IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 31, 2018, accepted November 29, 2018, date of publication December 10, 2018,
date of current version December 31, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2885705

Automatic Inference of Task Parallelism in Task-Graph-Based Actor Models

ABU NASER MASUD, BJÖRN LISPER, AND FEDERICO CICCOZZI, (Member, IEEE)
Mälardalen Real-Time Research Center, School of Innovation, Design and Engineering, Mälardalen University, 722 20 Västerås, Sweden

Corresponding author: Federico Ciccozzi (federico.ciccozzi@mdh.se)

This work was supported by the Knowledge Foundation through the SPACES[1] (rn. 20140165) and MOMENTUM[2] (rn. 20160153) Projects led by Mälardalen University, and run in cooperation with Ericsson AB, Evidente East AB, Alten Sweden AB, and SAAB AB.

ABSTRACT Automatic inference of task level parallelism is fundamental for ensuring many kinds of safety and liveness properties of parallel applications. For example, two tasks running in parallel may be involved in data races when they have conflicting memory accesses, or one is affecting the termination of another by updating shared variables. In this paper, we have considered a task-graph-based actor model, used in signal processing applications (e.g., baseband processing in wireless communication, LTE uplink processing) that are deployed on many-core platforms, in which the actors, task-graphs, and tasks are the active entities running in parallel. The actors invoke task graphs, which in turn invoke tasks, and they communicate through message passing, thus creating different kinds of dependencies and parallelism in the application. We introduce a novel *May Happen in Parallel* (MHP) analysis for complex parallel applications based on our computational model. The MHP analysis consists of (i) data-flow analysis applicable to parallel control-flow structures inferring MHP facts representing pairs of tasks running in parallel, (ii) identification of all direct and indirect communication by generating a context-free grammar and enumerating valid strings representing parallelism and dependencies among active entities, and (iii) inferring MHP facts when multiple task-graphs communicate. Our analysis is applicable to other computational models (e.g. Cilk or X10) too. We have fully implemented our analysis and evaluated it on signal processing applications consisting of a maximum of 36.57 million lines of code representing 232 different tasks. The analysis took approximately 7 minutes to identify all communication information and 10.5 minutes to identify 12052 executable parallel task-pairs (to analyse for concurrency bugs) proving that our analysis is scalable for industrial-sized code-bases.

INDEX TERMS May happen in parallel, data flow analysis, actor model, parallel task graph, graph reachability, UML profile.

I. INTRODUCTION

Software running on modern systems, from smartphones to autonomous vehicles, is becoming extremely complex and performance-demanding at a daunting pace. Parallel hardware, in terms of the so-called multiple core[3] processors, has been introduced to meet demands for performance while keeping power consumption under control. The computational increments brought by multiple core chips moved the performance quest from hardware to software. To exploit the physical parallelism of multiple core chips at its uttermost, we need in fact efficient parallel software. Along with efficiency, other crucial safety, liveness and security aspects such as data race freedom, atomicity, deadlock freedom, starvation freedom, to mention a few, are at least as important in many applications.

To design software able to fully leverage the computational power given by multi-core processors while keeping other aspects of parallel computation in balance is still a huge challenge in the software industry. The reason is that parallel programming is by nature more complex than sequential programming, which has shaped the mind-set of software developers throughout the years. Moreover, analysing, debugging, and testing parallel software is also more challenging than their sequential counterparts due to the enormous amount of

[1] http://www.es.mdh.se/projects/411-Static_Program_Analysis_for_Complex_Embedded_systems

[2] www.es.mdh.se/projects/458-MOMENTUM

[3] Note that we use 'multiple core' as a generic term for both many-core and multi-core.

information produced during the interleaved, nondeterministic execution of parallel processes.

Methodological aids are essential in the early development of parallel software for tuning performance and meeting correctness requirements. Understanding the inherent parallelism of a software application is essential for discovering many concurrency issues including data races, atomicity violations, deadlocks, livelocks, starvation, to mention a few. For example, in order to detect data races in an application, we need to identify two instances of program instructions, running in parallel, that have conflicting memory accesses. Historically, the analysis discovering parallelism of a given software is the *May Happen in Parallel* (MHP) analysis. The MHP problem aims at determining the set of pairs of program instructions whose instances can run in parallel during the execution of a given software system.

It is hard to find precise solutions to the MHP problem for many classes of programs. For instance, it becomes NP-complete when concurrent programs use rendezvous-like mechanisms for synchronous communication [1], and even undecidable when procedure calls are introduced [2]. Existing tractable solutions of the MHP problem target various concurrent computational models and languages such as the *fork-join* parallelism of Cilk [3], the async-finish parallelism of X10 [4]–[6], the rendezvous-like synchronisation of Ada [1], the concurrent object model of ABS [7], and the Java concurrency model [8], [9].

However, parallelism is not only brought along by software code, but can be hidden beneath high-level functional models as well as the complex distributed build and deployment processes. When it comes to modern complex software systems, such as telecom applications which we experimented with, existing solutions for MHP analysis are not applicable since these systems are described in terms of high-level functional models combined with code. Thus, in order to discover parallelism from different layers of complex software systems, we need an analysis targeting both the high-level models and the code.

In this paper, we provide **a novel static MHP analysis** for highly parallel software systems developed according to a task-graph-based actor model. This computational model is a variation of the actor-based concurrency model introduced by Agha *et al.* [10], [11]. In this model, a number of actors can run in parallel. Actors are lightweight processes without shared states; any actor may invoke instances of *task-graphs* describing execution scenarios for tasks on multi-core processors. More specifically, task-graphs are directed acyclic graphs representing the synchronisation of tasks via barriers. Task schedulers dispatch tasks on processors according to the precedence specified by the graphs. Tasks are basically sequential C or DSP C code possibly containing instructions to send asynchronous messages to actors or task-graphs.

Detecting all pairs of program instructions that can possibly run in parallel is a control-flow problem which requires first detecting and then navigating the (parallel) control-flow structure of the program. The kind of synchronisation primitives, together with other constructs such as pointers, used in programs decide the degree of complexity of detecting parallel control-flow structures. Once a parallel control-flow is detected, iterative data-flow-based techniques can be applied to collect parallel program instructions.

Inferring precise task-parallelism is a challenging problem that needs to take into account both the order of task invocation and dependencies among tasks created through message passing.

A. CONTRIBUTIONS

In contrast to existing solutions for MHP analysis, which are performed on software code having specific kinds of concurrency constructs, our contribution provides a novel MHP analysis performed on a task-graph-based actor model. This model can be obtained by leveraging high-level software functional models to infer the parallel control-flow of tasks in combination with the code implementing those tasks. We provide six contributions.

Contribution 1: We develop a data-flow analysis technique applicable to parallel control-flow structures such as task-graphs to infer task-level parallelism. We validated the analysis by experimenting on parallel task-graphs obtained from C code. However, the method is generic and applicable to parallel task-graphs obtained from other programming languages, for inferring properties not only limited to MHP facts. To show that, we illustrate by examples how to derive, from X10 parallel code, parallel task-graphs on which the analysis is directly applicable.

Contribution 2: In our computational model, it is possible that multiple parallel task-graphs are involved in direct or indirect communication by sending and receiving messages which restrict the task-level parallelism but increase the flow of information among task-graphs. We extend the data-flow analysis mentioned in Contribution 1 to obtain precise task-level parallelism from multiple parallel task-graphs involved in direct communication. We provide methods based on this data-flow analysis to infer non-concurrency among tasks invoked from a task-graph which is used to obtain precise task-parallelism from multiple task-graphs involved in indirect communication.

Contribution 3: We provide a mechanism based on generating *context-free grammar* (CFG) and a *string enumeration* technique from the CFG to automatically obtain all forms of communication to be used by the data-flow analysis technique, which is essential to infer all task-parallelism from the given system.

Contribution 4: We provide the theoretical worst-case asymptotic complexity of all algorithms provided by Contributions 1-3.

Contribution 5: We implemented the algorithms for the MHP analysis using SWI-Prolog. Methods to obtain communication information from a given system are implemented in the *Racer* tool. The *Racer* tool is implemented in the Clang/LLVM framework to obtain data race information from code (written in C or DSP C) executing in parallel.

We evaluated our implementation on signal processing applications obtained from communication-heavy telecom industry [12]. Our evaluation systems consisted of a maximum of 36.57 million lines of code representing 232 tasks.

Contribution 6: We define our task-graph models in terms of the Unified Modeling Language (UML).[4] More specifically, we designed and implemented a UML profile for MHP analysis (MHPp), which allows the intensional modelling of task-graph-based actor models and the back-propagation[5] of MHP analysis results to them for enhanced understandability.

B. ORGANIZATION OF THE ARTICLE

The remainder of this article is organized as follows. Section II describes the details of the task-graph-based actor model (computational model). Here, we formally define the *parallel task-graph* (PTG), and its relation with *actors* and tasks. We depict also the dependencies and parallelism that can arise due to interactions during the execution among actors, PTGs and tasks.

Section III describes the UML profile MHPp for MHP analysis and back-propagation of analysis results, and how we used it for describing our PTG running example.

In Section IV, we explain the semantics of the computational model. We define the program states and provide the operational semantics for PTGs. We have not provided the rules for actors and tasks because their individual execution is sequential in nature, which we consider intuitive.

Our MHP analysis is explained in all its details in Section V. Section VI explains the implementation of the entire approach, and Section VII evaluates the implementation on signal processing applications and analyses the experimental results. In Section VIII, we show how to translate code from the X10 parallel programming language into PTGs in order to illustrate the applicability analysis to other languages through PTGs.

Section IX gives an overview of the existing literature on MHP analysis and Section X concludes the paper.

II. COMPUTATIONAL MODEL

The computational model that we consider is used in signal processing applications (e.g., baseband processing in wireless communication, LTE uplink processing) that are deployed on many-core platforms [12], [14]. These applications are developed using a variant of the actor-based concurrency model, where an application consists of a number of actors.

Actors are lightweight processes without shared states and have mailboxes to buffer incoming messages. An actor selectively retrieves messages (one at a time) from its mailbox and either creates new actors, sends messages to other actors, or invokes different kinds of task-graphs based on the pattern of the retrieved message. Fig. 1(a) illustrates the relation among actors, task graphs and tasks.

[4]http://www.uml.org/

[5]Note that we have introduced the notion of back-propagation for model investigation and optimization in [13].

Task-graphs describe possible scenarios in which tasks can be executed on multiple core processors. A task-graph is basically a directed acyclic graph where the nodes represent tasks, barriers to which some tasks must synchronise, or other graph instructions such as conditional *if-then-else*. Edges represent the order of execution and dependencies among tasks. Each task executes on a dedicated core.

Fig. 1(b) illustrates the skeleton of a simple task-graph where horizontal rectangular nodes represent tasks to be executed, horizontal bars represent barrier nodes where tasks and data are to be synchronised, diamond-shaped node represents conditional instruction, and vertical rectangular nodes represent join nodes in which several task nodes are joined in order to be connected with a barrier node. The edges are directed, and arrows represent the direction of the data-flow.

In the telecom application used for our experiment, tasks are written in a proprietary dialect of the C language, and consist of application-specific functions that copy global data into core memory at the start and write back into the global memory at the end of the execution. The tasks themselves are sequential, but they may send asynchronous messages to actors or task-graphs. Individual tasks are independent meaning that the execution of tasks is neither interrupted by nor dependent upon other tasks. Once activated from a task-graph, tasks are dynamically scheduled to run on different cores by task schedulers.

Usually a graph engine, also called a *basic operating system* (BOS) [12], is designed to execute task-graphs; the BOS has usually a small footprint due to limited memory resources. The BOS operates in one or more cores and can receive graphs from outside the chip. Once a graph is received, it is preprocessed and stored as a condensed collection of linked lists. The BOS then starts the graph execution by finding a core supporting the execution of the first task, moving the required input data into the core's memory, and finally triggering it to start executing the task. As soon as the task finishes its execution, control and output data are moved back to the BOS. Data-flow between tasks may be synchronised through barriers, or tasks may run to completion without any synchronisation.

A number of task-graph instances may be executed in every time frame as shown in Fig. 1(c). The instances may originate from the same actor or different actors and are supposed to operate on different data areas.

There are several levels of parallelism in this model: (i) a single task-graph may invoke several tasks to be executed in parallel on different cores (i.e., *intra-graph* task-parallelism), (ii) several task-graphs running in parallel may invoke multiple tasks to be executed in parallel on different cores (i.e., *inter-graph* task-parallelism), and (iii) different actors running in parallel may invoke different task-graphs to run in parallel.

The semantics is provided at the level of task-graphs. The formal definition of a task-graph is as follows:

Definition 1 (Parallel Task-Graph (PTG)): A *Parallel task-graph* (PTG) is defined as a directed graph $\mathfrak{g} = (N, E)$

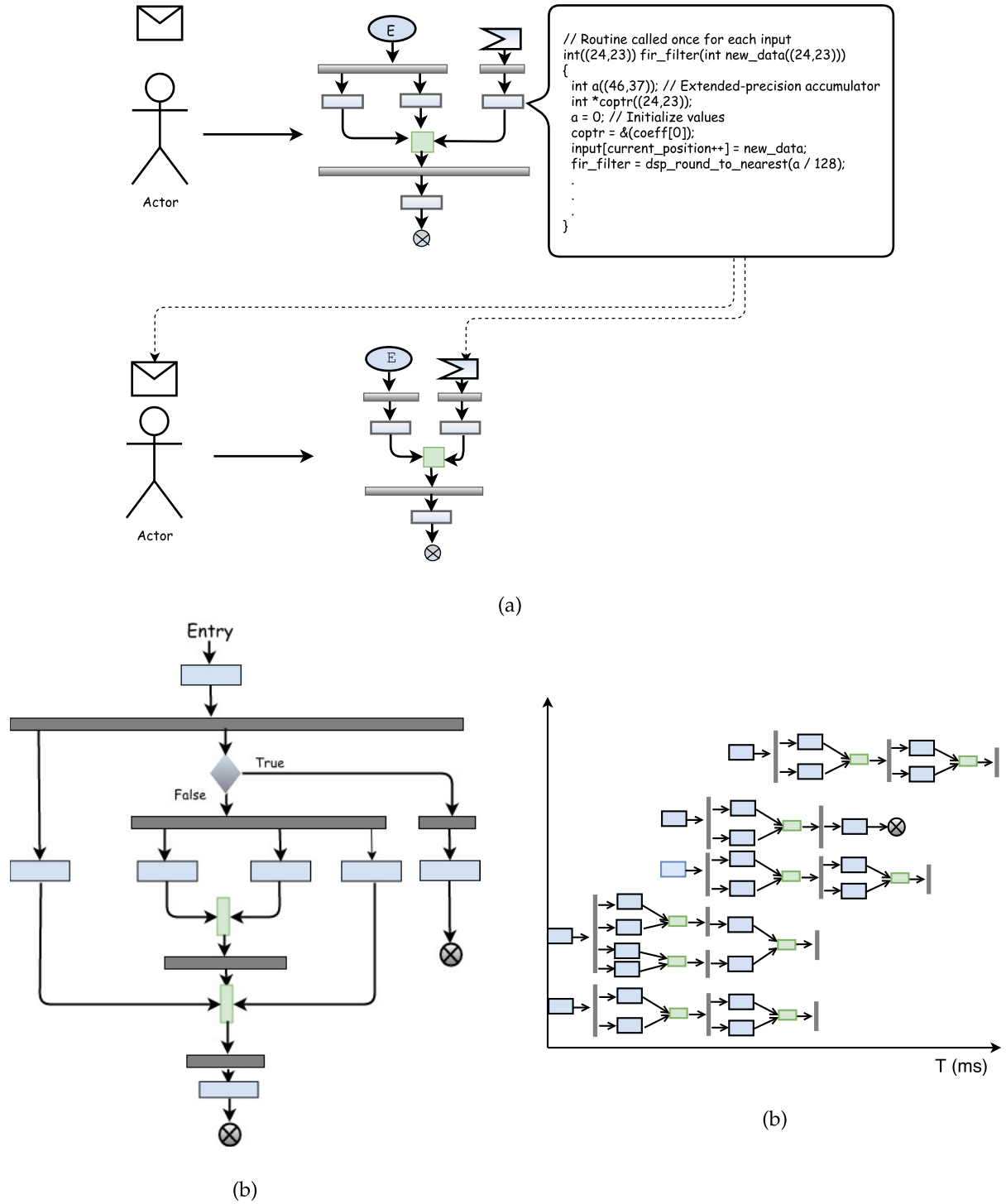


FIGURE 1. (a) Actors invoke PTGs which in turn invoke tasks and tasks may send messages to actors or task graphs, (b) a single task-graph where horizontal rectangles represent task nodes, horizontal bars represent barrier nodes, vertical rectangles represent join nodes and diamond-shaped nodes represent conditional nodes, and (c) multiple task-graphs running in parallel [12].

where

1) N is a set of nodes, $E \subseteq N \times N$ is a relation describing the possible parallel flow of execution in the graph, and any $n_1 \rightarrow n_2$ such that $(n_1, n_2) \in E$ describes the direction of the flow.

2) Nodes in N can be of one of the following types:
   - a *conditional* node n representing a condition c (denoted by $[c]^n$),
   - an *assignment* node,
   - a special *entry* node,

- a special *exit* node,
- a *barrier* node for possible synchronisation,
- a *task* node invoking the execution of an asynchronous task,
- a *msg-receiver* node to receive a message,
- a *join* node in which execution of multiple nodes join before connecting with a barrier node.

Task nodes can be further classified as *single-task* nodes invoking an instance of a task, or *multi-task* nodes invoking multiple instances of the same task (possibly operating on different global data).

3) The *entry* and the *msgReceiver* node have no predecessors and the *exit* node no successors. A *conditional* node has exactly two successors (labeled *true* and *false*, respectively), *barrier* nodes may have multiple successors, *join* nodes may have multiple predecessors, and all other nodes have exactly one successor and/or one predecessor.

We use the following denotation for the nodes of a PTG. The *entry*, the *exit*, any *join* and *barrier* nodes are represented by n_e, n_x, n_{join}, and n_b. We sometimes write $n : T$ to represent the condition that n is a node of type T where $T \in \{join, barrier, task, single\text{-}task, multi\text{-}task\}$. $[x := e]^n$ represents that the value of expression e is assigned to the variable x at node n. A single task node n is represented by $[m(\bar{e})]^n$ (or $[m(\bar{e})]^n_1$) that executes the function m with actual arguments $\bar{e} = e_1, \ldots, e_i$ when invoked. m is the initial function of the task node n that may have nested calls to other functions. $[m(\bar{e})]^n_k$ represents that $k \geq 1$ instances of the task starting with the m function will be executed. An example of a PTG is shown in Fig. 3.

III. MHPP UML PROFILE AND RUNNING EXAMPLE

From the myriad of general-purpose and domain-specific modelling languages (DSMLs) that have been defined to date, UML has established itself for industrial modelling [15].

UML is general-purpose, but it provides powerful profiling mechanisms to constrain and extend the language to achieve DSMLs, so called UML profiles. Through UML and profiles, the developer can fully describe software functionalities. Among the others, state-machine and activity diagrams are the most used when it comes to model behaviours of software systems with UML. In this work, we focus on activity diagrams since they represent the way task-graph-based actor models are modelled in practice. In the remainder of this section, we describe and show how we leveraged this mechanism to extend UML activity diagrams for (i) intensional modelling of task-graph-based actor models and (ii) back-propagation of analysis results to them, in terms of the MHPp profile.

A core activity of the creation of a UML profile is the definition of stereotypes. A stereotype represents a profile class which describes how an existing UML metaclass is extended by the profile. Stereotypes in the MHPp profile enable the use of MHP analysis specific terminology in place of the original one of the extended metaclasses. Stereotypes defined

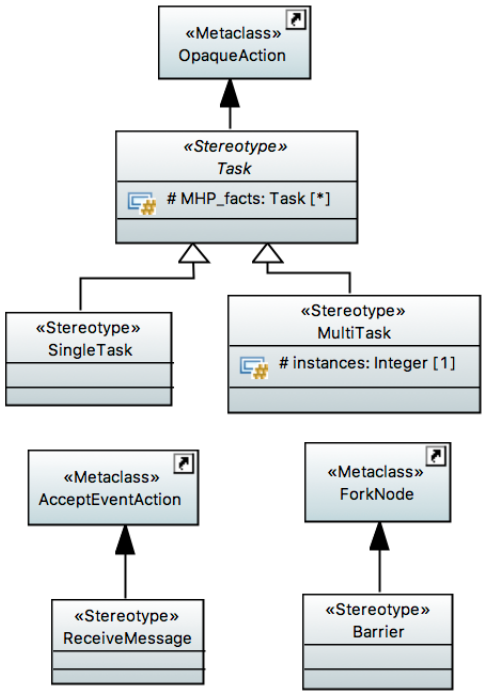


FIGURE 2. MHPp profile.

in the MHPp profile are shown in Fig. 2. More specifically we can see the following stereotypes (denoted in the figure as elements of type «Stereotype»):

- `Task`: it extends the UML metaclass `OpaqueAction`, representing nodes of a UML activity diagram, in order to introduce the concept of task. The stereotype owns a property, `MHP_facts`, representing an unordered and unbounded list of elements of type `Task`. This property allows to back-propagate MHP analysis results to the UML model (as shown in Section V). This stereotype is defined as abstract, and further specialised into:
  - `SingleTask`: it represents the concept of a task with a single instance;
  - `MultiTask`: it represents the concept of a task with possibly multiple instances. The number of instances is represented by an Integer value in the property `instances`.
- `Barrier`: it extends the `ForkNode` UML metaclass in order to introduce the concept of the barrier.
- `ReceiveMessage`: it extends the `AcceptEvent Action` UML metaclass to introduce the concept of the incoming message for the task-graph to be triggered by external task-graphs.

In Fig. 3 we depict a task-graph model described in terms of a UML activity diagram profiled with MHPp; note that we use this model in the remainder of the paper to explain our analysis. In the diagram, activities (denoted as *t_x* and stereotyped as «SingleTask», like *t_0*, or «MultiTask», like *t_2*) represent *task* nodes. Note that *t_x* represents a so called atom, which carries the information of the related node (*n_x* in the Figure). Note that in the case of «MultiTask» elements,

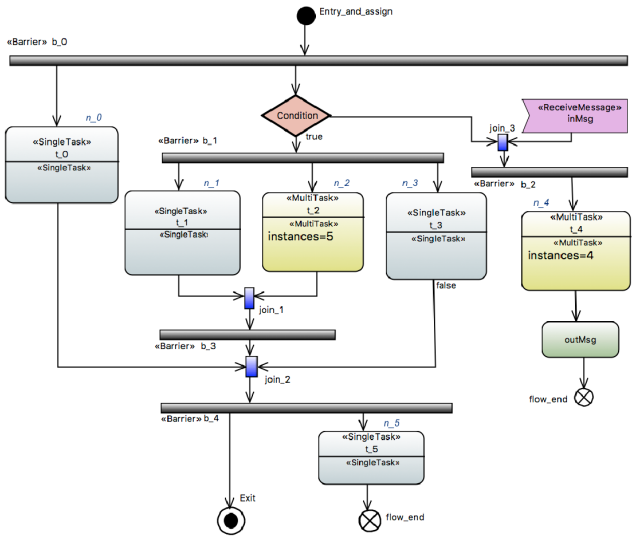


FIGURE 3. UML activity diagram describing task-parallelism in a single PTG.

the number of instances is specified too (e.g., 5 in *t_2*). Forks (denoted as *b_y* and stereotyped as «Barrier») represent *barrier* nodes. Joins (denoted as *join_z*) represent *join* nodes. Additionally, we have an *entry* node (*Entry_and_assign*), which represents the starting point of the overall activity. The flow can terminate in *exit* nodes (denoted as *flow_end* if a single flow terminates and *Exit* if the overall activity terminates). *Condition* represents a *conditional* node. We do not explicitly model assignment operations through *assignment* nodes since they can be embedded in other nodes (e.g., *entry* node *Entry_and_assign*).

In our computational model, we entail the possibility for running tasks to send messages to actors or task-graphs. Note that in the application type that we consider, message sending is not explicitly defined at graph level, but rather coded in the functionality carried by the individual task. Thereby, in our activity, we only model message receiving (*inMsg*, stereotyped as «ReceiveMessage») nodes. The activity diagram is complemented with task-specific algorithmic behaviors described for each task in terms of DSP C code.[6]

Note that the use of the MHPp profile and in general UML is not a requirement for our approach to be applicable. PTGs can, in fact, be described using other modelling formalisms.

IV. OPERATIONAL SEMANTICS

In this section, we provide the operational semantics for our computational model, i.e. task-graphs. We do not give the semantics for the sequential part, as it is standard and moreover not needed to support the MHP analysis. Furthermore, we abstract away actors for the sake of brevity, since actors are stateless and generate task-graphs sequentially. The operation of task schedulers and the BOS' activities are not included into the semantics either, since we prefer to base the MHP analysis on a semantic model that captures the inherent parallelism in the task-graphs rather than being dependent on a particular implementation of the underlying run-time system.

We define the semantics of our computational model in terms of program states. A program state is a tuple $(\mathbb{G}, \mathbb{T}, \mu)$ consisting of:

- a set of task-graph states $\mathbb{G}$,
- a set of task states $\mathbb{T}$ for all tasks invoked by the graphs, and
- a mapping function μ that maps global variables to their values (a *global store*).

Each task-graph state in $\mathbb{G}$ is a triple $\langle id, \mu_l, \mathcal{N} \rangle$ consisting of:

- the graph identifier id uniquely identifying the graph,
- a mapping function μ_l that maps graph-local and some auxiliary variables to their values (a *graph-local store*), and
- $\mathcal{N}$, the set of PTG nodes that are ready to be executed, possibly in parallel.

Each task state in $\mathbb{T}$ is a tuple $\langle tid, id, \mu_t, \mathcal{I} \rangle$ where

- tid is a task id that uniquely identifies the task,
- id is the identifier of the PTG that invoked this task,
- μ_t is the mapping function that maps task local variables to their values (a *task-local store*), and
- I is the sequence of the remaining task instructions to be executed.

The notation $a : \mathbb{A}$ is shorthand for $\mathbb{A} \cup \{a\}$, $\mathrm{i}.I$ is the sequence of instructions I preceded by the single instruction i, ϵ represents the empty sequence, and $[\![e]\!]_\mu$ denotes the result of evaluating the expression e with values of variables given by the mapping function μ. The function $\mu[x \leftarrow v]$ is defined by $\mu[x \leftarrow v](x) = v$, and $\mu[x \leftarrow v](y) = \mu(y)$ when $y \neq x$. $exec(\mathrm{i}, \mu, \mu_G)$, where i is an instruction and μ, μ_G are stores, is a store with the same domain as μ. Intuitively, this is the store μ updated with the effects of executing i in an environment where variable values are given by μ_G.

The following functions simplify the presentation of the operational semantic rules in Fig. 4:

- $newtasks(id, m(e_1, \ldots, e_l), \mu, k)$: creates $k \geq 1$ new task states $T^{tid_1}, \ldots, T^{tid_k}$ where $T^{tid_i} = (tid_i, id, \mu_t, body(m))$ for $1 \leq i \leq k$, tid_i is the fresh unique task identifier of the newly created task, id is the identifier of the PTG that invokes the task, μ_t is the initial store of the task consisting of values $[\![e_j]\!]_\mu$ for the j-th formal argument of m for all $1 \leq j \leq l$, and $body(m)$ contains the sequence of instructions of the m function to be executed.
- $succ(n)$ and $pred(n)$ denote the sets of successor and predecessor nodes of n.
- $tsucc(n)$ and $fsucc(n)$ denote the (singleton) set of successors in the *true* and in the *false* branch of the conditional node n.

[6]Note that the modelling language and the MHPp profile do not enforce any specific language for the definition of behaviours.

$$\frac{}{(\langle id, \mu_l, \{n_e\}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, succ(n_e)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; startGraph$$

$$\frac{[\![c]\!]_{\mu_l \cup \mu} = true}{(\langle id, \mu_l, [c]^n : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, \mathcal{N} \cup tsucc(n)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; condTrue$$

$$\frac{[\![c]\!]_{\mu_l \cup \mu} = false}{(\langle id, \mu_l, [c]^n : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, \mathcal{N} \cup fsucc(n)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; condFalse$$

$$\frac{}{(\langle id, \mu_l, [x := e]^n : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l[x \leftarrow [\![e]\!]_{\mu_l \cup \mu}], \mathcal{N} \cup succ(n)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; assign$$

$$\frac{T^{tid_1}, \ldots, T^{tid_k} = newtasks(id, m(\bar{e}), \mu_l \cup \mu, k), \mu'_l = \mu_l[s_{tid_1} \leftarrow t^\circ, \ldots, s_{tid_k} \leftarrow t^\circ]}{(\langle id, \mu_l, [m(\bar{e})]^n_k : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu'_l, \mathcal{N} \cup succ(n)\rangle : \mathbb{G}, \mathbb{T} \cup \{T^{tid_1}, \ldots, T^{tid_k}\}, \mu)} \; task$$

$$\frac{\forall n \in pred(n_{join})\ (S_n = \emptyset \vee \forall s \in S_n.\mu_l(s) \in \{t^\bullet, \top\})}{(\langle id, \mu_l, n_{join} : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, \mathcal{N} \cup succ(n_{join})\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; join\text{-}sync$$

$$\frac{pred(n_b) = \{n\},\ (S_n = \emptyset \vee \forall s \in S_n.\mu_l(s) \in \{t^\bullet\})}{(\langle id, \mu_l, n_b : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, \mathcal{N} \cup succ(n_b)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; barrier$$

$$\frac{n : msg\text{-}receiver, \mu_l(r_n) = \top}{(\langle id, \mu_l, n : \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l[r_n \leftarrow \bot], \mathcal{N} \cup succ(n)\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; msgReceive$$

$$\frac{\mathrm{i} \neq send(msg, d), \mu'_t = exec(\mathrm{i}, \mu_t, \mu_t \cup \mu), \mu' = exec(\mathrm{i}, \mu, \mu_t \cup \mu)}{(\langle id, \mu_l, \mathcal{N}\rangle : \mathbb{G}, \langle tid, id, \mu_t, \mathrm{i}.I\rangle : \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l, \mathcal{N}\rangle : \mathbb{G}, \langle tid, id, \mu'_t, I\rangle : \mathbb{T}, \mu')} \; seqExecution$$

$$\frac{\mathrm{i} = send(msg, id, n), \mu' = \mu[buff \leftarrow add((id, n, msg), \mu(buff))}{(\mathbb{G}, \langle tid, id, \mu_t, \mathrm{i}.I\rangle : \mathbb{T}, \mu) \rightsquigarrow (\mathbb{G}, \langle tid, id, \mu_t, I\rangle : \mathbb{T}, \mu')} \; msgSend$$

$$\frac{(id, n, msg) \in \mu(buff), \mu_l(r_n) = \bot, \mu' = \mu[buff \leftarrow elem((id, n, msg), \mu(buff))]}{(\langle id, \mu_l, \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l[r_n \leftarrow \top], \mathcal{N} \cup \{n\}\rangle : \mathbb{G}, \mathbb{T}, \mu')} \; msgTrigger$$

$$\frac{}{(\langle id, \mu_l, \mathcal{N}\rangle : \mathbb{G}, \langle tid, id, \mu_t, \epsilon\rangle : \mathbb{T}, \mu) \rightsquigarrow (\langle id, \mu_l[s_{tid} \leftarrow t^\bullet], \mathcal{N}\rangle : \mathbb{G}, \mathbb{T}, \mu)} \; endTask$$

$$\frac{}{(\langle id, \mu_l, \{n_x\}\rangle : \mathbb{G}, \mathbb{T}, \mu) \rightsquigarrow (\mathbb{G}, \mathbb{T}, \mu)} \; endGraph$$

FIGURE 4. Operational semantics.

- *add(e,l)* is the list l after inserting the element e, and *elem(e,l)* is the list l after removing the element e.

Suppose $\langle id, \mu_l, \mathcal{N}\rangle$ is the graph state of a PTG. The domain of function μ_l consists of graph-local variables and some auxiliary variables representing the execution status of some PTG nodes. Auxiliary variables include $s_{tid_1}, \ldots, s_{tid_k}$ for a task node creating $k \geq 0$ task states $T^{tid_1}, \ldots, T^{tid_k}$, and r_n for a *msg-receiver* node n. These auxiliary variables take values from the set $\{t^\circ, t^\bullet, \bot, \top\}$ where $t^\bullet$ (or t°) represents that the execution of the PTG task node that the auxiliary variable represents finishes (resp. is yet to finish), $\bot$ represents that the execution of the corresponding task node is not yet started or the *msg-receiver* node is yet to receive a message, and $r_n = \top$ represents that node n has received a message. If a PTG task node n invokes a task with task state T^{tid}, then s_{tid} is the global variable to the newly created task, but a local variable to the PTG. Given any node n, S_n represents the set of auxiliary variables related to node n. $S_n = \emptyset$ if n is not a task or a *msg-receiver* node. Any global store μ contains a buffer *buff* for the incoming messages of PTGs such that $\mu(buff)$ is the possibly empty list of incoming messages of the program. For any $(id, n, msg) \in \mu(buff)$, the PTG having the identifier id contains the *msg-receiver* node n to receive the message *msg*. Note that this buffer is different from the actor mailboxes which we do not model in the semantics for brevity. We assume that there is no naming conflict among global, graph-local, and task-local variables for simplicity. Global variables can be shared and updated by all the entities (i.e. PTGs and tasks), graph-local variables are visible only in the local PTG nodes and can be updated using the values of variables local to the same graph, and of global variables. Task-local variables, finally, are visible (and updatable) only within their respective tasks, using task-local and global variable values.

We assume that the program starts its execution from a non-empty set of PTGs having the graph state $\mathbb{G}$, no tasks have started their execution yet (i.e. $\mathbb{T} = \emptyset$), and μ_0 is the initial mapping of the global shared memory. Thus, the initial

program state is $\sigma_0 = (\mathbb{G}, \emptyset, \mu_0)$. For any graph state $\langle id, \mu_l, \mathcal{N}\rangle \in \mathbb{G}$, $\mu_l(s) = \perp$ for all auxiliary variables s. As the program executes, there are nondeterministic changes of state from σ to σ', i.e. $\sigma \rightsquigarrow \sigma'$. Nondeterminism comes from the task scheduler, which decides which tasks are executed on which cores, the concurrent execution of parallel task-graphs, and the invocation of new graphs by actors running in parallel.

Intuitively, the semantic rules in Fig. 4 represent execution as follows:

- *startGraph*: the graph $\langle id, \mu_l, \{n_e\}\rangle$ is nondeterministically chosen and the execution continues to the successor node(s) of n_e.
- *condTrue*: the graph state $\langle id, \mu_l, [c]^n : \mathcal{N}\rangle$ is selected, c is evaluated to be *true* with respect to the local store μ_l and the global store μ, and the successor node of n in the *true* branch will be executed next together with the nodes in $\mathcal{N}$.
- *condFalse*: the graph state $\langle id, \mu_l, [c]^n : \mathcal{N}\rangle$ is selected, c is evaluated to be *false* with respect to the local store μ_l and the global store μ, and the successor node of n in the *false* branch will be executed next together with the nodes in $\mathcal{N}$.
- *assign*: given the graph state $\langle id, \mu_l, [x := e]^n : \mathcal{N}\rangle$, the execution updates the local store μ_l by evaluating e with respect to μ_l and μ and assigning it to x. The successor node of n will be executed simultaneously with the nodes in $\mathcal{N}$.
- *task*: this rule is applicable when n is a *single-task* or a *multi-task* node. $k \geq 1$ new tasks with task states $T^{tid_1}, \ldots, T^{tid_k}$ are created, the value of auxiliary variables $s_{tid_1}, \ldots, s_{tid_k}$ are updated to t°, and all successor nodes of n will be executed simultaneously with the nodes in $\mathcal{N}$.
- *join-sync*: execution reaches the join node n_{join}. For any $n \in pred(n_{join})$, $S_n = \emptyset$ if n is not a task node or a a *msg-receiver* node (usually a *barrier* node). For all $n \in pred(n_{join})$ such that n is a task or a a *msg-receiver* node, if $\mu_l(s) \in \{t^\bullet, \top\}$ for all $s \in S_n$ then all preceding tasks have finished their execution or *msg-receiver* node received messages. The barrier node that immediately succeeds n_{join} will be executed simultaneously with the nodes in $\mathcal{N}$.
- *barrier*: execution reaches the barrier node n_b from a node n which might be a join, task, another barrier, or a msg-receiver node. If n_b is reached from a task node n, then $\mu_l(s) = t^\bullet$ for all $s \in S_n$. If the execution passes the join node, all preceding tasks supposed to synchronise with the barrier have finished their execution (see *join-sync* rule). $S_n = \emptyset$ if n is not a task node. Thus, all successor nodes of n_b (usually creating multiple parallel tasks) will be executed simultaneously with the nodes in $\mathcal{N}$.
- *msgReceive*: the *msg-receiver* node n has received a message (i.e. $\mu_l(r_n) = \top$). Thus, the successor node of n will be executed simultaneously with the nodes in $\mathcal{N}$. We reset μ_l to $\mu_l[r_n \leftarrow \perp]$ such that n may be triggered again by receiving new messages.
- *seqExecution*: this rule states how that statement $\mathfrak{i}$ of the task with the task state $\langle tid, id, \mu_t, \mathfrak{i}.I\rangle$ executes. The task local mapping μ_t and the global mapping function μ are updated to μ_t' and μ' due to executing $\mathfrak{i}$ when $\mathfrak{i}$ is not a message send instruction.
- *msgSend*: the task with the task state $\langle tid, id, \mu_t, \mathfrak{i}.I\rangle$ executes a message send instruction $\mathfrak{i} = send(msg, id, n)$ that sends the message msg to the *msg-receiver* node n in PTG id. This updates the buffer $\mu(buff)$ in the global store μ by including the element (id, n, msg).
- *msgTrigger*: the buffer *buff* in the global store contains a message to the *msg-receiver* node n in PTG id. Node n is added to the set of parallel executing nodes $\mathcal{N}$, the status variable r_n is set to $\top$ in the graph-local store μ_l and the message (id, n, msg) is removed from *buff* in the global store μ. This is the preprocessing step to proceed execution by node n.
- *endTask*: the task $\langle tid, id, \mu_t, \epsilon\rangle$ has finished its execution. ϵ represents that no more task instructions are remaining to be executed. Thus, the auxiliary variable s_{tid} in the graph state ($\langle id, \mu_l, \mathcal{N}\rangle$ is upgraded to $t^\bullet$.
- *endGraph*: the graph $\langle id, \mu_l, \{n_e\}\rangle$ is nondeterministically chosen and the execution of this graph terminates as the exit node n_x is reached.

V. OUR MHP ANALYSIS

This section describes our MHP analysis tailored for programs written according to the computational model introduced in Sec. II. The analysis is formulated in several stages with increasing complexity. First, we describe how to perform the analysis over a single task-graph that may invoke multiple parallel tasks. Next, we show how to extend the previous analysis to several graphs being executed in parallel, followed by a whole program analysis.

A. TASK-PARALLELISM IN A SINGLE TASK-GRAPH

We define a forward data-flow analysis [16] that infers task-parallelism from a single task-graph. Note that there exist previous works to infer MHP information [7], [8], [17] by data-flow analysis. Our technique differs from these by the graph representation of the program and the domain in which the data-flow operations are performed. The analysis infers task-parallelism from a single PTG, and it is extended to deal with multiple PTG's in Sec. V-B and V-C. The final whole program analysis in Section V-D is not a data flow analysis, but is based on context-free grammars.

The data-flow analysis is performed on a PTG and can determine precisely the tasks that may execute in parallel once invoked from the PTG. Let us use the following notation:

- $\mathcal{G}$ is the set of all PTGs,
- $\mathfrak{g}$ (possibly subscripted) denotes a single PTG,
- N is the set of all PTG nodes,

- $\mathcal{T}$ is the set of all tasks, and
- $\mathcal{B}$ is the set of all barriers.

We shall use the letters t, and b (possibly subscripted) to denote a single task, and barrier respectively, and $\mathtt{t}(n)$ denotes the type of node n (see Definition 1).

The MHP analysis on a PTG can be expressed by the instance $(\mathcal{L}, \mathcal{F}, \mathfrak{g}, I, \iota, \kappa)$ of the monotone data-flow framework [16], where:

- $\mathcal{L}$ is the data-flow domain (aka property space),
- $\mathcal{F}$ is the function space for the transfer functions,
- $\mathfrak{g} = (N, E)$ is the PTG,
- $I = \{entry\}$ is the set of so called extremal labels from which the analysis starts,
- $\iota = \emptyset$ is the initial data-flow value for the nodes in I,
- and the mapping function $\kappa : N \rightarrow \mathcal{F}$ selects the appropriate transfer function f_n for the node $n \in N$.

The choice of the data-flow domain and the transfer functions are explained as follows. The set $\mathcal{T}_A$ contains all the MHP *facts* (called "atoms") of the following forms:

- an atom t representing an instance of a task in $\mathcal{T}$ that is not synchronised by any barrier,
- an atom t^+ representing multiple instances of a task in $\mathcal{T}$ that is not synchronised by any barrier,
- an atom b representing an instance of a barrier in $\mathcal{B}$,
- an atom d representing a conditional node in the PTG,
- an atom $t \triangleright b$ (or $t^+ \triangleright b$) denoting an instance (or multiple instances) of a task in $\mathcal{T}$ that t represents which is synchronised at an instance of a barrier in $\mathcal{B}$ that b represents,
- an atom $t \triangleright d$ (or $t^+ \triangleright d$) representing that the task(s) represented by the atom t (or t^+) is followed by the conditional node in PTG that d represents,
- atoms px, $px \triangleright y$ such that $x \in \{t, t^+\}$ and $y \in \{b, d\}$ representing the fact that the corresponding atoms without the p-tag are originated in a parallel branch,
- a special atom ε such that $\varepsilon x = x$ and $S \cup \{\varepsilon\} = S$ for any atom x and a set of atoms S.

Note that the above kinds of facts illustrate the origin of the generated facts and their synchronisation pattern, which will help to decide the MHP pairs of tasks after the data-flow analysis. For any generated fact $t \triangleright d$, the task node represented by t may have different types of synchronisation in the two different branches of the conditional node represented by d. For example, in Fig. 5(a), the fact $t \triangleright d$ is generated for node n before the conditional node, which is evaluated to t, in the *true* branch, and $t \triangleright b$, in the *false* branch. We sometimes use variables x, y to hold atom values of the above types.

If the execution of a task node $n \in N$ ends at a barrier represented by the barrier atom b or is followed by a conditional node represented by the atom d, then $\mathcal{B}_n$ denotes the singleton set b or $\{d\}$. $\mathcal{B}_n$ is empty for all other cases. For any task or barrier node n, $\mathcal{B}_n^{\leftarrow}$ denotes the set of all barrier atoms b such that n is reachable from the barrier nodes generating atom b. We also consider the set $\mathcal{J}_n$ consisting of all task nodes from which the join node n is reachable. For example, $\mathcal{B}_n = \{b_4\}$ and $\mathcal{B}_n^{\leftarrow} = \{b_1, b_2, b_3\}$ in Fig. 5(b), and $\mathcal{J}_{join_1} = \{n_1, n_2\}$ in Fig. 3. $\mathcal{B}_n$ is used to decide the synchronisation of n, $\mathcal{B}_n^{\leftarrow}$ is used to discard relevant facts propagated due to parallel branching from the ancestor nodes, and $\mathcal{J}_n$ decides if an atom should be present as a parallel atom or not at a join node (explanation follows). The set $\mathcal{T}_B$ consists of all the barrier atoms.

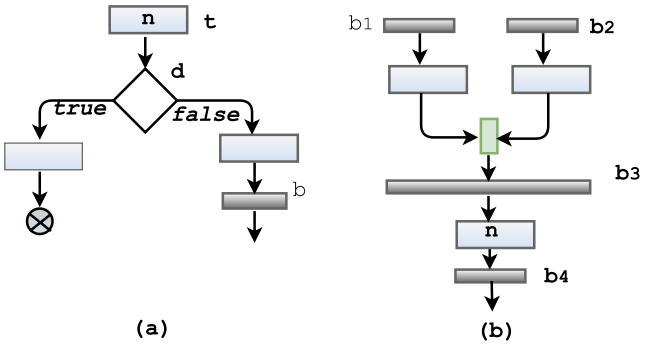


FIGURE 5. Synchronisation pattern of task nodes in the PTG segment: node ***n*** runs to completion in the ***true*** branch and synchronises with the barrier ***b*** in the ***false*** branch of the conditional node represented by ***d*** in Fig. (a), and synchronises with the barrier b_4 in Fig. (b).

The property space of the analysis is $\mathcal{L} = \mathcal{P}(\mathcal{T}_A)$, and $(\mathcal{P}(\mathcal{T}_A), \subseteq)$ forms a complete lattice. Note that the set $\mathcal{T}_A$ is finite as $\mathcal{T}$, $\mathcal{B}$, and the number of conditional nodes in the PTG are finite. $\mathcal{F}$ contains *transfer functions* updating the data-flow values. Given the set $\mathcal{A}$ of data-flow facts for any $n \in N$, the transfer function $f_n \in \mathcal{F}$ where $f_n : \mathcal{L} \rightarrow \mathcal{L}$ is defined as follows:

$$f_n(\mathcal{A}) = (\mathcal{A} \setminus kill(n)) \cup gen(n) \tag{1}$$

The definitions of *kill* and *gen* sets are as follows:

$$kill(n) = \begin{cases} \{x \triangleright b : b \in \mathcal{B}_n^{\leftarrow}, x = \mathfrak{a}(m), m \in N\} & \text{if } n : barrier \\ \{px : \mathcal{X}(x) = \mathfrak{a}(m), m \in \mathcal{J}_n\} & \text{if } n : join \\ \emptyset & \text{otherwise} \end{cases}$$

$$gen(n) = \begin{cases} \{\mathfrak{a}(n)\} & \text{if } (n : task \text{ or } n : barrier), \\ & \mathcal{B}_n = \emptyset \\ \{\mathfrak{a}(n) \triangleright y\} & \text{if } n : task, y \in \mathcal{B}_n \\ \emptyset & \text{otherwise} \end{cases}$$

In the definitions of *kill* and *gen*, $\mathfrak{a}(n)$ stands for the fixed atom for node n if n is a *single-task*, *multi-task*, *barrier*, or *conditional* node. The $\mathcal{X}$ function as defined in Eq. (6) extracts task atoms only. For example, $\mathcal{X}(pt \triangleright b) = t$. Intuitively, the *kill* set of a *barrier* node n consists of all atoms $x \triangleright b$ such that the atom x is generated at some node $m \in N$ by the $\mathfrak{a}$ function (i.e. $x = \mathfrak{a}(m)$) and is supposed to synchronise at a barrier represented by the barrier atom b, and hence $b \in \mathcal{B}_n^{\leftarrow}$. Usually, the atom $x \triangleright b$ is present in the dataflow set $\mathcal{A}$ due to the fact that a task is invoked earlier from the PTG. It should already have terminated its execution at n represented by the barrier atom b, and thus the atom $x \triangleright b$ gets killed at the barrier node n due to the $kill(n)$ set in Eq. (1).

If n is a join node, then $kill(n)$ includes all parallel atoms px such that the constituent task atom $\mathcal{X}(x)$ of px originates from some node m (i.e. $\mathcal{X}(x) = \mathfrak{a}(m)$) and n is reachable from m. If there is a path from m to n, then we have either one of the following two cases. The generated atom at n (i.e. $\mathcal{X}(x)$ possibly synchronized with a barrier atom) may flow to n. So, we have both x and px atoms at n, and x can no more be a parallel atom after the join node n. Alternatively, the generated atom at m may be killed before reaching n. If we choose px not to be killed at n, it would be killed at the barrier node n' immediately after n because of the first case of the *kill* function. However, $kill(n)$ does not include all other parallel atoms px' such that x' is generated at m' and there is no path from m' to n. If we do not include px into the kill set of n, we shall potentially transfer px beyond n and will result in incorrect parallelism.

The *gen* set for a *single-task*, *multi-task*, or *barrier* node n contains the atom $\mathfrak{a}(n)$ unless the execution of n ends at the barrier represented by b or n is followed by a conditional node represented by the atom d, in which case the *gen* set contains the atom $x \triangleright y$ and $y = b$ or $y = d$. The *kill* and *gen* sets are empty for all other kinds of nodes.

MHP facts are collected into the sets $\mathcal{A}_\circ(n)$ and $\mathcal{A}_\bullet(n)$, which are valid at the entry and exit of node $n \in N$. Equations (2)–(6), as shown at the bottom of this page, are formulated for the forward dataflow analysis that collects the MHP facts for each node $n \in N$. $\mathcal{A}_\bullet(n)$ can be obtained by simply applying the transfer function f_n. The set $\mathcal{A}_\circ(n)$ is constructed by combining the $Flow(n)$ and $Effect(n)$ sets: $Flow(n)$ collects all dataflow facts from the predecessor nodes of n, and $Effect(n)$ set contains all dataflow facts due to facts present in the parallel branches of n. In obtaining $Flow(n)$, we apply the functions $\mathfrak{f}_c$ to $\mathcal{A}_\circ(n')$ if the predecessor node n' of n is a *conditional* node; otherwise, $\mathcal{A}_\circ(n)$ contains elements of $\mathcal{A}_\bullet(n')$. The $\mathfrak{f}_c$ function either removes or modifies some specific types of atoms from $\mathcal{A}_\circ(n')$, and is defined as follows:

$$\mathfrak{f}_c(n, n', T) = (T \setminus \{x \triangleright d : d = \mathfrak{a}(n')\}) \cup D(n, n', T)$$

where

$$D(n, n', T) = \begin{cases} \{x : x \triangleright d \in T, d = \mathfrak{a}(n')\} & \text{if } \mathcal{B}_n{=}\emptyset \\ \{x \triangleright b : x \triangleright d \in T, d{=}\mathfrak{a}(n'), b \in \mathcal{B}_n\} & \text{if } \mathcal{B}_n{\neq}\emptyset \end{cases}$$

So, this function returns a set of facts containing elements of T but removing any fact $x \triangleright d \in T$ when n' is a conditional node represented by d, and instead, either $x \triangleright b$ or x is included depending on the set $\mathcal{B}_n$. Applying the function $\mathfrak{f}_c$ to $\mathcal{A}_\bullet(n')$ in Eq. (2) provides more precise synchronisation to the MHP atoms. Moreover, *Effect* accumulates facts from the set $gen(k)$ such that node k is in the parallel branch of node n. Eq. (4) models this update, which is explained later in this section. We thus get a system of equations where the least solution can be obtained by fixpoint iteration.

MHP Analysis for PTG: Algorithm 1 performs the fixpoint computations for the MHP analysis of a single PTG. It is a variation of the standard worklist algorithm [16], which collects MHP facts for every node of the graph according to Eq. (2). Initially, the worklist W contains the edge from the *entry* node, $\mathcal{A}_\circ$ and $\mathcal{A}_\bullet$ are empty sets. $\mathcal{B}_n$, $\mathcal{B}_n^{\leftarrow}$ and $\mathcal{J}_n$ are computed by applying Algorithm 2, described later in this section. During the fixpoint iteration of the algorithm, an edge (n, m) (i.e., a worklist element) of the given PTG is removed from the worklist, and $\mathcal{A}_\bullet(m)$ is updated by applying the transfer function computing MHP facts (line 5). Next, depending on the types of nodes n and m, the facts may be propagated through (i) the successor edges (line 7-8), (ii) other parallel branches connected to n when n is a barrier node (line 9-13), and (iii) distant parallel branches identified by the facts $px \in \mathcal{A}_\bullet(n)$ (line 14-20).

For each selected edge (n, m) from the worklist W during the fixpoint iteration, Algorithm 1 performs the following

$$\begin{aligned} \mathcal{A}_\bullet(n) &= f_n(\mathcal{A}_\circ(n)) \\ \mathcal{A}_\circ(n) &= Flow(n) \cup Effect(n) \end{aligned} \tag{2}$$

where

$$Flow(n) = \begin{cases} \emptyset & \text{if } n : entry \\ \bigcup_{n' \in pred(n)} \begin{cases} \mathfrak{f}_c(n, n', \mathcal{A}_\bullet(n')) & \text{if } n' : cond \\ \mathcal{A}_\bullet(n') & \text{otherwise} \end{cases} \end{cases} \tag{3}$$

$$Effect(n) = \bigcup_{k \in Parallel(n)} \{px : x \in gen(k), \forall y, d . x \notin \{y \triangleright d\}\} \tag{4}$$

$$Parallel(n) = \begin{cases} \{k : (n', k) \in E, k \neq n\} & \text{if } (n', n) \in E \wedge n' : barrier \\ \{k : px \in \mathcal{A}_\bullet(k), k \in N, \mathcal{X}(x) = \mathfrak{a}(n)\} & otherwise \end{cases} \tag{5}$$

$$\mathcal{X}(x) = \begin{cases} t & \text{if } x \text{ is of the form } t, t^+, t \triangleright y, t^+ \triangleright y, pt \triangleright y, pt^+ \triangleright y \\ \varepsilon & otherwise \end{cases} \tag{6}$$

Algorithm 1: MHPAnalysisOfPTG

```
   /* Initialization                                   */
1  forall (n ∈ N) do A_∘(n) := A_•(n) := ∅
2  W := {(entry, n) : (entry, n) ∈ E}
3  Compute-B_n J_n-and-B_n^← () /* Apply Alg. 2 */
4  while (W ≠ ∅) do /* Fixpoint iteration */
5      W := W \ {(n, m) : (n, m) = select(W)}
       /* apply transfer function */
6      A_•(m) := f_m(A_∘(m))
7      forall ((m, p) ∈ E) do
8          updateWListandPropagateFacts(m, p, A_•(m))
       /* propagate facts to the parallel
          branches of the barrier node */
9      if (n : barrier ∧ A_•(m) \ A_∘(m) ≠ ∅) then
10         S := A_•(m) \ A_∘(m)
11         T := {px : x ∈ S, ∃b, y, d .x ∉ {b, y ▷ d}}
12         forall ((n, m') ∈ E where m' ≠ m) do
13             updateWListandPropagateFacts(n, m', T)
       /* propagate facts to the distant
          parallel branches */
14     if (m : single-task ∨ m : multi-task) then
15         N_par := {k : px ∈ A_•(m), X(x) = α(k)}
16         T_par := {px : px ∈ A_•(m)}
17         T_self := {px : x ∈ A_•(m) \ T_par, ∃b, y, d .x ∉
           {b, y ▷ d})}
18         forall (k ∈ N_par where T_self ⊈ A_∘(k)) do
19             A_∘(k) := A_∘(k) ∪ T_self
20             W := W ∪ {(k', k) : (k', k) ∈ E}

21 Function updateWListandPropagateFacts(m, p, T)
22     if (m : cond) then
23         T := f_c(p, m, T)
24     if (T ⊈ A_∘(p)) then
25         W := W ∪ {(m, p)}
26         A_∘(p) := A_∘(p) ∪ T
```

Algorithm 2: Compute-$\mathcal{B}_n\mathcal{J}_n$-and-$\mathcal{B}_n^{\leftarrow}$

```
/* Initialization */
forall (n ∈ N) do
    A(n) := ∅, J(n) := ∅, B_n := ∅, B_n^← := ∅
W := {(entry, n) : (entry, n) ∈ E}
/* Visiting each edge exactly once */
while (W ≠ ∅) do
    (n, m) := select(W)
    W := W \ {(n, m)}
    if (n : barrier) then
        K := {x : x ∈ A(n), ∃n' : task. x = α(n')}
        A(m) := A(m) ∪ A(n) \ K
    else
        A(m) := A(m) ∪ A(n)
    J(m) := J(m) ∪ J(n)
    Gen := {x : x = α(m)}
    if (A(m) ∪ Gen) ⊈ A(m) then
        A(m) := A(m) ∪ Gen
        J(m) := J(m) ∪ Gen
        W:=W∪{(m, p) : (m, p) ∈ E}
forall (n ∈ N) do
    if (n : task ∨ n : barrier) then
        B_n^← := {b : b ∈ A(n), ∃p : barrier. b = α(p)}
    if (n : join) then
        J_n := {m ∈ N : m : task, ∃x ∈ J(n).X(x) =
        α(m)}
    if (n : barrier ∨ n : cond) then
        forall (p : task ∈ N where α(p) ∈ A(n)) do
            B_p := {α(n)}
```

steps:

- **Propagating facts to successors:** For each successor node p of m, *updateWListandPropagateFacts* function is called to possibly propagate facts from $\mathcal{A}_\bullet(m)$ to $\mathcal{A}_\circ(p)$ and update the worklist W. In *updateWListandPropagateFacts*, T (i.e. $\mathcal{A}_\bullet(m)$) is possibly modified by applying the function $\mathfrak{f}_c$ if m is a *conditional* node. Finally, if T contains facts not present in $\mathcal{A}_\circ(p)$, they are copied to $\mathcal{A}_\circ(p)$ and the edge (m, p) is included into the worklist W in order to process the facts in $\mathcal{A}_\circ(p)$ in the successive iterations.
- **Propagating facts to immediate parallel branches of barrier nodes:** Since parallel branches should have symmetric information, copying the MHP facts in relevant parallel branches is required for the soundness of the analysis. If n is a *barrier* node and applying transfer function f_m to the set $\mathcal{A}_\circ(m)$ generates new facts (i.e. $\mathcal{A}_\bullet(m) \setminus \mathcal{A}_\circ(m) \neq \emptyset$), then facts originated at m are copied to other parallel branches connected to n. In doing so, a temporary set T is obtained from the set $\mathcal{A}_\bullet(m) \setminus \mathcal{A}_\circ(m)$, but adding a p-tag to denote that the atom comes from the relevant parallel branches. However, facts like single barrier atoms or atoms whose synchronisation is not yet decided, such as those synchronised to d atoms (e.g. $x \triangleright d$), are not included into T, as propagating these facts in parallel branches would not generate any valid MHP information. Next, the dataflow information in $\mathcal{A}_\circ(m')$ is possibly updated from T and the worklist W may include the parallel edge (n, m'), different from (n, m), by calling the function *updateWListandPropagateFacts* described above. Thus, the MHP facts created at m are copied into the set $\mathcal{A}_\circ(m')$ such that m' is in the immediate parallel branch of m, and (n, m') is added to the worklist so that the new facts are propagated through that branch. This is the application of the $\mathit{Effect}(n)$ function in Eq. (4) that updates the set $\mathcal{A}_\circ(m')$ due to changes in $\mathcal{A}_\bullet(m)$.

- **Propagating facts to distant parallel branches:** If m is a single or multi-task node that creates a new fact x, but also receives fact py (with a p-tag), then the node k representing fact y is a distant parallel node to m. Thus, $\mathcal{A}_\circ(k)$ should receive the fact px due to symmetry, and lines 14-20 in Algorithm 1 perform this task. N_{par} contains the nodes k parallel to m. The set T_{self} contains all atoms tagged with p originated in this branch excluding lone barrier atoms, and atoms whose synchronisation is not yet evaluated (e.g. $y \triangleright d$). Next, for each node $k \in N_{par}$ placed at a parallel branch of m, if T_{self} contains more elements than $\mathcal{A}_\circ(k)$ (i.e. $T_{self} \not\subseteq \mathcal{A}_\circ(k)$), T_{self} is copied into $\mathcal{A}_\circ(k)$ and the parallel edge $(k', k) \in E$ is added into the worklist W to propagate this information through this branch. This is another application of the $Effect(n)$ function in Eq. (4) that updates the set $\mathcal{A}_\circ(k)$ due to changes in $\mathcal{A}_\bullet(m)$.

Algorithm 1 terminates when the fixpoint is reached, and the sets $\mathcal{A}_\circ$ and $\mathcal{A}_\bullet$ cannot get any new fact. Obtaining the *kill* and *gen* sets require computing the $\mathcal{B}_n$, $\mathcal{J}_n$, and $\mathcal{B}_n^{\leftarrow}$ sets. Algorithm 2 computes these sets as follows:

- it uses the worklist W to traverse the edges of the PTG in the forward direction.
- we maintain the sets $A(n)$ and $J(n)$ for each node $n \in N$ that might contain atoms of barrier nodes, task nodes, or decision nodes.
- while visiting an edge $(n, m) \in W$, first, we transfer the contents of $A(n)$ to $A(m)$ and $J(n)$ to $J(m)$. If n is a barrier node, everything except task atoms is transferred to $A(m)$ as the tasks are to be synchronized with this barrier. Next, we update $A(m)$ and $J(m)$ by including the atoms generated from the node m, and schedule all successors of m to W if the updated $A(m)$ contains more facts than the old $A(m)$.
- We reach a fixpoint when W is empty. For each $n \in N$, $\mathcal{B}_n^{\leftarrow}$ includes all the barrier atoms from $A(n)$. For each barrier or conditional node n, generating an atom x such that $A(n)$ includes the task atom t of node p, we then get $\mathcal{B}_p = \{x\}$. If n is a join node then, for all task nodes m of $\mathcal{J}_n$, the atom t of m is included in the $J(n)$ set indicating that n is reachable from m.

Now, one can get answers to the following queries from the solution space.

Query 1. What are the tasks that may run in parallel with the given task invoked from the given task-graph?

Assume that $n \in N$ is *single-* or *multi-task* PTG node invoking one or more tasks (i.e. $gen(n) \neq \emptyset$). The task atom in $gen(n)$ represents the task(s) that originates from n. All task atoms (possibly synchronized with barrier atoms) in $\mathcal{A}_\bullet(n)$ represent the tasks that are still active in this context, and thus may run in parallel with the task(s) of n. $\mathbb{M}_n$, the set of all pairs of tasks invoked by n, and tasks that may run in parallel with these, is defined by the following equation:

$$\mathbb{M}_n = \{(t_1, t_2) : t_1 \in S_\bullet \setminus S, t_2 \in S\} \cup \{(t, t) : t^+ \in S\} \quad (7)$$

where

$$S_\bullet = \mathcal{X}(\mathcal{A}_\bullet(n))$$
$$S = \mathcal{X}(gen(n))$$
$$\mathcal{X}(A) = \bigcup_{a \in A} \{\mathcal{X}(a)\}$$

Query 2. What are the pairs of tasks invoked from a given task-graph that may run in parallel?

All MHP facts for the PTG $\mathfrak{g} = (N, E)$ can be obtained as follows:

$$\mathbb{M}_\mathfrak{g} = \bigcup_{n \in N,\ n:task} \mathbb{M}_n \quad (8)$$

Example 1: Let us consider the PTG in Fig. 3. Let the tasks T_i, and barriers B_i be represented by the t-atoms t_i, and b-atoms b_i, respectively, for $1 \leq i \leq 5$. The data-flow analysis as explained above provides the following MHP facts (after removing the lonely b-atom and the symbol $\triangleright$ for simplification), as part of the sets $\mathcal{A}_\circ(n)$ and $\mathcal{A}_\bullet(n)$ respectively, for all task nodes once the fixpoint is reached.

Node n	$\mathcal{A}_\circ(n)$	$gen(n)$
n_0	$pt_1b_3, pt_2^+b_3, pt_3b_4, pt_4^+$	t_0b_4
n_1	$pt_0b_4, pt_2^+b_3, pt_3b_4$	t_1b_3
n_2	$pt_0b_4, pt_1b_3, pt_3b_4$	$t_2^+b_3$
n_3	$pt_0b_4, pt_1b_3, pt_2^+b_3$	t_3b_4
n_4	pt_0b_4, pt_5	t_4^+
n_5	pt_4^+	t_5

Node n	$\mathcal{A}_\bullet(n)$
n_0	$t_0b_4, pt_1b_3, pt_2^+b_3, pt_3b_4, pt_4^+$
n_1	$pt_0b_4, t_1b_3, pt_2^+b_3, pt_3b_4$
n_2	$pt_0b_4, pt_1b_3, t_2^+b_3, pt_3b_4$
n_3	$pt_0b_4, pt_1b_3, pt_2^+b_3, t_3b_4$
n_4	pt_0b_4, t_4^+, pt_5
n_5	pt_4^+, t_5

The following MHP facts are obtained from (7), and (8):

$$\mathbb{M}_{n_0} = \{(t_1, t_0), (t_2, t_0), (t_3, t_0), (t_4, t_0)\}$$
$$\mathbb{M}_{n_1} = \{(t_0, t_1), (t_2, t_1), (t_3, t_1)\}$$
$$\mathbb{M}_{n_2} = \{(t_0, t_2), (t_1, t_2), (t_3, t_2), (t_2, t_2)\}$$
$$\mathbb{M}_{n_3} = \{(t_0, t_3), (t_1, t_3), (t_2, t_3)\}$$
$$\mathbb{M}_{n_4} = \{(t_0, t_4), (t_5, t_4), (t_4, t_4)\}$$
$$\mathbb{M}_{n_5} = \{(t_4, t_5)\}$$
$$\mathbb{M}_\mathfrak{g} = \{(t_0, t_1), (t_0, t_2), (t_0, t_3), (t_0, t_4), (t_1, t_2), (t_1, t_3), (t_2, t_3), (t_5, t_4), (t_2, t_2), (t_4, t_4)\}$$

In Fig. 6 we can see how the MHP facts are reported back to the UML model to allow the developer to grasp at a glance the analysis results in relation to the task model (e.g., $\mathbb{M}_{n_0}$ is represented in the figure by 'M_n_0').

Worst-Case Asymptotic Complexity: Algorithm 1 performs the data-flow analysis of a PTG which is a directed acyclic graph. A PTG edge (n, m) is visited in each iteration of the *while-loop*. For each successor p of m, an edge is included into the worklist W by calling the

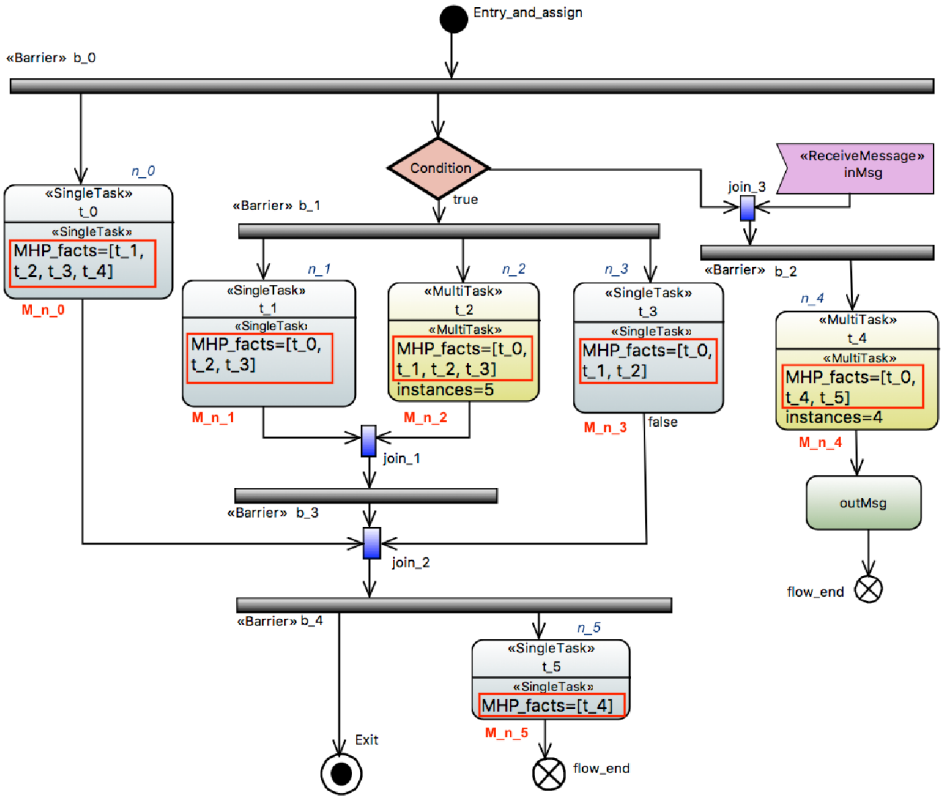


FIGURE 6. **MHP analysis results back-propagated to UML model.**

updateWListandPropagateFacts function at line 8 if $\mathcal{A}_\circ(p)$ does not contain all facts derived from $\mathcal{A}_\bullet(m)$. The two *if* instructions at lines 9 and 14 cause some edges (in parallel branches) to be visited more than once; the MHP facts are copied into the relevant parallel branches if the facts are not there yet, and parallel edges are stored into the worklist W in order to visit them later. Thus, the number of times an edge can be visited is controlled by the number of MHP facts that can be present in its $\mathcal{A}$ set. Any such set may contain task atoms, barrier and conditional atoms, and atoms due to synchronisation between tasks and barriers.

Consider the PTG (N, E) in which there are at most $\mathbf{t}$ *single-task* or *multi-task* nodes, and $\mathbf{b}$ *barrier* or *conditional* nodes. An $\mathcal{A}$ set can contain at most $\mathbf{t}$ task atoms, $\mathbf{b}$ barrier and conditional atoms, and $\mathbf{t}$ task atoms synchronised with barrier atoms by the operator $\triangleright$ since a single task can synchronise its execution with at most one barrier. This implies that the maximum cardinality of any $\mathcal{A}$ set is $2 * \mathbf{t} + \mathbf{b}$. Thus, it holds that $|\mathcal{A}| \leq 2 \cdot |N|$ since $2 * \mathbf{t} + \mathbf{b} \leq 2 * |N|$. Furthermore, $2 \cdot |E| \cdot |N|$ is an upper bound on the number of times that edges of the given PTG are visited by the algorithm. By choosing a suitable data structure all set operations and transfer functions can be computed in time $O(|\mathcal{A}|)$. So, $O(|E| \cdot |N|^2)$ is the worst-case asymptotic time complexity of the *while*-loop in Algorithm 1.

Algorithm 1 calls Algorithm 2 to compute the $\mathcal{B}_n^{\leftarrow}$, $\mathcal{B}_n$ and $\mathcal{J}_n$ sets. The complexity of Algorithm 2 is dominated by the while-loop and the second for-loop. In the while-loop, the A sets collect task, barrier or conditional atoms. The maximum size of any A set is $\mathbf{t} + \mathbf{b} \leq |N|$ and the set operations in each iteration of the while loop thus take at most $O(|N|)$ time if choosing a suitable data structure for the sets. Moreover, any particular edge can be visited at most $|N|$ times due to changes in the A set. Thus, the while loop of Algorithm 2 has $O(|E| \cdot |N|)$ worst-case asymptotic time complexity. The second for-loop iterates at most $|N|$ times, computing $\mathcal{B}_n^{\leftarrow}$ from $A(n)$ requires at most $|N|$ operations, and the inner for-loop iterates at most $|N|$ times. Thus, the asymptotic worst-case time complexity of the second for-loop is $O(|E| \cdot |N|)$, and Algorithm 2 is $O(|E| \cdot |N|)$ by assuming $|E| \geq |N|$ in the worst-case. Thus, the complexity of Algorithm 1 is dominated by the while-loop which is $O(|E| \cdot |N|^2)$.

Equations (7) and (8) are used to compute MHP facts after completing the data-flow analysis. $2 \cdot |N|$ is the upper bound of the size of any $\mathcal{A}_\bullet$ set, and any *gen* set has at most one element. Thus (7) requires $O(|N|)$ operations, and (8) requires $O(|N|^2)$ operations.

B. COMMUNICATION AMONG TASK-GRAPHS

In this section, we consider scenarios in which PTGs may communicate with each other. A task originated from one PTG may send a message to another PTG immediately before finishing execution. Upon receiving the message, the second PTG may invoke a task to execute. In Fig. 7 (a) and (b), tasks are invoked from nodes n_2 and n_4 in PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$

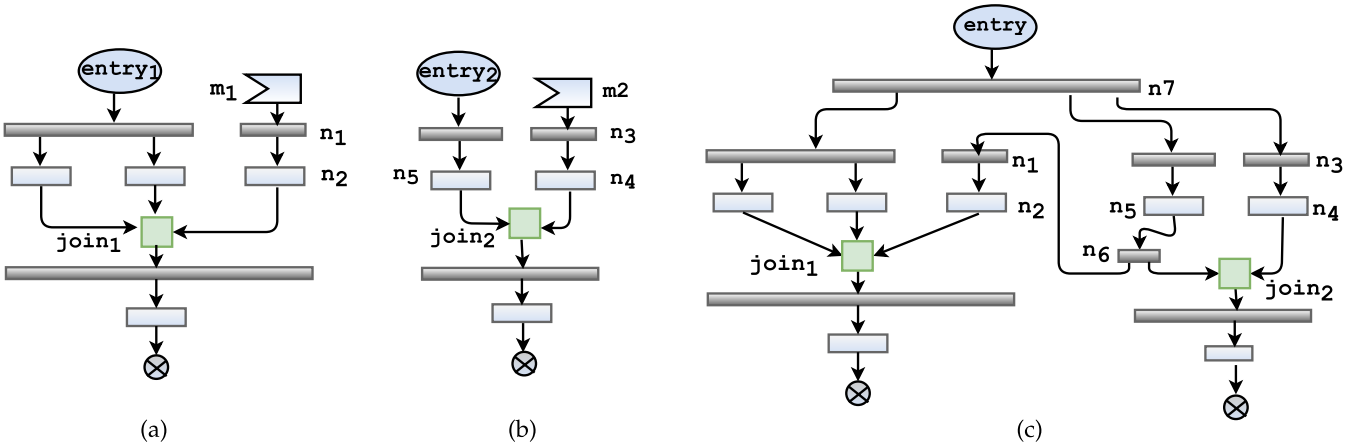


FIGURE 7. Direct communication between PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$: node n_5 in PTG $\mathfrak{g}_2$ sends a message to $\mathfrak{g}_1$ which should be captured at m_1. Dataflow analysis in Algorithm 1 can be performed over the combined PTG in Fig. (c). (a) PTG $\mathfrak{g}_1$. (b) PTG $\mathfrak{g}_2$. (c) A PTG obtained by combining $\mathfrak{g}_1$ and $\mathfrak{g}_2$.

upon receiving messages m_1 and m_2 respectively. The MHP analysis in Section V-A is only able to compute MHP facts from individual PTGs and does not consider the implicit dependencies created due to message transfers among PTG nodes of different PTGs. Thus, applying Algorithm 1 on communicating PTGs will only give us partial results. For example, consider the PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$ in Fig. 7. The previous MHP analysis will completely miss any MHP facts involving node n_2 as the edge from n_2 to the $join_1$ node will never be included into the worklist during the fixpoint iteration. Similarly, node n_4 of PTG $\mathfrak{g}_2$ will also not be taken into account in the MHP analysis results. Moreover, any inter-PTG parallelism (i.e. MHP facts involving tasks originated at different PTGs) between $\mathfrak{g}_1$ and $\mathfrak{g}_2$ will also not be generated.

Our solution to this scenario is to (i) combine the communicating PTGs into a single PTG by connecting the relevant task and barrier nodes such that the semantics of message send and receive is retained, and (ii) perform the analysis specified in Section V-A. For this, we need to identify the tasks that send messages and their destinations. These message send options are not explicitly modelled in the PTGs, but rather carried by the behavioural DSP C code of the related tasks; therefore, we need to syntactically analyse the code. We detect the message senders and receivers by traversing the abstract syntax tree of the C code and look for the particular commands sending messages. After analysing the C code, we obtain a set of pairs of PTG nodes such that each pair (n, m) consists of a task node n sending a message (i.e. the task invoked from n sends a message before its termination) which is received by the node m.

Suppose we have two PTGs $\mathfrak{g}_1 = (N_1, E_1)$ and $\mathfrak{g}_2 = (N_2, E_2)$, and the set $\{(n_1, m_1), \ldots, (n_k, m_k)\}$ of message senders and receivers. We also assume the set of message receiving nodes $\{m_{k+1}, \ldots, m_{k+j}\} \subset N_1 \cup N_2$ for which no message senders are either obtained due to lack of information, or outside the scope of graphs $\mathfrak{g}_1$ and $\mathfrak{g}_2$. We then connect these PTGs according to the following definition:

Definition 2 (Combining PTGs): We obtain the combined PTG (N, E) as follows:

1) $N = (N_1 \cup N_2) \setminus \{e_1, e_2, m_{k+1}, \ldots, m_{k+j}\} \cup \{e, q, q_1, \ldots, q_k\}$ where e_1, e_2 are the entry nodes of $\mathfrak{g}_1$ and $\mathfrak{g}_2$, e is the new entry node of the combined PTG, and $q, q_1, \ldots, q_k$ are $k+1$ new barrier nodes,
2) E is the least set of edges satisfying the following:
   a) $(E_1 \cup E_2) \setminus E' \subseteq E$ where $E' = \{(e_1, b_1), (e_2, b_2), (m_1, l_1), \ldots, (m_{k+j}, l_{k+j})\}$, b_1, b_2 are the successors of the entry nodes e_1, e_2, and l_i is the successor node of message receiving node m_i for $i = 1, \ldots, k+j$,
   b) $\{(e, q), (q, b_1), (q, b_2), (q_1, l_1), \ldots, (q_k, l_k)\} \subset E$, where $q, q_1, \ldots, q_k$ are new barrier nodes,
   c) $\{(n_1, q_1), \ldots, (n_k, q_k)\} \subset E$, that is: each message sending node n_i is connected to the barrier node q_i,
   d) for any $(n_i, n_i') \in E_1 \cup E_2$ holds that $(q_i, n_i') \in E$,
   e) finally $\{(q, l_{k+1}), \ldots, (q, l_{k+j})\} \subset E$, connecting the barrier node q, which is connected with the new entry node, to all nodes connected with the message receiving nodes for which no message senders are known.

Intuitively, for each task node $n_i \in N_1 \cup N_2$ sending a message received by $m_i \in N_1 \cup N_2$, we:

(i) introduce a new barrier node $q_i \in N$ and an edge $(n_i, q_i) \in E$,
(ii) remove the edge $(n_i, n_i') \in E_1 \cup E_2$, if any, and add the edge $(q_i, n_i') \in E$, and
(iii) remove the edge $(m_i, m_i') \in E_1 \cup E_2$ but add the edge $(q_i, m_i') \in E$.

For example, in Fig. 7(a) (b), assume that the task originated from node n_5 in $\mathfrak{g}_2$ sends a message which is supposed to be received by m_1 in $\mathfrak{g}_1$. In Fig. 7(c), we include a new barrier node n_6 and the edges (n_5, n_6), (n_6, n_1), and $(n_6, join_2)$ but remove the edges $(n_5, join_2)$, and (m_1, n_1). Moreover, for each message receiving node m_i for which no message sender is

known, we remove the edge (m_i, m_i') but add an edge (q, m_i') where q is the new barrier node connected with the new entry node. The reason for this connection is to assume the worst-case scenario that the message has arrived at the beginning. For example, in Fig. 7(b), the message sender for m_2 is not known; we thus introduce the edge from the new barrier node n_7 to n_3 which was connected with m_2.

If we have multiple communicating PTGs, we connect them in a pairwise fashion according to Definition 2, and finally, perform the analysis in Section V-A to get the MHP results.

C. TASK-PARALLELISM IN MULTIPLE TASK-GRAPHS

Here, we consider PTGs that may run in parallel but do not communicate explicitly via messages among themselves. Rather, one graph may send a message to an actor that may invoke another graph and run in parallel. So, even though there is no explicit connection between them, there are implicit dependencies that should be taken into account in order to be precise. Let us consider two PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$ being executed in parallel. Two tasks invoked from two different PTGs running in parallel do not synchronise their actions with any barrier, as barriers are task-local. If there are no other means of synchronisation, any task invoked from the first graph may run in parallel with another task invoked from the second graph. Suppose $\mathcal{T}_1$ and $\mathcal{T}_2$ are the sets of task atoms generated from the PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$. One may be tempted to compute the MHP facts for $\mathfrak{g}_1$ and $\mathfrak{g}_2$ as follows:

$$\mathbb{M}_{\mathfrak{g}_1 \| \mathfrak{g}_2} = \mathbb{M}_{\mathfrak{g}_1} \cup \mathbb{M}_{\mathfrak{g}_2} \cup \mathcal{T}_1 \times \mathcal{T}_2 \tag{9}$$

where $\mathcal{T}_1 \times \mathcal{T}_2 = \{(t_1, t_2) : t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2\}$.

There are two potential problems in applying Eq. (9) to compute all MHP facts from a parallel application. Even though tasks invoked by two different PTGs running in parallel do not synchronise their actions explicitly via barriers, there may still be implicit dependencies and Eq. (9) may provide imprecise results in such cases. A running task may send a message to an actor, which in turn can invoke a new task-graph. Clearly, all tasks that are invoked by the first graph, and have finished their execution before the message is sent, will not execute in parallel with the tasks invoked by the new task-graph. The second potential problem is the generalization of the previous computation for $n > 2$ PTGs. Thus, we need a more precise method to compute the MHP facts of n task-graphs running in parallel.

To achieve a more precise analysis, we need a precedence or exclusion relation $\preceq$ among tasks invoked from a PTG based on the task invocation and completion order, and the mutual exclusion property of task invocation. $t \preceq t'$ will hold if either t has finished execution before t' starts, or only one of t or t' can be invoked from the originating PTG. If $t \preceq t'$, where the PTG $\mathfrak{g}$ is invoked by the task t' (i.e. t' sends a message by to an actor), then we know that any task invoked from $\mathfrak{g}$ can never run in parallel with t; including this information will give us more precise results. $t \preceq t'$ also implies that t and t' cannot run in parallel. The analysis to determine if two tasks cannot execute together is historically called *Cannot-Happen-Together* (CHT), or *non-concurrency analysis* [18]. The CHT problem is the inverse of the MHP problem. A task pair (t_1, t_2) is in the solution of the CHT problem iff it is not in the solution of the MHP problem. However, just by knowing that (t_1, t_2) is in the solution of the CHT problem, we cannot tell whether $t_1 \preceq t_2$ or $t_2 \preceq t_1$ is true. Thus, the relation $\preceq$ carries more information about the possible order of task executions than the MHP facts computed by the MHP analysis in Section V-A.

Definition 3 (Precedence or Exclusion (POE) Relation $\preceq$): The POE relation $\preceq$ is the union of the *strict precedence* relation $\lhd$ and the *mutual exclusion* relation $\triangledown$ (i.e. $\preceq = \lhd \cup \triangledown$), where $\lhd$ and $\triangledown$ are defined as follows:

- $t \lhd t'$ holds iff t has finished execution before t' starts, and
- $t \triangledown t'$ holds iff either t or t' (but not both) can be invoked from the originating PTG.

Note that $\lhd$ is transitive, and $\triangledown$ is symmetric. We can obtain an under-approximation of the strict precedence relation $\lhd$ from the following static properties of PTGs:

$t \lhd t'$ holds iff there exists a path from a task node generating atom t to the task node generating t' and the path includes a barrier node in-between.

The existence of the barrier node in between ensures that t has finished running before t' starts. Thus, the relation $\lhd$ among tasks invoked from PTG $\mathfrak{g}$ can be obtained by solving the graph reachability problem. In particular, we perform the following graph reachability problem after completing the data-flow analysis illustrated in Section V-A.

We navigate the PTG $\mathfrak{g}$ forward from all *barrier* nodes. Suppose n is a *barrier* node, and $\mathcal{A}_\circ(n)$ contains the MHP facts valid immediately before node n. Then, $\mathcal{X}(kill(n) \cap \mathcal{A}_\circ(n))$ contains all task atoms representing the tasks invoked from $\mathfrak{g}$ and that have completed their execution. During the navigation of the PTG $\mathfrak{g}$ from node n, we consider all reachable *single-* or *multi-task* nodes m. As m is reachable from n, we thus infer $t \preceq t'$ for all $t \in \mathcal{X}(kill(n) \cap \mathcal{A}_\circ(n))$ and $t' \in gen(m)$.

The mutual exclusion relation $t \triangledown t'$ holds if t and t' are originated at mutually exclusive branches. Intuitively, if t is originated in the *true* branch, then t' is originated in the false branch of a *cond* node, or vice versa. Nodes n_1 and n_2 are at the *mutually exclusive* branches of a PTG $\mathfrak{g}$ iff there exists a cond node m such that m has two sucessor nodes m_1 and m_2, there exist paths $[m_1..n_1]$ and $[m_2..n_2]$ and there does not exist paths $[m_1..n_2]$, and $[m_2..n_1]$. However, we can obtain the relation $\triangledown$ from the solution of the MHP problem and the relation $\lhd$ by considering the following facts:

- the CHT problem is the inverse of the MHP problem (i.e. $CHT = \overline{MHP}$).
- the CHT solution includes the relation $\lhd$ and $\triangledown$ (that is, $CHT = \triangledown \cup \{(t_1, t_2), (t_2, t_1) : t_1 \lhd t_2\}$), and
- $\triangledown$ is symmetric.

Thus, we obtain $\triangledown$ according to the following equation:

$$\triangledown = \overline{MHP} \setminus (\triangleleft \cup \{(t_1, t_2) : (t_2, t_1) \in \triangleleft\}) \tag{10}$$

Algorithm 3 describes how the analysis infers the POE relation $\preceq$ from the PTG $\mathfrak{g} = (N, E)$. The **while** loop (lines 4-15) computes the *strict precedence* relation $\triangleleft$ and line 16 computes the final POE relation $\preceq$ from the equation 10. Note that all task pairs $(t_1, t_2) \notin \mathbb{M}_{\mathfrak{g}}$ that are not computed by the **while** loop in Algorithm 3 must be in the mutually exclusive branches of the graph $\mathfrak{g}$. We included all such pairs (t_1, t_2) and (t_2, t_1) in the PO_F relation. The expression $V[n \leftarrow true]$ denotes an array for which $V[n \leftarrow true][m] = V[m]$ if $n \neq m$, and $V[n \leftarrow true][n] = true$.

Algorithm 3: PrecedenceOrExclusionRelation

```
   /* Initialization */
1  forall (n ∈ N) do V[n] := false
2  W := {(n, F, V') : F = 𝒳(kill(n) ∩ 𝒜∘(n)), n ∈
   N, F ≠ ∅, V' = V}
3  PO := ∅
4  while (W ≠ ∅) do
5  |  W := W \ {(n, F, V) : (n, F, V) = select(W)}
   |  /* Remove from W */
6  |  forall ((n, m) ∈ E where V[m] = false) do
7  |  |  if (n : task) then
8  |  |  |  PO := PO∪{(t, t') : t ∈ F, t' ∈ 𝒳(gen(m))}
9  |  |  |  F' := F ∪ {t : t' ∈ 𝒳(gen(m))}
10 |  |  |  W := W ∪ {(m, F', V[n ← true])}
11 |  |  else
12 |  |  |  W := W ∪ {(m, F, V[n ← true])}
13 |  |  end
14 |  endfor
15 end
16 PO_F := PO ∪ {(t1, t2), (t2, t1) : (t1, t2) ∉ 𝕄_𝔤, (t1, t2) ∉
   PO, (t2, t1) ∉ PO}
17 return PO_F
```

Example 2: Consider the MHP facts generated in Example 1, and the *barrier* node b_3 in the PTG in Fig. 3. The set of tasks that have completed their execution after node b_3 is $\mathcal{X}(kill(b_3) \cap \mathcal{A}_\circ(b_3)) = \{t_1, t_2\}$. Next, we navigate the PTG in the forward direction and reach the *single-task* node t_5 such that $gen(t_5) = \{t_5\}$. Thus, we infer the relations $t_1 \preceq t_5$ and $t_2 \preceq t_5$.

Let us consider two parallel PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$ such that a task t invoked from the PTG $\mathfrak{g}_1$ sends a message to an actor invoking $\mathfrak{g}_2$. Suppose that we have computed the POE relation $\preceq$ among tasks referred in $\mathfrak{g}_1$. Next, we filter out all task atoms from $\mathcal{T}_1$ that have completed their execution before task t started, and compute the set $\mathcal{T}_1^{\preceq t}$ as:

$$\mathcal{T}_1^{\preceq t} = \{t' : t' \in \mathcal{T}_1, t' \not\preceq t, t' \neq t\}$$

The following equation defines the set of pairs of tasks invoked from PTGs $\mathfrak{g}_1$ and $\mathfrak{g}_2$ that may run in parallel, and provides more precise results than equation 9:

$$\mathbb{M}_{\mathfrak{g}_1 \| \mathfrak{g}_2} = \mathbb{M}_{\mathfrak{g}_1} \cup \mathbb{M}_{\mathfrak{g}_2} \cup (\mathcal{T}_1^{\preceq t} \times \mathcal{T}_2) \tag{11}$$

In order to generalize (11) for more than two PTGs, we need contextual information of PTGs that should indicate if one is originated due to the action of another PTG. Let us consider that, for every PTG $\mathfrak{g}$, we have a unique context $C(\mathfrak{g}) = \langle t, \mathfrak{g}' \rangle$ denoting that $\mathfrak{g}$ is originated due to sending a message from the task t invoked by the PTG $\mathfrak{g}'$. If $\mathfrak{g}$ is not originated due to the action of another PTG, then its contextual information is set to $C(\mathfrak{g}) = \perp$. On one hand, the uniqueness constraint of contextual information forbids two different PTGs being present in the same PTG context. On the other hand, it allows the same PTG to be present in the context of two different PTGs, meaning that two different PTGs can originate due to the actions of a single graph. Moreover, we forbid creating cycles of context information of the form $C(\mathfrak{g}_1) = \langle t_1, \mathfrak{g}_2 \rangle, C(\mathfrak{g}_2) = \langle t_2, \mathfrak{g}_3 \rangle, \ldots C(\mathfrak{g}_k) = \langle t_k, \mathfrak{g}_1 \rangle$ as it does not make any sense that $\mathfrak{g}_k$ is originated due to the action of $\mathfrak{g}_1$ which in turn invokes $\mathfrak{g}_1$. We also extend the POE relation $\preceq$ among the inter-PTG tasks such that $t'' \preceq t'$ holds for all $t' \in \mathcal{T}_i$, $t'' \in \mathcal{T}_j$ but either $t'' \preceq t$ or $t'' = t$ for the context $C(\mathfrak{g}_i) = \langle t, \mathfrak{g}_j \rangle$. Clearly, all tasks t'' of $\mathcal{T}_j$ has finished execution before $\mathcal{T}_i$ is invoked. Algorithm 4 describes how to compute the MHP facts for n PTGs in the presence of contextual information.

Algorithm 4: MayHappenParallelTasks($\mathfrak{T} = (V, R), C$)

```
1  𝕄_{𝔤1‖...‖𝔤η} := ⋃_{i=1...n} 𝕄_{𝔤i}
2  ∀1 ≤ i ≤ η.Facts[i] := ∅
3  while (R ≠ ∅) do
4  |  Select 𝔤i → 𝔤j ∈ R such that ¬∃𝔤k.𝔤k → 𝔤i ∈ R
   |  (t, 𝔤i) ← C(𝔤j)
5  |  𝒯_prev := {t' : (t', t'') ∈ Facts[i], t' ⋠ t, t' ∉ 𝒯i}
6  |  𝒯i^{≼t} := {t' : t' ∈ 𝒯i, t' ⋠ t, t ≠ t'}
7  |  Facts[j] := Facts[j] ∪ 𝒯i^{≼t} × 𝒯j ∪ 𝒯_prev × 𝒯j
8  |  if ¬∃(𝔤i → 𝔤k ∈ R ∧ j ≠ k) then V := V \ {𝔤i}
9  |  R := R \ {𝔤i → 𝔤j}
10 |  𝕄_{𝔤1‖...‖𝔤η} := 𝕄_{𝔤1‖...‖𝔤η} ∪ Facts[j]
11 end
12 while (𝔤i, 𝔤j ∈ V ∧ i ≠ j) do
13 |  Facts[i] := Facts[i] × Facts[j]
14 |  𝕄_{𝔤1‖...‖𝔤η} := 𝕄_{𝔤1‖...‖𝔤η} ∪ Facts[i]
15 |  V := V \ {𝔤j}
16 end
17 return 𝕄_{𝔤1‖...‖𝔤η}
```

Suppose that we have η parallel PTGs $\mathfrak{g}_1, \ldots, \mathfrak{g}_\eta$, and $C(\mathfrak{g}_i)$ is the context information of $\mathfrak{g}_i$ for all $i = 1, \ldots, \eta$. A graph of PTGs $\mathfrak{T} = (V, R)$ can be constructed from the contextual information where $V = \{\mathfrak{g}_1, \ldots, \mathfrak{g}_\eta\}$ is the set of vertices, and $R = \{\mathfrak{g}_i \rightarrow \mathfrak{g}_j : C(\mathfrak{g}_j) = \langle t, \mathfrak{g}_i \rangle, 1 \leq i, j \leq \eta\}$ is the set of edges. $\mathfrak{T}$ must be a forest (i.e. multiple disjoint trees) due to the possible disjointness of edges and the unique context information. Note that the unique context

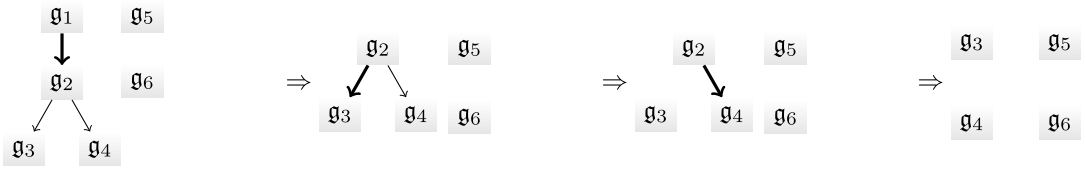


FIGURE 8. Tree structure of PTGs processed by Algorithm 4. Thick edges are the edges currently being processed.

constraints and the prohibition of forming cycles of contextual information make $\mathfrak{T}$ an acyclic graph. For any $\mathfrak{g}_k \in V$, and $\mathfrak{g}_i \rightarrow \mathfrak{g}_j \in E$, it may be the case that $\mathfrak{g}_i$ is the only parent of $\mathfrak{g}_j$, and $\mathfrak{g}_k$ is not connected with either $\mathfrak{g}_i$ or $\mathfrak{g}_j$. Intuitively, Algorithm 4 computes global MHP facts as follows:

- The global MHP facts $\mathbb{M}_{\mathfrak{g}_1 \| \ldots \| \mathfrak{g}_\eta}$ include all local MHP facts $\mathbb{M}_{\mathfrak{g}_i}$ obtained from the data-flow analysis for each graph $\mathfrak{g}_i$ for $i = 1, \ldots, \eta$ (line 1).
- Next, we consider all edges $\mathfrak{g}_i \rightarrow \mathfrak{g}_j \in R$ such that $\mathfrak{g}_i$ is the current root node of a tree in $\mathfrak{T}$. MHP facts of two combined PTGs $\mathfrak{g}_i$ and $\mathfrak{g}_j$ are computed by applying part of Eq. (11) (line 7). $\mathcal{T}_{prev}$ contains all tasks that are active when the PTG $\mathfrak{g}_j$ is invoked and may run in parallel with the tasks invoked from $\mathfrak{g}_j$. If vertex $\mathfrak{g}_i$ has only one successor, then it is removed from V (line 8) as $Facts[j]$ contains the MHP facts for the two combined PTGs. The edge $\mathfrak{g}_i \rightarrow \mathfrak{g}_j$ is removed from E as well, and the computed facts are stored into the global MHP facts $\mathbb{M}_{\mathfrak{g}_1 \| \ldots \| \mathfrak{g}_\eta}$. This process is repeated (lines 3-11) until there are no edges in $\mathfrak{T}$, and Fig. 8 illustrates this process.
- $\mathfrak{T}$ now consists of the set V of isolated vertices, and for any $\mathfrak{g}_i \in V$, $Facts[i]$ contains combined MHP facts for all PTGs connected with $\mathfrak{g}_i$. The final facts are computed by repeatedly applying the operator $\times$ over $Facts[i]$ and $Facts[j]$ such that $\mathfrak{g}_i, \mathfrak{g}_j \in V$, and are stored into the global MHP facts (lines 12-15). We define the operation $Facts[i] \times Facts[j]$ as follows:

$$Facts[i] \times Facts[j] = \bigcup_{k,l \in \{1,2\}} T_{(i,k)} \times T_{(j,l)}$$

  where,
  - $T_{p,1} = \{t : \exists t'.(t, t') \in Facts[p]\}$,
  - $T_{p,2} = \{t : \exists t'.(t', t) \in Facts[p]\}$, and
  - $p \in \{i, j\}$.

Worst-Case Asymptotic Complexity: Algorithm 3 computes the POE relation after the data-flow analysis and its complexity is dominated by the while-loop. Initially, the worklist contains information of all *barrier* nodes. For each *barrier* node, all reachable nodes from the *barrier* node are visited exactly once. Thus, the loop iterates at most $\mathbf{b} \cdot |N|$ times (let $\mathbf{b}$ be the maximum number of *barrier* nodes in any PTG). All operations in the while-loop require constant time except the computation of the PO set that requires linear time with respect to the size of the F set. The size of F is bounded by the upper bound $2 \cdot |N|$ on the size of the $\mathcal{A}$ set. Line 16 computes the final partial orders in PO_F set which can be computed by at most $|N|^2$ iterations. Thus, the complexity of Algorithm 3 can be expressed by $O(\mathbf{b} \cdot |N|^2)$.

Let us consider the PTGs (N_i, E_i), for any $1 \leq i \leq \eta$, and Algorithm 4, which computes MHP facts for η PTGs. We break the complexity analysis of this algorithm into the following stages:

- The complexity of line 1, which combines Algorithm 1, with complexity $O(|E_i| \cdot |N_i|^2)$, and Equation 8, with complexity $O(|N_i|^2)$ to η PTGs (N_i, E_i) for $i = 1, \ldots, \eta$, can be expressed by $O(\eta \cdot |E| \cdot |N|^2)$ where $|E| = max(|E_1|, \ldots, |E_\eta|)$ and $|N| = max(|N_1|, \ldots, |N_\eta|)$.
- The complexity of the algorithm is further dominated by the $\times$ operations at line 6 and 12.
- Line 6 requires computing the POE relations for the PTGs with complexity $O(\mathbf{b} \cdot |N|^2)$. The total complexity of computing the POE relations for η PTGs is $O(\eta \cdot \mathbf{b} \cdot |N|^2)$.
- Let us assume that the maximum number of pairs of tasks in the $Facts[i]$ set is k for all $1 \leq i \leq \eta$. Computing the set $Facts[i]^{\preceq t}$ requires computing the partial order relation with complexity $O(\mathbf{b} \cdot |N|^2)$ followed by the set operations with linear complexity with respect to the size of the set $Facts[i]$ which is proportional to k. $Facts[i]^{\preceq t}$ may contain at most k elements. The instruction $Facts[i] \times Facts[j]$ will compute k^2 elements with complexity $O(k^2)$. The two while-loops control the maximum number of $\times$ operations. The first while-loop will iterate at most the number of edges of the forest $\mathfrak{T}$, and the second while-loop will iterate at most the number of isolated vertices after removing all edges in the first loop. If the forest $\mathfrak{T}$ is a tree, the second loop will not iterate at all, and if $\mathfrak{T}$ does not have any edge, the first loop will not iterate at all. Since each PTG has a unique context $\mathfrak{T}$ can have at most $\eta - 1$ edges, and the maximum number of times the $\times$ operation can be applied is $\eta - 1$. Thus, since the cost of each $\times$ operation is $O(k^2)$ and each operation produces elements in a multiplicative manner, the complexity of the two while-loops is $O(k^\eta)$.
- The maximal number of elements in any $Fact[i]$ set k can be proportional to $|N|^2$ (at initialization) even though it is much less in practice. Thus, the worst-case complexity of the two while-loops is $O(|N|^{2\eta})$, which is also the worst-case complexity of Algorithm 4.

D. WHOLE SYSTEM ANALYSIS

In Sec. V-C, we assume that we have η parallel PTGs $\mathfrak{g}_i = (N_i, E_i)$ and their contextual information $C(\mathfrak{g}_i)$ for $1 \leq i \leq \eta$. In this section we define a method to systematically identify an overapproximation of all those PTGs $\mathfrak{g}_i$ that may run in parallel, and their contextual information $C(\mathfrak{g}_i)$. These results can be used by Algorithm 4 to compute the global MHP facts. We focus on a system that starts its execution by creating a number of actors. We are interested in those actor behaviors that either invoke PTGs or create other actors based on the type of the received message. Intuitively, we perform the whole system analysis in the following steps:

- First, we define a generic context-free grammar (CFG) that captures various kinds of code parallelism and dependencies in any actor-based system developed according to the computational model in Sec. II.
- Next, we scan the related code and generate an instance of the previously defined CFG. The generated instance of the CFG captures parallelism and dependencies specific to the given system.
- The parallelism in the system is detected by analysing the strings that are generated by the instantiated grammar.

1) DEFINING THE GENERIC CFG

We assume a unique nonterminal symbol for each kind of actor, multiple actors of the same kind, actor receiving a specific kind of message, and PTG. We also assume a unique terminal symbol (atom) for each task and PTG. Fig. 9(a) illustrates the generic CFG that we defined for capturing various code dependencies and parallelism, and we would like to generate an instance of it. In the following, we discuss the production rules of the CFG.

$$
\begin{array}{lcl}
S & \rightarrow & ActInit \\
ActInit & \rightarrow & par\,(\,Actor,\,\ldots,\,Actor\,) \\
Actor & \rightarrow & ActMsg \quad Actor \mid \epsilon \\
ActMsg & \rightarrow & Actor \mid TaskGraph \\
TaskGraph & \rightarrow & GAtom \quad Tasks \\
Tasks & \rightarrow & Task\,Tasks \mid Task \\
Task & \rightarrow & \xrightarrow{TaskAtom} \quad ActMsg \mid \epsilon
\end{array}
$$

(a)

$$
\begin{array}{lcl}
S & \rightarrow & par\,(\,A_1, A_2\,) \\
A_1 & \rightarrow & AM_{11} \mid AM_{12} \\
A_2 & \rightarrow & AM_{21} \\
AM_{11} & \rightarrow & TG_1 \\
AM_{12} & \rightarrow & TG_2 \\
AM_{21} & \rightarrow & TG_3 \\
TG_1 & \rightarrow & GA_1\ T_1 \\
TG_2 & \rightarrow & GA_2\ T_2 \\
TG_3 & \rightarrow & GA_3\ T_3
\end{array}
\qquad
\begin{array}{lcl}
GA_1 & \rightarrow & g_1 \\
GA_2 & \rightarrow & g_2 \\
GA_3 & \rightarrow & g_3 \\
T_1 & \rightarrow & \epsilon \\
T_2 & \rightarrow & \epsilon \\
T_3 & \rightarrow & \xrightarrow{t} \quad AM_{12}
\end{array}
$$

(b)

FIGURE 9. (a) Generic CFG capturing parallelism and dependencies among actors, PTGs, and tasks. (b) An instance of (a) having two actors receiving three different types of messages and invoking three PTGs.

S is the initial nonterminal symbol of the grammar. The right-hand side of the *ActInit* rule contains nonterminals representing the various kinds of actors created initially. We do not keep information of all actors of the same kind as they will have similar behaviors (i.e create similar PTGs or similar actors) and there may be infinitely many such actors. Rather, if several actors of the same kind are created, we create the nonterminal (say A) representing one of the actors and another nonterminal (say A^*) representing other actors of that kind running in parallel with the actor that A represents. This abstraction is sound, since it creates an overapproximation of all MHP facts. The third production rule (*Actor*) states that each actor may receive different types of message and dispatch them one by one, or have no observable behavior (ϵ). We assume that the message type can be determined statically and this reflects our benchmark application code. Each actor receiving a specific kind of message is represented by a unique nonterminal symbol in the grammar. The fourth rule (*ActMsg*) states that, on receipt of the message, the actor may create a new actor, invoke a PTG, or simply ignore the message. The fifth rule (*TaskGraph*) is applicable when a PTG is invoked by an actor and it generates a unique graph atom (from *GAtom*) for the PTG (e.g. $\mathfrak{g}$ representing that PTG) followed by a nonterminal representing one or more kinds of tasks invoked by the graph. The final rule either generates an empty string, or a string, e.g. $\xrightarrow{t}$, where the terminal symbol t represents the *Task* generated by the *TaskAtom*, followed by the nonterminal *ActMsg* representing an actor receiving a specific kind of message. This grammar captures parallelism of both actors and PTGs, as well as dependencies like actors creating other actors, or invoking PTGs due to a task action (i.e. sending messages).

2) GENERATING AN INSTANCE OF THE GENERIC CFG

Let us now describe the step-by-step procedure to create an instance of this generic CFG. The input to this procedure is the *abstract syntax tree* (AST) of the corresponding system and the PTGs. We start by navigating the ASTs of the system initialization procedure and the actor code. The steps are the following:

1) First, we get a list of actors created during system initialization and create the nonterminals A_i ($i \geq 1$) for the created actors. If there are multiple actors of the same kind, we create two nonterminals A_i and A_{i+1}: A_i is the unique actor and A_{i+1} represents the remaining actors. Two production rules, $ActInit \rightarrow par(A_1, \ldots, A_n)$ and $S \rightarrow ActInit$, are created, where n is the number of representative actors. Suppose *Actors* is the set of all actors contained in the right side of the *ActInit* rule. Next, we perform steps 2-6 for each actor in *Actors*.
2) For each actor A_i in *Actors* that is not yet processed, we navigate the AST of the corresponding actor code, statically determine the different types of messages being handled by an instance of this actor, and mark A_i as processed.

3) We create a unique nonterminal AM_j ($j \geq 1$) for each type of message being handled by A_i and create the production rule $A_i \rightarrow AM_j$. If no such message is found, we create the production rule $A_i \rightarrow \epsilon$.
4) We navigate the AST of the message handler for each type of actor message AM_j created in the previous step. We create the production rule $AM_j \rightarrow A$ if the message handler creates a new actor, where, the unique nonterminal A represents the created actor. A is created afresh if *Actors* does not contain an actor representing either the type of A, or multiple actor instances of type A. If A is created anew, then it is included into the *Actors* set. However, if the message handler invokes a PTG, then we produce $AM_j \rightarrow TG$, where TG represents the PTG being invoked.
5) For each created PTG TG in the previous step, we visit the nodes of the PTG and obtain the set Tsk of tasks invoked from the graph. We create the rules $Tasks \rightarrow T$ for each $T \in Tsk$, and the rule $TG \rightarrow g \;\; Tasks$ where g is the fresh atom representing the PTG TG.
6) We visit the AST of each task T invoked in the previous step and identify the message send instructions (if any) in T. We create the rule $T \xrightarrow{t} AM$ for each such message send instruction, where t represents the unique terminal symbol for task T, and AM represents a specific type of actor receiving an specific kind of message created in step 3. However, we create the rule $T \rightarrow \epsilon$ if no such message send instruction exists in the AST of T.

Example 3: Fig. 9(b) shows an example of the CFG generated from code containing two different actors A_1 and A_2. Actor A_1 may receive messages of type AM_{11} and AM_{12}. Similarly, AM_{21} is the type of the message that actor A_2 may receive. Note that A_1 and A_2 are the abstract representation of the created actors overapproximating their behavior. A_1 (and A_2) invokes PTGs of type TG_1 and TG_2 (TG_3). PTG TG_3 invokes a task sending a message of type AM_{12} to A_1. The set of terminal symbols in this CFG is $\{g_1, g_2, g_3, t\}$.

3) ANALYSIS OF THE GENERATED STRINGS

Once the grammar is obtained, the next step is to build valid sentences stating which PTGs to be invoked in parallel. There are efficient techniques to infer all sentences of a given length from a CFG (see e.g., [19], [20]). Note that the language of the generated CFG may be infinite, but it is sufficient to infer finite sentences capturing all MHP behaviors. This is because the MHP computation problem infers all pairs of program instructions that possibly execute in parallel, and there is only a finite number of program instructions.

We generate all valid strings of the form $par(T_1, \ldots, T_k)$, where T_i is a tree whose nodes are PTG atoms. Edges have the form $\xrightarrow{t}$ and t indicates a task atom. Let us assume the production $S \rightarrow par(A_1, \ldots, A_k)$, and that the code contains $n > 0$ different types of PTGs. Next, we shall apply a string enumeration technique to generate all valid strings from each nonterminal A_i containing at most $2n$ PTG atoms such that there does not exist more than two atoms representing the same kind of PTG. We record the generation of the PTG atoms during the string enumeration procedure in order to keep track of the maximal number of generated atoms of the same kind. Thus, for instance, if two strings $g_1 \xrightarrow{t_2} g_2$ and $g_1 \xrightarrow{t_3} g_3$ are generated, we get a tree with g_1 as the root node, and the two branches $g_1 \xrightarrow{t_2} g_2$ and $g_1 \xrightarrow{t_3} g_3$. For any generated string $g_1 \xrightarrow{t_2} g_2$ (for example), we obtain the contextual information $C(g_2) = \langle t_2, g_1 \rangle$.

Many techniques are available to enumerate strings of a fixed length from a given CFG. For the completeness of this paper, we present an iterative algorithm to generate fixed length strings according to [19], which is based on the standard *inductive concatenation scheme*. Suppose $\mathcal{P}$ is the set of productions, $\mathcal{N}$ is the finite set of nonterminal symbols, and Σ is the finite set of terminal symbols of the given grammar. We generate the grammar according to Fig. 9(a) and convert it to *Chomsky Normal Form* (CNF) [21], that does not have any ϵ-productions, and all productions are in the following form:

$$\begin{aligned} \textit{Nonterminal production}: &\quad A \rightarrow BC, \text{ with } B, C \in \mathcal{N} \\ \textit{Terminal production}: &\quad A \rightarrow a, \text{ with } a \in \Sigma \end{aligned}$$

Suppose $\mathcal{P}$ contains the productions in CNF form. Let A^i denote the set of all strings generated by the nonterminal A in iterations 0 to i. For each nonterminal $A \in \mathcal{N}$, we compute the following set of strings iteratively:

$$\begin{aligned} A^0 &= \{a \in \Sigma | A \rightarrow a \in \mathcal{P}\} \\ A^{i+1} &= A^0 \cup \{u.v \mid \exists B, C \in \mathcal{N} : A \rightarrow BC \in \mathcal{P} \wedge u \in \\ &\qquad B^i \wedge v \in C^i\} \end{aligned}$$

Thus, for the production $S \rightarrow par(A_1, \ldots, A_k) \in \mathcal{P}$, we generate all strings of length at most $2n$ for each nonterminal A_i by using the above iterative method. More details can be found in [22].

Example 4: Let us consider the CFG in Fig. 9(b). The CNF of this grammar without ϵ-production and including the first production is as follows:

$$\begin{aligned} S &\rightarrow par(A_1, A_2) \\ A_1 &\rightarrow g_1 \mid g_2 \\ A_2 &\rightarrow GA_3 \;\; T_3 \\ GA_2 &\rightarrow g_2 \\ GA_3 &\rightarrow g_3 \\ T_3 &\rightarrow TA \;\; GA_2 \\ TA &\rightarrow \xrightarrow{t} \end{aligned}$$

We obtain the results presented in Table 1 by applying the string enumeration technique explained above.

TABLE 1. String enumeration steps on the CNF grammar.

Nonterminal N	N^0	N^1
A_1	$\{g_1, g_2\}$	$\{g_1, g_2\}$
GA_2	$\{g_2\}$	$\{g_2\}$
GA_3	$\{g_3\}$	$\{g_3\}$
TA	$\{\xrightarrow{t}\}$	$\{\xrightarrow{t}\}$
T_3	$\emptyset$	$\{\xrightarrow{t} g_2\}$
A_2	$\emptyset$	$\{g_3 \xrightarrow{t} g_2\}$

Thus, the enumeration algorithm generates the following valid sentences:

$$par(g_1, g_3 \xrightarrow{t} g_2), \quad par(g_2, g_3 \xrightarrow{t} g_2)$$

Thereby, we have two scenarios. In the first one g_1 and g_3 may run in parallel, and at some point g_3 invokes a task that introduces g_2 to run in parallel with both g_1 and g_3. The second scenario is similar, except for the fact that g_1 is replaced by g_2.

Worst-Case Asymptotic Complexity: The steps in generating a CFG from an actor-based system can be optimized so that the ASTs and the PTGs are not visited more than a fixed number of times. Thus, the complexity of generating a CFG is linear with respect to the size of the system. Enumerating finite-length strings from the CFG by the presented iterative algorithm takes lower degree polynomial time with respect to the size of the generated string [22].

E. INSTRUCTION-LEVEL PARALLELISM

Our analyses are able to infer all task-level parallelism invoked from the PTGs. However, it may be required to infer instruction-level parallelism of the parallel programming code used by the actor systems. We infer instruction-level parallelism as follows. Suppose $\mathbb{M}$ contains the global task-level MHP facts. Given any task t, assume that $ins(t)$ denotes the set of all instructions belonging to the task t. The following equation infers instruction-level MHP facts from the results in $\mathbb{M}$:

$$i\mathbb{M} = \{(p, q) : p \in ins(t), q \in ins(t'), (t, t') \in \mathbb{M}\}$$

VI. IMPLEMENTATION AND EXPERIMENTAL SETUP

We targeted telecom systems in which the behaviour of actors is implemented with pthreads, PTGs are represented by XML-based graphs, and tasks are written in DSP C. The systems include components such as uplink, downlink, and scheduling. Each of the components has its own actor model, a different set of PTGs and tasks. For the implementation we used SWI-Prolog[7] to implement Algorithms 1-4. We chose SWI-Prolog mainly to use (i) the logical inference engine provided by the language, and (ii) the XML library since PTGs are described in an XML-based format. After parsing the XML files, we store the PTGs as Prolog predicates and perform the data-flow and MHP analysis on those predicates. For each *single-* or *multi-task* node, the XML file specifies the name of the initial function and the location of the code to invoke that task. The relevant code of each task is distributed over multiple files. The application is written in DSP C.

We developed a tool called *Racer* to detect data races in the Clang/LLVM framework.[8] Tasks send and receive messages by invoking special functions. We generate the AST of the code in Clang and traverse it to detect functions sending and receiving messages. The AST traversal algorithm is implemented in the *Racer* tool by using the Clang AST matcher library.

Even though the source code of each task is distributed over many files, we are only aware of the location of the initial function from which the task starts execution. The XML file also provides the name of the directories containing C files; while some of C files belong to a given task, some others do not. The message transfer events of each given task may be located in a function, deep into the call hierarchy from the initial function. So, we generate the call graph of each task and perform a reachability check of functions sending and receiving messages from the initial function. The reachability check confirms the owner of the message transfer events, i.e., the specific task. The DSP C code includes function pointers, thus we implemented Steensgaard's pointer analysis [23] to resolve function calls through pointers that are used during the call graph construction. We used this sound (but less precise) analysis since it scales to millions of lines of code. Our MHP analysis tool also builds the compilation database of all C files for all tasks. This compilation database is used by the *Racer* tool to parse the C files and generate the ASTs.

All experiments have been performed on a Linux x86-64 machine with a 2.30 GHz processor. The benchmark application code was obtained from an industrial repository in the telecom domain.[9] Time was measured by the "time" predicate of SWI-Prolog, which gives time in seconds as a floating point number with 3 digits after the decimal point precision.

VII. EXPERIMENTAL RESULTS

Table 2 provides the experimental results performed on our benchmark application. The columns *#Node* and *#Tasks* specify the size of PTGs and the actual number of tasks, each related to DSP C code. The *#Pairs(MHP)* column indicates the number of pairs of tasks that may run in parallel, which is obtained from Eq. (8) after the analysis of Algorithm 1. The column *Time (Algorithm 1)* specifies the execution time required by the data-flow analysis, while *Time (total)* specifies the total running time including parsing, generating PTGs in prolog format, and generating compile commands for Clang (among other implementation specific jobs). The *#Events* column gives the total number of messages received by the PTGs. The *MLOC* column gives the number of millions

[7] http://www.swi-prolog.org/

[8] https://clang.llvm.org/

[9] Note that for confidentiality reasons, we cannot disclose details of the code used for the experiments.

TABLE 2. Example benchmarks and results of MHP analysis (via Algorithm 1).

PTGs	#Node	#Tasks	#Pairs (MHP)	Time(s) (Algorithm 1)	Time(s) (Total)	#Events (Msg)	MLOC	Time(s) (C Code)	Time(ms) (CG)
TBAdminProactor (G_1)	27	5	10	0.023	2.527	0	0	0	0
hdbaseTestReactor (G_2)	15	5	5	0.004	2.835	0	0	0	0
ProCellRandActivator (G_3)	17	5	6	0.007	3.213	0	0	0	0
ProCellSetupActor (G_4)	26	8	13	0.021	3.221	0	0	0	0
ProCellActivator (G_5)	36	9	20	0.046	2.550	4	1.1	33.514	22
DlCellReactor (G_6)	52	13	37	0.106	3.180	8	5	91.123	107
shedCheckActor (G_7)	90	18	137	0.360	3.199	4	1.41	29.911	36
DldBmProActor (G_8)	157	50	395	11.295	15.289	6	16.31	293.64	389
DbmProActor (G_9)	178	44	494	4.043	6.530	14	5.10	106.821	131
CommActivator (G_{10})	335	75	2307	26.931	37.670	18	7.65	155.452	200
Total	933	232	3424	42.836	80.214	54	36.57	446.185	885

of lines of DSP C code searched for the message transfer instructions. The *Time (C Code)* column shows the execution time of the *Racer* tool required to find the tasks sending messages, and the *Time (CG)* column shows the time required to build and analyse the call graph of the relevant code. All times are measured in seconds except *Time (CG)*, which is measured in milliseconds.

The results in Table 2 come from the analysis of PTGs individually. As the final row indicates, we required only 80 seconds for the total processing time generating 3424 pairs of tasks that may run in parallel. The data-flow analysis performed by Algorithm 1 requires only ∼43 seconds. PTG G_{10} contributes with most MHP-pairs and requires the longest analysis time (27 seconds by the implementation of Algorithm 1). Comparing the results of PTGs G_8 and G_9, G_8 has more task nodes and requires longer analysis time, but less total number of nodes and produces less MHP pairs. This reveals that it is the number of tasks that most heavily influences the execution time of data-flow analysis. This is theoretically correct since a bigger number of task nodes (along with barrier nodes) generates more facts and requires more time to reach a fixpoint. However, the graph structure should also influence the execution time of the data-flow analysis, as illustrated later in this section.

The last 6 PTGs (i.e. G_5–G_{10}) are involved in message transfer events for which we analysed 36.57 MLOC (DSP C) in approximately 7 minutes in order to find tasks involved in message transfer events. The reason for such a short analysis time, considering the huge number of code lines, is that (i) Steensgaard's pointer analysis requires almost linear time, (ii) the complexity of our AST traversal algorithm is linear, and (iii) building call graphs and reachability check in any call graph require at most linear time with respect to the size of the code. We performed the reachability check on the call graph of a given task in a bottom-up manner in order to confirm whether the method sending messages is reachable from the tasks' initial method (i.e. the task is sending a specific message). The *Time (CG)* column indicates that building and analysing call graphs requires very negligible time compared to the total time required to identify message transfer events. Moreover, the time to identify the message transfer events is 5.6 times bigger than the total processing

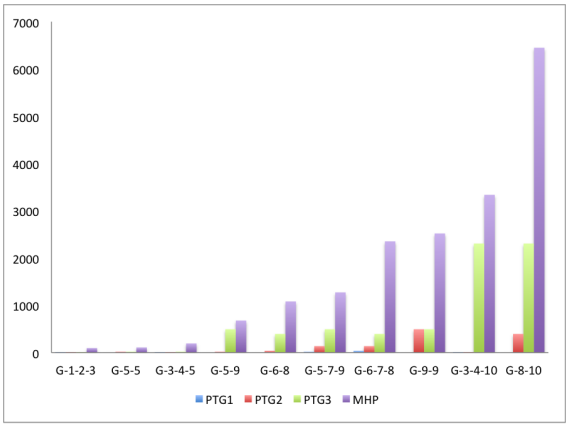


FIGURE 10. Comparing the number of MHP pairs (Y-axis) of different PTGs (X-axis). The X-axis represents 10 different test cases each containing 3 PTGs. The third PTG is the combination of the first and the second (or a copy of the first) PTG.

time of the graphs (including data-flow analysis of individual PTGs). However, once the analysis of the C code for finding message transfer events is completed, no further code analysis is required and detecting parallelism among multiple PTGs can benefit from it.

Table 3 provides the MHP analysis results obtained from Algorithm 1 after combining relevant PTGs according to Section V-B. As seen from the table, the number of task nodes in PTGs usually dominate the analysis time and the number of MHP pairs. One exception is the case in which we analysed the combination of PTGs G_5, G_7, G_9. All these

TABLE 3. Results of MHP analysis (via Algorithm 1) after combining PTGs.

PTGs	#Nodes	#Tasks	#MHP pairs	Time (s)
G_1, G_2, G_3	63	15	96	0.281
G_5, G_5	76	18	109	0.698
G_3, G_4, G_5	84	22	194	1.121
G_5, G_9	229	53	678	16.889
G_6, G_8	211	63	1082	19.432
G_5, G_7, G_9	318	71	1274	218.812
G_6, G_7, G_8	303	81	2353	35.569
G_9, G_9	370	88	2520	87.063
G_3, G_4, G_{10}	384	88	3337	79.710
G_8, G_{10}	496	125	6448	125.240

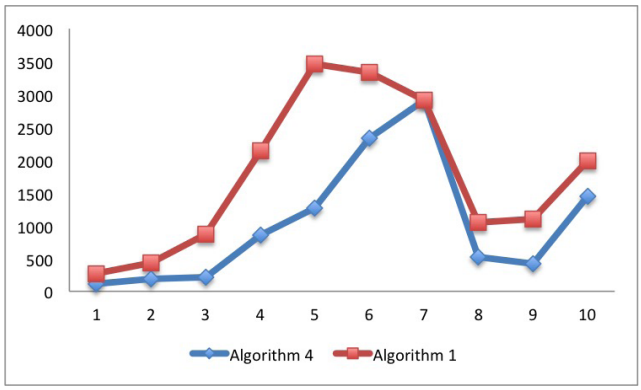


FIGURE 11. Comparing the number of MHP pairs (Y-axis) obtained from Algorithm 1 and Algorithm 4 of different PTGs (X-axis).

PTGs are involved in heavy communication and therefore data-flow facts generated in one PTG are copied to distant parallel branches of other PTGs. This increases the flow of facts in all segments of the graph, and thus, reaching a fixpoint requires longer time. The combination of PTGs (G_5, G_5) (or (G_9, G_9)), as indicated in the table, represents multiple instances of the same PTG are running. We obtain these combinations by first creating a copy of G_5 (or G_9) and then combining it with G_5 (or G_9). Fig. 10 shows a comparison of the number of MHP pairs obtained by analysing the combined PTGs with respect to the number of MHP pairs obtained by analysing the constituent PTGs. The X- and Y-axis represent the test cases and the number of MHP pairs. The third bar in each group of PTGs represents the number of MHP pairs obtained by analysing the combined PTGs. Not surprisingly, we obtained an exponential increment of the number of MHP pairs, in comparison with the results of its constituent PTGs, when the PTGs are combined and they are involved in little or no communication. However, the results of the PTG groups (G_5, G_9), (G_5, G_7, G_9) and (G_3, G_4, G_{10}) do not provide an exponential increase of the number of MHP pairs when compared with the number of MHP pairs of individual PTGs. After examining the graphical representation of the connected PTGs, it reveals that each group of PTGs is involved in inter-graph communication that restricts the parallelism of the combined PTGs.

We have tested the implementation of Algorithm 4 on a number of industrial use-cases, and compared the obtained results (i.e. the number of MHP pairs) with the results of Algorithm 1 combining PTGs according to Section V-B. Note that Algorithm 4 uses Algorithm 1 to compute MHP pairs of individual PTGs. Moreover, it takes into account the task-actor-PTG communication. Algorithm 1 is unaware of the task-actor communication and it assumes that all PTGs are running in parallel if there is no communication among them. The comparison is presented in Fig. 11, which shows that Algorithm 4 is more precise than Algorithm 1 in 9 out of 10 cases. In case 7, there is no task-actor communication involved, thus the two algorithms provide the same results.

In order to manually validate the correctness of our results, we generated a graphical representation of PTGs in which all tasks running in parallel with a given task were marked. We inspected the marked graphs to check the correctness. We inspected all results of smaller PTGs and took random samples from larger PTGs.

The major application of our MHP analyses is for detection of data races. A data race may appear if there are two tasks t, t' that might run in parallel and where there is an instruction in t, and one in t' such that both may access the same memory address, and at least one instruction is a write. On each pair of tasks obtained from our MHP analyses we ran our *Racer* tool on the code for the tasks, checking for memory conflicts. *Racer* uses Steensgaard's pointer analysis to detect possible aliasing. During the experiment, our analyses reported 12052 MHP task-pairs (after duplicates removal). Thus, the *Racer* tool had 12052 test-cases for race detection and it reported 39 potential data races after nearly 9.5 hours of computation to analyse 36.57 MLOC code. Note that *Racer* is not optimized, in the sense that a task appearing in multiple MHP pairs is analysed multiple times with respect to memory accesses; we expect *Racer* to be much faster after removing this redundant analysis.

VIII. APPLICABILITY TO OTHER COMPUTATIONAL MODELS

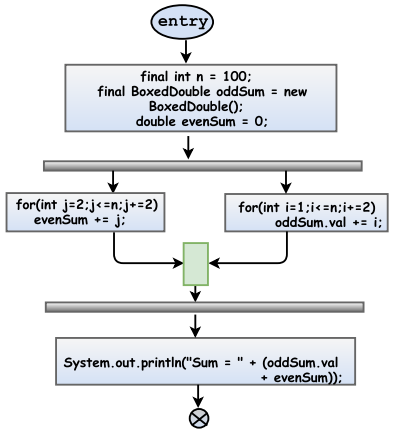
Our theoretical development is not limited to the computational model that we describe in Section II. We can in fact analyse subclasses of other programming models from which it is possible to extract PTGs, such as Cilk [24] or X10 [25]. In the following, we provide a brief introduction to X10 followed by an illustration on how to generate PTGs from X10 code samples; our analyses can be applied to the generated PTGs.

X10 is a class-based object-oriented programming language designed for high-performance, high-productivity computing [25]. X10 targets program execution on a wide range of computers such as multi-core processors, large clusters of parallel processors, hardware accelerators, embedded processors and supercomputers. The concept of *place* is central to the language. A *place* is conceptually a computational unit with a finite number of threads called *activities* having an uniform access to a finite amount of shared memory. The global address space is divided into multiple places. X10 provides various concurrency constructs such as creating and synchronising parallel activities, enforcing mutual exclusion, and synchronising parallel activities through barriers. In the following, we briefly illustrate some of the concurrency constructs of X10:

- *async S* – creates a new child activity to execute S asynchronously at the *place* of the parent activity. Execution returns immediately to the parent activity meaning that the remaining instructions of the parent activity and the newly created activity may run in parallel.
- *finish S* – it guarantees that the parent activity must wait after the *finish* for all the child activities created within S to terminate.

```
public static void main(String[] args) {
  final int n = 100;
  final BoxedDouble oddSum = new BoxedDouble();
  double evenSum = 0;
  finish {
    async { // Compute oddSum in child activity
      for(int i=1;i<=n;i+=2)
      oddSum.val += i;
    }
   // Compute evenSum in parent activity
   for(int j=2;j<=n;j+=2)
   evenSum += j;
  } // finish
 System.out.println("Sum = " + (oddSum.val + evenSum));
} // main()
```

(a)



(b)

FIGURE 12. (a) X10 code containing async-finish parallel constructs, (b) PTG representation of the X10 code in Fig. (a).

- *atomic S* – the *atomic* construct executes *S* atomically pretending a single-step execution. It offers the mutual exclusive operation by multiple parallel activities to shared data located at the same *place*. However, *S* may not spawn another activity of use blocking statement such as *finish*.
- *async clocked(cl) S* – the language supports the synchronisation of dynamically varying collection of activities through the *clocks* that act as barriers. This statement creates an asynchronous activity to execute *S*; its execution should be synchronised with other activities that are registered with the clock *cl*. A single clock may have several phases of barriers and the instruction *Clock.advanceAll()* executed by an activity causes it to synchronise at this phase and the remaining instructions of the activity should synchronise to the next phase.

Let us now see how we can generate PTGs from X10 code samples. The code in Fig. 12(a) illustrates the *async-finish* parallelism in X10. The parent activity is creating an asynchronous child activity preceded by a *finish* command. So, the child activity and the remaining instruction of the parent activity inside the *finish* block should synchronise their execution. We create the corresponding PTG in Fig. 12(b) such that the instructions of the parent activity are divided into three tasks: (i) *task-1* contains all the instructions before the *finish* block, (ii) *task-2* contains the instructions inside the *finish* block but excluding the *async* instruction, and (iii) *task-3* includes the remaining instructions of the parent activity. After *task-1*, we create a barrier node to fork an asynchronous activity due to the *async* command, and *task-2*. *Task-2* and the asynchronous task should synchronise their execution due to the *finish* command, and *task-3* should be forked from a barrier node, after which it should run to completion.

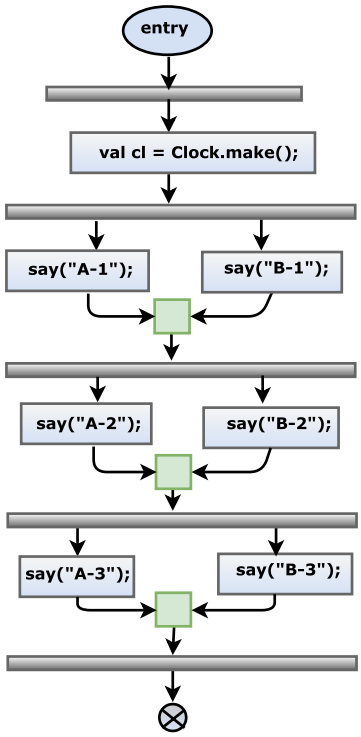
Fig. 13(a) depicts an example of clock-based synchronisation in which the parent activity invokes a child activity to be synchronised with the parent due to the *finish* command. The child activity invokes two asynchronous activities *A* and *B*, which are registered to the clock *cl*. Both activities should advance their execution in three different phases. The corresponding PTG in Fig. 13(b) depicts this scenario in which both activities *A* and *B* are divided into three subtasks A_i and B_i for $i \in [1..3]$. The sequence of executions is the following: both A_1 and B_1 are invoked from the same barrier node and joined at the same *join* node; the *join* node is connected with a barrier node forking A_2 and B_2, and the process is repeated for A_3, and B_3.

We now consider a more complex code in which *async* and *atomic* instructions are inside the loops, as in Fig. 14(a). We need to extend the node types used in Def. 1 in order to deal with this problem. The *join* nodes in Def. 1 merge parallel control flow. However, the code in Fig. 14(a) requires merging sequential control flow as well. We thus introduce an additional join node type *sjoin* such that a *sjoin* node merges multiple sequential control flow edges with the semantics that control transfers to its sucessor node immediately if the *sjoin* node is reached via one of its predecessor nodes. In Fig. 14(b), nodes J_1, J_2, J_4 are the *sjoin* nodes. In previous examples, we divided the instructions of an activity into one or more subtasks; this can not be done here. Either the loop instruction should belong to a single subtask or it should be broken apart. Due to the existence of the *async* instruction in the loop body, the loop instruction cannot belong to a single subtask. Here, we model the two loop instructions by the conditional nodes C_1, C_2 and C_3 of the PTG shown in Fig. 14(b).

The *async* instruction is creating a child activity and the remaining instruction of the parent activity (in the inner **for** loop) is atomic. Thus, we create two subtasks T_1 and T_2 corresponding to these instructions, which are forked from a single barrier. Subtask T_2 is connected to the conditional node C_3 via a *sjoin* node J_2 through a feedback loop. Node C_3 represents the conditional part of the inner *for* loop. Thus, as long as $j \leq n$ is evaluated *true*, subtasks T_1 and T_2 are spawned, and continues to the next iteration when T_2 finishes.

```
public static void main(String[] args) {
  finish async{
    val cl = Clock.make();
    async clocked(cl) {// Activity A
      say("A-1");
      Clock.advanceAll();
      say("A-2");
      Clock.advanceAll();
      say("A-3");
    }// Activity A
    async clocked(cl) {// Activity B
      say("B-1");
      Clock.advanceAll();
      say("B-2");
      Clock.advanceAll();
      say("B-3");
    }// Activity B
  }
}
```

(a)



(b)

FIGURE 13. (a) X10 code containing clock-based synchronisation primitives, (b) PTG representation of the X10 code in Fig. (a).

The *false* edge of C_3 and the barrier node B are connected with the *join* node J_3. The node B ensures that all subtasks T_1 terminates when $j \leq n$ is evaluated *false*. This is due to the *finish* construct, as the parent activity should wait for any child activity to terminate when executed inside a *finish* block.

The condition of the inner for loop is modeled by two condition nodes C_2 and C_3. The *false* edge of C_2 ensures that no subtasks T_1 and T_2 are ever created. The *true* edge of C_2 is connected with the subgraph modeling the body of the inner for loop. The *sjoin* node J_4 merges both branches of C_2 and followed by the task node incrementing the outer loop counter. The subtask incrementing the outer loop counter is connected through a feedback loop to the conditional node C_1 through the *sjoin* node J_1 representing the conditional part of the outer *for* loop. Now, in order to perform our analyses, we simply consider that (i) the *kill* and *gen* sets of any *sjoin* node are empty, and (ii) the subtasks T_k are *multi-task* nodes creating multiple instances of themselves for $k = 1, 2$. Then, we can use Algorithm 1 to infer all task-level parallelism.

The resulting PTG in Fig. 14(b) contains cycles. This poses some challenges for the expected properties of PTGs as well as the MHP analysis. The main cause is that in a PTG with cycles, the same task node may be executed several times. This can create new opportunities for concurrent executions of tasks, which may violate the expected properties of PTGs and thus may render the current MHP analysis unsound. For the example in Fig. 14 the analysis will yield a sound result, given the modifications in the analysis indicated above. However it is not hard to come up with examples of cyclic PTGs where, for instance, different executions of the same task will trigger the concurrent execution of tasks in both branches from a conditional node thus violating the mutual exclusivity of these branches. In such situations the current MHP analysis will be unsound. There are two ways to deal with this situation:

- modify the MHP analysis in order to be sound also in the presence of several executions of the same task, or
- define rules for ''well-formedness'' of cyclic PTGs such that the important properties of acyclic PTGs, like precedence and mutual exclusivity of task executions, hold also for the cyclic PTGs under consideration. It is of particular interest if translations into PTGs from other parallel models can be shown to always yield well-formed PTGs.

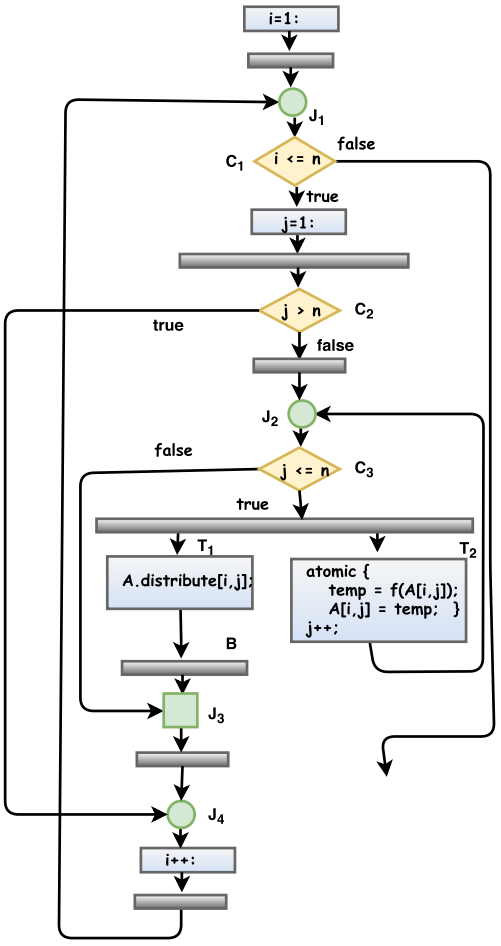
We leave the investigations of these issues as future work.

IX. RELATED WORK

Execution order of statements in parallel or concurrent programs is inherently related to inferring MHP information. Some early works on inferring execution order of statements from concurrent programs are due to Bristow *et al.* [26], and Callahan and Subhlok [27]. They considered a computational model in which there are specific synchronisation points in the program. Duesterwald and Soffa [28] extended the work of Callahan and Subhlok to infer non-concurrency analysis in the rendezvous model that addressed the interprocedural

```
for(i = 1;i <= n;i++)
  finish {
  for(j=1;j<=n;j++) {
    async (A.distribute[i,j])
    atomic {
      temp = f(A[i,j]);
      A[i,j] = temp;
    }
  }
}
```

(a)



(b)

FIGURE 14. (a) X10 code segment containing a *async* statement inside loops, (b) PTG representation of the X10 code in Fig. (a).

effects on the concurrency analysis. This was later extended by Masticola and Ryder [18] to handle binary semaphores, and applied iterative refinement techniques to obtain more precise results. Naumovich and Avrunin [17] presented a conservative approach of computing the MHP facts for programs consisting of rendezvous-like synchronisations. It works on a *trace flow graph*, which is a forest of control-flow graphs adding special communication nodes and edges for synchronous communication. Later, they extended their work together with Clarke [8] to handle the concurrency model of java. All these works apply data-flow analysis techniques to infer the MHP information once the parallel control-flow is generated. The complexity of these approaches are mostly due to different kinds of synchronisation primitives, which either create too many synchronisation points or complicate the generation parallel control-flow. In our case, we take advantage of the high-level software design to generate parallel control-flow of task-graphs; however, the complexity of our approach comes from the parallel execution of multiple task-graphs, the parallel execution of actors, and the communication between actors and task-graphs.

Albert *et al.* [29] proposed MHP analysis for concurrent objects. They used abstract interpretation [30] based techniques to infer local (method-level) MHP information followed by constructing MHP graph and solving the reachability problem to infer global MHP information. Their method is sound, and the worst-case computational complexity is cubic with respect to the number of nodes in the MHP graph. This work was further extended (i) in [31] that takes into account the interprocedural synchronisation that prevails in most concurrent languages, and (ii) in [7] that includes a proof of soundness, and an improvement of accuracy for conditional statements. With respect to our solution, our computational model is different from theirs and their solution is not applicable to it. We applied a data-flow analysis technique which has lower order polynomial complexity to infer MHP facts from PTGs. Moreover, analysing multiple PTGs executing in parallel is done by inferring non-concurrency analysis, and dependencies among task-graphs and actors are inferred by generating specific context-free grammars from the system code followed by enumerating strings from the generated grammar.

Agarwal *et al.* [4] proposed static MHP analysis of X10 programs. These programs may contain concurrency primitives such as *async*, *finish*, and *atomic* that allows them to devise more precise and efficient algorithms based on traversing the Program Structure Tree. In doing so, they first perform a Never-Execute-In-Parallel analysis followed by a Place-Equivalence analysis that decides whether all instances of two given operations execute at the same place. Lee *et al.* [5] proposed algorithms for static MHP analysis of X10 like languages consisting of async-finish parallelism and procedures. They solve the MHP decision problem in linear time by reducing the program to *constrained dynamic pushdown networks*. The MHP computation problem is solved by running a type-based analysis followed by utilising the MHP decision algorithm to refine the results. Even though *async-finish* parallel constructs have some similarity with barrier-based synchronisation, nevertheless, our computational model is not comparable with the parallel computational model of X10 and their type-based analysis technique is different from ours, which uses data-flow analysis inferring partial order relations and deriving context-free grammars.

To summarize, existing MHP analyses differ either by the computational model of the targeted language, or by the kinds of synchronisation constructs used by the language.

X. CONCLUSION AND FUTURE WORK

Parallel programming is idiosyncratically more complex than sequential programming. Methodological aids are essential for developers to ensure the safety and liveness properties of parallel applications such as data race detection, termination analysis, deadlock detection, resource analysis, and determinism checking. A crucial preparation activity for these analyses is the MHP analysis, which identifies pairs of tasks runnable in parallel. In this paper, we introduce a novel MHP analysis tailored to a computational model based on task-graphs and actors. Moreover, we provide a UML profile specific for this computational model, both for modelling graphs and for back-propagating analysis results to the modelled graphs for further investigation.

We developed a data-flow analysis technique applicable to parallel control-flow structures that can infer all MHP facts from individual task graphs. Our technique can be applied to parallel graphs obtained from other programming languages for inferring properties not only limited to MHP facts. We have shown by examples how to transform X10 code into parallel task-graphs, on which our analysis is directly applicable. In our computational model, actors and task-graphs may involve in direct or indirect communication either by transmitting asynchronous messages or acting on the received messages. These message transfer events restrict task-level parallelism due to dependencies between tasks but increase the flow of information among task-graphs. We generate a context-free grammar (CFG) of the system and enumerate all valid strings from the generated CFG that represent all communication information among parallel task-graphs. Our data-flow analysis technique is extended by utilising the obtained communication information and via inferring non-concurrency among tasks to generate all MHP facts from the underlying system. We have described the theoretical worst-case asymptotic complexity of the algorithms conceived by our solution. We obtained a polynomial time complexity for analysing a single PTG, and exponential time complexity for analysing n PTGs. However, by statically determining the maximum number of PTGs that can run in parallel, exponential time complexity can be reduced to higher order polynomial time complexity.

We implemented the algorithms, and evaluated them on a set of real-world signal processing applications from the telecom domain consisting of 36.57 MLOC, representing 232 different tasks. The analysis required approximately 7 minutes to identify all communication information, and 10.5 minutes to generate 12052 pairs of tasks running in parallel. Our *Racer* tool took around 9.5 hours to identify 39 potential data races from 36.57 MLOC, which shows that the analysis can scale to realistic applications.

As a future work, our plan is to transform other programming models into task models and analyse them through our methods, as well as to develop other analyses, based on our data-flow analysis technique, to infer other kinds of program properties from parallel applications.

REFERENCES

[1] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Inf.*, vol. 19, no. 1, pp. 57–84, 1983.
[2] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 416–430, Mar. 2000.
[3] H. Seidl and B. Steffen, "Constraint-based inter-procedural analysis of parallel programs," *Nordic J. Comput.*, vol. 7, no. 4, pp. 375–400, Dec. 2000.
[4] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel analysis of x10 programs," in *Proc. 12th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2007, pp. 183–193.
[5] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong, "Efficient may happen in parallel analysis for async-finish parallelism," in *Proc. 19th Int. Static Anal. Symp.*, in Lecture Notes in Computer Science, vol. 7460. Berlin, Germany: Springer, 2012, pp. 5–23.
[6] J. K. Lee and J. Palsberg, "Featherweight x10: A core calculus for async-finish parallelism," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2010, pp. 25–36.
[7] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin, "May-happen-in-parallel analysis for actor-based concurrency," *ACM Trans. Comput. Logic*, vol. 17, no. 2, pp. 11:1–11:39, Dec. 2015.
[8] G. Naumovich, G. S. Avrunin, and L. A. Clarke, "An efficient algorithm for computing MHP information for concurrent Java programs," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 338–354, Oct. 1999.
[9] P. Lammich and M. Müller-Olm, "Precise fixpoint-based analysis of programs with thread-creation and procedures," in *Proc. 18th Int. Conf., Concurrency Theory (CONCUR)*, Lisbon, Portugal, L. Caires and V. T. Vasconcelos, Eds. Berlin, Germany: Springer, 2007, pp. 287–302.
[10] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, Jan. 1997.
[11] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
[12] M. Lundqvist, A. Mallo, P. Brauer, and D. Engdal, "BOS: A task-graph processing software framework for the epiphany many-core architecture," in *Proc. 6th Swedish Workshop Multicore Comput. (MCC)*, Nov. 2013, pp. 1–4.
[13] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1085–1100, 2013.
[14] M. Själander, S. A. McKee, P. Brauer, D. Engdal, and A. Vajda, "An LTE uplink receiver PHY benchmark and subframe-based power management," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2012, pp. 25–34.
[15] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869–891, Jun. 2013.
[16] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. New York, NY, USA: Springer, 2010.
[17] G. Naumovich and S. G. Avrunin, "A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 6, pp. 24–34, Nov. 1998.
[18] S. P. Masticola and B. G. Ryder, "Non-concurrency analysis," *SIGPLAN Notices*, vol. 28, no. 7, pp. 129–138, Jul. 1993.
[19] C. C. Florêncio, J. Daenen, J. Ramon, J. van den Bussche, and D. V. Dyck, "Naive infinite enumeration of context-free languages in incremental polynomial time," *J. Universal Comput. Sci.*, vol. 21, no. 7, pp. 891–911, Jul. 2015.
[20] Y. Dong, "Linear algorithm for lexicographic enumeration of CFG parse trees," *Sci. China Ser. F, Inf. Sci.*, vol. 52, no. 7, pp. 1177–1202, 2009.
[21] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2006.
[22] P. Dömösi, "Unusual algorithms for lexicographical enumeration," *Acta Cybernetica*, vol. 14, no. 3, pp. 461–468, 2000.
[23] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Princ. Programming Lang. (POPL)*, New York, NY, USA, 1996, pp. 32–41.
[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Notes*, vol. 30, no. 8, pp. 207–216, Aug. 1995.
[25] V. Saraswat *et al.*, "X10 language specification-version 2.3," IBM, New York, NY, USA, Tech. Rep., 2012.

[26] G. Bristow, C. Drey, B. Edwards, and W. Riddle, "Anomaly detection in concurrent programs," in *Proc. 4th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA, 1979, pp. 265–273.
[27] D. Callahan and J. Sublok, "Static analysis of low-level synchronisation," *SIGPLAN Notes*, vol. 24, no. 1, pp. 100–111, Nov. 1988.
[28] E. Duesterwald and M. L. Soffa, "Concurrency analysis in the presence of procedures using a data-flow framework," in *Proc. Symp. Test., Anal., Verification (TAV4)*, New York, NY, USA, 1991, pp. 36–48.
[29] E. Albert, A. E. Flores-Montoya, and S. Genaim, "Analysis of may-happen-in-parallel in concurrent objects," in *Proc. Int. Conf. Formal Techn. Distrib. Syst.*, vol. 7273, H. Giese and G. Rosu, Eds. Berlin, Germany: Springer, Jun. 2012, pp. 35–51.
[30] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 1977, pp. 238–252.
[31] E. Albert, S. Genaim, and P. Gordillo, "May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronisation," in *Proc. 22nd Int. Static Anal. Symp. (SAS)*, S. Blazy and T. Jensen, Eds. Berlin, Germany: Springer, Sep. 2015, pp. 72–89.

ABU NASER MASUD received the Ph.D. degree in computer science *(awarded Apto Cum Laude)* from the Technical University of Madrid, in 2013. He was one of the Erasmus Mundus Scholars to complete European Masters in Computational Logic from the Technical University of Madrid, Spain, and the University of Technology Dresden, Germany. He was a Researcher and also a Post-Doctoral Researcher at Mälardalen University, from 2013 to 2017, and an also an Assistant Professor at Khulna University, Bangladesh, in 2006. He is currently a Senior Lecturer with the Computer Science and Software Engineering Division, Mälardalen University.

He was involved in several research projects funded by the European commission (HATS), Madrid Regional Government (PROMETIDOS-CM), EU Marie Curie (APARTS), and KKS HÖG (SPACES and MOMENTUM) related to software resource analysis, WCET analysis, static program analysis, program transformation, and analysis and transformation of executable models.

BJÖRN LISPER received the M.Sc. degree in engineering physics, in 1980, and the Ph.D. degree in computer science, in 1987, both from the KTH Royal Institute of Technology, Sweden. He was also an appointed "docent" in computer systems, in 1991. He is currently a Professor in computer engineering with Mälardalen University, where he leads the Programming Languages Research Group. His current research interests are in programming language issues, targeting program analysis, especially with respect to timing properties. He has been a Core Member of the FP7 NoE ArtistDesign, where he was the Leader of the timing analysis activity. He coordinated the FP7 ICT Project ALL-TIMES, and the Marie Curie IAPP Project APARTS. He is also a member of the IFIP WG 10.2 on Embedded Systems. He is the Chair of the COST Action IC1202 Timing Analysis on Code-Level.

FEDERICO CICCOZZI received the Ph.D. degree from the School of Innovation, Design and Engineering, Mälardalen University, Sweden, in 2014. He is currently an Associate Professor with the School of Innovation, Design and Engineering, Mälardalen University. His research focuses on the definition of metamodels and model transformations for several automation aspects in the model-driven development of component-based embedded real-time systems, such as code generation, preservation of system properties, back-propagation, to mention a few. Moreover, he carries out research in the areas of multi-paradigm modeling, model versioning, co-evolution and synchronization, as well as the application of model-driven and component-based techniques to multi-robot systems. He has co-authored over 75 publications in journals and international conferences and workshops in these areas. He has been serving the community as the Conference Track and Workshop Organizer, an Expert Panelist, a Program Committee Member, and also as a Reviewer for conferences, workshops, and international journals. In his research activity, he has collaborated with several companies and research institutions such as Ericsson, ABB, Alten, Thales, Saab, CEA list, and so on. More information is available at http://www.es.mdh.se/staff/266-Federico_Ciccozzi.