Transactor-based Formal Verification of Real-time Embedded Systems

D. Karlsson, P. Eles, Z. Peng
Department of Computer and Information Science, Linköpings universitet, Sweden
{danka, petel, zebpe}@ida.liu.se

ABSTRACT

With the increasing complexity of today's embedded systems, there is a need to formally verify such designs at mixed abstraction levels. This is needed if some components are described at high levels of abstraction, whereas others are described at low levels. Components in single abstraction level designs communicate through channels, which capture essential features of the communication. If the connected components communicate at different abstraction levels, then these channels are replaced with transactors that translate requests back and forth between the abstraction levels. It is important that the transactor still preserves the external characteristics, e.g. timing, of the original channel. This paper proposes a technique to generate such transactors. According to this technique, transactors are specified in a single formal language, that is capable of capturing timing aspects. The approach is especially targeted to formal verification.

1. INTRODUCTION

Developers of embedded systems face an ever-increasing complexity of their designs. At the same time, they also face an ever-decreasing time-to-market. A common way to deal with this challenge is to divide the design into several components, each component with its own responsibilities and functionality. These components can either be reused from a repository or be developed in-house. In the former case, the developers do not have to design the whole system from scratch, but they can rely on partial designs of some functionality developed previously. However, in some cases, it might be more efficient to develop the desired, customised, functionality from scratch.

This divide-and-conquer technique is usually combined with an iterative top-down approach, where the system is initially defined at a high level of abstraction, leaving out most low-level details. The design is then gradually refined and more and more details are put into place. During this process, some parts of the system will be described at high-level, and other parts at low level.

This situation, together with the fact that verification and test consume a significant part of the total development cost, stresses the need for efficient verification methods that target systems described at mixed abstraction levels.

The above-mentioned problem is traditionally solved in an unsystematic manner, where developers rewrite properties and modify the system in an ad hoc manner in order to match the mixed level model. Lately, a more systematic approach, involving *transactors*, has been proposed [4], [5].

The key issue of the problem lies in the fact that two (or more) components described at different abstraction levels cannot communicate with each other, since they, in principle, use different protocols. One component uses a more high-level protocol than the other. A transactor is a mechanism that bridges this gap by translating the high-level requests into their low-level ditto and vice versa. Moreover, evaluations have shown that using a transactor-based verification approach is more effective than a traditional RTL verification flow with respect to both fault and assertion coverage [1]. Using transactors moreover helps in reusing testbenches as well as assertions in the refinement process.

A few work have been performed in the area of automatically generating this type of transactors, based on protocol conversion techniques [2], [3]. Bombieri et al. [4] start from a master-bus-slave communication framework that contains information on how communication is carried out at different abstraction levels on the specified infrastructure (bus). From this framework, the authors extract a master, bus or a slave transactor from a high to low level or vice versa. Their extraction algorithm is based on Extended Finite State Machines. It does, however, not handle timing aspects explicitly and is only applicable on bus-based protocols.

Balarin et al. [5] use Sequential Extended Regular Expressions (SERE) to specify the relation between the two interfaces of the transactor and to automatically generate the corresponding transactor. The transactors are generated in a programming language such as C++, Verilog or SCE-MI, in order to facilitate integration with existing simulation tools. The approach supports to a lesser extent formal methods, and it completely lacks the support for time.

Protocols are often described using various kinds of regular expression-like languages. Although SEREs [5] in principle are sufficiently expressive, they do not support the notion of time. Timed Regular Expressions [7], on the other hand, lack several useful features, such as variables and conditions.

The approach proposed in this paper combines SEREs with timed regular expressions by adding a timing feature on top of SEREs. We call the resulting language Timed SERE (TSERE). By doing this, we are able to create transactors suitable for formal verification in a component-based real-time setting with mixed abstraction levels. The approach moreover widens the scope of responsibility of transactors from a pure protocol converter to a semi-refined communication channel.

The paper is organised into 7 sections. Section 1 introduces and motivates transactor-based verification. Next, Section 2 provides an overview of the proposed approach. Section 3 presents the Petri-net based design representation that is used throughout the paper, and Section 4 defines the Timed Sequential Extended Regular Expression language that is used for specifying transactors. Section 5 describes the mechanism to generate timed Petri-nets from the formal description and Section 6 presents a few case studies. Section 7 concludes the paper.

2. OVERVIEW

In the proposed approach, a system consists of several communicating components, as indicated in Figure 1. Each component implements a well-defined functionality, and they interact with other components and the rest of the system through ports, depicted in the figure with circles at the edges of the component.

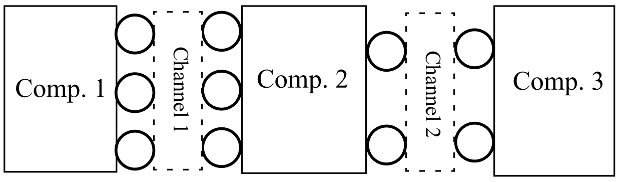


Figure 1. Targeted System Topology

Channels are inserted between communicating components. The channels model the protocol, delays, noise and other peculiarities that can occur in the communication. They are hence only an artefact for high-level models, that will not occur or be synthesised in the final implementation. Channels can, from a modelling point of view, be regarded as a special type of components, and are depicted with dotted lines. Though not exemplified in the figure, channels can connect an arbitrary number of components.

During the development phase, it is often desirable to check if certain temporal logic properties are satisfied in the system. Such analysis can be obtained by feeding a model of the system into a model checking tool together with properties to be verified. This procedure gives a formal proof whether the properties are satisfied in the system or not [8].

At the same time, the components are iteratively refined and more and more details are added to the system. This naturally leads to a situation where some parts of the system are more refined than others. However, it is still desirable to occasionally verify the system to ensure that the recently performed refinement steps did not violate any, possibly critical, properties.

When refining the components, the interfaces of those components are simultaneously refined. However, the interfaces are shared or connected with other components, that are not yet refined. This creates an incompatibility of interfaces between the involved components and channels. In order to overcome this problem, the channel is replaced by a *transactor* between the incompatible interfaces, as demonstrated in Figure 2. A transactor can thus be seen as a channel connecting components at different levels of abstraction, or a semi-refined channel. The transactor shall encapsulate the same external behaviour as the channel it replaces with respect to delays, noise etc.

The transactor takes high-level requests and translates them into low-level ones, and vice versa. It is described in Timed Sequential Extended Regular Expressions (TSERE), which is both intuitive and sufficiently expressive for this purpose. The TSEREs (and thereby also the transactors) are given either by the designer himself, or, in a standardised context, by a third-party provider.

The example in Figure 3 will be used to explain the approach in more detail. A sender repeatedly sends messages to a receiver over a channel. At a high level of abstraction (Figure 3(a)), it takes 2 time units for the message to be transported between the two components. This delay is implemented in the channel interconnecting the components.

At a low level of abstraction (Figure 3(b)), the message is refined into two: address and data. The protocol that the sender and receiver have agreed upon states that these messages should be sent sequentially with 1 time unit in between. It moreover takes 1 time unit for each message to reach the receiver. The sender thus sends the data at the same time as the address reaches the receiver. It should be noted that the total timeframe for sending a message in the two abstraction levels is the same. In both cases, this takes 2 time units. Thus, the channel preserves its external behaviour between abstraction levels.

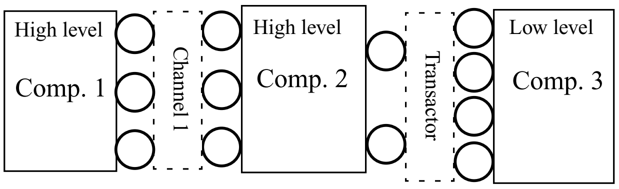


Figure 2. System at mixed abstraction level with transactor

At one moment, during the refinement phase, only one of the components is refined. Assume that this component is the receiver (Figure 3 (c)). At this stage, the sender and receiver adhere to different protocols and cannot communicate with either of the high-level or low-level channels. Instead, the channel is replaced with a transactor that translates the high-level message into the stipulated sequence of low-level ones. The transactor consequently has to analyse the message from the sender and divide it into two. The first message should contain the destination address, whereas the second one should contain the data. The transactor then forwards the two pieces to the receiver with 1 time unit difference.

The transactor can be said to be a mix of the two versions of the channel. It, however, also contains additional protocol information not explicit in the channels, e.g. how to split the high-level message and the time separation between the address and data transmission. Therefore, the information captured in the channels is not sufficient for formulating the TSEREs. In addition, the transactor respects the external timing behaviour of the channels.

3. VERIFICATION FLOW AND DESIGN REPRESENTATION

This section introduces the verification flow and the Petri-net based design representation used in this paper.

3.1 Verification Flow

Figure 4 presents the overall verification flow where the work described in this paper is put into context. The flow centers around a component-based verification methodology [8], which accepts three entities as input: a mixed-level model, transactor and Timed Computation

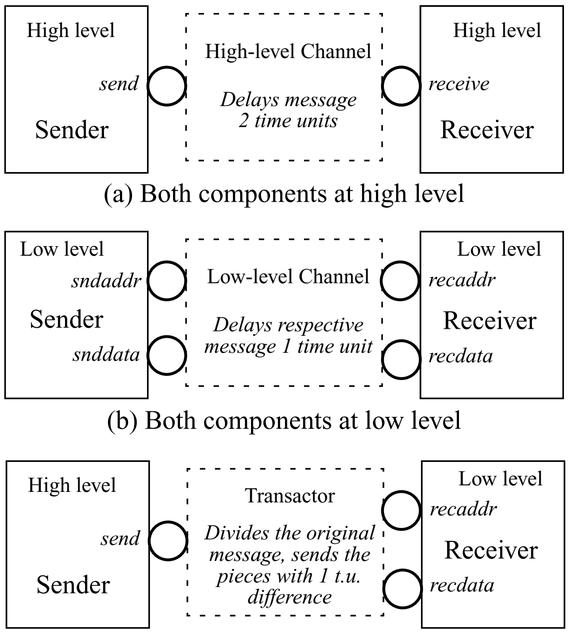

High level
send
Sender
High-level Channel
Delays message 2 time units
High level
receive
Receiver
(a) Both components at high level
Low level
sndaddr
Sender
snddata
Low-level Channel
Delays respective message 1 time unit
Low level
recaddr
Receiver
recdata
(b) Both components at low level
High level
send
Sender
Transactor
Divides the original message, sends the pieces with 1 t.u. difference
Low level
recaddr
Receiver
recdata
(c) Sender at high level, Receiver at low level

Figure 3. Explanatory example

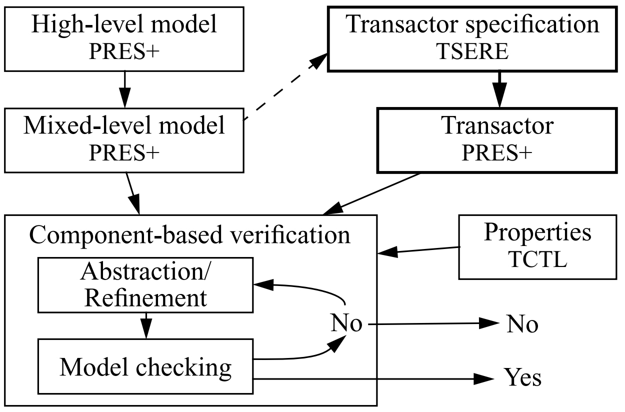


Figure 4. Verification flow

Tree Logic (TCTL) properties [9].

The mixed-level model is obtained from traditional refinement steps of a high-level model. The designer then writes TSEREs describing the communication discrepancies arised from the mixed abstraction levels in the semi-refined design and generates a transactor out of them (the focus of this paper). The TCTL formulas express the real-time properties to be verified.

In the verification methodology, an abstraction of the model is first obtained with respect to the components and channels referred to by the properties. The abstracted model is then input to the UPPAAL model checker [10], by first translating the Petri-net model [11] into Timed Automata [12], the input language of UPPAAL. If the result of the model checking was false, the model might need to be refined (relative to the abstraction done in the verification methodology, not the design itself) based on diagnostic information obtained from the model checker. In case the refinement of the abstraction fails, the properties are concluded not to be satisfied. If, on the other hand, the model checking result was true, it can be concluded that the properties hold in the model.

3.2 The Design Representation: PRES+

The components as well as the system as a whole are assumed to be modelled in a design representation called *Petri-net based Representation for Embedded Systems* (PRES+) [11]. It is a Petri-net based representation with the extensions listed below. Figure 5 shows an example of a PRES+ model.

1. Each token has a value and a timestamp associated to it.
2. Each transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. If the time delay interval is not explicitly stated, it is assumed to be [0..0]. In Figure 5, the functions are marked on the outgoing edges from the transitions.
3. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.

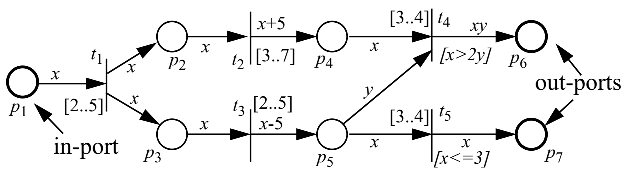


Figure 5. A simple PRES+ net

4. The transitions may have guards. A transition can only be enabled if the value of its guard is true (transitions t_4 and t_5).

Places without incoming arcs are called *in-ports*, and places without outgoing arcs are called *out-ports*. A common name for in-ports and out-ports respectively, is *ports*. Components are subnets of the whole model, delimited by ports.

4. TIMED SEQUENTIAL EXTENDED REGULAR EXPRESSIONS

The proposed approach introduces Timed Sequential Extended Regular Expressions (TSEREs) for the specification of transactors. TSEREs consist of three types of entities: basic entities, terms and operators.

4.1 Basic Entities

Basic entities cannot be standalone TSEREs, but constitute a part of terms. They are used as building blocks for storage, communication and computation. The 3 categories of basic entities are shown below:

1. Variables: *a*, *b*, *c*
   Variables are used to store and retrieve values. Variables are associated to a datatype. Unless explicitly stated otherwise, the datatype used in all examples is integer. The scope of a variable stretches from its first occurrence to the end of the sequence (see the sequence operator below) of that first occurrence.
2. Port labels: *!send*, *?rec*
   Port labels are used to define the interaction with other components. *!* denotes the sending of a (possibly empty) message on the subsequent out-port, and *?* denotes receiving of a message from the specified in-port.
3. Arithmetic expressions: $(a + b) \cdot 3$
   Arithmetic expressions perform a computation on other basic entities, following standard syntax. This entity allows to express data processing.

4.2 Terms

Terms describe an action by combining basic entities. There are 3 different types of terms, listed below:

1. Assignments: $a \leftarrow 3$, $!send \leftarrow 0$, $b \leftarrow ?rec$
   The variable or out-port on the left-hand side of the arrow is updated to the value of the variable, in-port or arithmetic expression on the right-hand side.
2. Guards: $a = 4$, $?rec > 10$
   Guards compare the value of a variable or in-port with the evaluation of an arithmetic expression. If the guard evaluates to true, nothing happens. Otherwise, the TSERE fails (or, loosely speaking, reaches a dead end).
3. Delays: [0..0], [3..5]
   Delays denote the passing of time. They are expressed as intervals, with the connotation that an arbitrary amount of time from the interval may elapse. This feature is crucial in the context of real-time systems.

4.3 Operators

In addition to terms, TSEREs can be recursively combined to express more complex behaviour with the following operators. Assume α and β being arbitrary TSEREs.

1. Sequence: $\alpha;\beta$
   α occurs immediately before β.
2. Choice: $\alpha + \beta$
   Either α or β occurs.
3. Concurrency: $\alpha|\beta$, $\alpha^{|n}$
   α and β occur concurrently. The concurrency operator is not considered to have occurred until both α and

β have fully occurred. $\alpha^{|n}$ denotes n concurrent copies of α.

4. Iteration: α^n, α^∞, α^*, α^+
   The iteration operators denote a sequence of recurring α. The length of that sequence depends on the type of iteration. α^n denotes a sequence of length n and $n = \infty$ signifies an infinitely long sequence. Such a sequence can only be escaped if placed inside the choice operator. α^* denotes a sequence where n is arbitrarily chosen between $0 \le n \le \infty$, and in the case of α^+, n is arbitrarily chosen from $1 \le n \le \infty$.

4.4 Example

Returning to the example introduced in Figure 3, the high-level and low-level channels and the transactor can be expressed with the following TSEREs:

1. High-level channel: $(m \leftarrow ?send;[2..2];!rec \leftarrow m)^\infty$
2. Low-level channel:
   $(a \leftarrow ?sndaddr;[1..1];!recaddr \leftarrow a;$
   $d \leftarrow ?snddata;[1..1];!recdata \leftarrow d)^\infty$
3. Transactor:
   $(m \leftarrow ?send;[1..1];!recaddr \leftarrow m.addr;$
   $[1..1];!recdata \leftarrow m.data)^\infty$

The infinite iteration on the whole expression is necessary to enable the transactor to process several requests. Without the iteration, the transactor and channels would stop working after the first request.

As another example, consider a variant of the low-level channel where either the address and data are sent simultaneously, or we receive a reset request. (eq. 1) shows the corresponding TSERE.

$$(((a \leftarrow ?sndaddr;[1..1];!recaddr \leftarrow a)| (d \leftarrow ?snddata;[1..1];!recdata \leftarrow d)) + ?reset)^\infty \quad \text{(eq. 1)}$$

If statements can be expressed using guards together with the choice operator. In combination with iteration, this structure allows formulating bounded loops, as demonstrated in (eq. 2).

$$\alpha^n \Leftrightarrow i \leftarrow 0;((i < n;\alpha;i \leftarrow i+1)^\infty + (i = n)) \quad \text{(eq. 2)}$$

5. TRANSACTOR GENERATION

To generate a transactor is a two-step process. First, the behaviour of the transactor must be described with TSEREs. This must be done in such a way that each high-level request is mapped onto low-level ones, while preserving the external behaviour, e.g. timing. Once a TSERE for the transactor is developed, that TSERE is automatically translated into an equivalent PRES+ model. This section provides details on how this is done.

Regular expression based languages have a very strong relation with finite automata (and therefore also with PRES+), which makes such conversion relatively straight-forward [13]. Each basic entity, term and operator is mapped onto a PRES+ pattern, which directly reflects the semantics of that entity. The patterns have one entry place and one exit place, indicated in figures by a loose incoming and outgoing arc respectively. A token arriving in the entry place of a pattern enables the execution of that pattern, i.e. the occurrence of its corresponding TSERE. After executing the pattern/expression, a token should, by convention, be put in the exit place to indicate its completion. Figure 6 presents the patterns corresponding to basic entities, Figure 7 the patterns corresponding to the terms and Figure 8 the patterns corresponding to the operators.

5.1 Patterns for Basic Entities

Variables are represented by a place (Figure 6(a)), initially without a token. When the variable is assigned a value for the first time, and the variable enters its scope, a token containing the initial value is put in the place. From that point on, a token shall always reside in that place during the whole lifetime of the variable. The last term in the sequence, where the scope of possibly several variables ends, should consume the tokens in the places corresponding to those variables. This procedure reduces statespace by not storing values when not needed, and therefore mitigates the effects of statespace explosion. This is particularly important when it comes to formal verification and model checking.

Port labels are also modelled with a single place (Figure 6(b)). These places will serve as ports of the transactor. *?* labels serve as in-ports and *!* labels as out-ports. Therefore, the transactor can only consume tokens from *?* label ports, and analogously only put tokens in *!* label ports.

Arithmetic expressions are modelled in two stages: fetching variable values and computation (Figure 6(c)). The value of each variable involved in the expression must be explicitly fetched and stored in a temporary place. This arrangement is due to the fact that PRES+ transitions only are associated to one function. Without the fetching steps, the involved variables would change values to the value of the expression, which is not the desired behaviour.

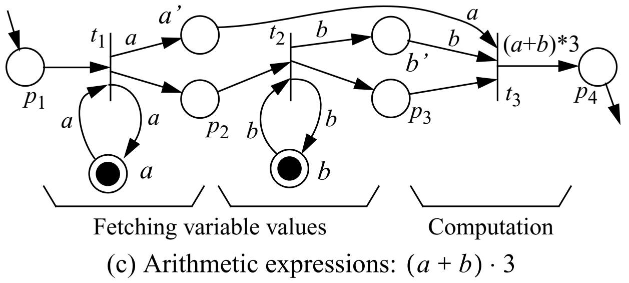
The fetching of variable values is realised by transitions t_1 and t_2 in Figure 6(c), for variables a and b respectively. The transitions consume the token from the variable place and immediately puts it back with the same value. In the case of *?* port labels, the token is never put back. A copy of the value is moreover stored in a temporary place, *a'* and *b'* respectively. These tokens are then used in the final computation stage, transition t_3, instead of directly accessing the variable places. The fetching stages and the final computation stage are connected in a sequence with the help of intermediate places, p_1 to p_4. The result of the expression is located in the exit place of the arithmetic expression.

5.2 Patterns for Terms

Assignments are realised in a similar way as variable fetching, with the difference that the value of the token is updated (Figure 7(a)). The new value is located in the entry place in the case of arithmetic expression, or, in the case of a constant, the transition function is set to that constant. Attention must be paid to if the assignment denotes the initial assignment to the variable in question or not. If it is, there is no token in the variable place to be

(a) Variables: a

(b) Port labels: *!send*, *?rec*



(c) Arithmetic expressions: $(a + b) \cdot 3$

Figure 6. PRES+ patterns for TSERE basic entities

consumed and consequently there shall not be an arc from the place to the transition. If the assignment is an update of an already initialised variable, the token must, on the contrary, be consumed before the update is actuated. In the case of *!* port labels, tokens are never consumed from within the transactor. As an optimization when the new value is an arithmetic expression, the assignment can be merged with the computation stage of the arithmetic expression.

Guards are implemented as variable fetching without creating a temporary copy, with the addition that the transition guard is set to the TSERE guard expression (Figure 7(b)).

Delays are modelled with a transition with the time delay interval stipulated by the TSERE delay expression (Figure 7(c)). The modelling of delays is preferably optimised by moving the time delay interval to the first transition of the subsequent TSERE, if such exists.

5.3 Patterns for Operators

The operator patterns combine several subpatterns to form a more complex behaviour. In Figure 8, the subpatterns are drawn as clouds with arrows from/to its entry and exit places. The resulting complex pattern is also assigned entry and exit places, indicated in the figures in the same way as with the terms.

Sequences are realised by merging the exit place of the first subpattern with the entry place of the second (Figure 8(a)). The entry place of the first subpattern becomes the entry place of the whole sequence, and the exit place of the second subpattern becomes the exit place of the whole sequence. In this way, when the first subpattern has finished executing, a token is put in the shared middle place, which enables the execution of the second subpattern.

In the pattern for the choice operator (Figure 8(b)), the entry and exit places of the subpatterns are merged, so that all subpatterns share the same entry place and the same exit place. When a token appears in the entry place, this leads to the enabling of all subpatterns, out of which one is chosen randomly. If the first term of a subpattern is a guard that evaluates to false, that subpattern can naturally not be chosen.

When a token arrives in the entry place of the concurrency pattern (Figure 8(c)), the entry places of each subpattern must also be marked to enable the execution of each corresponding subpattern. This is achieved by introducing an additional transition (t_1) with the entry places of all subpatterns as output and the entry place of the whole pattern as input. A similar, but contrary, construct is also inserted at the exit places (t_2), implementing the synchronisation of the subpatterns upon their completion. The concurrency operator is not considered completed until all subexpressions are completed.

Iteration is accomplished by connecting the exit place of the subpattern to its entry place via a transition (t_1 in Figure 8(d)). This procedure can, in the case of α^∞ and α^*, be optimised by instead merging the entry and exit places of the subpattern. The entry place of the subpattern is also the entry place of the iteration. For α^* iterations, the exit place is the same as the entry place, whereas for α^+ the exit place of the iteration is the exit place of the subpattern. α^∞ iterations do not have an exit place due to their infinite nature.

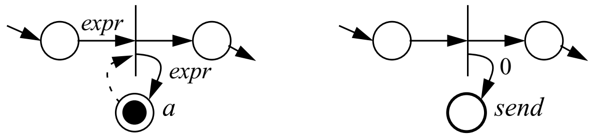
Finite loops are implemented using a loop counter (Figure 8(e)), which is initially set to 0 (t_1) and increased by one after each iteration (t_2). Guards on transitions t_2 and t_3 ensure that the loop is broken when the loop counter has reached n, the specified number of iterations.

When a PRES+ model has been generated for the whole TSERE, an initial token is put in the entry place of the final model, to indicate the first term.

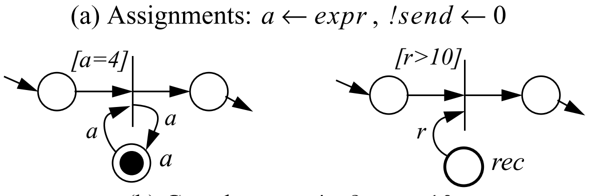
5.4 Examples

Let us continue the sender and receiver example introduced in Figure 3, and where the TSEREs for the channels were listed in Section 4.4. Figure 9 provides the PRES+ models resulting from the presented approach, including certain optimizations.

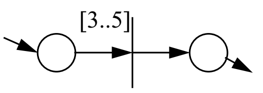
The core of the transactor is a sequence of reading and writing on ports combined with simple arithmetic expressions (Figure 9(a)). Transitions t_2 and t_4 model the variable fetching stages of the arithmetic expressions, while transitions t_3 and t_5 combine the computation stages with



(a) Assignments: $a \leftarrow expr$, $!send \leftarrow 0$

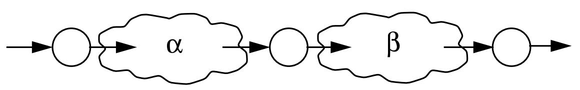


(b) Guards: $a = 4$, $?rec > 10$

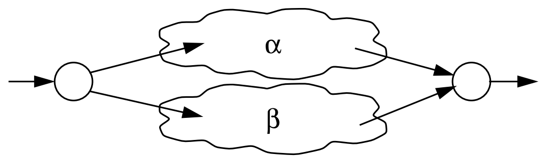


(c) Delays: [3..5]

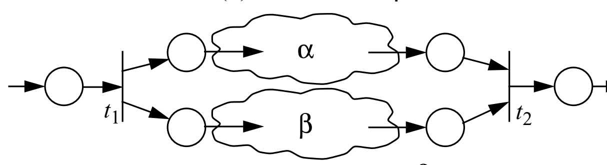
Figure 7. PRES+ patterns for TSERE terms



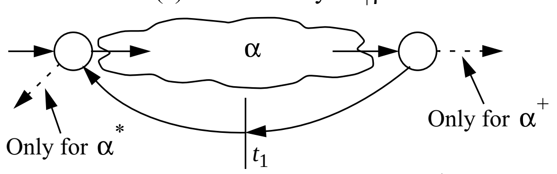
(a) Sequence: $\alpha;\beta$



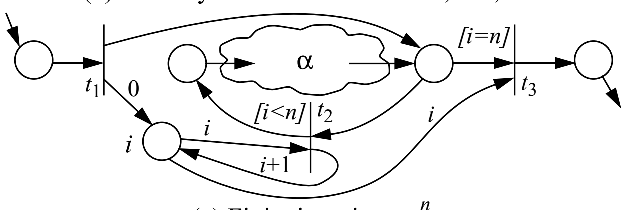
(b) Choice: $\alpha + \beta$



(c) Concurrency: $\alpha|\beta$



(d) Possibly infinite iteration: α^∞, α^*, α^+



(e) Finite iteration: α^n

Figure 8. PRES+ patterns for TSERE operators

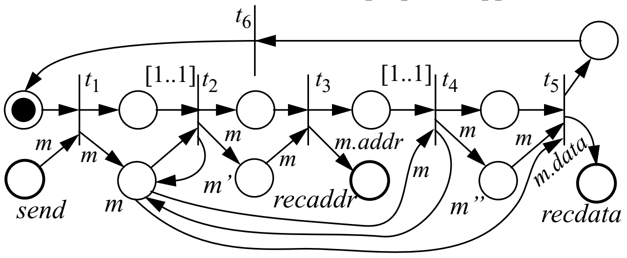
the assignment on ports *recaddr* and *recdata* respectively (optimization). The delays are moreover added to the first transitions in the subsequent terms, in this case t_2 and t_4. It should moreover be noted how the scope of variable *m* is modelled. Transition t_1 realises the first assignment to *m*, therefore it only puts a token with the initial value in place *m*. As transition t_5 is the last transition in its scope, it consumes the token, no matter it needs the value or not. Transition t_6 models the infinite loop.

Figure 9(b) presents the PRES+ model corresponding to (eq. 1). Inside the iteration, there is a choice between either two concurrent statements or a single reading of reset. If the reset is not immediately present, the two concurrent sequences are launched. If the reset is present, there is a non-deterministic choice between the two options. The loop is in this figure optimised in the sense that the exit place of the choice operator is merged with its entry place.

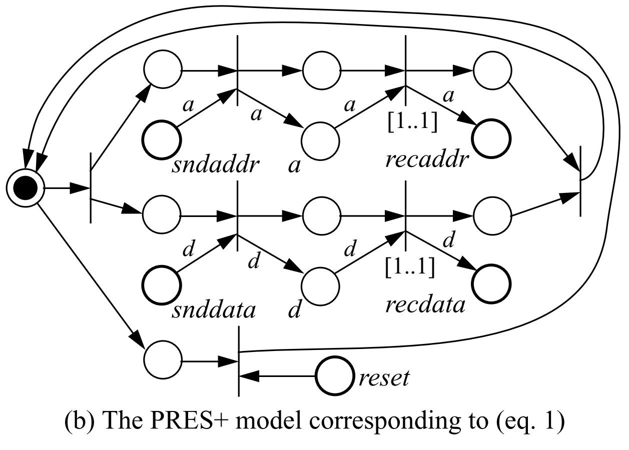
6. CASE STUDIES

The proposed approach has been applied on two examples: the example from Figure 3 and an AMBA-based protocol. The models were formally verified on high, low and mixed levels of abstraction using a Linux machine with an Intel Pentium 4, 2.8GHz processor and 2GB of memory. The AMBA example was moreover verified with different configurations on the number of masters (M) and slaves (S). Both examples were checked for the same two properties: no deadlock and that sent messages will arrive at their destinations.

Table 1 and Table 2 present the verification times in seconds for the respective example. The tables moreover indicate the sizes of the TSEREs, which define the channels/transactors, as the number of terms and operators in the expression. The size of the entire verified PRES+ model is indicated by the number of transitions. These numbers only give a hint to the size of the examples and are not directly related to verification time. These results indicate the reasonableness of the proposed approach.



(a) The generated transactor from Figure 3(c)



(b) The PRES+ model corresponding to (eq. 1)

Figure 9. Examples of PRES+ models generated from TSEREs

7. CONCLUSIONS

This paper has presented an approach to generate transactors for real-time embedded systems, suitable for formal verification. The approach assumes a design where components communicate over channels, and that those channels capture all the characteristics of the communication. During the development, more and more components are refined leading to a model with mixed abstraction levels. In such models, the components cannot directly communicate due to protocol discrepancies. In order to overcome these discrepancies, the channels interfacing components of different abstraction levels are replaced with transactors. The behaviour of the transactors, i.e. the mapping of requests between abstraction levels, is described using TSEREs, which are automatically converted into the design representation used, PRES+. The resulting PRES+ model can then be analysed by a formal verification tool.

8. REFERENCES

[1] N. Bombieri, F. Fummi, G. Pravadelli, "On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL", in *Proc. DATE*, 2006

[2] J. Akella, K. McMillan, "Synthesizing Converters between Finite State Protocols", in *Proc. ICCD*, 1991, pp. 410-413

[3] R. Passerone, J.A. Rowson, "Automatic Synthesis of Interfaces between Incompatible Protocols", in *Proc. DAC*, 1998, pp. 8-13

[4] N. Bombieri, F. Fummi, G. Pravadelli, "A TLM Design for Verification Methodology", in *Research in Microelectronics and Electronics*, 2006, pp. 337-340

[5] F. Balarin, R. Passerone, "Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation", in *Proc. DATE*, 2006

[6] F. Plasil, S. Visnovsky, M. Besta, "Bounding Component Behaviour via Protocols", in *Proc. TOOLS*, 1999, pp. 387-398

[7] E. Asarin, P. Caspi, O. Maler, "A Kleene Theorem for Timed Automata", in *Proc. LICS*, 1997, pp. 160-171

[8] D. Karlsson, P. Eles, Z. Peng, "Formal Verification of Component-based Designs", in *Journal of Design Automation for Embedded Systems*, Vol. 11, No. 1, March 2007, pp. 49-90

[9] R. Alur, C. Courcoubetis, D.L. Dill, "Model Checking for Real-time Systems", in *Theoretical Computer Science*, 1990, pp. 414-425

[10] UPPAAL homepage: http://www.uppaal.com/

[11] L.A. Cortés, P. Eles, Z. Peng, "Verification of Embedded Systems using a Petri Net based Representation", in *Proc. ISSS*, 2000, pp. 149-155

[12] R. Alur, D.L. Dill, "A theory of timed automata", in *Theoretical Computer Science*, 1994, pp. 126:183-235

[13] D. C. Kozen, "Automata and Computability", 1997, Springer Verlag.

Table 1: Results from the example in Figure 3

Abstraction level	TSERE	PRES+	No d.lock	Will arr.
High	4	8	0.12s	0.13s
Low	7	13	0.06s	0.09s
S High - R Low	6	11	0.11s	0.06s

Table 2: Results from the AMBA example

M-S	Abstr. level	TSERE	PRES+	No dlock	Will arr.
1 - 1	High	22	30	0.33s	0.12s
	Low	25	52	0.19s	0.22s
	M High-S Low	23	44	0.19s	0.17s
	M Low-S High	36	53	0.30s	0.40s
1 - 2	High	35	48	0.50s	0.46s
	Low	28	71	0.80s	1.68s
	M High-S Low	26	62	0.24s	0.35s
	M Low-S High	47	69	1.44s	3.57s
2 - 1	High	22	32	0.19s	0.43s
	Low	32	68	0.48s	1.53s
	M High-S Low	23	46	0.38s	0.84s
	M Low-S High	42	69	1.43s	6.59s
2 - 2	High	36	50	5.01s	18.99s
	Low	35	86	5.43s	22.57s
	M High-S Low	26	64	5.39s	17.77s
	M Low-S High	54	85	42.06s	200.5s