](image)

**Fig. 12.** Observer automata implementing the \textit{synchronous before} property.

### IV. APPLICATION EXAMPLE

This section illustrates the presented concepts using a real world example. It also demonstrates the developed tool support for assisting the verification process. The used GUI elements have been integrated into \texttt{FINSSTRUCT}, a tool providing a graphical interface for analysing networks created in the robotics framework \texttt{FINROC}. As realistic application example, a part of the behaviour-based control system of the autonomous bucket excavator \texttt{THOR} has been chosen.

![The autonomous bucket excavator THOR.](image)

**Fig. 13.** The autonomous bucket excavator \texttt{THOR}.

The core of this system has been specified with the finite-state machine (FSM) depicted in Fig. 14. It realises an autonomous excavation process that includes scanning the environment, digging through the soil, dumping the spoil, and moving between the different positions. The FSM has been transformed into a network of around 20 interconnected behaviours as described in [18]. Basically, the transformation algorithm generates a CBS for each state, which is active if and only if the system is in this particular state. Furthermore, a standard behaviour is created for realising the task the excavator shall fulfil in each state.

![FSM of the complete excavator operation (LRF: laser range finder, PCC: point cloud collector—stores a point cloud generated by the LRF).](image)

**Fig. 14.** FSM of the complete excavator operation (LRF: laser range finder, PCC: point cloud collector—stores a point cloud generated by the LRF).

This initial network has been extended with numerous other behaviours for fulfilling subtasks needed in several states or for fusing the control values different behaviours want to send to the excavator’s actuators, resulting in a \texttt{BBS} of approx. 90 behaviours (see Fig. [15]).

One of the behaviours realising subtasks (which is actually a behaviour group) is \((G)\) \textit{Approach Target Pose}, which is responsible for moving the excavator’s arm to the desired pose. According to the specification as FSM, it should be used in states \( s_2 \) (\textit{Approaching Excavation Position}), \( s_5 \) (\textit{Approaching Dumping Position}), and \( s_6 \) (\textit{Emptying Bucket}). As this subnet along with the other additional behaviours has been added without following an algorithm, the presence of errors is not unlikely. Due to missing or incorrect inter-behaviour connections, it could easily happen that \((G)\) \textit{Approach Target Pose} gets active in an undesired system state, leading to unexpected movements of the excavator’s arm, which could cause harm to people and damage objects in the machine’s environment. Figure [15] gives an impression of the complexity of the network. It can be seen that identifying incorrect connections manually is very difficult. Hence, the presented verification technique shall be used to determine in which states \((G)\) \textit{Approach Target Pose} is used, i.e. in which system states the excavator’s arm can move.

For this purpose, the relevant part of the system is modelled using the technique described in Sec. III-C which results in a system of around 160 synchronised automata. With the aid of a graphical interface that is part of \texttt{FINSSTRUCT}, a query graph corresponding to the property \( \text{eventually}(a(G)\ \text{Approach Target Pose} \equiv 1) \) is created (see Fig. 16). Using an automated process, the query graph is transferred into a corresponding \texttt{UPPAAL} observer automaton, which in turn is combined with the model of the system. It is then checked whether the observer automaton reaches an acceptance state—which it does. In order to gain more information, \texttt{UPPAAL} is called from within \texttt{FINSSTRUCT} to create a trace leading to a system state in which the property holds. The visualisation of this trace in \texttt{FINSSTRUCT} (see...
Fig. 15. The resulting behaviour-based system visualised by FINSTRUCT. It consists of the network created from the specification provided as Moore machine (see Fig. 14) and a number of interconnected behaviours added subsequently. Colours: same as of behaviour symbols introduced in Sec. III-A; double frame: behaviour groups; white: non-behaviour modules.

Fig. 16. FINSTRUCT’s GUI for defining query graphs. The depicted graphs correspond to the queries given in this section. The interface can be used to directly call UPPAAL’s verifier to check whether the queries corresponding to the query graphs are fulfilled or not.

Fig. 17) shows that (F) Approach Position is also active and stimulates (G) Approach Target Pose. It also shows that (F) Approach Position is active because of Approach Excavation Position being active, which in turn is stimulated by (CBS) Approach Excavation Position, the CBS corresponding to state $s_2$ of the Moore machine depicted in Fig. 14.

The next step is to create a query graph representing the property $\text{requires\_strict}(\alpha_{(CBS)}\text{ Approach Excavation Position} \equiv 1, \alpha_{(G)}\text{ Approach Target Pose} \equiv 1)$ and check it with UPPAAL’s verifier. The check yields false, i.e. the property does not hold. Again, the trace produced by UPPAAL (leading to a counter-example) is visualised in FINSTRUCT in order to identify the second state in which (G) Approach Target Pose is used: $s_5$ (Approaching Dumping Position). Creating another query graph for the property

\[
\text{requires\_strict}(\alpha_{(CBS)}\text{ Approach Excavation Position} \equiv 1 \lor \alpha_{(CBS)}\text{ Approach Dumping Position} \equiv 1, \alpha_{(G)}\text{ Approach Target Pose} \equiv 1)
\]

and analysing the resulting trace yields the next state in which (G) Approach Target Pose gets active, namely $s_6$ (Emptying Bucket).

This process could be continued iteratively to identify all system states in which the excavator’s arm can move until no further state is found. If among these states was one in which the arm should not move, the cause for this misbehaviour could be identified (again with the proposed concept) and removed, yielding an increased safety of the autonomous excavator. Such a cause would possibly be that another behaviour was—by mistake—connected to (F) Approach Position. This behaviour could be, for example, Evaluate Scan Data.
V. CONCLUSION AND FUTURE WORK

This paper has presented a novel approach for using tools to support the verification of complex behaviour networks. Properties to check can be input graphically as query graphs, which are automatically transferred into observer automata. Using these automata, it can be checked easily whether a property holds or not. Traces for counter-examples can be displayed directly in the visualisation of the behaviour network in question. Analysing complex networks has shown to yield challenges with respect to the computational effort of the model checking. A paper focussing on quantitative aspects like the model size and required verification time is already in process. While certain aspects of the work at hand are tailored specifically to the iB2C (e.g. the concrete modelling of a behaviour), concepts like applying model checking to models of behaviour networks or specifying queries as graphs can be applied to other architectures as well, provided these architectures are well-defined. In the context of future work, techniques for reducing this effort shall be developed. One starting point is the optimisation of the current UPPAAL models. Furthermore, iterative verification approaches taking into account prior knowledge about a behaviour network shall be investigated. For this purpose, a collaboration with researchers of the university’s Concurrency Theory Group shall be established.

ACKNOWLEDGMENT

The authors gratefully acknowledge Prof. Roland Meyer from the Concurrency Theory Group of the University of Kaiserslautern for his helpful comments and suggestions. Furthermore, the authors acknowledge the technical work done by the students Maryla Rittmann and Stephan Rohr. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 285417.

REFERENCES