Automated Translation and Analysis of a ToolBus Script for Auctions

Wan Fokkink[2,1], Paul Klint[1,3], Bert Lisser[1], and Yaroslav S. Usenko[4,1]

[1] Software Engineering Cluster,
Centrum voor Wiskunde en Informatica,
Amsterdam, The Netherlands
[2] Theoretical Computer Science Section,
Vrije Universiteit Amsterdam, The Netherlands
[3] Programming Research Group,
Universiteit van Amsterdam, The Netherlands
[4] INRIA Lille - Nord Europe,
Parc Scientifique de la Haute Borne
40, avenue Halley
Bat.A, Park Plaza
59650 Villeneuve d'Ascq, France

Abstract. ToolBus allows to connect tools via a software bus. Programming is done using the scripting language Tscript, which is based on the process algebra ACP. In previous work we presented a method for analyzing a Tscript by translating it to the process algebraic language mCRL2, and then applying model checking to verify certain behavioral properties. We have implemented a prototype based on this approach. As a case study, we have applied it on a standard example from the ToolBus distribution, distributed auction, and detected a number of behavioral irregularities in this auction Tscript.

1 Introduction

ToolBus [1,2] provides a simple, service-oriented view on organizing software systems by separating the *coordination* of software components from the actual *computation* that they perform. It organizes a system along the lines of a programmable software bus. Programming is done using the scripting language Tscript that is based on the process algebra ACP (Algebra of Communicating Processes) [3] and abstract data types. The tools connected to the ToolBus can be written in any language and can run on different machines.

A Tscript can be tested, as any other software system, to observe whether it exhibits the desired behavior. An alternative approach for analyzing communication protocols is model checking, which constitutes an automated check of whether some behavioral property is satisfied. This can be, roughly, a safety property, which must be satisfied throughout any run of the system, or a liveness property, which should eventually be satisfied in any run of the system. To perform model checking, the communication protocol must be specified in some

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 308–323, 2010.
© Springer-Verlag Berlin Heidelberg 2010

formal language, and the behavioral properties in some temporal logic. Strong points of model checking are that it attempts to perform an exhaustive exploration of the state space of a system, and that it can often be fully automated.

As one of the main aims of Tscript, Bergstra and Klint [2] mention that it should have "a formal basis and can be formally analyzed". The formal basis is offered by the process algebra ACP, but ways to formally analyze Tscripts were lacking until recently [4]. There a number of obstructions for an automatic translation from Tscript to ACP were classified, and solutions were proposed. Firstly, each Tscript process has a built-in queue to store incoming messages, which is left implicit in the process description; in mCRL2, all of these queues are specified explicitly as a separate process. Secondly, Tscript supports dynamic process creation; in mCRL2, we chose to start with a fixed number of ToolBus processes, and let a master process divide connecting tools over these processes. Thirdly, we expressed the iterative star operator of Tscript as a recursive equation in mCRL2. And fourthly, we developed some guidelines on how to deal with so-called result variables in Tscript.

The work in [4] was initiated by the developers of the ToolBus, who are keen to integrate model checking into the design process. Based on [4], we have now implemented a prototype translation from Tscript into the formal modeling language mCRL2 [5]. This language is also based on the process algebra ACP, extended with equational abstract data types [6]. As a result, Tscript can then be model checked using the mCRL2 or CADP toolset [7].

We report on an exploratory case study, to investigate in how far the automated translation from Tscript to mCRL2 can serve as a way to formally verify Tscripts. The case study concerns a distributed auction, in which the auction master and the bidders are cooperating from different computers. This auction Tscript has been used extensively for teaching purposes at various universities and in numerous demonstrations of the ToolBus. We translated the Tscript of the auction system to mCRL2, and analyzed the resulting model with several different approaches. We performed on-the-fly model checking with CADP. On-the-fly means that only the part of the state space needed for checking a property is generated; this is essential here, because the state space of the translated auction system is infinite. Moreover, we enriched the model with behavior from the environment, containing an error action that is triggered if a certain series of events occurs. To perform symbolic model checking, we translated the model and the property that we wanted to check into a Parametrized Boolean Equation System [8], and analyzed this symbolic object with the mCRL2 toolset.

This analysis revealed two deadlocks and a race condition in the auction system. First of all, a deadlock occurs when the master process is busy with a sale, and a new bidder connects to the system, but disconnects very quickly. In this case, the master wishes to synchronize with the bidder process, and will wait indefinitely for this synchronization. A second deadlock occurs since processes that subscribe to certain types of notes in the ToolBus, may never unsubscribe. This can happen when a process terminates after completing its task. Although not considered to be a real error, it does show up in our analysis. A third and

more serious error is that a bidder can, for a very short time slot, bid for the last item that has already been sold, while the master interprets this as a bid for the next item. Finally, we discuss a possible Denial of Service attack. We proposed fixes for the problems we found, and verified that with these fixes the system behaves correctly.

This paper is set up as follows. Section 2 gives a brief overview (taken from [4]) of the ToolBus and Tscript, and presents the auction example. Section 3 gives a brief overview (taken from [4]) of mCRL2 and CADP. Section 4 discusses the translation scheme from Tscript to mCRL2 that originates from [4]. Section 5 presents an analysis of the auction example using this translation scheme. Finally, Section 6 contains conclusions.

Related Work. Our work has its origins in the formal verification of interface languages [9,10]. The aim is to get a separation of concerns, in which the (in our case Tscript) interfaces that connect software components can be analyzed separately from the components themselves. Our work is closest in spirit to Pipa [11], an interface specification language for an aspect-oriented extension of Java called AspectJ [12]. In [11] it is discussed how one could transform an AspectJ program together with its Pipa specification into a Java program and JML specification, in order to apply existing JML-based tools for verifying AspectJ programs, see also [13].

Many publications on model checking and other verification experiments in process algebra present one of the following two setups. Either verification of a (hand-made) *model* is presented, without an implementation in mind, or a model is *reverse engineered* from the source code of a working system, and then analyzed. Here we mention the works that focus on using process algebra for both the (*forward*) development and the analysis of a system.

ToolBus is not the only system where process algebra is used as a scripting language to describe coordination of software components. Many of these put focus on the architectural design as well as on obtaining the working executable system by either code generation or interpretation of process algebra. Some of these make use of the verification possibilities process algebra-based tools like FDR2, CADP, μCRL, mCRL2 and CWB offer.

In [14] a distributed Java system based on CSP is proposed. In [15] a methodology for control system implementation is proposed based on the ideas of [14]. In [16] Analytical Software Design (ASD) method based on Sequence-Based Specifications (SBS) [17] is presented. As demonstrated in [18], the method allows for verified software development where a CSP model is generated from SBSs and verified in FDR2. Yet another CSP-based approach is CSP++ [19], which is a C++ library for executing CSP models.

The most recent version of CAESAR from the CADP toolset provides a functionality called EXEC/CAESAR for C code generation. This C code interfaces with the real world, and can be embedded in applications. This allows rapid prototyping directly from the LOTOS specification. The implementation of the process algebraic formalism χ [20], for modeling and analyzing the dynamics and control of, for instance, production plants, is also centered around the ToolBus.

In [21] a method of software integration based on χ is presented. It allows to generate source code and test cases from χ models.

As related work in the context of the ToolBus, Diertens [22,23] uses the ToolBus to implement a platform for simulation and animation of process algebra specifications in the language PSF. In this approach, Tscript is automatically generated from a PSF specification.

2 ToolBus and Tscript

The behavior of the ToolBus consists of the parallel composition of a variable number of processes. In addition to these processes, a variable number of external tools written in different languages may be connected to the ToolBus via network sockets or OS level pipes. All interactions between processes and connected tools are controlled by Tscripts, which are based on predefined communication primitives. The classical procedure interface (a named procedure with typed arguments and a typed result) is thus replaced by a more general behavior description.

A Tscript process is built from the standard process algebraic constructs: atomic actions (including the deadlock `delta` and the internal action `tau`), alternative composition $+$, sequential composition $\cdot$ and parallel composition $\|$. The binary star operation $p * q$ represents zero or more repetitions of p, followed by q. Atomic actions are parametrized with data parameters (see below), and can be provided with a relative or absolute time stamp. A process definition is of the form $Pname(x_1, \ldots, x_n)$ `is` P, with P a Tscript process expression and $x_1, \ldots, x_n$ a list of data parameters. Process instances may be created dynamically using the `create` statement.

The following communication primitives are available. A process can send a message (using `snd-msg`), which should be received, synchronously, by another process (using `rec-msg`). Furthermore, a process can send a note (using `snd-note`), which is broadcast to other, interested, processes. A process may `subscribe` and `unsubscribe` to certain notes. The receiving processes read notes asynchronously (using `rec-note`) at a low priority. Processes only receive notes to which they have subscribed. Communication between ToolBus and tools is based on handshaking communication. A process may send messages in several formats to a tool (`snd-eval`, `snd-do`, `snd-ack-event`), and can receive values (`rec-value`) and events (`rec-event`) from a tool.

The only values that can be exchanged between the ToolBus and connected tools are terms of some sort (basic data types booleans, integers, strings and lists). In these terms, two types of variables are distinguished: *value* variables whose value is used in expression, and *result* variables (written with a question mark) who get a value assigned to them as a result of an action or a process call. Manipulation of data is completely transparent, i.e., data can be received from and sent to tools, but inside ToolBus there are hardly any operations on them. ATerms [24] are used to represent data terms; ATerms support maximal subterm sharing, and use a very concise, binary format. In general, an adapter is

needed for each connected tool, to adapt it to the common data representation and message protocols imposed by ToolBus.

The ToolBus was introduced in the mid-1990s for the implementation of the ASF+SDF Meta-Environment [25,26] but has been used for the implementation of various other systems as well. The source code and binaries of the ToolBus and related documentation can be found at `www.meta-environment.org`.

2.1 The Auction Example

Consider a completely distributed auction, in which the auction master and the bidders are cooperating via a workstation in their own office. Challenges are how to synchronize bids, how to inform bidders about higher bids, and how to decide when the bidding is over. In addition, bidders may connect and disconnect from the auction whenever they want. This example is described in full detail in [2]. Since that time it has become a standard application of the ToolBus, most of all it has been used extensively in teaching and demonstrations. Its architecture is shown in Fig. 1, where ToolBus processes are represented by ellipses.

The auction is initiated by the process `Auction`, which executes the `master` tool (the user interface used by the auction master), and then handles connections and disconnections of new bidders, the introduction of new items for sale at the auction, and the actual bidding process. A delay is used to determine the end of the bidding activity per item. A `Bidder` process is created for each new `bidder` tool that connects to the auction; it describes the possible behavior of the bidder. The auxiliary process `ConnectBidder`, which handles the connection of a new bidder to the auction, consists of the following steps:

- Receive a connection request from some bidder. This may occur when someone executes a bidder tool outside the ToolBus (possibly on another computer). As part of its initialization, the bidder tool will attempt to make a connection with the ToolBus system running the auction Tscript.
- Create an instance of the process `Bidder` that defines the behavior of this particular bidder.
- Ask the bidder for its name, and send that name to the auction master.

The auxiliary process `OneSale` handles all steps needed for the sale of one item:

- Receive an event from the master tool announcing a new item for sale.
- Broadcast this event to all connected bidders, and perform one of the following four steps as long as the item is not sold:
  - Receive a new bid from one of the bidders. If the bid is too low, reject it and inform the bidder. If the bid is acceptable, inform the bidder and notify all bidders that a higher bid has been received.
  - Ask for a final bid if no bids were received during the last 10 seconds.
  - Declare the item sold if no new bids arrive within 10 seconds after asking for a final bid.
  - Connect a new bidder.

The Tscript of this auction system takes care of the issues mentioned earlier, i.e., synchronizing bids, informing bidders, and completing a sale.

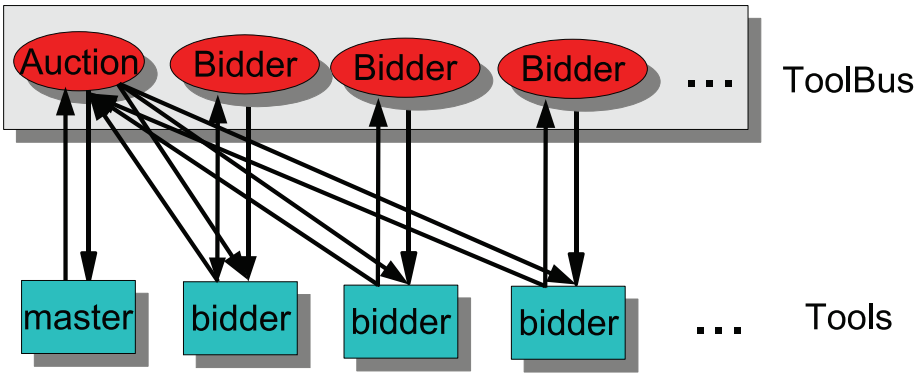

Fig. 1. Architecture of the auction application

3 mCRL2 and CADP

An mCRL2 [5] specification is built from the standard process algebraic constructs: atomic actions (including the deadlock δ and the internal action τ), alternative composition $+$, sequential composition $\cdot$ and parallel composition $\|$. One can define synchronous communication of actions. The following two operators combine data with processes. The sum operator $\sum_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for values d of sort D. The conditional operator $_ \rightarrow _ \diamond _$ describes the *if-then-else*. The process $b \rightarrow x \diamond y$ (where b is a boolean) has the behavior of x if b is true and the behavior of y otherwise.

Data elements are terms of some sort. Next to equational abstract data types, mCRL2 also supports built-in functional data types. Atomic actions are parametrized with data parameters, and can be provided with an absolute time stamp. A process definition is of the form $\mathsf{Pname}(x_1, \ldots, x_n) = P$, with P an mCRL2 process and $x_1, \ldots, x_n$ a list of parameters.

The mCRL2 toolset (`www.mcrl2.org`) supports formal reasoning about systems specified in mCRL2. It is based on term rewriting techniques and on formal transformation of process algebraic and data terms. mCRL2 specifications are first transformed to a linear form [5, Section 5], in a *condition-action-effect* style. The resulting specification can be simulated interactively or automatically, there are a number of symbolic optimization tools, and the corresponding Labeled Transition System (LTS) can be generated. This LTS can, in turn, be minimized modulo a range of behavioral semantics, and model checked with the mCRL2 toolset or the CADP toolset [7].

4 From Tscript to mCRL2

Both Tscript and mCRL2 are based on the process algebra ACP [3]. In spite of this common origin, the languages have some important differences. In [4], we proposed how these differences can be bridged. For instance, the binary

star operation in TScript can be encoded by means of recursive equations in mCRL2. And dynamic process creation in TScript can be modeled in mCRL2 by statically fixing the maximal number of process instances that can be active simultaneously; these process instances are present from the start, and the master process divides connecting tools over these processes. And the notion of discrete time in TScript can be modeled using a tick action synchronization (cf. [27,28,29]). Here we go over two of the main differences, and show how they relate to the auction example.

Asynchronous Communication. According to the semantics of the ToolBus, each process created by TScript has a queue for incoming notes. A `rec-note` will inspect the note queue of the current process, and if the queue contains a note of a given form, it will remove the note and assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-note` occurs.

mCRL2 contains no built-in primitives for asynchronous communication. Therefore, in mCRL2, note queues are handled by a separate AsyncComm process. It also takes care of subscriptions/unsubscriptions and lets any process send any note at any time. Any process can inspect its queue for incoming notes by synchronously communicating with AsyncComm.

$$
\begin{aligned}
&\mathsf{AsyncComm}(\mathit{subscribers}{:}\mathbf{L}(\mathit{Pid}), \mathit{queues}{:}\mathbf{L}(\mathbf{L}(\mathit{Msg}))) = \\
&\quad \textstyle\sum_{p:\mathit{Pid}} \mathsf{r_subscribe}(p) \cdot \mathsf{AsyncComm}(\mathit{subscribers} \triangleleft p, \mathit{queues}) \\
&\quad + \textstyle\sum_{p:\mathit{Pid}} \mathsf{r_unsubscribe}(p) \cdot \\
&\qquad \mathsf{AsyncComm}(\mathit{rem_list_elem}(\mathit{subscribers}, p), \mathit{set_elem}(\mathit{queues}, p, [\,])) \\
&\quad + \textstyle\sum_{\mathit{note}:\mathit{Msg}} \mathsf{r_snd_note}(\mathit{note}) \cdot \\
&\qquad \mathsf{AsyncComm}(\mathit{subscribers}, \mathit{distr_note}(\mathit{queues}, \mathit{subscribers}, \mathit{note})) \\
&\quad + \textstyle\sum_{p:\mathit{Pid}} \sum_{\mathit{ntype}:\mathit{NoteType}} \\
&\qquad (p < \#\mathit{queues} \wedge \mathit{has_note_of_type}(\mathit{queues}.p, \mathit{ntype})) \rightarrow \\
&\qquad \mathsf{s_rec_note}(p, \mathit{get_first_of_type}(\mathit{queues}.p, \mathit{ntype})) \cdot \\
&\qquad \mathsf{AsyncComm}(\mathit{subscribers}, \mathit{set_elem}(\mathit{queues}, p, \\
&\qquad\qquad \mathit{rem_first_of_type}(\mathit{queues}.p, \mathit{ntype})))
\end{aligned}
$$

The process AsyncComm is parametrized by the list of *subscribers* (for the sake of simplicity we assume that processes can subscribe/unsubscribe to all notes simultaneously), and by the list of note queues containing the pending notes for the subscribed processes. The four summands of the process definition reflect the four actions that AsyncComm react upon.

The first two summands handle the subscription and unsubscription. A process willing to subscribe performs $\mathsf{s_subscribe}(\mathit{id})$ action that synchronizes to the $\mathsf{r_subscribe}(p)$ action of the first summand. This can only happen for the value of p that is equal to id, and as a result of this action the id is added to *subscribers*. In a similar way an id of the unsubscribing process is removed from *subscribers* and its queue is emptied.

The third summand says that a sent note is distributed into the queues of all subscribers. The fourth summand deals with reception of a note by a process p. It can only happen if its queue has a note of the appropriate type. In

this case the first note of this type is taken from the queue, and is delivered to process p.

4.1 Structure of the Translator

The actual translation program is implemented as a sequence of transformation steps. The first step performs unfoldings of Tscripts and some other syntactic sugar removals as a Tscript to Tscript transformation implemented in ASF [26].

The simplified Tscript is then compiled by a part of the ToolBus system to an internal representation, containing a finite automata representation for each process in the Tscript. Each state n of such a representation is translated to an mCRL2 process of the form

$$\begin{aligned}\mathsf{P}_n(\overrightarrow{v{:}V}) = {} & \textstyle\sum_{\overrightarrow{v_1{:}V_1}} c_1(\overrightarrow{v}) \rightarrow \mathsf{a}_1(\overrightarrow{f_1}(\overrightarrow{v}), \overrightarrow{v_1}) \cdot \mathsf{P}_{n(1)}(\overrightarrow{g_1}(\overrightarrow{v}, \overrightarrow{v_1})) \\ & + \\ & \cdots \\ & + \textstyle\sum_{\overrightarrow{v_k{:}V_k}} c_k(\overrightarrow{v}) \rightarrow \mathsf{a}_k(\overrightarrow{f_k}(\overrightarrow{v}), \overrightarrow{v_k}) \cdot \mathsf{P}_{n(k)}(\overrightarrow{g_k}(\overrightarrow{v}, \overrightarrow{v_k}))\end{aligned}$$

where k is the number of outgoing transitions from state n and for any transition i such that $1 \leq i \leq k$, $n(i)$ is the next state of process P. The vector $\overrightarrow{v{:}V}$ represents the local variables of process P in state n. The vectors $\overrightarrow{v_i{:}V_i}$ represent the input variables (if any) that are being assigned by performing the action a_i of transition i. These variables are used to determine the values of the local variables in the next state using the vector of functions $\overrightarrow{g_i}(\overrightarrow{v}, \overrightarrow{v_i})$.

As the final step, the standard parts, like the AsyncComm process, are added to the generated mCRL2 model. As a result the generated mCRL2 model performs the actions of the Tscript. By performing deadlock or reachability analysis one can obtain a trace to an undesirable state of the mCRL2 model. The actions in this trace map directly to the actions of the Tscript.This gives a possibility to locate and correct the problem in the Tscript. The modified Tscript can be translated to mCRL2 again and the analysis can be repeated. In this iterative way one can get a Tscript where all formulated behavioral properties are satisfied. Executing this Tscript and performing some tests of the working system can reveal additional problems that can also be formulated as behavioral properties and checked with the mCRL2 level.

5 Analysis of the Auction System

We translated the Tscript of the auction system to mCRL2 using the prototype translator. A small example is given in Figure 2.

The structure of the resulting mCRL2 model is presented in Figure 3. Here each ToolBus process is represented by an mCRL2 process, and an extra process AsyncComm is added to model the asynchronous communication of ToolBus.

`Tscript` fragment

```
   tool bidder is {}
%% Declaration of the tool type "bidder"

   process ConnectBidder is
      let Bidder : bidder in
         rec-connect(Bidder?) . snd-msg(new(Bidder)) ...
      endlet
%% Suppose the auction runs on location ($HOST, $PORT). Then a tool of
%% type "bidder" can be launched on any client by entering the command:
%% wish-adapter $HOST $PORT -TB_TOOL_NAME bidder ...
```

becomes `mCRL2` fragment

```
   sort bidder; Msg = struct new(bidder)|any-higher-bid|... ;
%% Tool types are represented in mCRL2 as sorts. "Msg" is a sort
%% added to the mCRL2 specification to represent a Tscript message.

   act  rec-connect : bidder; snd-msg : Msg;
%% Some mCRL2 declarations of Tscript built-in actions.

   proc ConnectBidder() = sum(Bidder:bidder,
      rec-connect(Bidder) . ConnectBidder1(Bidder));
%% "sum(Bidder:bidder" introduces a local variable "Bidder" of sort "bidder".
%% "rec-connect" communicates with "snd-connect" defined in an environment.

   ConnectBidder1(Bidder : bidder) = snd-msg(new(Bidder)) ...
%% Here "Bidder" is initialised with the value received by "rec-connect".
```

Fig. 2. Fragment of a translation from TSCRIPT into mCRL2

The mCRL2 translation of the auction TSCRIPT has been analyzed[1] for the presence of deadlocks and some other behavioral properties. This revealed two deadlocks and a race condition. Moreover, we encountered a possible Denial of Service attack. We proposed fixes for the detected problems and verified that with these fixes the system behaves correctly.

Finding 1: two deadlocks due to a fast disconnect. One deadlock occurs when a newly connected bidder disconnects immediatly instead of sending its name. In this case the newly created `Bidder` process handles the disconnect, and the `Master` process keeps waiting for the bidder's name forever. This problem can be resolved by postponing the creation of the `Bidder` process till after the reception of the name of the new bidder.

Another problem occurs when the `Master` process is busy with `OneSale`, and a new bidder connects to the system. After executing `ConnectBidder(Mid,Bid?)`, the `Master` process attempts to do `snd-msg(Bid,new-item(Descr,HighestBid))`. This has to synchronize with the `rec-msg` of the connected new bidder. In case that bidder has already disconnected by that time, the synchronization is impossible and the `Master` process and the whole system deadlocks. Many solutions are possible for this problem. The one we chose consists of two parts.

[1] The source code of the auction script and the full mCRL2 model can be found at `www.win.tue.nl/~yusenko/sources/Auction/sources.zip`

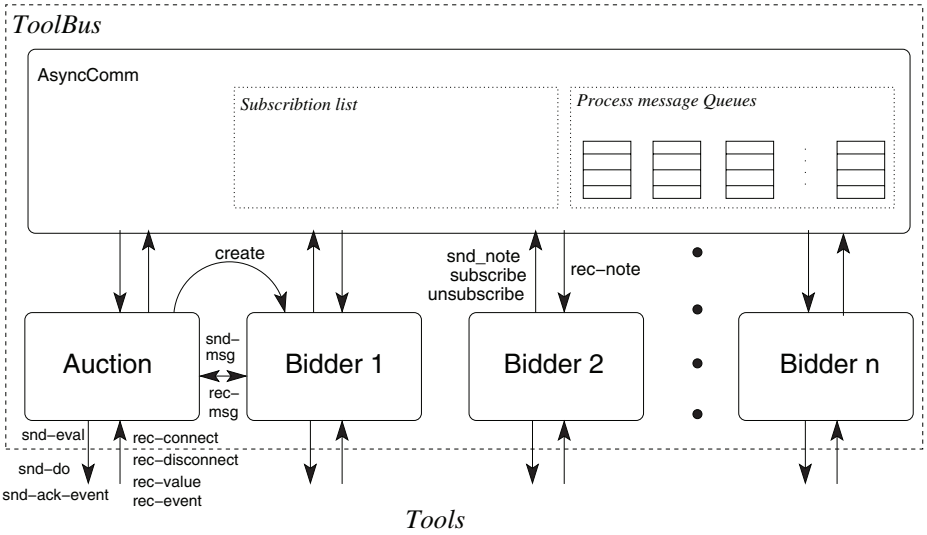

Fig. 3. Auction Tscript in mCRL2

1. The `Bidder` process has to perform a `rec-msg` before disconnecting. This patch alone, however, does not solve the problem, since it introduces another one. In case the bidder tool connects *not* during an ongoing sale, the `rec-msg` would wait for synchronization forever. That is why we need another patch as well.
2. When a new bidder connects at a moment that there is no sale, the `Master` process does `snd-msg(Bid,no-new-item)`. The newly-created `Bidder` process waits for either `new-item` or a `no-new-item` message before proceeding further, or receiving a disconnect from its tool.

After bringing this fix into the Tscript, we could regenerate the mCRL2 model and verify that this deadlock has been resolved.

Finding 2: a missing ***`unsubscribe`****.* The `Bidder` process contains no `unsubscribe` commands, also not before successful termination. This situation can be seen as a violation of the stylistic constraint that every subscribe command has a corresponding unsubscribe command. We found this situation by generating the underlying LTS of our mCRL2 model with the `lps2lts` tool. The tool reported a deadlock situation: due to the way we modeled the process creation/termination mechanism in mCRL2, the missing `unsubscribe` command lead to a deadlock. To be more precise, in case a new `Bidder` tool connects to the system after the missing `unsubscribe` command, the corresponding `Bidder` process cannot perform `subscribe` any longer. We resolved this problem by adding the missing `unsubscribe`.

Finding 3: buying the next item while bidding for the previous one. A third and more serious error that we detected is that a bidder can, for a very short time

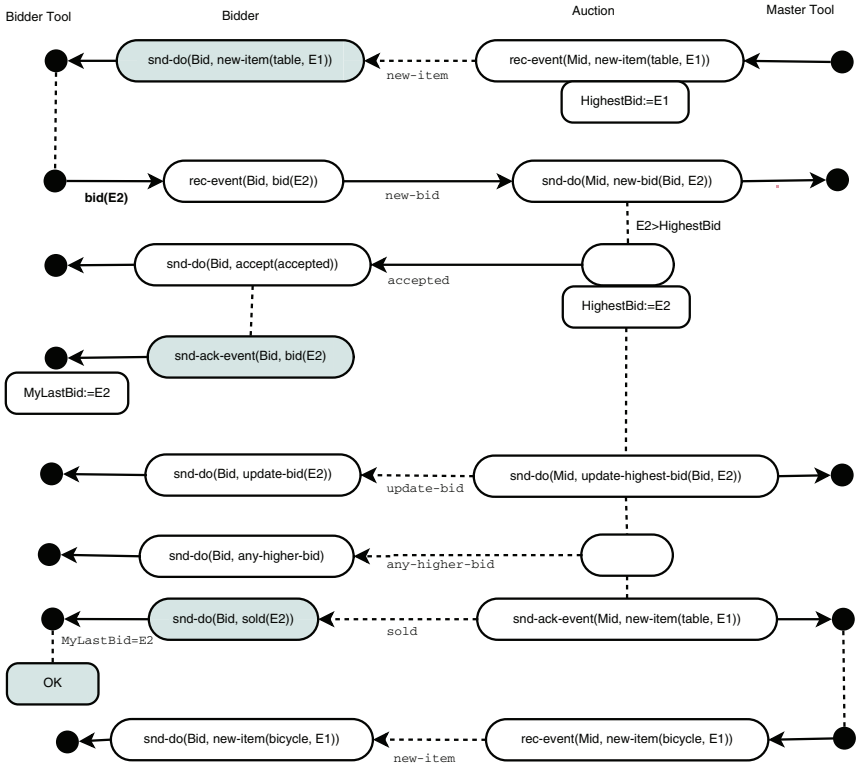

Fig. 4. Normal external behavior

slot, bid for the last item that has already been sold, while the master interprets this as a bid for the next item.

We demonstrate this race condition with a small example. Suppose that the auction master sells a table and a bicycle, both for the price of `E1`. Bidder `B1` intends to bid `E2` (a larger amount than `E1`) for the table. In the next paragraphs we explain the desired behavior of the auction, and possible erroneous behavior that may occur in this situation.

The desired scenario for the aforementioned example is as follows. After a new item (the table) is presented at the auction for the price of `E1`, the bidder bids `E2`. The bid is accepted, and the bidder is informed about this fact. Then every bidder receives a note that the current price is raised to `E2`. Bidder `B1` does not bid anymore, receives a note `any-higher-bid`, and receives a note that the item is sold for the price of `E2`. From this bidder `B1` can derive that he has bought the table. This sequence of events is depicted in Fig. 4.

However, using model checking, we found the following erroneous scenario. After the table is presented at the auction for the price of `E1`, the item is sold for this price. During the selling of that item, three broadcasts are performed

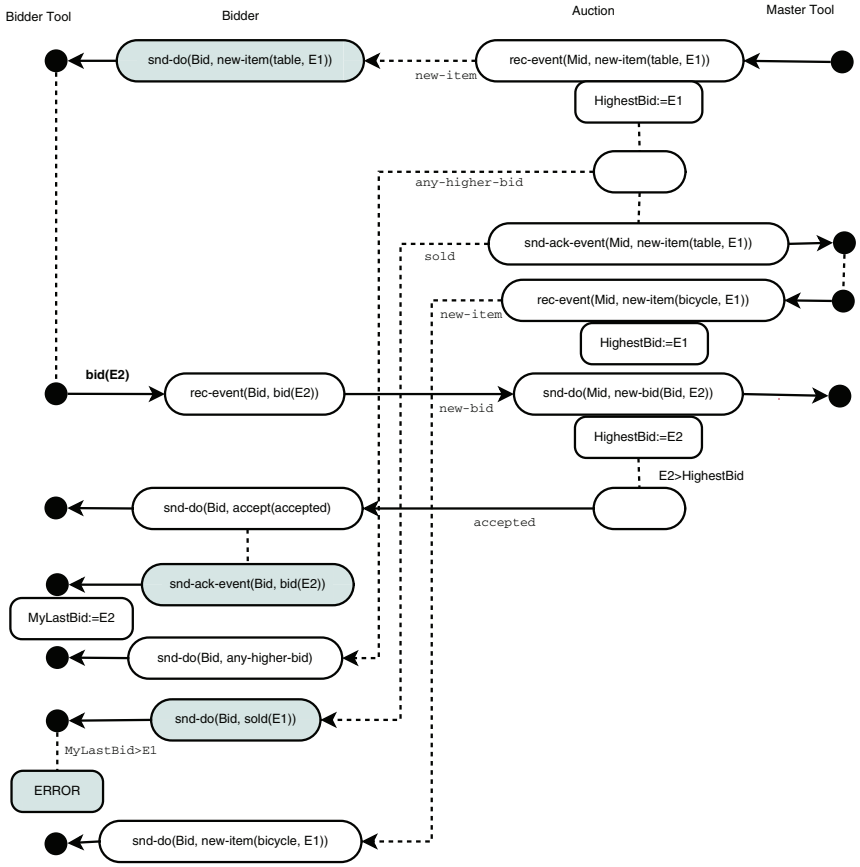

Fig. 5. Erroneous external behavior

to the bidders: `new-item`, `any-higher-bid`, and `sold`. Bidder `B1` bids `E2` under the illusion that he bids on the table, because the note `sold` has not arrived yet. The bid gets accepted, but the auction master thinks that this is a bid for the next item, being the bicycle. This sequence of events is depicted in Fig. 5.

To find this issue, we used the following property:

> If a `Bidder` tool bids m and this bid gets accepted (`accept(accepted)` message is received), then the following `sold(`n`)` message it receives should be such that $m \leq n$.

We used two methods to check this property. One way is in formulating the property as a μ-calculus [30] formula and performing on-the-fly model checking with the mCRL2 toolset or with the CADP tool `evaluator` [7]. Another way is in adding an observer process in parallel to the system. The process will observe the relevant actions in the way the environment (`Bidder` tool) does it. In case

the property violation is detected, the observer performs the `s_error` action. The presence of this action and a shortest trace to it can be detected with the mCRL2 tool `lps2lts` during the LTS generation. Concretely, the relevant part of the observer process look as follows:

$$
\begin{aligned}
&\mathsf{ToolBidder_1}(tid{:}TId, accepted{:}Bool, my_last_bid{:}Int) = \\
&\quad \textstyle\sum_{msg:Msg} \mathsf{r_ack_event}(tid, msg)\cdot \\
&\qquad (accepted \wedge is_bid1(msg)) \rightarrow \mathsf{ToolBidder_1}(tid, false, bid1_0(msg)) \\
&\qquad\qquad \diamond\, \mathsf{ToolBidder_1}(tid, accepted, my_last_bid) \\
&+ \textstyle\sum_{msg:Msg} \mathsf{r_do}(tid, msg)\cdot \\
&\qquad (is_sold0(msg) \rightarrow \\
&\qquad\quad ((my_last_bid > sold0_0(msg)) \rightarrow \mathsf{s_error}\cdot \\
&\qquad\qquad\qquad \mathsf{ToolBidder_1}(tid, accepted, 0) \\
&\qquad\qquad \diamond\, \mathsf{ToolBidder_1}(tid, accepted, 0) \\
&\qquad\quad) \\
&\qquad \diamond\, is_accept0(msg) \rightarrow \\
&\qquad\quad ((accept0_0(msg) = trm_accepted) \rightarrow \\
&\qquad\qquad\qquad \mathsf{ToolBidder_1}(tid, true, my_last_bid) \\
&\qquad\qquad \diamond\, \mathsf{ToolBidder_1}(tid, false, my_last_bid) \\
&\qquad\quad) \\
&\qquad \diamond\, \mathsf{ToolBidder_1}(tid, accepted, my_last_bid) \\
&\qquad) \\
&+ \ldots
\end{aligned}
$$

Here the first summand reacts to the r_ack_event action and, in case the last bid has been accepted, sets the value of the *my_last_bid* parameter to the amount in the last bid. The second summand reacts to r_do. Here two messages of the TSCRIPT are important: `sold` and `accept`. The latter one is used to set the *accepted* parameter to represent the fact that our last bid has been accepted. The former one leads to the check if our property $my_last_bid > sold0_0(msg)$ holds. In case it does not, the s_error action is executed.

We believe that this problem can be resolved in many different ways. However, not much can be done without increasing the level of detail in the communication protocol between the auction TSCRIPT and the tools. We propose a solution to this problem without changing this protocol. We extend the `bid` message that is synchronously communicated from the `Bidder` process to the `Auction` process with the description of the item for which the bid is valid. In case the `Auction` process receives a bid for a wrong item, it rejects this bid.

Additionally we had to add an extra condition to the case when a new bidder connects during an ongoing sale. Namely, we make it impossible to connect in this way if the item has already been sold. In this case the connection is performed after this sale round is finished.

To verify the fact that this solution actually works we had to decorate the `accept` and `sold` messages that are sent to the `Bidder` tool with the description and the amount information. An important assumption for our solution to work is that the consecutive sale items must have different descriptions. We could verify that the resulting TSCRIPT does not have the erroneous behavior.

An important observation related to this problem has been proposed by an anonymous reviewer. The root of the problem lays in the fact that a bid comes into a race condition with the `sold` message. In case a bid comes late and there is no next item to be sold, the bidder will have to wait for the rejection of its bid forever. To avoid this problem the reviewer proposed to allow a choice between `snd-msg(bid(...,...))` and `rec-note(sold(...))`. We implemented this fix and checked that the rejection is always received by the bidder tool.

Finding 4: infinite queues. Although using on-the-fly model checking we could detect some important issues, we could not analyze the entire behavior of the auction Tscript, due to the fact that its LTS is infinite. We could handle some sources of infinity, like infinite domains for data types, by bounding these domains.

Another source of infinity has to do with the asynchronous communication and queues for notes. For example, one process may keep sending notes while another process is not willing to receive them. Such a situation can happen in case one of the bidders keeps bidding very actively. The following part of the `Bidder` process illustrates such a phenomenon.

```
( rec-event(Bid, bid(Amount?)).
  ...
  + rec-note(update-bid(Amount?)) .
    snd-do(Bid, update-bid(Amount))
  + rec-note(any-higher-bid) . snd-do(Bid, any-higher-bid)
  + rec-disconnect(Bid).delta ) *
```

In case a `Bidder` tool is active in sending bids, the first alternative is enabled and can be chosen. As a result, the parallel OneSale process adds an `update-bid` note to the message queue of our `Bidder` process. For any fixed-size note queue, an overflow can be reached by executing a sufficiently large number of first alternatives without ever taking the second one.

This can be seen as a problem that has to be solved by the scheduler of the ToolBus. Another way to look at this problem is to see it as a possibility of a Denial of Service attack, leading to an overflow.

Some of such queue size problems can be tackled with the help of timing. We could impose that no time can progress as long as a process can receive a note (so-called maximal progress). However, the aforementioned problem can happen without reception of any notes and, therefore, without any progress of time at all. To solve this issue we chose to limit the number of bids per time unit that the `Bidder` process is willing to accept from its tool. By combining the two timing restrictions into the mCRL2 model, we could get to a finite LTS.

6 Conclusions and Future Work

Our general aim is to have a process algebra-based software development environment where both formal verification and production of an executable system is possible. We implemented a prototype translation from Tscript to mCRL2.

This translation makes it possible to verify TScript in an automated fashion, and to explore behavioral properties of executable software systems that have been built with the ToolBus.

We automatically translated a standard TScript application, a distributed auction, to mCRL2, and analyzed it using on-the-fly and symbolic model checking techniques. As a result, four flaws were detected in the original TScript description of this auction system. We could fix all the issues, and verified the correctness of the fixed TScript by automatically translating it to mCRL2. We could also execute and test the fixed model to ensure it still works.

In the future we aim at applying the presented techniques to analyze a large existing TScript with the help of model checking. An example of such a system is the the ASF+SDF Meta-Environment [25,26]. It is also of our interest to develop a new TScript from scratch in a way that formal verification with mCRL2 contributes to every stage of the development process.

References

1. Bergstra, J.A., Klint, P.: The ToolBus coordination architecture. In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 75–88. Springer, Heidelberg (1996)
2. Bergstra, J.A., Klint, P.: The discrete time ToolBus - a software coordination architecture. Sci. Comput. Program. 31(2-3), 205–229 (1998)
3. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
4. Fokkink, W., Klint, P., Lisser, B., Usenko, Y.S.: Towards formal verification of ToolBus scripts. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 160–166. Springer, Heidelberg (2008)
5. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The formal specification language mCRL2. In: Proc. Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings, vol. 06351 (2007)
6. Bergstra, J.A., Heering, J., Klint, P.: Module algebra. J. ACM 37(2), 335–372 (1990)
7. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
8. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci. 343(3), 332–369 (2005)
9. Wing, J.M.: Writing Larch interface language specifications. ACM Trans. Program. Lang. Syst. 9(1), 1–24 (1987)
10. Guaspari, D., Marceau, C., Polak, W.: Formal verification of Ada programs. IEEE Trans. Software Eng. 16(9), 1058–1075 (1990)
11. Zhao, J., Rinard, M.C.: Pipa: A behavioral interface specification language for aspectJ. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 150–165. Springer, Heidelberg (2003)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)

13. Larsson, D., Alexandersson, R.: Formal verification of fault tolerance aspects. In: Supplementary Proceedings of International Symposium on Software Reliability Engineering (ISSRE) Conference, pp. 279–280. IEEE, Los Alamitos (2005)
14. Hilderink, G.H., Bakkers, A.W.P., Broenink, J.F.: A distributed real-time Java system based on CSP. In: Proc. ISORC 2000, pp. 400–410. IEEE, Los Alamitos (2000)
15. Orlic, B., Broenink, J.F.: Design Principles of the SystemCSP Software Framework. In: McEwan, A.A., Ifill, W., Welch, P.H. (eds.) CPA 2007, pp. 207–228 (2007)
16. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. Electr. Notes Theor. Comput. Sci. 128(6), 127–144 (2005)
17. Prowell, S.J., Poore, J.H.: Foundations of sequence-based software specification. IEEE Trans. Software Eng. 29(5), 417–429 (2003)
18. Broadfoot, G.H.: Asd case notes: Costs and benefits of applying formal methods to industrial control software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 548–551. Springer, Heidelberg (2005)
19. Doxsee, S., Gardner, W.B.: Synthesis of C++ software from verifiable CSPm specifications. In: Proc. ECBS 2005, pp. 193–201 (2005)
20. Beek, B., Man, K.L., Reniers, M., Rooda, K., Schiffelers, R.: Syntax and consistent equation semantics of hybrid χ. J. Log. Algebr. Program. 68(1-2), 129–210 (2006)
21. Braspenning, N.C.W.M., van de Mortel-Fronczak, J.M., Rooda, J.E.: A model-based integration and testing method to reduce system development effort. Electr. Notes Theor. Comput. Sci. 164(4), 13–28 (2006)
22. Diertens, B.: Simulation and animation of process algebra specifications. Technical Report P9713, University of Amsterdam (1997)
23. Diertens, B.: Software (re-)engineering with PSF III: An IDE for PSF. Technical Report PRG0708, University of Amsterdam (2007)
24. van der Brand, M., de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. Softw., Pract. Exper. 30(3), 259–291 (2000)
25. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM 2(2), 176–201 (1993)
26. den van Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
27. Fokkink, W., Ioustinova, N., Kesseler, E., van de Pol, J., Usenko, Y.S., Yushtein, Y.A.: Refinement and verification applied to an in-flight data acquisition unit. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 1–23. Springer, Heidelberg (2002)
28. Blom, S., Ioustinova, N., Sidorova, N.: Timed verification with μCRL. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 178–192. Springer, Heidelberg (2004)
29. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In: ICECCS, pp. 35–46. IEEE, Los Alamitos (2007)
30. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. 46(3), 255–281 (2003)