Knowledge Based Optimization for Distributed Real-Time Systems

Mahieddine Dellabani, Jacques Combaz, Saddek Bensalem, Marius Bozga

▶ **To cite this version:**

Mahieddine Dellabani, Jacques Combaz, Saddek Bensalem, Marius Bozga. Knowledge Based Optimization for Distributed Real-Time Systems. 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Dec 2017, Nanjing, China. pp.751-756, 10.1109/APSEC.2017.106. hal-01888605

HAL Id: hal-01888605
https://hal.science/hal-01888605v1

Submitted on 5 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Knowledge Based Optimization for Distributed Real-Time Systems

Mahieddine Dellabani, Jacques Combaz, Saddek Bensalem and Marius Bozga
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
Email: {mahieddine.dellabani, jacques.combaz, saddek.bensalem, marius.bozga}@univ-grenoble-alpes.fr
http://www.verimag.fr/rsd

***Abstract*—The design and the implementation of distributed real-time systems has always been a challenging task. A central question being how to efficiently coordinate parallel activities by means of point-to-point communication so as to keep global consistency while meeting timing constraints. In the domain of safety critical applications, system predictability allows to pre-compute optimal scheduling policies. In this paper, we consider a larger class of systems represented as compositions of timed automata subject to multiparty interactions, for which an implementation method for distributed platforms and based on intermediate model transformation already exists. To improve this approach, we developed specific static analysis techniques that, combined with local and global knowledge of the system, checks particular conditions that enables to decrease the number of messages exchanged in the system for executing each interaction, as well as to remove unnecessary scheduling overhead in some cases.**

I. Introduction

Nowadays, real-time systems are widespread and span across several application domains. Such an evolution has introduced a growing urge towards the development of more sophisticated, and thus more complex, real-time systems such as air traffic control and automotive systems, or even to combine formerly isolated systems improving consequently functionalities and reducing costs. This trend has sustained a race for an increasing need of performance (availability, concurrency, resources, etc.), which led to a shift from the use of single processor based hardware platforms, to large sets of interconnected and distributed computing nodes. On the other hand, the emergence of some applications that are intrinsically distributed, for instance networked embedded systems, or other applications that have physical constraints which require the use of several dedicated computing units in specific locations, such as processing sensor data and controlling actuators, corroborates the necessity of this expansion.

Model-based design is one promising approach to deal with such complexity. This approach relies on the same semantics rules used consistently through the flow, to reach a concrete implementation. First, an application model flowing from the specification is established. This high level model defines a platform free abstraction of the application behavior. This abstraction may use some high level primitives that are, in practice, rarely part of the built-in primitives offered by distributed platforms. Then, intermediate model transformations preserving the functional properties of the original model can be progressively derived until reaching the code generation phase. The big challenge being how to switch from a high level model using particular primitives, to an intermediate distributed model, where only messages-passing and internal computations are allowed. In [1], [2], the author extensively studied such transformation (send-receive model) for scheduling multiparty interactions in a distributed real-time context. We contribute to this research field by proposing a compositional verification method for pre-computing local and global knowledge aiming to enhance this method. In particular, we leverage such information to minimize the number of message exchanges and avoid unnecessary scheduling overhead in the proposed protocols. Note that how execution times, communication delays and clocks skew may impact synchronization protocols in the satisfaction of timing constraints is a related topic, but is beyond the scope of this paper.

The rest of the paper is organized as follows. Section II introduces the preliminary definitions of timed automata with respect to multiparty interactions, as well as predicate definitions needed for the rest of the paper. Then, we explain why *conflict* detection between interactions is needed to reach a correct send-receive model (Section III). Thereafter, Section IV first describes the existing method used to determine the set of *conflicting interactions*, then presents our method based on static analysis using compositional verification. Finally, qualitative and quantitative experiments on various examples are given to confirm the interest of the approach.

II. Timed Systems and Properties

In the framework of the present paper, components are timed automata and systems are compositions of timed automata with respect to multiparty interactions. The timed automata we use are essentially the ones from [3], however, slightly adapted to embrace a uniform notation throughout the paper.

Definition 1 (Component): A *component* is a tuple $B = (\mathcal{L}, \ell_0, \mathcal{A}, \mathcal{T}, \mathcal{X}, tpc)$ where $\mathcal{L}$ is a finite set of *locations*, $\ell_0 \in \mathcal{L}$ is an *initial* location, $\mathcal{A}$ a finite set of *actions*, $\mathcal{T} \subseteq \mathcal{L} \times (\mathcal{A} \times \mathcal{C} \times 2^{\mathcal{X}}) \times \mathcal{L}$ is a set of *transitions* labeled with an action, a guard, and a set of clocks to be reset, $\mathcal{X}$ is a finite set of *clocks*, and $tpc : \mathcal{L} \rightarrow \mathcal{C}$ assigns a *time progress condition*, $tpc(\ell)$ to each location $\ell \in \mathcal{L}$, where $\mathcal{C}$ is the set of *clock constraints* defined by the following grammar:

$$\mathcal{C} := \mathit{true} \mid x \sim ct \mid x - y \sim ct \mid \mathcal{C} \wedge \mathcal{C} \mid \mathit{false},$$

with $x, y \in \mathcal{X}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $ct \in \mathbb{Z}$. Time progress conditions are restricted to conjunctions of constraints of the form $x \leq ct$ with $ct \in \mathbb{Z}_{\geq 0}$.

Throughout the paper, we assume components that are deterministic timed automata, that is, at a given location ℓ, for a given action a, there is up to one outgoing transition from ℓ labeled by a. Given a timed automaton $(\mathcal{L}, \ell_0, \mathcal{A}, \mathcal{T}, \mathcal{X}, tpc)$, we write $\ell \xrightarrow{a,g,r} \ell'$ if there exists a transition $\tau = \big(\ell, (a, g, r), \ell'\big) \in \mathcal{T}$. Let $\mathcal{V}$ be the set of all clock valuation functions $v : \mathcal{X} \to \mathbb{R}_{\geq 0}$. For a clock constraint C, $C(v)$ is a boolean value corresponding to the evaluation of C on v. For a valuation $v \in \mathcal{V}$, $v + \delta$ is the valuation satisfying $(v + \delta)(x) = v(x) + \delta$, while for a subset of clocks r, $v[r]$ is the valuation obtained from v by resetting clocks of r, i.e. $v[r](x) = 0$ for $x \in r$, $v[r](x) = v(x)$ otherwise. We also denote by $C + \delta$ the clock constraint C shifted by δ, i.e. such that $(C + \delta)(v)$ iff $C(v + \delta)$.

Definition 2 (Semantics): A component $B = (\mathcal{L}, \ell_0, \mathcal{A}, \mathcal{T}, \mathcal{X}, tpc)$ defines the labeled transition system (LTS) $(\mathrm{Q}, \mathcal{A} \cup \mathbb{R}_{>0}, \rightarrow)$ where $\mathrm{Q} \subseteq \mathcal{L} \times \mathcal{V}(\mathcal{X})$ denotes the *states* of B, and $\rightarrow \subseteq \mathrm{Q} \times (A \cup \mathbb{R}_{>0}) \times Q$ denotes the set of transitions between states according to the rules:

- $(\ell, v) \xrightarrow{a} (\ell', v[r])$ if $\ell \xrightarrow{a,g,r} \ell'$, and $g(v)$ is true.
- $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$ if $tpc(\ell)(v + \delta)$ for $\delta \in \mathbb{R}_{>0}$.

An *execution sequence* of B from a state (ℓ, v) is a path in the LTS starting at (ℓ, v) and that alternates action steps and time steps, that is:

$$(\ell, v) \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_i} (\ell_n, v_n), n \in \mathbb{Z}_{>0}, \sigma \in \mathcal{A} \cup \mathbb{R}_{>0}.$$

We say that a state (ℓ, v) is *reachable* if there is an execution sequence from the initial state (ℓ_0, v_0) leading to (ℓ, v), where v_0 assigns 0 to all clocks. In this paper, we always assume components with *well-formed guards*, that is, transitions $\ell \xrightarrow{a,g,r} \ell'$ satisfy $g(v) \Rightarrow tpc(\ell)(v) \wedge tpc(\ell')(v[r])$ for any $v \in \mathcal{V}$. This ensures that the reachable states always satisfy the time progress conditions, i.e. if (ℓ, v) is reachable then we have $tpc(\ell)(v)$. Notice that the set of reachable states is in general infinite, but it can be partitioned into a finite number, $J \in \mathbb{Z}_{>0}$, of symbolic states [4], [5], [6]. A symbolic state is defined by a pair (ℓ, ζ) where, ℓ is a location of B, and ζ is a zone, i.e. a set of clock valuations defined by a clock constraint (as defined in Definition 1). Efficient algorithms for computing symbolic states and operations on zones are described in [5]. Given symbolic states $\{(\ell_j, \zeta_j)\}_{j \in J}$ of B, the predicate *Reach*(B) characterizing the reachable states can be formulated as: $Reach(B) = \bigvee_{j \in J} \mathsf{at}(\ell_j) \wedge \zeta_j$, where $\mathsf{at}(\ell_j)$ is true on states whose location is ℓ_j, and clock constraint ζ_j is applied to clock valuation functions of states.

We define the predicate $Enabled(a)$ characterizing states (ℓ, v) at which an action a is enabled, i.e. such that $(\ell, v) \xrightarrow{a} (\ell', v')$. It can be written:

$$Enabled(a) = \bigvee_{\ell \in \mathcal{L}_a} \mathsf{at}(\ell) \wedge g.$$

where $\mathcal{L}_a$ is the set of locations enabling action a, i.e. $\mathcal{L}_a = \{\ell \in \mathcal{L} | \ell \xrightarrow{a,g,r} \ell'\}$. In our framework, components communicate by means of *multiparty interactions*. A multiparty interaction is a rendez-vous synchronization between actions of a fixed subset of components. It takes place only if all the participants agree to execute the corresponding actions. Given n components B_i, $i = 1, \dots, n$, with disjoint sets of actions $\mathcal{A}_i$, an interaction is a subset of actions $\alpha \subseteq \cup_{1 \leq i \leq n} \mathcal{A}_i$ containing at most one action per component, i.e. $\alpha \cap \mathcal{A}_i$ is either empty or a singleton $\{a_i\}$. We denote by $\mathrm{part}(\alpha)$, the set of components *participating* in α, that is, $\mathrm{part}(\alpha) = \{B_i\}_{i \in I}$.

In practice, we do not explicitly build compositions of components. We interpret their semantics at runtime by evaluating enabled interactions based on current states of components.

Definition 3 (Semantics of a Composition): Given a set of components $B_i = (\mathcal{L}_i, \ell_0^i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{X}_i, tpc_i)$ and an interaction set γ. The semantics of the composition $S = \gamma(B_1, \cdots, B_n)$ w.r.t. the set of interactions γ, is the LTS $(\mathrm{Q}_s, \gamma \cup \mathbb{R}_{>0}, \rightarrow_\gamma)$ where:

- $\mathrm{Q}_s = \mathcal{L} \times \mathcal{V}(\mathcal{X})$ is the set of global states, where $\mathcal{L} = \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$ and $\mathcal{X} = \bigcup_{i=1}^{n} \mathcal{X}_i$.
- $\rightarrow_\gamma$ is the set of labeled transitions defined by the rules:
  - $(\ell, v) \xrightarrow{\alpha}_\gamma (\ell', v')$ for $\alpha = \{a_i\}_{i \in I} \in \gamma$, if $\forall i \in I.\ (\ell_i, v_i) \xrightarrow{a_i} (\ell'_i, v'_i)$ and $\forall i \notin I.\ (\ell_i, v_i) = (\ell'_i, v'_i)$.
  - $(\ell, v) \xrightarrow{\delta}_\gamma (\ell, v + \delta)$ for $\delta \in \mathbb{R}_{>0}$ if $\forall i \in \{1, \cdots, n\}.\ tpc(\ell_i)(v_i + \delta)$.

For an interaction $\alpha \in \gamma$, we denote by $Enabled(\alpha)$ the predicate characterizing states (ℓ, v) from which α can be executed. It is defined as follow:

$$Enabled(\alpha) = \bigwedge_{i \in I} Enabled(a_i)$$

The definitions of execution sequences and reachable states are trivially extended to compositions of components.

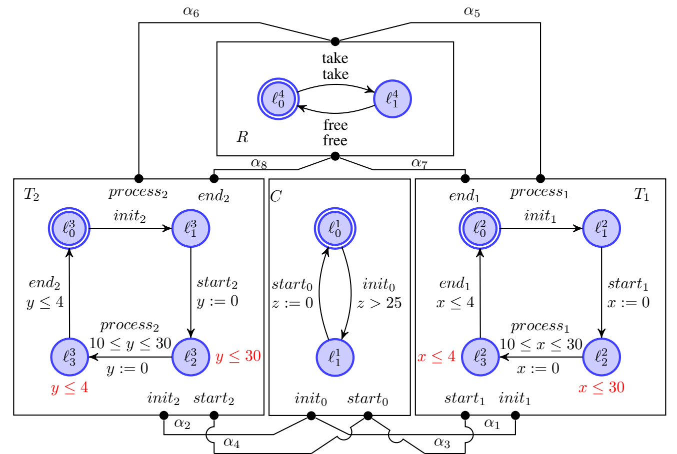

Fig. 1: Task Manager

Example 1 (Running Example): Let us consider as a running example the composition of four components C, T_1, T_2, and R of Figure 1. Component C represents a controller that initializes then releases tasks T_1 and T_2. Tasks use the shared resource R during their execution. To implement such behavior, we consider the following interactions between C,

R, and T_1: $\alpha_1 = \{init_0, init_1\}$, $\alpha_3 = \{start_0, start_1\}$, $\alpha_5 = \{take, process_1\}$, $\alpha_7 = \{free, end_1\}$, and similar interactions α_2, α_4, α_6, α_8 for task T_2, as shown by connections on Figure 1. The controller is responsible for firing the execution of each task. First, it non-deterministically initializes one of the two tasks, i.e. executes α_1 or α_2, and then releases it through interaction α_3 or α_4. Tasks perform their processing independently of the controller, after being granted an access to the shared resource (α_5 or α_6). When finished, a task releases the resource (interactions α_7 or α_8) and go back to its initial location.

III. Send-Receive Model for Distributed Real-Time Systems

In the previous section, we introduced timed automata model that describes a high level representation of systems execution. However, this type of model does not provide any details on how an implementation of multiparty interactions can be derived in a distributed context. In this section, we explain how, from a high level model, a concrete implementation of systems with multiparty interactions can be achieved.

In a distributed context, components communicate through asynchronous message passing. Consequently, components are able to either send a message, to wait for a notification message or to execute an internal computation. In order to obtain such behavior, we follow the approach of [1], [2], in which an intermediate transformation is applied on the initial model, aiming to explicitly express the ongoing communication mechanism. The resulting *send-receive* model describes the execution of each interaction as a two-way handshake protocol involving asynchronous messages exchange between the transformed components and a third party coordinator called *scheduler*. Moreover, to avoid centralized coordination, which is often inefficient, this approach has been extended to the use of several schedulers [1], [2], each one being responsible of a subset of interactions, named an interaction partition (Figure 2). The purpose behind this practice is to: *(i)* spread the workload across independent and concurrent schedulers, and *(ii)* to map schedulers as close as possible to components that they effectively handle (w.r.t. to their interactions partition), which brings back the communication overhead between the components to the same magnitude. Interaction partitioning is not addressed here but is a crucial concern for load-balancing and for tuning the system to achieve a desired level of performance. This transformation describes the following protocol. Once they complete the execution of transitions, components inform schedulers about their current state (i.e. enabled transitions and timing constraints) via *offer* messages. When a scheduler gathers enough offers to determine the execution of an interaction, it sends notification messages to components participating in the selected interaction, informing them thus, about the transitions to be executed.

Example 2: Figure 2 depicts a send-receive model for Example 1 with two schedulers. Scheduler S_1 (resp. S_2) handles the interactions $\{\alpha_1, \alpha_3, \alpha_5, \alpha_7\}$ (resp. $\{\alpha_2, \alpha_4, \alpha_6, \alpha_8\}$). Each component sends information about its current state to its corresponding scheduler through offers (o_i with $i \in \{1, \cdots, 4\}$) and receive a notification from the latter (n_j with $j \in \{1, \cdots, 4\}$). Moreover, schedulers may rely on a third-party arbiter when scheduling *conflicting* interactions. In this case, they emit a request (req_k with $k \in \{1, \cdots, 2\}$) and wait for a notification granting (resp. denying) them the execution of an interaction (ok_k resp. $fail_k$).

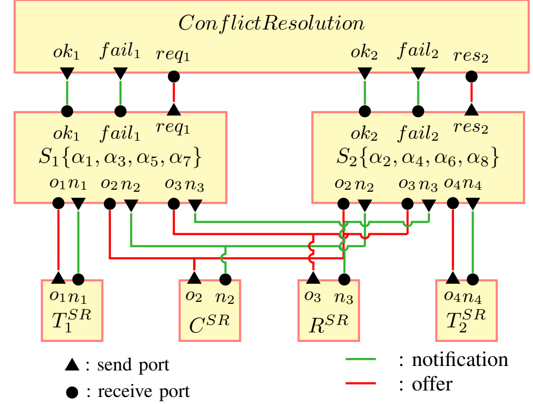

Fig. 2: Send-Receive Model for Example 1 (Two Schedulers)

Conflict Resolution

Decentralizing the schedulers generates situational conflict between interactions, that is, if two interactions handled in separate schedulers share a component, they cannot execute in parallel since they share at least one component. We call such interactions, conflicting interactions. A simple solution to resolve such conflicts is to enforce a *conflict-free* partitioning of interactions, which consists in grouping conflicting interactions together in a scheduler that, internally, will solve the conflict between the interactions as described in [2], and separate only the conflict-free interactions. In spite of that, this solution will restrict the choice for distributing interactions across schedulers. Thus, another method [7] is to use an extra component called *conflict resolution protocol (CRP)*. This component implements an algorithm based on the idea of message-count technique [8]. This technique is based on counting the number of times that a component participates in an interaction. Conflicts are then resolved by ensuring that each participation number is used only once. Effectively, this is achieved by counting the number of the interaction offer for each component. Then, conflict resolution is simply achieved by comparing the offer numbers of participating components with numbers of their last execution.

In what follows, we give a formal definition of conflicts. After that, we explains how the set of conflicting interactions is over-approximated in [2], and propose a compositional verification based method aiming to enhance the conflict detection computation.

IV. Conflict Detection

As explained in previous section, two interactions α and β sharing a subset of components cannot execute concurrently.

However, we consider that they are not conflicting if the composition cannot reach a state from which both can potentially execute. Clearly, such interactions do not require conflict resolution as they cannot be scheduled based on common offers. Conflicts are defined as follows:

Definition 4 (Conflicting Interaction): Let $S = \gamma(B_1, \ldots, B_n)$ be a composition. Two interactions α and β of γ are *conflicting*, if $part(\alpha) \cap part(\beta) \neq \emptyset$ and there exists a reachable state from which both α and β are enabled, i.e. a state satisfying:

$$Conf(S, \alpha, \beta) = Reach(S) \wedge Enabled(\alpha) \wedge Enabled(\beta) \quad (1)$$

As explained earlier, conflicts arising from distributing the execution of interactions across multiple independent schedulers require a third party arbiter (CRP), resulting in additional exchange of messages and calculation overhead.

In general, conflicts as presented in Definition 4 can be very hard to characterize for real life case studies since they depend on the reachability of particular states. In [2], the computation of the conflicting interactions set relies on over-approximation. It depends on a notion of *potential conflicts* that can be detected by simple syntactic pre-checks, and are used to quickly exclude conflicts based on the fact that when two interactions are not potentially conflicting, they are also not conflicting. Particularly, the following syntactic characterization of conflicts is used:

Definition 5 (Potential Conflict): Two interactions α and β are *potentially* conflicting if $part(\alpha) \cap part(\beta) \neq \emptyset$ and for each shared component $B_i \in part(\alpha) \cap part(\beta)$ there exists a location ℓ_i of B_i, and two transitions $\ell_i \xrightarrow{a_i, g_i, r_i} \ell_i'$ and $\ell_i \xrightarrow{b_i, g_i', r_i'} \ell_i''$ from ℓ_i, such that $a_i \in \alpha$ and $b_i \in \beta$.
Hereinafter, *false* conflicts refer to potential conflicts as defined in Definition 5 but that are not conflicts w.r.t Definition 4.

A syntactic characterization of (potential) conflicts can be safely used for scheduling interactions. However, it may lead to unnecessary resolution of (false) conflicts, inducing thus, unnecessary messages exchange and conflict resolution computation. Thus, we propose a more advanced method for detecting conflicts between interactions. Our technique includes two main steps: the first step *(i)* consists of constructing the set of potentially conflicting interactions based on Definition 5. Then, the second step *(ii)* calculates then combines local and global knowledge of the system on the form of invariants. The latter will represents an over-approximation of the reachable states of the system. After that, by replacing $Reach(S)$ by its over-approximation $\widetilde{Reach(S)}$ in Equation 1, it verifies whether potentially conflicting interactions are conflicting or not by checking the following pre-condition:

$$\widetilde{Conf(S, \alpha, \beta)} = \widetilde{Reach(S)} \wedge Enabled(\alpha) \wedge Enabled(\beta) \quad (2)$$

Notice that since $Reach(S) \Rightarrow \widetilde{Reach(S)}$, we obtain that $Conf(S, \alpha, \beta) \Rightarrow \widetilde{Conf(S, \alpha, \beta)}$.

A potential conflict of two interactions is a *false* conflict either: *(i)* because the system cannot reach a global location enabling both interactions, or *(ii)* because both interactions are not enabled at the same time due to timing constraints. In the following, we show how to compute invariants for removing false conflicts of types *(i)* and *(ii)*. This invariants combined with individual reachable states of components will represent our over-approximation.

A. Removing False Conflicts using Linear Invariants

Locations reachable in a composition $S = \gamma(B_1, \ldots, B_n)$ are necessary combinations of reachable locations of individual components B_i. However, in general not all combinations are reachable since components are not fully independent, as they synchronize through interactions in the composition. A typical example of this can be found in example of Figure 1. Components T_1 (resp. T_2) may reach location ℓ_1^2 (resp. ℓ_1^3) by executing action $init_1$ (resp. $init_2$), but in the composition T_1 and T_2 cannot be simultaneously at locations ℓ_1^2 and ℓ_1^3. This is due to interactions $\alpha_1 = \{init_0, init_1\}$ and $\alpha_2 = \{init_0, init_2\}$ with component C: executing α_1 disables α_2, and vice versa. That is, the potential conflict between interactions α_3 and α_4 can be excluded if we consider reachable locations of the composed system.

Following [9], we use linear invariants to characterize reachable locations of compositions. They are formalized as follows.

Definition 6 (Linear Invariant): Let $S = \gamma(B_1, \ldots, B_n)$ be a composition and $\mathcal{L} = \bigcup_{1 \leq i \leq n} \mathcal{L}_i$ all components locations, $\mathcal{L}_i$ being the locations of B_i. A *linear invariant* of S is a linear equality constraint of the form: $\sum_{\ell \in \mathcal{L}} u_\ell \cdot \mathsf{at}(\ell) = u_0$, where u_ℓ, $\ell \in \mathcal{L}$, and u_0 are integers, in which predicates $\mathsf{at}(\ell)$, $\ell \in \mathcal{L}$, are interpreted as 0 for false and 1 for true, and holding on all reachable states, i.e. such that $Reach(S) \Rightarrow \sum_{\ell \in \mathcal{L}} u_\ell \cdot \mathsf{at}(\ell) = u_0$.
Using linear invariants for verification of systems is not a new idea. They corresponds to the notion of *S-invariant* in the Petri net community [10], which is determined to be appropriate for proving non-coverage of subsets of individual locations, thing that corresponds exactly to what needed to prove that two interactions cannot be enabled from the same location. To compute linear invariants for a composition $S = \gamma(B_1, \ldots, B_n)$, we consider its untimed version $\tilde{S}$ abstracting all timing aspects of S (i.e. obtained from S by relaxing guards and time progress conditions of components). Note that linear invariants for $\tilde{S}$ are also a linear invariants for S, since reachable locations of S are necessary included in reachable locations of $\tilde{S}$. We can directly apply results of [9] to $\tilde{S}$, in which linear invariants are characterized by a system of linear equations solved using standard techniques such as Gauss-Jordan elimination or LU-factorization, to compute a finite set of linear invariants $I(S) = \bigwedge_{k \in K} \sum_{\ell \in \mathcal{L}} u_\ell^k \cdot \mathsf{at}(\ell) = u_0^k$. A potential conflict between interactions α and β is a false conflict if the following formula is not satisfiable:

$$\bigwedge_{1 \leq i \leq n} Reach(B_i) \wedge I(S) \wedge Enabled(\alpha) \wedge Enabled(\beta) \quad (3)$$

In practice we use untimed versions of $\mathit{Enabled}$. Checking only locations as linear invariants do not involve clocks, which is illustrated below considering the running example.

Example 3: Let us reconsider the example of Figure 1. Among the linear invariants computed using the method of [9], we focus on following:

$$\begin{cases} 1 \cdot \mathsf{at}(\ell_1^2) + 1 \cdot \mathsf{at}(\ell_1^3) - 1 \cdot \mathsf{at}(\ell_1^1) = 0 & (4) \\ 1 \cdot \mathsf{at}(\ell_3^2) + 1 \cdot \mathsf{at}(\ell_3^3) - 1 \cdot \mathsf{at}(\ell_1^4) = 0. & (5) \end{cases}$$

We deduce from Equation (4) that $\mathsf{at}(\ell_1^2)$ and $\mathsf{at}(\ell_1^3)$ cannot be true simultaneously, that is, components T_1 and T_2 cannot be simultaneously at the corresponding locations. Consequently, we can directly infer that interactions α_3 and α_4 are not conflicting, even though they are potentially conflicting. Likewise, with (5) we exclude the conflict between α_7 and α_8.

B. Removing False Conflicts using History Clocks

As they completely abstract time, linear invariants capture only partially the system dynamics. For example, a global location may be not reachable because component locations having disjoint time progress conditions, or an interaction may not be enabled from a state because of an empty timing constraint. Such properties require extra relationships relating clocks of different components that are not available in $\mathit{Reach}(B_i)$ as it is is restricted to clocks of a single component.

We follow the approach of [11] for reinforcing our approach with global invariants on clocks. They are induced by simultaneity of transitions execution when executing an interaction and the synchrony of time progress. To compute such invariants, additional *history* clocks are first introduced in components. History clocks are associated to actions of components and to interactions, and reset upon their execution. They do not modify the behavior since they are not involved in timing constraints. They only reveal local timing of components, relevant to the interaction layer, which allows to infer further properties referred as *history clocks inequalities* in [11], expressing the fact that history clock of an interaction are necessary equal to history clocks of its actions after its execution and until the execution of another interaction involving these actions. Our method combines history clocks inequalities $\mathcal{E}(S)$ and symbolic states of components to identify false conflicts where the following is not satisfiable:

$$\bigwedge_{1 \le i \le n} \mathit{Reach}(B_i) \wedge \mathcal{E}(S) \wedge \mathit{Enabled}(\alpha) \wedge \mathit{Enabled}(\beta) \quad (6)$$

Example 4: We illustrate the application of (6) for checking conflicts by considering again the example of Figure 1. It can be shown that the potential conflict between α_5 and α_6 cannot be removed using (only) linear invariants. In the following, we prove that these interactions are actually not conflicting using history clocks inequalities.

Since action $start_0$ of C is synchronized with either $start_1$ of T_1 or $start_2$ of T_2, and since history clocks h_a of an action a is reset whenever a is executed, by [11] the history clock inequalities for $start_0$ are: $(h_{start_0} = h_{start_1} \le h_{start_2} - 25) \vee (h_{start_0} = h_{start_2} \le h_{start_1} - 25)$. This equation states that h_{start_0} is equal to the history clock corresponding to the last synchronization, i.e. either h_{start_1} or h_{start_2}, and is lower than history clocks of previous synchronizations. Value 25 is obtained considering *separation constraints* computed from symbolic states of components and interactions [11]: two occurrences of $start_0$ are separated by at least 25 time units because of timing constraints of C, and so too occurrences of $start_1$ or $start_2$ which can only execute jointly with $start_0$. To relate history clocks with components clocks, we simply include history clocks when computing symbolic states of components (i.e. *Reach* for components), which is used to establish here that $x = h_{start_1}$ and $y = h_{start_2}$ when components T_1 and T_2 are respectively at locations ℓ_2^2 and ℓ_2^3. We obtain then: $x \le y - 25$ or $y \le x - 25$.

By definition of $\mathit{Enabled}$ we have $\mathit{Enabled}(\alpha_5) = \mathsf{at}(\ell_2^2) \wedge (10 \le x \le 30)$. Similarly, $\mathit{Enabled}(\alpha_6) = \mathsf{at}(\ell_2^3) \wedge (10 \le y \le 30)$. We obtain then: $\mathit{Enabled}(\alpha_5) \wedge \mathsf{at}(\ell_2^3) \Rightarrow y \le 5 \;\wedge\; \mathit{Enabled}(\alpha_6) \wedge \mathsf{at}(\ell_2^2) \Rightarrow x \le 5$. This proves that α_5 and α_6 are not conflicting.

V. Implementation and Experiments

The described methods have been implemented using our framework BIP [12]. BIP is a component-based and model-based framework for constructing systems by superposing three layers of modeling: Behavior, Interactions and Priorities. It is intended for the design and analysis of complex, heterogeneous embedded applications, where components are modeled as timed automata and synchronize by means of multiparty interactions. Our implementation is integrated as a filter in the middleend of the real-time BIP compiler. The filter compiler implements all the methods described in this paper. The BIP compiler takes as inputs a real-time BIP model and a file expressing the interactions partitioning. First, the frontend builds a system representation of the input model. Then, based on the interaction partition, the middlend computes first the set of potentially conflicting interactions. Thereafter, reachable states of individual components are computed, as well as the linear invariants and the history clocks inequalities. Additional enabledness predicates are calculated since they are needed in Equation 2. Finally, for each two potentially conflicting interactions we check the satisfiability of:

$$\bigwedge_{1 \le i \le n} \mathit{Reach}(B_i) \wedge I(S) \wedge \mathcal{E}(S) \wedge \mathit{Enabled}(\alpha) \wedge \mathit{Enabled}(\beta)$$

This verification is achieved using the Yices solver [13]. If the result is unsatisfiable, then the two interactions are not conflicting. Otherwise, if the result is satisfiable, Yices generates a counter-example that may or not be a false-positive counter-example. The experiments have been conducted on a HP machine with Ubuntu 12.04, an Intel® Core™ i5-4300U processor of frequency 1.90GHz×4, and 7.7GiB memory.

A. Benchmarks

In order to attest that conflict detection is a matter of concern that should not be neglected during the implementation of

distributed real-time systems, we first performed several measurements on the Task Manager model in different settings: *(i)* we used 6 variants (including the initial model) each one respectively involves 2, 10, 20, 30, 40 and 50 tasks, and for each model *(ii)* we used several partitions of interactions (the more partition we have the higher is the risk of conflicts). Finally, we used the two following benchmarks to test our conflict detection method:

a) Train Gate Controller: This example defines a system composed of a controller, a gate and a couple of trains [3]. The controller lowers and raises the gate when a train is approaching, respectively exiting the gate.

b) Gear Controller: The Gear controller system describes the control system responsible for the gear change inside a vehicle. The used model encompasses the formal models of the gear controller and its environment. The hole system includes five components: an interface, a controller, a clutch, an engine and a gear-box. The case study [14] was initially designed in UPPAAL and has been translated to BIP.

B. Results

Table I summarizes for each experiment the number of components (n), interactions (i), partitions (p), potential conflicts (c), false conflicts (f), and gives also the total verification time of our methods combined.

We can notice from the Task Manger experiments that increasing the number of interactions partitions will increase the number of potential conflicts, that is, the more distributed the system is the more conflicts it contains. It also gives an indication on how much conflict resolution will be needed during execution. On the other hand, increasing the number of tasks increases the number of interactions in the system, and thus, the number of potential conflicts, especially when involving the same components as in our case the shared resource (R) and the controller (C). Nevertheless, conflict detection highly depends on the model and is highly coupled with its determinism degree. The other experiments showed also interesting results in terms of conflict reduction rate and execution time on real life case studies.

TABLE I: Experimental results

Model	n	i	p	c	f	t
Task Manager	4	8	2	4	3	60.55ms
	12	40	2	40	30	367.19ms
			5	64	48	521.45ms
			10	68	51	545.34ms
	22	80	2	80	60	2.14s
			10	144	108	3.41s
			20	148	111	3.69s
	32	120	30	228	171	8.41s
	42	160	40	308	231	20.72s
	52	200	2	200	150	22.56s
			5	320	240	34.78s
			10	360	270	37.72s
			25	384	288	41.69s
			50	388	291	45.35s
Train Gate Controller	22	45	20	74	37	813,62ms
Gear Controller	5	32	4	8	3	5,94s

VI. Conclusion

We presented an advanced method for conflict detection, allowing the generation of enhanced distributed implementations from model described using multiparty interactions. The proposed method leverages components and system information to avoid unnecessary invocation of the CRP during the scheduling of interactions, which would induce additional exchange of messages and an overhead computation, meaning additional latency. The implementation of the presented approach is fully automated and has been encompassed in the middleend filter of the real-time BIP compiler. Several tests were conducted on models of different size and with various distributed settings (interactions partitioning). There are still many challenging open problems in the implementation of distributed real-time systems. An immediate direction is to extend this method to systems with data variables. Another interesting aspect is to study the effect of clock drifts on the scheduling of interactions and conflict resolution. In real life clocks are not perfect and are subject to drifts, which may introduce anomalies during system execution.

References

[1] A. Triki, “Distributed implementations of timed component-based systems.” Ph.D. dissertation, Grenoble Alpes University, France, 2015.

[2] A. Triki, B. Bonakdarpour, J. Combaz, and S. Bensalem, “Automated conflict-free concurrent implementation of timed component-based models,” in *NASA Formal Methods - 7th International Symposium*, 2015.

[3] R. Alur and D. L. Dill, “A theory of timed automata,” *Theor. Comput. Sci.*, 1994.

[4] S. Tripakis, “The analysis of timed systems in practice,” Ph.D. dissertation, Joseph Fourier University, 1998.

[5] J. Bengtsson and W. Yi, “On clock difference constraints and termination in reachability analysis of timed automata,” in *ICFEM*, 2003.

[6] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” *Inf. Comput.*, 1994.

[7] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, “A framework for automated distributed implementation of component-based models,” *Distributed Computing*, 2012.

[8] R. Bagrodia, “Process synchronization: design and performance evaluation of distributed algorithms,” *Software Engineering, IEEE Transactions on*, 1989.

[9] S. Bensalem, M. Bozga, B. Boyer, and A. Legay, “Incremental generation of linear invariants for component-based systems,” in *13th International Conference on Application of Concurrency to System Design*, 2013.

[10] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, 1989.

[11] S. B. Rayana, L. Astefanoaei, S. Bensalem, M. Bozga, and J. Combaz, “Compositional verification for timed systems based on automatic invariant generation,” *Logical Methods in Computer Science*, 2015.

[12] A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis, “Using bip for modeling and verification of networked systems – a case study on tinyos-based networks,” in *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*, 2007.

[13] B. Dutertre and L. de Moura, “The yices smt solver,” SRI International, Tech. Rep., 2006.

[14] M. Lindahl, P. Pettersson, and W. Yi, “Formal Design and Analysis of a Gearbox Controller,” *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 2001.